XSLT 実習マニュアル

第三版

大黒学

XSLT 実習マニュアル・第三版
著者——大黒学

2005 年　1 月 30 日（日）　第零版発行
2006 年　1 月 17 日（火）　第一版発行
2006 年　2 月 22 日（水）　第二版発行
2007 年　2 月 15 日（木）　第三版発行

目次

第 1 章　XML の基礎

1.1　XML の基礎の基礎

1.1.1　この章について

この「XSLT 実習マニュアル」という文章は、XSLT というものについて解説することを目的として書かれたものです。読者として想定しているのは、XSLT について予備知識のまったくない方々です。

ですから、まず最初に、そもそも XSLT というのはいったい何なのか、ということについて説明しないといけません。しかし、XSLT が何なのかということを理解するためには、その前に、XML というものについて理解しておく必要があります。

そういうわけで、この章では、XSLT についての説明に先立って、まず XML について簡単に説明しておきたいと思います。

1.1.2　文書

XML について説明する前に、XML を理解するために必要となる若干の予備知識について説明しておきたいと思います。

最初の予備知識は、「文書」(document) という言葉の意味です。この言葉は、少なくともコンピュータのソフトに関連する文脈では、何らかの情報を表現しているもの、という意味で使われます。「文書」と同じ意味を持つ言葉として「データ」(data) という言葉もあります。「データ」と「文書」は、意味としては同じですが、ニュアンスが少し違いますので、文脈によって使い分けることもあります。

1.1.3　言語

文書は、多くの場合、文字を並べることによって作られます。文字を並べることによって何らかの情報を表現するためには、あらかじめ、文字の並べ方とその意味に関する規則を定めておく必要があります。そのような規則は、「言語」(language) と呼ばれます。

言語は、文字の並べ方に関する規則と、文字列と意味との対応を定めた規則から構成されます。前者は「構文論」(syntax) または「文法」(grammar) と呼ばれ、後者は「意味論」(semantics) と呼ばれます。

世の中にはさまざまな言語がありますが、それらは、「自然言語」(natural language) と「人工言語」(artificial language) という二つの種類のどちらかに分類されます。

自然言語というのは、人間の社会の中で自然に発生してきた言語のことです。自然言語の例としては、日本語、中国語、アラビア語、スペイン語などがあります。

それに対して、人工言語というのは、人間が意図的に作り出した言語のことです。たとえば、プログラムを書くために使われる、awk、Prolog、ML、C++、Ruby などの、「プログラミング言語」(programming language) と呼ばれるさまざまな言語は、人工言語の例です。

1.1.4　マークアップ言語

文書というものは、たいていの場合、その内部に構造を持っています。たとえば、住所録という文書は、氏名と住所の組がいくつか集まってできているという構造を持っています。

文書をコンピュータで処理する場合、その構造をコンピュータに理解してもらうためには、構造を記述するための文字列を文書の中に埋め込むことが必要になります。たとえば、

井原量子岩手県盛岡市近松知美愛媛県松山市上田桃子山梨県甲府市

と書かれた住所録の構造をコンピュータに理解させることは困難ですが、

```
person(name(井原量子),address(岩手県盛岡市)),
person(name(近松知美),address(愛媛県松山市)),
person(name(上田桃子),address(山梨県甲府市))
```

と書き直せば、その構造を理解させることが少しは容易になります。

構造を記述するために文書の中に埋め込まれる文字列は、「マークアップ」(markup) と呼ばれます。上の例では、`person` と `name` と `address` という名前と、丸括弧とコンマという区切り文字が、マークアップとして使われています。

文書の中にマークアップを埋め込むことによってその構造を記述するためには、マークアップに関する構文論と意味論から構成される言語を定める必要があります。そのような言語は、「マー

名前	用途
XHTML	ウェブページの記述。
SVG	ベクター形式のグラフィックスの記述。
SMIL	マルチメディアのコンテンツの記述。
MathML	数式の記述。
OWL	ウェブのオントロジーの記述。
XSLT	XML 文書の変換の記述。
XSL-FO	文書の組版の記述。
XML Schema	XML 応用言語の定義の記述。

表 1.1: XML 応用言語の実例

クアップ言語」(markup language) と呼ばれます。マークアップ言語の例としては、troff、LaTeX、HTML、RD などがあります。

1.1.5 メタ言語

文書の構造を記述するために文書の中に埋め込むマークアップの構文は、マークアップ言語を設計する人が自由に決めることができます。しかし、さまざまな文書のマークアップの構文が文書の種類ごとにまちまちだというのは、かなり不便なことです。

文書を処理するプログラムは、それらの文書の構造を解析しないといけません。もしも、文書のマークアップの構文が統一されていないとすると、プログラムの中に含まれる、文書の構造を解析する部分は、文書の種類ごとに違ったものを書かないといけなくなります。もしもマークアップの構文が統一されていれば、文書の構造を解析する部分は、さまざまなプログラムで共有することができますので、とても便利です。

言語を定義するための一般的な規則から構成されている言語は、「メタ言語」(metalanguage) と呼ばれます。メタ言語を定めておいて、それに基づいて個々の具体的な言語を設計することによって、それらの言語の基本的な部分を統一することができます。

マークアップ言語に関しても、マークアップの構文をメタ言語として定めておいて、そのメタ言語に基づいて個々のマークアップ言語を設計するようにすれば、マークアップの構文を統一することができます。

1.1.6 SGML と XML

1986 年、国際標準化機構 (International Organization for Standardization, ISO) という組織は、SGML(Standard Generalized Markup Language) というマークアップ言語のメタ言語を制定しました。しかしこの言語は、柔軟性があまりにも高すぎて、それを処理するためにはきわめて複雑なプログラムを書く必要がある、などの理由で、あまり普及しませんでした。

1998 年、W3C(World Wide Web Consortium) という組織は、XML(Extensible Markup Language) というマークアップ言語のメタ言語を「勧告」(recommendation) として公開しました。XML は、SGML に対して、柔軟性を低くしてプログラムを書きやすくしたり、新しい概念を導入したりするという改良を加えてできた言語です。

XML の仕様にしたがって書かれた文書は、「XML 文書」(XML document) と呼ばれます。そして、XML 文書の構造を解析するプログラムの部品は、「XML プロセッサー」(XML processor) と呼ばれます。

1.1.7 XML 応用言語

XML というのはマークアップ言語のメタ言語ですから、それ自体は、マークアップの構文を定めているだけで、特定の目的を持つ文書を書くためのマークアップ言語ではありません。ですから、特定の目的を持つ XML 文書を書くためには、その目的のためのマークアップ言語を XML に基づいて設計する必要があります。

XML に基づいて設計されたマークアップ言語は、「XML 応用言語」(XML application) と呼ばれます。XML 応用言語にはさまざまなものがありますが、それらのうちで比較的重要と思われるものを表 1.1 に挙げてみました。

1.2　要素

1.2.1　要素と要素型

XML 文書は、小さな部品が組み合わさって大きな部品ができている、という構造を持っています。XML 文書の構造を作るための部品となる文字列は、「要素」(element) と呼ばれます。

要素は、かならず何らかの種類に属しています。要素の種類は、「要素型」(element type) と呼ばれます。要素型は、「要素型名」(element type name) と呼ばれる名前によって識別されます。

1.2.2　タグ

XML 文書の中に要素を書くためには、その要素型と、それがどこから始まってどこで終わっているのかということを示すマークアップを書く必要があります。そのようなマークアップは、「タグ」(tag) と呼ばれます。

すべてのタグは、小なり (less than, <) という文字で始まって大なり (greater than, >) という文字で終わります。つまり、タグというのはかならず、

< 文字列 >

という形になっているということです。

タグは、3 種類に分類することができます。3 種類のタグは、それぞれ、「開始タグ」(start-tag)、「終了タグ」(end-tag)、「空要素タグ」(empty-element tag) と呼ばれます。それぞれのタグは、

開始タグ	< 要素型名 >	例 `<tamanegi>`
終了タグ	</ 要素型名 >	例 `</daikon>`
空要素タグ	< 要素型名 />	例 `<hakusai/>`

という構文を持っています。このように、タグの種類は、スラッシュ(slash, /) という文字の有無と位置によって示されます。

なお、開始タグと空要素タグの内部には、「属性指定」と呼ばれるものを書くこともできるのですが、それについてはもう少しあとで説明します。

1.2.3　要素の作り方

ひとつの要素は、一組の開始タグと終了タグのペアによって作られるか、または、ひとつの空要素タグによって作られます。開始タグと終了タグのペアによって作られる要素というのは、

開始タグ	文字列	終了タグ

という形の文字列のことです。たとえば、

```
<renkon>I am the content of this element.</renkon>
```

というのは、開始タグと終了タグのペアによって作られた要素の例です。開始タグと終了タグのあいだに書かれた文字列は、要素の「内容」(content) と呼ばれます。上の例では、

```
I am the content of this element.
```

という文字列が要素の内容になっています。

空要素タグは、それ自体がひとつの要素になります。たとえば、

```
<ninjin/>
```

というのは、空要素タグによって作られた要素の例です。空要素タグによって作られた要素は、内容を持ちません。

要素の内容は、いくつかの要素を含んでいてもかまいません。つまり、要素は「入れ子にする」(nest) ことができるということです。たとえば、

```
<words>I am <em>not</em> a cat.</words>
```

という要素は、その中に `<em>not</em>` という要素が入れ子になっています。

1.2.4 ホワイトスペース

空白 (space)、タブ (tab)、復帰 (carriage return)、改行 (line feed) という文字は、総称して「ホワイトスペース」(whitespace) と呼ばれます。

開始タグまたは終了タグを書くとき、大なり (>) の左側には 0 個以上のホワイトスペースを書いてもかまいません。また、空要素タグを書くとき、スラッシュ大なり (/>) の左側にも 0 個以上のホワイトスペースを書くことができます。

1.2.5 属性

要素は、1 個の文字列を保持することができる容器のようなものをいくつでも持つことができます。それらの容器は、「属性」(attribute) と呼ばれます。属性は、「属性名」(attribute name) と呼ばれる名前によって識別されます。

それぞれの属性が保持している文字列は、その属性の「属性値」(attribute value) と呼ばれます。

要素が持っている属性に対して文字列を設定したいときは、その要素の開始タグまたは空要素タグの中に、「属性指定」(attribute specification) と呼ばれる文字列を書きます。属性指定というのは、

属性名="文字列"

という構文を持つ文字列のことです。このような文字列を書くことによって、二重引用符 (double quote, ") で囲まれた内側にある文字列を、属性名で指定された属性に対して属性値として設定することができます。

たとえば、distance という名前の属性に 83km という属性値を設定したいという場合は、

```
distance="83km"
```

という属性指定を書きます。

属性指定を書くことのできる場所は、開始タグまたは空要素タグの要素型名の右側です。ひとつのタグの中には、属性指定を何個でも書くことができます。ただし、要素型名と属性指定とのあいだと、属性指定と属性指定とのあいだには、少なくとも 1 個のホワイトスペースを書く必要があります。たとえば、

```
<student grade="3" class="d" no="28" sex="female"
    name="Mita Hiroko" tel="0874-20-9517"/>
```

というのは、6 個の属性指定を含む空要素タグの例です。

なお、属性値として設定する文字列は、二重引用符ではなくてアポストロフィー (apostrophe, ') で囲んでもかまいません。文字列を二重引用符で囲んだ場合、その文字列の中に二重引用符を書くことはできませんが、アポストロフィーで囲んだ場合は、二重引用符を書くことができます。逆に、文字列をアポストロフィーで囲んだ場合、アポストロフィーは書くことができなくて、二重引用符は書くことができます。

1.2.6 文字データ

XML 文書のうちでマークアップではない部分のことを、「文字データ」(character data) と呼びます。たとえば、

```
<words>I am <em>not</em> a cat.</words>
```

という要素から文字データだけを取り出すと、

```
I am not a cat.
```

という文字列になります。

1.3 整形式の XML 文書

1.3.1 正しさの二つのレベル

XML というメタ言語が定めている構文にしたがって書かれた文書は、「整形式の XML 文書」(well-formed XML document) と呼ばれます。

この節では、XML 文書が整形式であるためにはそれをどのように書かないといけないのか、つまり XML 文書の構文について説明したいと思います。

実は、XML 文書の「正しさ」には、「整形式である」というレベルよりもさらに高いレベルの正しさがあります。整形式よりも高いレベルの正しさを持つ XML 文書は、「妥当な XML 文書」(valid XML document) と呼ばれます。

妥当な XML 文書については、次の節で説明したいと思います。

1.3.2　XML 文書の構成

XML 文書は三つの部分から構成されます。それらの部分は、「XML 宣言」(XML declaration)、「文書型宣言」(document type declaration)、「ルート要素」(root element) と呼ばれます（ルート要素は、「文書要素」(document element) と呼ばれたり、「XML インスタンス」(XML instance) と呼ばれたりすることもあります)。

これらの三つの部分のうちで、ルート要素は、XML 文書が整形式であるために絶対に必要ですが、XML 宣言と文書型宣言は書いても書かなくてもどちらでもかまいません。

なお、文書型宣言については次の節で説明することにします。

1.3.3　XML 宣言

XML 宣言は、文書を書くために使われている XML のバージョンを XML プロセッサーに伝えるための記述です。これは、書かなかったとしても間違いとは言えないのですが、書いておくことが望ましい記述です。なお、XML 宣言を書く場合は、かならず XML 文書の先頭に書かないといけません。

XML 宣言は、基本的には、

```
<?xml version="バージョン"?>
```

と書きます。現在のところ、XML のバージョンは 1.0 だけしかありませんので、XML 宣言は、

```
<?xml version="1.0"?>
```

と書けばいい、ということになります。

XML 宣言の中には、XML のバージョンだけではなくて、文書を書くために使われている符号化方式の名前を書くこともできます。「符号化方式」(encoding) というのは、いわゆる「文字コード」(character code) のことです。日本語の XML 文書を書く場合は、次のいずれかの符号化方式を使います。

UTF-8	UCS Transformation Format 8bit
UTF-16	UCS Transformation Format 16bit
ISO-2022-JP	いわゆる JIS コード
Shift_JIS	Windows で使われているもの
EUC-JP	UNIX で使われているもの

符号化方式の名前を XML プロセッサーに伝えたい場合は、

```
<?xml version="バージョン" encoding="符号化方式"?>
```

という形の XML 宣言を書きます。たとえば、符号化方式として Shift_JIS を使っているということを XML プロセッサーに伝えたいならば、

```
<?xml version="1.0" encoding="Shift_JIS"?>
```

という XML 宣言を書けばいいわけです。

1.3.4　ルート要素

ルート要素というのは、XML 文書の本体に相当する部分です。

ルート要素は、ひとつの要素になっていないといけません。つまり、ひとつの XML 文書に含まれるすべての要素は、ルート要素を根とするひとつの木になっている必要があるということです。

XML 宣言と文書型宣言を書くことは、XML 文書が整形式であるための必要条件ではありません。ですから、整形式の XML 文書に最低限必要なのはルート要素だけです。極端な例ですが、

```
<a/>
```

という 4 文字の文字列も、立派な整形式の XML 文書です。

1.3.5 注釈

XML 文書を書くとき、プログラムによって処理されるべき内容ではないけれども、その文書を読む人間に伝えたいことを、その文書の一部分として書いておきたい、ということがしばしばあります。そのような、プログラムではなくて人間に読んでもらうために文書の中に埋め込まれた文字列は、「注釈」(comment) と呼ばれます。

注釈を書くときは、XML プロセッサーが、「ここからここまでは注釈だ」ということを認識することができるように、文字列を注釈にするための文法にしたがって書く必要があります。XML の文法では、注釈は、

<!-- 文字列 -->

というように、<!-- で始まって --> で終わるように書く、と定められています。<!-- と --> とのあいだには、基本的にはどんな文字列を書いてもかまわないのですが、その文字列の中に、連続する 2 個のマイナス、つまりマイナスマイナス (--) という文字列を書くことはできません。

注釈は、基本的には XML 文書の中のどこに書いてもかまいません。ただし、タグの中に注釈を書いたり、XML 宣言よりも前に注釈を書いたりすることはできません。

1.3.6 実体参照

XML は、文字列に名前を与えておいて、その文字列を名前で参照することができるという機能を持っています。文字列に与えられた名前は「実体名」(entity name) と呼ばれ、文字列を名前で参照するために書かれる文字列は「実体参照」(entity reference) と呼ばれます。実体参照は、

& 実体名 ;

という構文を持つ文字列です。たとえば、 tamanegi というのが実体名だとするとき、

```
&tamanegi;
```

という実体参照は、その実体名が与えられている文字列を参照します。

文字列に名前を与えるためには、「実体宣言」(entity declaration) と呼ばれるものを書く必要があります。ただし、XML では、五つの文字に対しては最初から名前が与えられています。それらの五つの文字と実体名は次のとおりです。

小なり (<)	lt
大なり (>)	gt
二重引用符 (")	quot
アポストロフィー (’)	apos
アンパサンド (&)	amp

XML 文書の中では、小なり (<) という文字はタグの始まりという意味で使われますので、それ以外の目的で小なりを使うことはできません。アンパサンド (ampersand, &) という文字も、実体参照の始まりという意味で使われますので、それ以外の目的では使えません。しかし、小なりやアンパサンドを書く代わりに、

```
&lt;  &amp;
```

という実体参照を書くことによって、独自の目的で小なりやアンパサンドを使うことが可能になります。

また、二重引用符 (") で囲まれた場所に二重引用符を書くことはできませんし、アポストロフィー (’) で囲まれた場所にアポストロフィーを書くことはできません。しかし、この問題も、それらの文字の代わりに実体参照を書くことで解決することができます。

1.3.7 文字参照

XML は、文字コードにしたがって文字をあらわしている整数を指定することによって文字を参照することができるという機能を持っています。整数で文字を参照する記述は、「文字参照」(character reference) と呼ばれます。文字参照は、

&# 10 進数 ; または &#x 16 進数 ;

という構文を持つ文字列です。この中の 10 進数または 16 整数として、文字をあらわす整数を書くことによって、その文字を参照することができます。たとえば、 A は大文字の A を参照する文字参照で、 b は小文字の b を参照する文字参照です。

文字参照は、ホワイトスペースに属する文字を記述するためにしばしば使われます。つまり、空白を 、タブを 	 、復帰を
 、改行を
 と書くわけです。

1.3.8　CDATA セクション

XML プロセッサーは、小なり (<) という文字を、タグの始まりをあらわす特別な文字だと解釈します。また、アンパサンド (ampersand, &) という文字も、実体参照または文字参照の始まりをあらわす特別な文字だと解釈します。

しかし、場合によっては、小なりやアンパサンドを XML とは異なる意味で使っている文字列を XML 文書の中に書く必要が生じることもあります。独自の意味を持つ小なりやアンパサンドを XML 文書の中に書くための方法としては、 < や & を使うという方法があるわけですが、方法はそれだけではなくて、もうひとつあります。それは、「CDATA セクション」(CDATA section) と呼ばれるものを XML 文書の中に書くという方法です。

CDATA セクションというのは、

<![CDATA[文字列]]>

という形の記述のことです。CDATA セクションの内容、つまり、<![CDATA[の直後から]]> の直前までの文字列は、その構造を解析されることなく、文字列そのものとして扱われます。つまり、書かれている文字が書かれているとおりのものとして解釈されるということです。ですから、その文字列の中に小なりやアンパサンドが含まれていたとしても、それらは、XML での特殊な意味を持つ文字だとは解釈されないわけです。

CDATA セクションの内容としては、基本的にはどんな文字列を書いてもかまわないのですが、ひとつだけ例外があります。CDATA セクションの内容として、]]> という文字列を含む文字列を書くことはできません。

1.4　妥当な XML 文書

1.4.1　妥当な XML 文書とは何か

前の節で説明したように、整形式の XML 文書というのは、XML の構文にしたがって書かれた文書のことです。そして妥当な XML 文書というのは、整形式の XML 文書よりも高いレベルの正しさを持つ XML 文書のことです。

XML 文書が妥当であるための条件は、整形式であることと、そして、文書型宣言を持っていて、その文書型宣言の中に書かれている制約を満足していることです。

1.4.2　文書型宣言

文書型宣言 (document type declaration) というのは、基本的には、XML 文書の構文に対する制約 (constraint) を記述したものだと考えることができます。文書型宣言を書く場所は、XML 宣言とルート要素とのあいだです。

文書型宣言は、DTD(document type definition) と呼ばれる文法にしたがって記述します。

文書型宣言は、

<!DOCTYPE 名前 [マークアップ宣言 ···]>

という構文を持つ文字列です。この中の「名前」のところには、ルート要素の要素型名を書きます。

文書型宣言の中には、「マークアップ宣言」(markup declaration) と呼ばれるものを何個でも好きなだけ書くことができます。

1.4.3　要素型宣言

マークアップ宣言にはいくつかの種類があって、そのひとつに、「要素型宣言」(element type declaration) と呼ばれるものがあります。これは、要素の内容についての制約を記述するための宣言です。

要素型宣言は、

```
<!ELEMENT 名前 内容仕様>
```

という構文を持つ文字列です。この中の「名前」のところには、制約を記述する対象となる要素の要素型名を書きます。そして「内容仕様」のところには、その要素の内容についての制約を書きます。たとえば、内容仕様として EMPTY と書くことによって、その要素は常に空要素でなければならないという制約を記述することができます。ですから、

```
<!ELEMENT hoge EMPTY>
```

という要素型宣言は、hoge という要素は常に空要素でなければならないという意味になります。

妥当な XML 文書というのは、整形式になっていて、文書型宣言を持っていて、その文書型宣言の中に書かれている制約を満足している XML 文書のことですから、たとえば次の XML 文書は、妥当な XML 文書の一例です。

```
<?xml version="1.0"?>
<!DOCTYPE namako [ <!ELEMENT namako EMPTY> ]>
<namako/>
```

1.4.4 システム識別子

文書型宣言は、別のファイルの中に書かれたマークアップ宣言の列を参照することも可能です。

マークアップ宣言の列を格納したファイルは、「DTD ファイル」(DTD file) と呼ばれます。DTD ファイルの標準的な拡張子は .dtd です。

DTD ファイルの内容を参照する文書型宣言は、

```
<!DOCTYPE 名前 外部識別子 [
    マークアップ宣言 ・・・
]>
```

と書きます。「外部識別子」(external identifier) というのは、XML 文書の外部にあるファイルを識別するための記述のことです。外部識別子は、基本的には、

```
SYSTEM "システム識別子"
```

と書きます。「システム識別子」(system identifier) というのは、ファイルを識別するためのパス名などの文字列のことです。XML 文書と同じディレクトリにあるファイルを参照する場合は、そのファイル名がシステム識別子になります。

なお、文書型宣言の中にマークアップ宣言をまったく書かない場合、それを囲む角括弧は省略することができます。

たとえば、

```
<!DOCTYPE hitode SYSTEM "hitode.dtd">
```

という文書型宣言を書くことによって、その XML 文書と同じディレクトリにある hitode.dtd という DTD ファイルの内容を参照することができます。

1.4.5 公開識別子

標準的な XML 応用言語を使って XML 文書を書くときに、その XML 応用言語の DTD ファイルを識別するための外部識別子の中に、「公開識別子」(public identifier) と呼ばれる文字列を書くことが要求される場合もあります。

公開識別子というのは、ファイルを識別するための文字列の一種で、XML 応用言語を定義した組織の名前やデータの種類などから構成される文字列です。たとえば、XHTML 1.1 という XML 応用言語の DTD ファイルは、

```
-//W3C//DTD XHTML 1.1//EN
```

という公開識別子によって識別されます。

公開識別子を使ってファイルを識別する場合は、外部識別子として、

```
PUBLIC "公開識別子" "システム識別子"
```

という形の記述を書きます（たとえ公開識別子を書く場合でも、システム識別子を省略することはできません）。たとえば、XHTML 1.1 という XML 応用言語の DTD ファイルを識別する外部

識別子は、

```
PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"
```

と書きます。

1.5　名前空間

1.5.1　名前空間とは何か

XML 文書は、何らかの XML 応用言語を使って書かれるわけですが、ひとつの XML 文書の中で使われる XML 応用言語は、ひとつだけとは限りません。つまり、ひとつの XML 文書の中に、いくつかの異なる XML 応用言語で定義された要素型名や属性名が混在していることもあるということです。

XML 応用言語で使われる要素型名や属性名は、その言語を設計する人間や機関が自由に決めることができます。ですから、それぞれの XML 応用言語で、同じ名前が異なる目的で使われていることもあります。たとえば、`note` という名前が、ある言語では音符をあらわす要素の要素型名として使われていて、別の言語では注記をあらわす要素の要素型名として使われているかもしれません。

同じ名前を異なる目的で使っている複数の XML 応用言語を使ってひとつの XML 文書を書く場合には、その名前がどちらの言語で定義されているものなのかということを識別することができないといけません。

XML では、「名前空間」(namespace) と呼ばれる領域の中に要素型名や属性名があると考えることによって、ひとつの XML 文書の中で使われる名前が、どの XML 応用言語で定義されたものなのかということを識別することができるようになっています。

1.5.2　URI

「資源」(resource) という言葉は、しばしば、プログラムがアクセスする対象となるさまざまなものの総称として使われます。

プログラムが資源にアクセスするためには、その資源を指定する記述が必要になります。資源を指定する記述を書く方法は、二つあります。ひとつは資源の位置を記述する方法で、その方法で書かれた文字列は URL(uniform resource locator) と呼ばれます。そしてもうひとつは、資源を特定することのできる名前によってそれを記述する方法で、その方法で書かれた文字列は URN(uniform resource name) と呼ばれます。

URL というのは、広告などでよく見かける、

```
http://www.example.com/product/
```

というような文字列のことです。また、ディレクトリの相対的な位置を記述することによってファイルを指定する、

```
../../index.html
```

というような文字列も URL の一種です。

URN は、`urn:` という文字列の右側に、資源を特定することのできる名前を書いたものです。資源を特定することのできる名前の例としては、出版物を指定するために使われる ISBN(International Standard Book Number) と呼ばれる番号があります。書籍の奥付などに記載されている 10 桁の数字の列が ISBN です。ISBN を使うことによって、たとえば、

```
urn:isbn:4-00-327921-2
```

というような URN を書くことができます。

XML では、何らかの資源を指定するために、URI(uniform resource identifier) と呼ばれる文字列を使います。URI というのは、URL と URN の上位概念です。つまり、XML では URL と URN の両方を使うことができるということです。

1.5.3　名前空間名

名前空間は、「名前空間名」(namespace name) と呼ばれる名前によって識別されます。名前空間名としては、`http:` で始まる URI が使われます。たとえば、XHTML という XML 応用言語で使われる要素型名や属性名は、

```
http://www.w3.org/1999/xhtml
```

という名前空間名を持つ名前空間の中にあります。

URI というのは、本来は何らかの資源を指定するためのものですが、名前空間名として使われている URI は、かならずしも何らかの資源を指定しているとは限りません。

1.5.4 QName

名前空間の中にある要素型名や属性名は、XML 文書の中では、「QName」(qualified name) と呼ばれる名前によって識別されます。

QName は、

名前空間接頭辞 : ローカル部分

という構造を持つ名前です。「名前空間接頭辞」(namespace prefix) というのは名前空間を識別するための接頭辞のことで、「ローカル部分」(local part) というのは名前空間の中にある名前のことです。

たとえば、`mus` という名前空間接頭辞によって識別される名前空間の中にある `note` という要素型名は、

```
mus:note
```

という QName によって識別されます。

1.5.5 名前空間宣言

名前空間接頭辞を使うためには、あらかじめ、名前空間名と名前空間接頭辞とを結び付けておく必要があります。名前空間名と名前空間接頭辞は、「名前空間宣言」(namespace declaration) と呼ばれる特殊な属性指定を書くことによって結び付けることができます。

名前空間宣言は、

`xmlns:` 名前空間接頭辞 `="` 名前空間名 `"`

という構造を持つ属性指定です。このような属性指定を要素の開始タグの中に書いておくと、その属性指定を含んでいる要素自身と、それに含まれる要素で、その名前空間接頭辞を使うことができます。たとえば、

```
<mus:music xmlns:mus="http://www.example.org/music"
           xmlns:doc="http://www.example.org/document">
  <mus:note pitch="C" length="4"/>
  <doc:note>I am a comment.</doc:note>
</mus:music>
```

というように、名前空間と名前空間接頭辞とを結び付けて、その名前空間接頭辞を使って QName を書くことができます。

1.5.6 デフォルト名前空間

XML 文書の中で、名前空間接頭辞が省略された場合は特定の名前空間が指定されたものとみなす、という設定をすることも可能です。そのように設定された名前空間は、「デフォルト名前空間」(default namespace) と呼ばれます。

特定の名前空間をデフォルト名前空間に設定したいときは、「デフォルト名前空間宣言」(default namespace declaration) と呼ばれる属性指定を書きます。デフォルト名前空間宣言は、

`xmlns="` 名前空間名 `"`

と書きます。

デフォルト名前空間を設定することによって、たとえば、

```
<music xmlns="http://www.example.org/music"
       xmlns:doc="http://www.example.org/document">
  <note pitch="C" length="4"/>
  <doc:note>I am a comment.</doc:note>
</music>
```

というように、名前空間接頭辞を省略することができるようになります。

第 2 章　XSLT の基礎

2.1　XSLT の基礎の基礎

2.1.1　XSLT の概要

XSLT(Extensible Stylesheet Language Transformations) は、XML 応用言語のひとつで、XML 文書の変換を記述することを目的とする言語です。XSLT 1.0 と呼ばれる、この言語の最初のバージョンは、1999 年に W3C によって勧告として公開されました。

「変換」(transformation) というのは、何らかの規則にもとづいて、何かの形を別の形へ変えるということです。XSLT によって記述されるのは、XML 文書を別の形の XML 文書へ変えるという変換です。ただし、基本的にはそうなのですが、XML 文書を XML ではない文書に変えるという変換を XSLT を使って記述することも可能です。つまり、XML 文書から、プレーンテキストや、XML 以外のマークアップ言語（たとえば troff や LaTeX など）の文書への変換も、XSLT を使って記述することができるということです。

XSLT を使って記述された文書は、「XSLT スタイルシート」(XSLT stylesheet) と呼ばれます。XSLT は XML の応用言語ですから、XSLT スタイルシートは、変換の対象だけではなくて、自分自身も XML 文書になります。

XSLT スタイルシートに記述されている変換を XML 文書に対して実行して、その結果を出力する、という動作をするプログラムは、「XSLT プロセッサー」(XSLT processor) と呼ばれます。XSLT プロセッサーの実例としては、Xalan、XT、Saxon などがあります。

2.1.2　ツリー

全体がいくつかの部分に分かれていて、それぞれの部分がさらにいくつかの部分に分かれていて……という構造は、全体が根に相当して、それぞれの部分が枝でつながっている、木のような形になっていると考えることができます。そのような木の形の構造は、「ツリー」(tree) と呼ばれます。たとえば、身近なツリーの例として、ディレクトリ（フォルダ）の構造を挙げることができます。そしてまた、XML 文書の構造も、ツリーの一例です。

XSLT によって記述される変換というのは、XML 文書のツリーを別の形のツリーに変形することだと考えることができます。XSLT スタイルシートによって変換される対象となるツリーは「ソースツリー」(source tree) と呼ばれ、変換によって作られるツリーは「結果ツリー」(result tree) と呼ばれます。また、XSLT スタイルシートから作られたツリーは、「スタイルシートツリー」(stylesheet tree) と呼ばれます。

XSLT プロセッサーは、ツリーからツリーへの変換が終わったのち、結果ツリーを文字列に変換して、その結果として得られた文字列を出力します。ツリーを文字列に変換することを、ツリーを「シリアライズする」(serialize) と言います。

以上の説明にもとづいて、XSLT プロセッサーの動作を先ほどよりももう少し細かく書くと、

(1) XML 文書と XSLT スタイルシートを読み込む。
(2) XML 文書を解析してソースツリーを作り、XSLT スタイルシートを解析してスタイルシートツリーを作る。
(3) スタイルシートツリーにもとづいてソースツリーを変換して結果ツリーを作る。
(4) 結果ツリーをシリアライズして、その結果を出力する。

ということになります。

2.1.3　ノード

あるものが別のものの「一部分になっている」という関係は、「親子関係」(parent-child relationship) と呼ばれます。A と B という二つのものがあって、B が A の一部分になっているとき、A は B の「親」(parent) と呼ばれ、B は A の「子供」(child) と呼ばれます。

ツリーは、「ノード」(node) と「枝」(branch) と呼ばれる 2 種類のものから構成されます。枝というのは親子関係をあらわしていて、ノードというのは、親子関係によって結ばれているそれぞれのものをあらわしています。

なお、ツリーを図で描く場合は、根になっているノードをいちばん上に描いて、下に向かって枝が伸びているように描くというのが普通です。

XML 文書を構成するそれぞれのノードは、「ノード型」(node type) と呼ばれるいくつかの種

類に分類することができます。ノード型としては、要素ノード (element node)、テキストノード (text node)、注釈ノード (comment node)、ルートノード (root node)、属性ノード (attribute node) などがあります。

2.1.4 要素ノード

XML 文書を構成するそれぞれの要素は、「要素ノード」(element node) と呼ばれるノードになります。

要素ノードは、自分の一部分になっている要素ノードを自分の子供として持つことになります。たとえば、

```
<words><something/></words>
```

という要素ノードは、<something/> という要素ノードを自分の子供として持っています。

2.1.5 テキストノード

ルート要素の内側にある、タグを含まないで連続している文字列は、「テキストノード」(text node) と呼ばれるノードになります。

要素ノードは、自分の一部分になっているテキストノードを、自分の子供として持つことになります。たとえば、

```
<words>I am <em>not</em> a cat.</words>
```

という要素ノードは、'I am ' と ' a cat.' という二つのテキストノードと、em という要素型の要素ノードを自分の子供として持っていて、'not' というテキストノードを自分の孫（つまり子供の子供）として持っています。

ホワイトスペースだけから構成される文字列もテキストノードになる、という点に注意してください。たとえば、

```
<words>
  <em>I am not a cat.</em>
</words>
```

という要素ノードは、ホワイトスペースだけから構成される 2 個のテキストノードと、em 要素のノードとを子供として持っています。

ちなみに、ルート要素の外側にあるホワイトスペースは、ツリーを作るときに無視されますので、テキストノードにはなりません。

2.1.6 注釈ノード

XML 文書の中にある注釈は、「注釈ノード」(comment node) と呼ばれるノードになります。

要素ノードは、自分の一部分になっている注釈ノードを、自分の子供として持つことになります。たとえば、

```
<words><!--I am a comment.--></words>
```

という要素ノードは、

```
<!--I am a comment.-->
```

という注釈ノードを自分の子供として持っています。

2.1.7 ルートノード

XML 文書のツリーの根に相当するノード、つまり XML 文書の全体は、「ルートノード」(root node)、と呼ばれます。

ルートノードは、ルート要素のノードや、ルート要素の外側に書かれた注釈のノードなどを子供として持つことになります。たとえば、

```
<?xml version="1.0"?>
<!--I am a comment.-->
<words>I am <em>not</em> a cat.</words>
```

という XML 文書のルートノードは、

```
<!--I am a comment.-->
```

という注釈のノードと、

```
<words>I am <em>not</em> a cat.</words>
```

というルート要素のノードを自分の子供として持っています。

言葉が似ていて紛らわしいのですが、ルートノードとルート要素のノードとは、同じものではなくて、親子の関係にある別々のノードだという点に注意してください。

2.1.8　属性ノード

要素が持っている属性のそれぞれは、「属性ノード」(attribute node) と呼ばれるノードになります。

ツリーを構成しているそれぞれのノードは、親子関係によって結ばれているわけですが、XML 文書のツリーの場合、要素ノードと属性ノードとのあいだの関係だけは例外で、それは「子供として持つ」という関係ではなくて、「属性ノードとして持つ」という関係です。たとえば、

```
<line length="30cm"/>
```

という要素ノードは、

```
length="30cm"
```

という属性ノードを、属性ノードとして持っています。

2.2　XSLT スタイルシート

2.2.1　XSLT スタイルシートのルート要素

前の節ですでに説明したように、XSLT は XML の応用言語ですから、XSLT で記述された文書、すなわち XSLT スタイルシートは、それ自体が XML 文書になります。

XSLT スタイルシートは、整形式の XML 文書でないといけませんが、妥当な XML 文書ではなくてもかまいませんので、通常、文書型宣言は不要です。

XSLT の要素型名は、

```
http://www.w3.org/1999/XSL/Transform
```

という名前空間の中にありますので、それらを使う場合は名前空間宣言を書く必要があります。名前空間接頭辞としては、`xsl` というものを使うのが普通です。

XSLT スタイルシートのルート要素は、`xsl:stylesheet` という要素型を持つ要素です。この要素を書く場合は、それが持っている `version` という属性に、スタイルシートを書くために使う XSLT のバージョンを設定する必要があります。ですから、その開始タグは、通常、

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

と書くことになります。

なお、`xsl:stylesheet` には、`xsl:transform` という別名があります。XSLT スタイルシートのルート要素は、これらの二つの名前のうちのどちらを使って書いてもかまいません。

2.2.2　トップレベル要素

XSLT スタイルシートのルート要素の子供は、0 個以上の要素から構成されます。ルート要素の子供になっている要素は、「トップレベル要素」(top-level element) と呼ばれます。

トップレベル要素は、XSLT で定義されている要素型を使って作ります。XSLT ではさまざまな要素型が定義されていますが、それらの中には、トップレベル要素を作るために使うことのできるものとそうでないものとがあります。XSLT 1.0 は、トップレベル要素を作るために使うことのできる要素型として、12 個の要素型を定義しています。

トップレベル要素は、どんな順序で書いてもかまいません。また、それらの順序は意味を持ちません。ただし、第 7.1 節で登場する `xsl:import` という要素型の要素だけは例外です。その要素は、かならず、それ以外の要素型のトップレベル要素よりも前に書かないといけません。

2.2.3　式

XSLT スタイルシートの中では、「式」(expression) と呼ばれるものを書くことによって、さまざまなものごとを記述することができるようになっています。

XSLT で使われる式は、XPath(XML Path Language) と呼ばれる言語にしたがって書くことになっています。ですから、XPath という言語は XSLT という言語の一部分だということにな

ります。しかし、W3C による XSLT の勧告は、XPath の仕様には言及していません。W3C は、XSLT とは別の勧告として XPath の仕様を公開しているのです。その理由は、XSLT だけに留まらず、さまざまな言語が式の言語を共有できるようにするためです。式の言語として XPath を利用している言語としては、XSLT のほかに、XPointer や XQuery などがあります。

式は、何らかの動作をあらわしていると解釈することができます。式があらわしている動作を実行することを、式を「評価する」(evaluate) と言います。

式を評価すると、その結果として何らかのデータが得られます。式を評価した結果として得られたデータのことを、その式の「値」(value) と呼びます。

式の値は、「データ型」(data type) と呼ばれるいくつかの種類に分類することができます。データ型としては、数値 (number)、文字列 (string)、真偽値 (boolean)、ノード集合 (node-set) などがあります。ちなみに、ノード集合というのは、その名前のとおり、ノードを要素とする集合のことです。

2.2.4 ロケーションパス

XSLT によって記述される変換というのは、XML 文書のツリーを別の形のツリーに変形することだと考えることができます。ツリーの変形を記述するためには、ツリーを構成しているノードのうちのいくつかを、その位置関係にもとづいて指示する記述を書くことが必要になります。

位置関係にもとづいてツリーの中にあるいくつかのノードを指示する記述は、「ロケーションパス」(location path) と呼ばれます。ロケーションパスを書くための規則は、XPath の中で定められています。

ロケーションパスの書き方が XPath によって規定されているということは、ロケーションパスというのは式の一種だということを意味しています。ですから、ロケーションパスというのは評価することができて、評価の結果として何らかの値が得られるということになります。ロケーションパスを評価することによって得られる値は、それによって指示されたノードから構成されるノード集合です。

ロケーションパスの全貌については次の章で解説することになるのですが、ここでは、その一例としてスラッシュ(slash, `/`) というひとつの文字だけで構成されるロケーションパスを紹介しておきたいと思います。スラッシュというのは、ソースツリーのルートノードという位置を指示するロケーションパスです。ですから、このロケーションパスを評価すると、ソースツリーのルートノードだけから構成されるノード集合（つまりソースツリーの全体）が、その値として得られます。

2.3 テンプレートルール

2.3.1 テンプレートルールの基礎

XSLT スタイルシートのトップレベル要素の中で、もっとも中心的な役割を果たすのは、「処理の対象になっているノードがこのような条件を満足しているならば、そのノードを対象としてこのような動作をする」という形の規則をあらわしている要素です。そのような規則をあらわしている要素は、「テンプレートルール」(template rule) と呼ばれます。

XSLT プロセッサーが XML 文書を変換する上でもっとも中心となる処理は、処理の対象になっているノードについて、それを処理するテンプレートルールを探して、そのテンプレートルールを使ってそのノードを処理することです。そのような処理をすることを、ノードにテンプレートルールを「適用する」(apply) と言います。

テンプレートルールは、ノードに関する条件をあらわす記述と、その条件を満足するノードに対して実行される動作をあらわす記述、という 2 種類の記述を含んでいます。

2.3.2 パターン

テンプレートルールに含まれている 2 種類の記述のうちのひとつで、ノードに関する条件をあらわしているものは、「パターン」(pattern) と呼ばれます。パターンがあらわしている条件を満足しているノードは、そのパターンと「一致する」(match) と言われます。

パターンを書くための規則は、式を書くための規則の部分集合です。言い換えれば、すべてのパターンは式として評価することも可能です。しかし、その逆は真ではありません。つまり、すべての式がパターンになるわけではありません。

ちなみに、スラッシュ(/) という文字は、ロケーションパスだけではなくてパターンとしても使うことができます。パターンとしてのスラッシュは、ルートノードだけと一致します。

2.3.3　テンプレート

テンプレートルールに含まれている 2 種類の記述のうちのひとつで、ノードを対象として実行される動作をあらわしているものは、「テンプレート」(template) と呼ばれます。テンプレートがあらわしている動作を実行することは、テンプレートを「インスタンス化する」(instantiate) と言われます。

テンプレートは、基本的には、自分自身をそのまま結果ツリーに挿入するという動作をあらわしています。たとえば、

```
<result>Good morning.</result>
```

というテンプレートをインスタンス化すると、そのテンプレート自身が結果ツリーに挿入されることになります。

2.3.4　変換の開始

XSLT プロセッサーは、読み込んだ XML 文書からソースツリーを作り、読み込んだ XSLT スタイルシートからスタイルシートを作り、そしてルートノードだけから構成される結果ツリーを作ったのち、ソースツリーの変換を開始します。

XSLT プロセッサーがソースツリーの変換を開始して、まず最初にすることは、ソースツリーのルートノードと一致するパターンを持つテンプレートルールを探して、見付かったテンプレートルールをルートノードに適用することです。

そして、ルートノードに適用されたテンプレートルールに含まれているテンプレートのインスタンス化が終了するとともに、変換も終了します。ですから、それ以外のテンプレートルールは、ルートノードに適用されたテンプレートルールから間接的に指示されることによってノードに適用されることになります。

2.3.5　xsl:template 要素

テンプレートルールは、xsl:template という要素型を使って作られる要素です。この要素は、トップレベル要素でないといけません。

テンプレートルールを書くためには、パターンとテンプレートをその中に書く必要があります。パターンは、xsl:template 要素が持っている match という属性に、属性値として設定します。そしてテンプレートは、xsl:template 要素の内容として書きます。ですから、テンプレートルールというのは、

```
<xsl:template match="パターン">
  テンプレート
</xsl:template>
```

という形の要素として書けばいい、ということになります。たとえば、

```
<xsl:template match="/">
  <result>Good morning.</result>
</xsl:template>
```

というテンプレートルールは、処理の対象になっているノードがルートノードならば、

```
<result>Good morning.</result>
```

というテンプレートをインスタンス化する、という規則をあらわしています。

それでは、XSLT プロセッサーを使って、実際に XML 文書を変換してみましょう。まず、次の XSLT スタイルシートを入力して、greet.xsl というファイルに保存してください。

XSLT スタイルシートの例　greet.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result>Good morning.</result>
  </xsl:template>
```

```
</xsl:stylesheet>
```

ちなみに、XSLT スタイルシートを格納するファイルの名前には、このように .xsl という拡張子を付けることになっています。

次に、変換の対象となる XML 文書を入力しましょう。先ほど入力した XSLT スタイルシートは、変換の対象がどんな XML 文書であっても常に同じものを出力しますので、どんな XML 文書を入力してもかまいません。たとえば、empty.xml というファイルに、次のようなものを保存してください。

XML 文書の例　empty.xml

```
<?xml version="1.0"?>
<empty/>
```

XSLT プロセッサーを使って、empty.xml を greet.xsl で変換すると、その結果として次のような XML 文書が得られます（ただし、改行を補って読みやすくしてあります）。

変換結果　empty.xml + greet.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>Good morning.</result>
```

2.4 基本的な命令

2.4.1 命令の基礎

前の節で説明したように、テンプレートをインスタンス化すると、基本的には、そのテンプレートがそのまま結果ツリーに挿入されます。しかし、テンプレートの能力はそれだけではありません（もしもテンプレートの能力がそれだけしかないとすると、ソースツリーの内容を反映した結果ツリーを作ることはできない、ということになってしまいます）。

テンプレートの中には、「命令」(instruction) と呼ばれる要素を書くことができます。命令というのは、テンプレートの中に書かれる要素のうちで、自分自身を結果ツリーに挿入するということ以外の動作をあらわしているもののことです。XSLT プロセッサーは、テンプレートをインスタンス化するとき、その中に命令が含まれていた場合、その命令をそのままの形で結果ツリーに挿入するのではなくて、その命令があらわしている動作を実行します。

命令があらわしている動作を実行することを、テンプレートを実行する場合と同じ言葉を使って、命令を「インスタンス化する」(instantiate) と言います。

テンプレートの中にある要素のうちで、命令ではないもの、つまり結果ツリーにそのまま挿入されるものは、「リテラル結果要素」(literal result element) と呼ばれます。

命令は、命令を作るための要素型を使って作られた要素です。XSLT 1.0 は、命令を作るために使うことのできる要素型として、18 個の要素型を定義しています。

この節では、命令を作るための要素型のうちで比較的基本的な、

```
xsl:copy-of      xsl:value-of
```

という二つの要素型を紹介したいと思います。

なお、この実習マニュアルの中では、「何々という要素型から作られた命令」のことを、「何々命令」と呼ぶことにします。たとえば、xsl:copy-of という要素型から作られた命令は、「xsl:copy-of 命令」と呼ばれることになります。

2.4.2 xsl:copy-of 命令

xsl:copy-of という要素型から作られた命令は、式を評価して、その式の値を結果ツリーに挿入する、という動作をあらわしています。

xsl:copy-of 命令は、select という属性を持っています。この属性に対しては、任意の式を属性値として設定することができます。xsl:copy-of 命令をインスタンス化すると、select 属性に設定されている式が評価されて、その値が結果ツリーに挿入されます。

式の値は、基本的にはそのまま結果ツリーに挿入されるのですが、式の値がルートノードの場合、実際に挿入されるのはルートノードそのものではなくて、ルートノードのすべての子供です。ですから、

```
<xsl:copy-of select="/"/>
```

という命令をインスタンス化した場合、ルートノードのすべての子供が結果ツリーに挿入されます。

次の XSLT スタイルシートは、読み込んだ XML 文書のルートノードのすべての子供を、result という要素の子供にして出力します。

XSLT スタイルシートの例　copyof.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:copy-of select="/"/></result>
  </xsl:template>
</xsl:stylesheet>
```

それでは、この XSLT スタイルシートを使って、次の XML 文書を変換してみてください。

XML 文書の例　notcat.xml

```
<?xml version="1.0"?>
<!--I am a comment.-->
<words>I am <em>not</em> a cat.</words>
```

すると、次のような結果が出力されるはずです（ただし、改行と空白を補って読みやすくしてあります。以下同様）。

変換結果　notcat.xml + copyof.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <!--I am a comment.-->
  <words>I am <em>not</em> a cat.</words>
</result>
```

2.4.3　xsl:value-of 命令

xsl:value-of という要素型から作られた命令は、式を評価して、その式の値を文字列に変換した結果を結果ツリーに挿入する、という動作をあらわしています。評価の対象となる式は、xsl:copy-of の場合と同じように、select という属性に対して属性値として設定します。

ノード集合を文字列に変換すると、そのノード集合に含まれているすべてのテキストノードを連結した文字列が得られます。ですから、

```
<xsl:value-of select="/"/>
```

という命令をインスタンス化すると、ソースツリーのルートノードに含まれているすべてのテキストノードを連結した結果が結果ツリーに挿入されることになります。

次の XSLT スタイルシートは、読み込んだ XML 文書に含まれているすべてのテキストノードを連結することによってできる文字列を、result という要素の内容にして出力します。

XSLT スタイルシートの例　valueof.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:value-of select="/"/></result>
  </xsl:template>
</xsl:stylesheet>
```

この XSLT スタイルシートを使って、先ほどの notcat.xml を変換すると、次のような結果が出力されます。

変換結果　notcat.xml + valueof.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>I am not a cat.</result>
```

2.5 テンプレートルールの適用

2.5.1 明示的なテンプレートルールの適用

これまでに書いてきたXSLTスタイルシートは、すべて、ルートノードに適用される1個のテンプレートルールだけから構成されているものでした。しかし、ルートノードに適用されるテンプレートルールだけで、目的とする変換を実現することができるとは限りません。

第2.3節で説明したように、XSLTプロセッサーは、ルートノードと一致するパターンを持つテンプレートルールを探して、それをルートノードに適用します。しかし、XSLTプロセッサーが自発的にテンプレートルールを適用するノードは、ルートノードだけです。それでは、ルートノード以外のノードに対してテンプレートルールを適用するためには、いったいどうすればいいのでしょうか。

`xsl:apply-templates`という要素型から作られた命令は、テンプレートルールをノードに適用するという動作をあらわします。ですから、この命令をテンプレートの中に明示的に書いておくことによって、ルートノード以外のノードに対してテンプレートルールを適用することができます。

この節では、`xsl:apply-templates`命令を使ってテンプレートルールをノードに適用する方法について説明したいと思います。しかし、そのためには、それを理解するための予備知識について、あらかじめ説明しておく必要があります。

2.5.2 カレントノード

予備知識のひとつ目は、カレントノードです。「カレントノード」(current node)というのは、テンプレートがインスタンス化されているときに、その処理の対象になっているノードのことです。

テンプレートルールの中に書かれたテンプレートは、パターンと一致したノードをカレントノードとしてインスタンス化されます。たとえば、

```
<xsl:template match="/">
  テンプレート
</xsl:template>
```

というテンプレートルールの中のテンプレートは、ルートノードをカレントノードとしてインスタンス化されます。

2.5.3 文脈ノード

予備知識の二つ目は、文脈ノードです。「文脈ノード」(context node)というのは、式が評価されているときに、その処理の対象になっているノードのことです。

テンプレートの中に書かれた式の文脈ノードは、基本的には、そのテンプレートのカレントノードと同じになります。カレントノードと文脈ノードとが同じものではなくなる場合もあるのですが、それについては第3.9節で説明します。

式の中で文脈ノードを求めたいときは、ドット(dot, `.`)というロケーションパスを書きます。このロケーションパスを評価すると、その値として、文脈ノードだけを要素とするノード集合が得られます。

2.5.4 パターンとしての要素型名

予備知識の三つ目は、パターンとしての要素型名です。要素型名は、その要素型を持つ要素ノードと一致するパターンになります。たとえば、`person`というパターンは、`person`という要素型を持つ要素ノードと一致します。ですから、

```
<xsl:template match="person">
  テンプレート
</xsl:template>
```

というテンプレートルールは、`person`という要素型を持つ要素ノードに対して適用されます。

もう少し具体的な例を書いてみましょう。たとえば、

```
<xsl:template match="person">
```

```
    <name><xsl:value-of select="."/></name>
  </xsl:template>
```

というテンプレートルールは、person 要素に適用されて、その中のテキストノードを内容とする name 要素を結果ツリーに挿入します。

さて、xsl:apply-templates 命令について理解するための予備知識の説明は以上で終わって、いよいよ本題に入ることにしたいと思います。

2.5.5　xsl:apply-templates 命令

xsl:apply-templates という要素型から作られた命令は、ノード集合を構成するそれぞれのノードに対して適切なテンプレートルールを適用するという動作をあらわします。

テンプレートルールを適用するノード集合は、属性指定を書くことによって指定することもできますが（第 4.3 節参照）、その指定は省略することもできて、省略すると、カレントノードのすべての子供から構成されるノード集合が適用の対象になります。ですから、

```
<xsl:apply-templates/>
```

という命令は、カレントノードのすべての子供を対象として適切なテンプレートルールを適用する、という動作をあらわします。たとえば、

```
<result><xsl:apply-templates/></result>
```

というテンプレートをインスタンス化すると、カレントノードのすべての子供を対象として適切なテンプレートルールが適用されて、その結果が、result という要素の子供として結果ツリーに挿入されます。

それでは、ルートノード以外のノードに適用されるテンプレートルールを含む XSLT スタイルシートを実際に書いて、それを使って XML 文書を変換してみましょう。

XSLT スタイルシートの例　apply.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="friends">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="person">
    <name><xsl:value-of select="."/></name>
  </xsl:template>
</xsl:stylesheet>
```

この XSLT スタイルシートは、3 個のテンプレートルールから構成されています。

1 個目は、ルートノードに対して適用されるテンプレートルールで、ルートノードのすべての子供を対象として適切なテンプレートルールを適用して、その結果を結果ツリーに挿入します。

2 個目は、friends 要素に対して適用されるテンプレートルールで、friends 要素のすべての子供を対象として適切なテンプレートルールを適用して、その結果を子供とする result 要素を結果ツリーに挿入します。

3 個目は、person 要素に対して適用されるテンプレートルールで、person 要素からテキストノードを取り出して、それを子供とする name 要素を結果ツリーに挿入します。

次に、次のような XML 文書を入力して、ファイルに保存してください。

XML 文書の例　friends.xml

```
<?xml version="1.0"?>
<friends>
  <person>Yokoyama Takashi</person>
  <person>Hamano Yoshio</person>
  <person>Nakayama Akira</person>
</friends>
```

それでは、friends.xml を apply.xsl で変換してみましょう。そうすると、次のような結果が得られます。

変換結果 friends.xml + apply.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <name>Yokoyama Takashi</name>
  <name>Hamano Yoshio</name>
  <name>Nakayama Akira</name>
</result>
```

2.5.6 組み込みテンプレートルール

ところで、XSLT プロセッサーは、処理の対象となったノードに適用することのできるテンプレートルールを探したけれども、それが見付からなかった、という場合、いったいどうするのでしょうか。

XSLT プロセッサーは、テンプレートルールが見付からなかった場合にノードに適用する、「組み込みテンプレートルール」(built-in template rule) と呼ばれるいくつかのテンプレートルールを持っています。

組み込みテンプレートルールのうちの主要なものとしては、次のようなものがあります。

- ルートノードまたは要素ノードに適用されるもの。そのルートノードまたは要素ノードのすべての子供を対象として適切なテンプレートルールを適用する。
- テキストノードに適用されるもの。そのテキストノードをそのまま結果ツリーに挿入する。
- 注釈ノードに適用されるもの。動作は何もしない。

ちなみに、先ほどの apply.xsl という XSLT スタイルシートの中に含まれている、

```
<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>
```

というテンプレートルールは、組み込みテンプレートルールで代用することが可能です。したがって、先ほどの apply.xsl は、次のように書き直したとしても同じ変換をあらわすことになります。

XSLT スタイルシートの例 builtin.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="friends">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="person">
    <name><xsl:value-of select="."/></name>
  </xsl:template>
</xsl:stylesheet>
```

2.5.7 ロケーションパスとしての要素型名

要素型名はロケーションパスの一種です。

要素型名が、式の一部分になっているのではなくて、単独で式になっているとき、それを評価すると、文脈ノードの子供のうちで、その要素型を持つすべての要素ノードから構成されるノード集合が、その値として得られます（文脈ノードの子供のうちで、という制約があることに注意してください)。

XSLT スタイルシートの例 etname.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="friends">
    <result><xsl:copy-of select="person"/></result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果 friends.xml + etname.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <person>Yokoyama Takashi</person>
  <person>Hamano Yoshio</person>
  <person>Nakayama Akira</person>
</result>
```

第3章 式

3.1 式の基礎

3.1.1 式についての復習

この章では、XSLT で使うことのできる式 (expression) について説明していきたいと思います。

式については、すでに前の章で簡単に紹介しましたが、この章を始めるに当たって、まず、そのときに説明したことについて復習しておくことにしましょう。

XSLT で使われる式を作るための言語は、XPath と呼ばれます。XPath は XSLT の一部分ですが、XPath の仕様は、XSLT とは別の独立した勧告で定義されています。

式は、何らかの動作をあらわしていると解釈することができます。式があらわしている動作を実行することを、式を「評価する」(evaluate) と言います。

式を評価すると、その結果として何らかのデータが得られます。式を評価したときの結果として得られたデータのことを、その式の「値」(value) と呼びます。

式の値は、「データ型」(data type) と呼ばれるいくつかの種類に分類することができます。データ型としては、数値 (number)、文字列 (string)、真偽値 (boolean)、ノード集合 (node-set) などがあります。

位置関係にもとづいてツリーの中にあるいくつかのノードを指示する記述は、「ロケーションパス」(location path) と呼ばれる式になります。ロケーションパスを評価すると、その値として、それによって指示されたノードから構成されるノード集合が得られます。

前の章では、ロケーションパスとして次のようなものを紹介しました。

スラッシュ(`/`)	ソースツリーのルートノード
ドット (`.`)	文脈ノード
要素型名	その要素型を持つすべての要素ノード

`xsl:copy-of` という要素型を持つ要素は、式を評価して、得られた値を結果ツリーに挿入する、という動作をあらわす命令です。評価させたい式は、この要素型の要素が持っている `select` という属性に対して属性値として設定します。たとえば、

```
<xsl:copy-of select="/"/>
```

という命令は、`/` という式を評価することによって得られた、ルートノードだけから構成されるノード集合を結果ツリーに挿入する、という動作をあらわしています。

`xsl:value-of` という要素型を持つ要素は、式を評価して、得られた値を文字列に変換した結果を結果ツリーに挿入する、という動作をあらわす命令です。評価させたい式は、`xsl:copy-of` 要素の場合と同じように、`select` という属性に対して属性値として設定します。たとえば、

```
<xsl:value-of select="/"/>
```

という命令は、`/` という式を評価することによって得られた、ルートノードだけから構成されるノード集合を、文字列に変換して、その結果を結果ツリーに挿入する、という動作をあらわしています。

3.1.2 リテラル

特定の文字列を具体的に記述したいときは、「リテラル」(literal) と呼ばれる式を書きます。

リテラルというのは、文字列をアポストロフィー (apostrophe, ’) または二重引用符 (double quote, ") で囲んだもの、つまり、

’文字列’ または "文字列"

という構文を持つ式のことです。たとえば、

```
'namako'   "I am a literal."
```

などは、リテラルの例です。

リテラルは式の一種です。したがって、それを評価すると、値が得られます。リテラルを評価することによって得られる値は、アポストロフィーまたは二重引用符によって囲まれた内側にある文字列です。

XSLT スタイルシートの例 literal.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:copy-of select="'How are you?'"/></result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果 empty.xml + literal.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>How are you?</result>
```

3.1.3 数値定数

特定の数値を具体的に記述したいときは、「数値定数」(number constant) と呼ばれる式を書きます。

数値定数というのは、基本的には、0 から 9 までの数字 (digit) を並べることによってできる列のことです。たとえば、609122 というのは数値定数の一例です。

数値定数は、式の一種です。数値定数の値は、それを 10 進数とみなしたときに、それがあらわしている数値です。たとえば、609122 という数値定数を評価すると、その値として 609122 という数値が得られます。

小数点の位置を示したいときは、その位置にドット (dot, .) を書きます。たとえば、

```
3.14159    .003724    684013.
```

などは、小数点の位置が明示された数値定数の例です。ドットを含んでいない数値定数は、その右端に小数点があると解釈されますので、684013. は、ドットを取り除いたとしても同じ意味になります。

XSLT スタイルシートの例 number.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:value-of select="2.71828"/></result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果 empty.xml + number.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>2.71828</result>
```

上の XSLT スタイルシートの例 (number.xsl) では、xsl:value-of を使うことによって明示的に数値を文字列に変換していますが、数値は、結果ツリーに挿入されるときに自動的に文字列に変換されますので、xsl:value-of の代わりに xsl:copy-of を使ったとしても、同じ結果が得られます。

なお、XPath の仕様では、マイナスの数値をあらわす数値定数を書くことはできません。それを記述するためには、第 3.3 節で説明することになる「演算子」と呼ばれるものを使う必要があります。

XPath の勧告は、数値そのものと、数値を具体的に記述した式のことを、どちらも「数値」(number) と呼んでいます。しかし、それでは紛らわしいと思われますので、この実習マニュア

ルでは、数値を具体的に記述した式のことを「数値定数」(number constant) と呼ぶことにしました。

3.1.4 関数呼び出し

XSLT プロセッサーは、「関数」(function) と呼ばれるものを何個も持っています。それぞれの関数は、それぞれに異なる何らかの動作をします。関数を動作させることを、関数を「呼び出す」(call) と言います。

関数は、何個かのデータを受け取って、それを処理して、その結果を返す、という動作をします。関数が受け取るデータは「引数」(argument) と呼ばれ、関数が返すデータは「戻り値」(returned value) と呼ばれます。関数の多くは 1 個以上の引数を受け取りますが、引数をまったく受け取らない関数もあります。

それぞれの関数は自分の名前を持っています。関数が持っている名前は、その関数の「関数名」(function name) と呼ばれます。

ここで、関数の一例として、`string-length` という名前を持つものを紹介しましょう。これは、1 個の文字列を引数として受け取って、その文字列の長さ（含まれている文字の個数）を戻り値として返す関数です。たとえば、この関数を呼び出して、引数として `nadeshiko` という文字列を渡したとすると、この関数は、戻り値として 9 という数値を返します。

関数を呼び出したいときは、「関数呼び出し」(function call) と呼ばれる式を書きます。関数呼び出しは、

関数名 (式 , ···)

という構文を持つ式です。関数呼び出しを評価すると、関数名で指定された関数が呼び出されて、丸括弧 (parenthesis, `()`) の中に書かれた式の値が引数としてその関数に渡されます。たとえば、

```
string-length('nadeshiko')
```

という関数呼び出しを評価すると、`string-length` という関数が呼び出されて、`nadeshiko` という文字列がその関数に引数として渡されることになります。

関数呼び出しを評価することによって得られる値は、その関数呼び出しによって呼び出された関数が返した戻り値です。たとえば、

```
string-length('nadeshiko')
```

という関数呼び出しを評価すると、その値として 9 という数値が得られます。

XSLT スタイルシートの例 `funcall.xsl`

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result>
      <xsl:copy-of select="string-length('nadeshiko')"/>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果 `empty.xml` + `funcall.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<result>9</result>
```

3.1.5 データ型の変換

何らかのデータが与えられたとき、そのデータに近い意味を持つ別のデータ型のデータを求めることを、データ型を「変換する」(convert) と言います。

XSLT プロセッサーは、与えられたデータのデータ型が、文脈によって必要とされるデータ型と異なっている場合、そのデータを必要なデータ型へ自動的に変換します。そのような変換は、「暗黙の」(implicit) 変換と呼ばれます。

たとえば、次の XSLT スタイルシートでは、`string-length` を呼び出す関数呼び出しの丸括弧の中にドット (`.`) というロケーションパスが書かれています。ドットの値は文脈ノードから

関数呼び出し	説明
boolean(a)	a を真偽値に変換します。
number(a)	a を数値に変換します。
string(a)	a を文字列に変換します。
format-number(n, f)	フォーマット f にしたがって数値 n を文字列に変換します。

表 3.1: 型変換関数

構成されるノード集合ですが、この文脈では文字列が必要とされていますので、XSLT プロセッサーはノード集合を暗黙のうちに文字列に変換します。

XSLT スタイルシートの例　impconv.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="friends">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="person">
    <xsl:copy-of select="."/>
    <length><xsl:copy-of select="string-length(.)"/></length>
  </xsl:template>
</xsl:stylesheet>
```

変換結果　friends.xml + impconv.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <person>Yokoyama Takashi</person><length>16</length>
  <person>Hamano Yoshio</person><length>13</length>
  <person>Nakayama Akira</person><length>14</length>
</result>
```

なお、データ型は、表 3.1 に示されているような「型変換関数」(type conversion function) と呼ばれる関数を使うことによって明示的に変換することも可能です。たとえば、

```
number('5.03724')
```

という関数呼び出しを評価すると、その値として 5.03724 という数値が得られます。

3.2 変数名

3.2.1 xsl:variable 要素

XSLT では、データに名前を与える記述というものを書くことができます。そのような記述によってデータに与えられた名前は、「変数名」(variable name) と呼ばれます。

変数名をデータに与えることを、変数名を「宣言する」(declare) と言います。

変数名を宣言したいときは、xsl:variable という要素型名を持つ要素を書きます。この要素は、テンプレートの中に書いた場合、命令としてインスタンス化されます。

xsl:variable 要素は、name と select という二つの属性を持っています。name には変数名を設定して、select には式を設定します。そうすると、name に設定された変数名が、select に設定された式の値に与えられます。たとえば、

```
<xsl:variable name="namako" select="67"/>
```

という要素を書くことによって、namako という変数名を 67 という数値に与えることができます。

3.2.2 変数参照

変数名をデータに与えると、その変数名を使って、そのデータをあらわす記述を書くことができるようになります。変数名を使ってデータを記述することを、変数名でデータを「参照する」(refer to) と言います。

変数名でデータを参照したいときは、「変数参照」(variable reference) と呼ばれる式を書きます。変数参照というのは、変数名の左側にドルマーク (dollar mark, $) を書いたもののことです。たとえば、namako という変数名が与えられているデータを参照したいときは、

```
$namako
```

という変数参照を書きます。

変数参照は式の一種です。変数参照を評価すると、その値として、その中の変数名が与えられているデータが得られます。

なお、変数名というのは式ではありませんが、変数名が与えられているデータのことを、その変数名の「値」(value) と呼びます。

XSLT スタイルシートの例 varname.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:variable name="a" select="20338"/>
    <xsl:variable name="b" select="'I am a string.'"/>
    <xsl:variable name="c" select="."/>
    <result>
      <data><xsl:copy-of select="$a"/></data>
      <data><xsl:copy-of select="$b"/></data>
      <data><xsl:copy-of select="$c"/></data>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果 notcat.xml + varname.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <data>20338</data>
  <data>I am a string.</data>
  <data>
    <!--I am a comment.-->
    <words>I am <em>not</em> a cat.</words>
  </data>
</result>
```

3.2.3 空ではない xsl:variable 要素

変数名を宣言する xsl:variable 要素の書き方としては、先ほど説明したもののほかに、もうひとつ別の書き方があります。それは、select 属性に属性値を設定する属性指定を書かないで、その代わりとして、開始タグと終了タグを使って空ではない xsl:variable 要素を作る、という書き方です。

select 属性の属性指定がなくて、空ではない xsl:variable 要素をインスタンス化すると、その内容がテンプレートとしてインスタンス化されます。ただし、その結果は、結果ツリーには挿入されません。xsl:variable 要素の内容をインスタンス化した結果は、結果ツリーとは別の場所に保管されます。そして、name 属性に設定された変数名は、そのインスタンス化の結果に与えられます。たとえば、

```
<xsl:variable name="namako">
  <words>I don't know what I am.</words>
</xsl:variable>
```

という xsl:variable 要素をインスタンス化すると、その結果として、namako という変数名が、

```
<words>I don't know what I am.</words>
```

という要素ノードに与えられることになります。

このように、空ではない xsl:variable 要素は、テンプレートをインスタンス化した結果に変数名を与えたいときに使うことができます。

XSLT スタイルシートの例 notempty.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:variable name="number" select="38026"/>
    <xsl:variable name="node">
      <data><xsl:copy-of select="$number"/></data>
    </xsl:variable>
    <result><xsl:copy-of select="$node"/></result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果　empty.xml + notempty.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result><data>38026</data></result>
```

3.2.4 変数名のスコープ

XSLT スタイルシートの中で、変数名でデータを参照することができる範囲のことを、その変数名の「スコープ」(scope) と呼びます。

xsl:variable 要素を、テンプレートの中に命令として書いた場合、それによって宣言された変数名のスコープは、その命令の直後から、その命令の親の終了タグまでです。

先祖の要素で宣言されている変数名と同じものを子孫の要素で宣言してもかまいません。その場合、子孫で宣言された変数名のスコープの中では、先祖で宣言された変数名が隠されることになります。

XSLT スタイルシートの例　scope.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result>
      <xsl:variable name="a" select="53901"/>
      <data><xsl:copy-of select="$a"/></data>
      <awabi>
        <data><xsl:copy-of select="$a"/></data>
      </awabi>
      <sazae>
        <xsl:variable name="a" select="'I am a string.'"/>
        <data><xsl:copy-of select="$a"/></data>
      </sazae>
      <data><xsl:copy-of select="$a"/></data>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果　empty.xml + scope.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <data>53901</data>
  <awabi>
    <data>53901</data>
  </awabi>
  <sazae>
    <data>I am a string.</data>
  </sazae>
  <data>53901</data>
</result>
```

3.2.5 グローバルな変数名とローカルな変数名

xsl:variable 要素は、テンプレートの中に命令として書くことができるだけではなくて、トップレベル要素として書くこともできます。トップレベル要素として書いた場合、それによって宣

言された変数名のスコープは、XSLT スタイルシートの全体になります。

XSLT スタイルシートの全体をスコープとする変数名は、「グローバルな」(global) 変数名と呼ばれます。それに対して、特定の要素の内部だけをスコープとする変数名は、「ローカルな」(local) 変数名と呼ばれます。

グローバルな変数名と同じものをローカルな変数名として宣言してもかまいません。その場合、ローカルな変数名のスコープの中では、グローバルな変数名が隠されることになります。

XSLT スタイルシートの例 global.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:variable name="a" select="77240"/>
  <xsl:template match="/">
    <result>
      <data><xsl:copy-of select="$a"/></data>
      <sazae>
        <xsl:variable name="a" select="'I am a string.'"/>
        <data><xsl:copy-of select="$a"/></data>
      </sazae>
      <data><xsl:copy-of select="$a"/></data>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果 empty.xml + global.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <data>77240</data>
  <sazae>
    <data>I am a string.</data>
  </sazae>
  <data>77240</data>
</result>
```

3.3 演算子

3.3.1 演算子とは何か

XPath では、「演算子」(operator) と呼ばれるいくつかの名前が定義されています。演算子の例としては、プラス (plus, `+`) やマイナス (minus, `-`) や `div` などがあります。

演算子というのは、何らかの動作に対して与えられた名前です。たとえば、`+` という演算子は、二つの数値を加算するという動作に与えられた名前で、`div` という演算子は、ひとつの数値をもうひとつの数値で除算するという動作に与えられた名前です。加算や除算のような、名前として演算子を与えられている動作は、「演算」(operation) と呼ばれます。

演算が処理の対象とするデータのことを、その演算の「オペランド」(operand) と呼びます。たとえば、`+` を使って 64 と 31 とを加算する場合、その演算のオペランドは、64 と 31 という数値です。

3.3.2 二項演算子と単項演算子

演算は、2 個のオペランドを処理するものと 1 個のオペランドを処理するものに分類することができます。2 個のオペランドを処理する演算は「二項演算」(binary operation) と呼ばれ、1 個のオペランドを処理する演算は「単項演算」(unary oparation) と呼ばれます。そして、二項演算に名前として与えられた演算子は「二項演算子」(binary operator) と呼ばれ、単項演算に名前として与えられた演算子は「単項演算子」(unary oparator) と呼ばれます。

ちなみに、`+` や `div` などの演算子は二項演算子です。`-` という演算子は、二項演算子として使われる場合と単項演算子として使われる場合とがあります。`-` が二項演算子として使われた場合は、ひとつの数値からもうひとつの数値を減算するという演算を指示していて、単項演算子として使われた場合は、ひとつの数値の符号（プラスかマイナスか）を反転させるという演算を指示しています。

3.3.3 演算子式

演算子は、「演算子式」(operator expression) と呼ばれる式の中に書くことができます。

二項演算子は、

式 | 二項演算子 | 式

という構文を持つ演算子式の中に書くことができます。この形の演算子式を評価すると、まず、演算子の左右に書かれた式が評価されて、それらの値がオペランドとして演算に渡されて、演算が実行されます。たとえば、

```
2000+83
```

という演算子式を評価すると、まず、+ の左右の数値定数が評価されて、それらの値をオペランドとして加算が実行されます。

単項演算子は、

単項演算子 | 式

という構文を持つ演算子式の中に書くことができます。この形の演算子式を評価すると、まず、演算子の右側に書かれた式が評価されて、その値がオペランドとして演算に渡されて、演算が実行されます。たとえば、

```
-400
```

という演算子式を評価すると、まず、- の右側の数値定数が評価されて、その値をオペランドとして、その符号を反転するという演算が実行されます。

演算子式を評価することによって得られる値は、その演算子式によって実行された演算の結果です。たとえば、

```
2000+83
```

という演算子式を評価すると、その値として 2083 という数値が得られます。

XSLT スタイルシートの例　opexpre.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="twonum">
    <result>
      <add><xsl:copy-of select="a + b"/></add>
      <div><xsl:copy-of select="a div b"/></div>
      <sub><xsl:copy-of select="a - b"/></sub>
      <rev><xsl:copy-of select="- a"/></rev>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　twonum.xml

```
<?xml version="1.0"?>
<twonum><a>60</a><b>7</b></twonum>
```

変換結果　twonum.xml + opexpre.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <add>67</add>
  <div>8.571428571428571</div>
  <sub>53</sub>
  <rev>-60</rev>
</result>
```

演算が処理の対象とするデータのことを、その演算の「オペランド」と呼ぶわけですが、この言葉は、二項演算子の左右に書かれた式や、単項演算子の右側に書かれた式のことを指示するために使われることもあります。また、二項演算子の左右に書かれたオペランドのうちで、左側にあるものは「左辺」(left operand) と呼ばれ、右側にあるものは「右辺」(right operand) と呼ば

/ //
|
単項の-
* div mod
+ 二項の-
> < >= <=
= !=
and
or

表 3.2: 演算子の優先順位

れます。

演算子とオペランドとのあいだには、1 個以上のホワイトスペースを書くことができます。逆に、 a+b というように、ホワイトスペースを書かないことも可能です。しかし、場合によってはホワイトスペースが絶対に必要になることもあります。たとえば、 div という演算子の前後に要素型名を書く場合は、ホワイトスペースが絶対に必要です。なぜなら、ホワイトスペースを入れないで div の前後に要素型名を書いたとすると、その全体がひとつの要素型名とみなされてしまうからです。

ちなみに、二項演算子のマイナスの前後に要素型名を書く場合も、ホワイトスペースは絶対に必要です。なぜなら、マイナスという文字は要素型名を作るために使うことのできる文字のひとつだからです。ただし、要素型名の先頭の文字としてマイナスを使うことはできませんので、単項演算子のマイナスの右側に要素型名を書く場合、ホワイトスペースは入れなくてもかまいません。

3.3.4 優先順位と結合規則

$\circ$ と $\bullet$ のそれぞれが二項演算子で、a と b と c のそれぞれが式だとしましょう。このとき、

$$a \circ b \bullet c$$

という演算子式は、二通りの解釈が可能です。ひとつは、

$$\boxed{a \circ b} \bullet c$$

という解釈で、もうひとつは、

$$a \circ \boxed{b \bullet c}$$

という解釈です。

このような、何通りかの解釈が可能な演算子式は、その中に含まれている演算子が持っている「優先順位」(precedence) と呼ばれる属性か、または「結合規則」(associativity) と呼ばれる属性にしたがって解釈されます。

優先順位というのは、演算子が左右の式と結合する強さの順位だと考えることができます。優先順位の高い演算子は、低い演算子に比べて、より強く左右の式と結合します。

表 3.2 は、XPath の演算子が持っている優先順位を示したものです（まだ説明していない演算子がたくさん含まれていますが、その点は気にしないでください）。この表は、上のほうに書かれている演算子ほど高い優先順位を持っていて、横に並べて書かれている演算子は同一の優先順位を持っているということを意味しています。

たとえば、 + と div とでは、 div のほうが + よりも高い優先順位を持っていますので、

$$a \texttt{ + } b \texttt{ div } c$$

という式は、

$$a \texttt{ + } \boxed{b \texttt{ div } c}$$

と解釈されます。

優先順位だけで、すべての演算子式の解釈を確定することができるわけではありません。たとえば、

a - *b* + *c*

という式は、- と + の優先順位が同一ですので、優先順位では解釈を確定することができません。

演算子は、優先順位だけではなくて、結合規則という属性も持っています。同一の優先順位を持つ演算子がひとつの式の中に何個も含まれているとき、それぞれの演算子が左右の式と結合する強さの差は、それらの演算子が持っている結合規則によって決定されます。左にあるものほど強くなるという結合規則は「左結合」(left-associativity) と呼ばれ、右にあるものほど強くなるという結合規則は「右結合」(right-associativity) と呼ばれます。ちなみに、XPath で定義されている二項演算子の結合規則は、すべて左結合です。

優先順位では解釈を確定することのできない演算子式は、それに含まれている演算子が持っている結合規則によって解釈されることになります。たとえば、

a - *b* + *c*

という式は、- と + がどちらも左結合ですので、

(*a* - *b*) + *c*

と解釈されることになります。

XSLT スタイルシートの例　precede.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="threenum">
    <result>
      <precedence>
        <xsl:copy-of select="a + b div c"/>
      </precedence>
      <associativity>
        <xsl:copy-of select="a - b + c"/>
      </associativity>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　threenum.xml

```
<?xml version="1.0"?>
<threenum><a>300</a><b>100</b><c>4</c></threenum>
```

変換結果　threenum.xml + precede.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <precedence>325</precedence>
  <associativity>204</associativity>
</result>
```

3.3.5 丸括弧式

演算子式を、優先順位や結合規則とは無関係に、式を書く人間の意図のとおりに解釈してほしいときは、その演算子式の中に「丸括弧式」(parenthesis expression) と呼ばれる式を書きます。

丸括弧式というのは、丸括弧 (parenthesis, ( )) で式を囲んだもの、つまり、

(式)

という形の式のことです。丸括弧式を評価すると、丸括弧の中の式が評価されて、その値が丸括弧式全体の値になります。

演算子式の一部分をひとつの式として解釈してほしいときは、その部分をひとつの丸括弧式にします。たとえば、

(*a* + *b*) div *c*

演算子式	説明
a + b	a と b とを加算します。
a - b	a から b を減算します。
a * b	a と b とを乗算します。
a div b	a を b で除算したときの商を求めます。
a mod b	a を b で除算したときのあまりを求めます。
- a	a の符号を反転させます。

表 3.3: 算術演算子

と解釈されるような式を書きたいならば、

(*a* + *b*) div *c*

というように、ひとつの式として解釈してほしい部分を丸括弧式にすればいいわけです。

XSLT スタイルシートの例 paren.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="threenum">
    <result>
      <precedence>
        <xsl:copy-of select="(a + b) div c"/>
      </precedence>
      <associativity>
        <xsl:copy-of select="a - (b + c)"/>
      </associativity>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果 threenum.xml + paren.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <precedence>100</precedence>
  <associativity>196</associativity>
</result>
```

3.4 数値

3.4.1 算術演算子

オペランドが数値で、結果も数値であるような演算は、「算術演算」(arithmetic operation) と呼ばれます。そして、算術演算に対して名前として与えられている演算子は、「算術演算子」(arithmetic operator) と呼ばれます。表 3.3 は、XPath で定義されている算術演算子を示しています。

XSLT スタイルシートの例 arithop.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="twonum">
    <result>
      <add><xsl:copy-of select="a + b"/></add>
      <sub><xsl:copy-of select="a - b"/></sub>
      <mul><xsl:copy-of select="a * b"/></mul>
      <div><xsl:copy-of select="a div b"/></div>
      <mod><xsl:copy-of select="a mod b"/></mod>
      <rev><xsl:copy-of select="- a"/></rev>
```

```
    </result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果　twonum.xml + arithop.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <add>67</add>
  <sub>53</sub>
  <mul>420</mul>
  <div>8.571428571428571</div>
  <mod>4</mod>
  <rev>-60</rev>
</result>
```

XPath では、マイナスの数値をあらわす数値定数というものは書くことができません。しかし、単項演算子のマイナスと数値定数から構成される演算子式を書くことによって、特定のマイナスの数値を記述することができます。たとえば、

```
-2308
```

という演算子式を書くことによって、マイナスの 2308 という特定の数値を記述することができます。

3.4.2 算術関数

引数が数値で、戻り値も数値であるような関数は、「算術関数」(arithmetic function) と呼ばれます。

XPath では、ceiling、floor、round という三つの算術関数が定義されています。これらの関数はいずれも、引数の小数点以下の部分を 0 にした結果を戻り値として返す、という動作をします。つまり、数値を整数に変換するわけです。ただし、XPath には整数というデータ型はありませんので、データ型が変換されるわけではありません。

ceiling と floor は、「天井」と「床」という関数名からわかるとおり、対照的な動作をする関数です。ceiling は引数を下回らない最小の整数を返し、floor は引数を上回らない最大の整数を返します。つまり、

引数	7.3	−7.3
ceiling	8.0	−7.0
floor	7.0	−8.0

というような動作をするわけです。

round は、引数を四捨五入することによって整数に変換する関数です。たとえば、引数が 7.3 ならば 7 を返し、7.6 ならば 8 を返します。

XSLT スタイルシートの例　arifunc.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="arifunc">
    <result>
      <ceiling><xsl:copy-of select="ceiling(a)"/></ceiling>
      <ceiling><xsl:copy-of select="ceiling(b)"/></ceiling>
      <floor><xsl:copy-of select="floor(a)"/></floor>
      <floor><xsl:copy-of select="floor(b)"/></floor>
      <round><xsl:copy-of select="round(a)"/></round>
      <round><xsl:copy-of select="round(c)"/></round>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　arifunc.xml

```
<?xml version="1.0"?>
```

文字	意味
0	この位置に常に数字を置く。
#	冗長な 0 ではないならばこの位置に数字を置く。
.	この位置に小数点を置く。
,	グループにする桁の個数を指定する。
%	数値を 100 倍して、その右側にパーセントを置く。

表 3.4: フォーマットパターンで使われる主要な特殊文字

```
<arifunc><a>7.3</a><b>-7.3</b><c>7.6</c></arifunc>
```

変換結果 arifunc.xml + arifunc.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <ceiling>8</ceiling><ceiling>-7</ceiling>
  <floor>7</floor><floor>-8</floor>
  <round>7</round><round>8</round>
</result>
```

3.4.3 数値をめぐるデータ型の変換

数値をあらわしてはいるけれどもデータ型が数値ではないデータは、number という型変換関数を使うことによって、そのデータ型を数値に変換することができます。たとえば、

```
number('4047.8')
```

という関数呼び出しで number を呼び出すと、その戻り値として 4047.8 という数値が得られます。

number は、数値をあらわしていないデータを引数として受け取った場合、「非数値」(not a number, NaN) と呼ばれる数値を返します (非数値のデータ型はあくまで数値です)。たとえば、

```
number('umiushi')
```

という関数呼び出しで number を呼び出すと、number は戻り値として非数値を返します。

数値を文字列に変換したいときは、string という型変換関数を使います。たとえば、

```
string(4037.8)
```

という関数呼び出しで string を呼び出すと、string は、その引数をあらわす 10 進数、つまり 4047.8 という文字列を戻り値として返します。

string は、非数値を引数として受け取った場合、NaN という文字列を戻り値として返します。

数値を文字列に変換する関数としては、string のほかに、format-number という関数もあります[1]。

format-number は、2 個の引数を受け取ります。1 個目の引数は変換の対象となる数値で、2 個目の引数は「フォーマットパターン」(format pattern) と呼ばれる文字列です。フォーマットパターンというのは、数値をどのような文字列に変換するのかという書式をあらわす文字列のことです。この関数は、引数として受け取ったフォーマットパターンにしたがって数値を文字列に変換して、その結果を戻り値として返します。

フォーマットパターンは、表 3.4 に示されているような、特殊な意味を持つ文字を使って書きます。フォーマットパターンを書くための規則は、Java の DecimalFormat というクラスで定義されているものと同じです。

XSLT スタイルシートの例 formnum.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="formnum">
    <result><xsl:apply-templates/></result>
  </xsl:template>
```

[1] format-number は、XPath ではなくて XSLT で定義されている関数です。

```
  <xsl:template match="arg">
    <value>
      <xsl:copy-of select="format-number(n, f)"/>
    </value>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　formnum.xml

```
<?xml version="1.0"?>
<formnum>
  <arg><n>3.14159</n>    <f>#.###</f></arg>
  <arg><n>5834217</n>    <f>#.###</f></arg>
  <arg><n>0</n>          <f>#.###</f></arg>
  <arg><n>0</n>          <f>0.0##</f></arg>
  <arg><n>283.74</n>     <f>00000.00000</f></arg>
  <arg><n>6341902538</n> <f>#,##0</f></arg>
  <arg><n>0.0745</n>     <f>#.####%</f></arg>
</formnum>
```

変換結果　formnum.xml + formnum.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <value>3.142</value>
  <value>5834217.</value>
  <value>.</value>
  <value>0.0</value>
  <value>00283.74000</value>
  <value>6,341,902,538</value>
  <value>7.45%</value>
</result>
```

3.5 真偽値

3.5.1 真偽値とは何か

何らかの条件が成り立っているかどうかということをあらわしているデータは、「真偽値」(boolean) と呼ばれます。条件が成り立っているということをあらわす真偽値は「真」(true) と呼ばれ、条件が成り立っていないということをあらわす真偽値は「偽」(false) と呼ばれます。

3.5.2 特定の真偽値を返す関数

XPath では、特定の真偽値をあらわす定数というものは定義されていません。しかし、特定の真偽値は、true または false という関数を呼び出す関数呼び出しを書くことによって記述することができます。

true と false は、引数を何も受け取らないで、特定の真偽値を返す関数です。true は真を返し、false は偽を返します。ですから、特定の真偽値をあらわす定数を書きたいときは、その代用品として true() または false() という関数呼び出しを書けばいいわけです。

3.5.3 真偽値をめぐるデータ型の変換

真偽値は、文脈が数値または文字列を必要とする場合、暗黙のうちに数値または文字列に変換されます。また、型変換関数の number または string を使うことによって、真偽値を明示的に数値または文字列に変換することもできます。真偽値から数値または文字列への変換は、次の表に示されている規則にしたがって実行されます。

	数値	文字列
真	1	true
偽	0	false

逆に、真偽値以外のデータは、文脈が真偽値を必要とする場合、自動的に真偽値に変換されます。また、型変換関数の boolean を使うことによって、データを明示的に真偽値に変換するこ

演算子式	説明
a = b	a と b とは等しい。
a != b	a と b とは等しくない。
a > b	a は b よりも大きい。
a < b	a は b よりも小さい。
a >= b	a は b よりも大きいかまたは等しい。
a <= b	a は b よりも小さいかまたは等しい。

表 3.5: 関係演算子

ともできます。真偽値以外のデータ型から真偽値への変換は、次の表に示されている規則にしたがって実行されます。

	数値	文字列	ノード集合
真に変換されるデータ	0 以外	空文字列以外	空集合以外
偽に変換されるデータ	0	空文字列	空集合

3.5.4 関係演算子

二つのオペランドのあいだに何らかの関係がある、という条件が成り立っているかどうかを調べる演算は、「関係演算」(relational operation) と呼ばれ、その演算をあらわしている演算子は「関係演算子」(relational operator) と呼ばれます。表 3.5 は、XPath で定義されている関係演算子を示しています。

なお、< という文字は、XML 文書の中ではタグの開始を意味していますので、XSLT スタイルシートの中で演算子の < または <= を使う場合は、< という文字の代わりに `<` という実体参照を書かないといけない、という点に注意が必要です。

関係演算を実行する演算子式を評価すると、その値として、関係が成り立っているかどうかをあらわす真偽値が得られます。たとえば、

```
5 > 8
```

という演算子式を評価すると、その値として偽という真偽値が得られます。

>、<、>=、<= のオペランドは、数値ではないならば暗黙のうちに数値に変換されます。しかし、XSLT プロセッサーの中には、それらの演算子のオペランドが数値ではなかった場合に警告のメッセージを出力するものもありますので、数値以外のデータは `number` を使って明示的に数値に変換しておいたほうがいいでしょう。

XSLT スタイルシートの例 `relatop.xsl`

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="relatop">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="operand">
    <xsl:variable name="a" select="number(a)"/>
    <xsl:variable name="b" select="number(b)"/>
    <value>
      <eq><xsl:copy-of select="$a = $b"/></eq>
      <ne><xsl:copy-of select="$a != $b"/></ne>
      <gt><xsl:copy-of select="$a > $b"/></gt>
      <lt><xsl:copy-of select="$a &lt; $b"/></lt>
      <ge><xsl:copy-of select="$a >= $b"/></ge>
      <le><xsl:copy-of select="$a &lt;= $b"/></le>
    </value>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 `relatop.xml`

```
<?xml version="1.0"?>
<relatop>
  <operand><a>5</a><b>8</b></operand>
  <operand><a>5</a><b>5</b></operand>
</relatop>
```

変換結果 relatop.xml + relatop.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <value>
    <eq>false</eq><ne>true</ne><gt>false</gt><lt>true</lt>
    <ge>false</ge><le>true</le>
  </value>
  <value>
    <eq>true</eq><ne>false</ne><gt>false</gt><lt>false</lt>
    <ge>true</ge><le>true</le>
  </value>
</result>
```

3.5.5 論理演算

オペランドが真偽値で、結果も真偽値であるような演算は、「論理演算」(logical operation) と呼ばれます。そして、論理演算に対して名前として与えられている演算子は、「論理演算子」(logical operator) と呼ばれます。XPath では、 and と or という二つの論理演算子が定義されています。

and は、二つのオペランドの論理積を求める演算子です。演算の結果は、オペランドが両方とも真のときだけ真で、どちらかが偽ならば偽になります。この演算子を使うことによって、A という条件と B という条件から、「A かつ B」(A and B) という条件を作ることができます。

or は、二つのオペランドの論理和を求める演算子です。演算の結果は、オペランドのどちらかが真ならば真で、両方とも偽のときだけ偽になります。この演算子を使うことによって、A という条件と B という条件から、「A または B」(A or B) という条件を作ることができます。

XSLT スタイルシートの例 logicop.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="logicop">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="operand">
    <xsl:variable name="a" select="string(a) = 'true'"/>
    <xsl:variable name="b" select="string(b) = 'true'"/>
    <value>
      <and><xsl:copy-of select="$a and $b"/></and>
      <or><xsl:copy-of select="$a or $b"/></or>
    </value>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 logicop.xml

```
<?xml version="1.0"?>
<logicop>
  <operand><a>true</a> <b>true</b></operand>
  <operand><a>true</a> <b>false</b></operand>
  <operand><a>false</a><b>true</b></operand>
  <operand><a>false</a><b>false</b></operand>
</logicop>
```

変換結果 logicop.xml + logicop.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <value><and>true</and> <or>true</or></value>
```

```
  <value><and>false</and><or>true</or></value>
  <value><and>false</and><or>true</or></value>
  <value><and>false</and><or>false</or></value>
</result>
```

3.5.6 論理関数

引数が真偽値で、戻り値も真偽値であるような関数は、「論理関数」(logical function) と呼ばれます。XPath では、not という論理関数が定義されています。この関数は、引数として 1 個の真偽値を受け取って、それを否定した結果を戻り値として返します。つまり、引数が真ならば偽を返し、偽ならば真を返すわけです。この関数を使うことによって、A という条件から「A ではない」(not A) という条件を作ることができます。

XSLT スタイルシートの例 not.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result>
      <true><xsl:copy-of select="not(true())"/></true>
      <false><xsl:copy-of select="not(false())"/></false>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果 empty.xml + not.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result><true>false</true><false>true</false></result>
```

3.6 文字列

3.6.1 文字列の長さ

文字列を構成している文字の個数を、その文字列の「長さ」(length) と言います。

文字列の長さは、string-length という関数を呼び出すことによって求めることができます。この関数は、引数として文字列を受け取って、その長さを戻り値として返します。たとえば、

```
string-length('dolphin')
```

という関数呼び出しを評価すると、7 という数値が値として得られます。

XSLT スタイルシートの例 length.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="length">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="arg">
    <value><xsl:copy-of select="string-length(.)"/></value>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 length.xml

```
<?xml version="1.0"?>
<length>
  <arg>1</arg>
  <arg>four</arg>
  <arg>This string consists of 38 characters.</arg>
  <arg></arg>
</length>
```

変換結果 length.xml + length.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <value>1</value>
  <value>4</value>
  <value>38</value>
  <value>0</value>
</result>
```

3.6.2 文字列の連結

何個かの文字列を並べることによってひとつの文字列を作ることを、それらの文字列を「連結する」(concatenate) と言います。

concat という関数は、2 個以上の文字列を引数として受け取って、それらの文字列を連結した結果を戻り値として返します。たとえば、

```
concat('st', 'ri', 'ng')
```

という関数呼び出しを評価すると、 string という文字列が値として得られます。

XSLT スタイルシートの例 concat.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="concat">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="arg">
    <value><xsl:copy-of select="concat(a, '/', b)"/></value>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 concat.xml

```
<?xml version="1.0"?>
<concat>
  <arg><a>positive</a><b>negative</b></arg>
  <arg><a>real</a><b>virtual</b></arg>
</concat>
```

変換結果 concat.xml + concat.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <value>positive/negative</value>
  <value>real/virtual</value>
</result>
```

3.6.3 ホワイトスペースの正規化

文字列に対して、

(1) 先頭と末尾にあるホワイトスペースをすべて取り除く。

(2) ホワイトスペースではない文字に囲まれたホワイトスペースの列を 1 個の空白に変換する。

という二つの処理をすることを、文字列のホワイトスペースを「正規化する」(normalize) と言います。

normalize-space という関数は、引数として 1 個の文字列を受け取って、その文字列のホワイトスペースを正規化した結果を戻り値として返します。たとえば、

```
normalize-space('   slow   is&#x9;&#xA;beautiful  ')
```

という関数呼び出しを評価すると、

```
'slow is beautiful'
```

という文字列が値として得られます。

XSLT スタイルシートの例 normal.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="normal">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="arg">
    <value><xsl:copy-of select="normalize-space(.)"/></value>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 normal.xml

```
<?xml version="1.0"?>
<normal>
  <arg>left                center               right</arg>
  <arg>
    northwest northeast
    southwest southeast
  </arg>
</normal>
```

変換結果 normal.xml + normal.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <value>left center right</value>
  <value>northwest northeast southwest southeast</value>
</result>
```

3.6.4 部分文字列の探索

文字列の一部分になっている文字列は、「部分文字列」(substring) と呼ばれます。そして、文字列の内部に特定の部分文字列が含まれているかどうかを調べることを、部分文字列を「探索する」(search) と言います。

部分文字列を探索したいときは、contains という関数を使います。この関数は、2 個の文字列を引数として受け取って、1 個目の文字列の中で 2 個目の文字列を探索します。そして、部分文字列が発見された場合は戻り値として真を返して、発見されなかった場合は偽を返します。たとえば、

```
contains('mitochondria', 'chond')
```

という関数呼び出しを評価すると、真という真偽値が値として得られ、

```
contains('mitochondria', 'vwxyz')
```

という関数呼び出しを評価すると、偽という真偽値が値として得られます。

starts-with という関数を使うと、文字列の先頭に特定の部分文字列があるかどうかを調べることができます。この関数は、2 個の文字列を引数として受け取って、1 個目の文字列の先頭に 2 個目の文字列があるかどうかを調べて、あるならば真を返して、なければ偽を返します。たとえば、

```
starts-with('mitochondria', 'mito')
```

という関数呼び出しの値は真になって、

```
starts-with('mitochondria', 'chond')
```

という関数呼び出しの値は偽になります。

XSLT スタイルシートの例 contains.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="contains">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="arg">
```

```
    <value>
      <c><xsl:copy-of select="contains(a, b)"/></c>
      <s><xsl:copy-of select="starts-with(a, b)"/></s>
    </value>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　contains.xml

```
<?xml version="1.0"?>
<contains>
  <arg><a>gymnosperm</a>  <b>gymno</b></arg>
  <arg><a>metamorphose</a><b>morph</b></arg>
  <arg><a>insemination</a><b>ovum</b></arg>
  <arg><a>invertebrate</a><b></b></arg>
  <arg><a></a>            <b></b></arg>
</contains>
```

変換結果　contains.xml + contains.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <value><c>true</c> <s>true</s></value>
  <value><c>true</c> <s>false</s></value>
  <value><c>false</c><s>false</s></value>
  <value><c>true</c> <s>true</s></value>
  <value><c>true</c> <s>true</s></value>
</result>
```

3.6.5　部分文字列の取り出し

substring という関数を使うことによって、文字列から部分文字列を取り出すことができます。この関数は、3 個の引数を受け取ります。1 個目はそこから部分文字列を取り出す文字列で、2 個目は取り出す部分文字列の先頭になる文字の番号で、3 個目は取り出す部分文字列の長さです。

文字列を構成するそれぞれの文字には、先頭から順番に、1 から始まる番号が与えられています。たとえば、mitochondria という文字列の中にある t という文字の番号は 3 ですので、

```
substring('mitochondria', 3, 6)
```

という関数呼び出しを評価すると、tochon という文字列が値として得られます。

なお、substring を使って、文字列の途中から右端までの部分文字列を取り出す場合は、3 個目の引数を省略することができます。たとえば、

```
substring('mitochondria', 5)
```

という関数呼び出しを評価すると、chondria という文字列が値として得られます。

XSLT スタイルシートの例　substr.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="substr">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="arg">
    <value><xsl:copy-of select="substring(s, i, n)"/></value>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　substr.xml

```
<?xml version="1.0"?>
<substr>
  <arg><s>chrysanthemum</s><i>5</i><n>7</n></arg>
  <arg><s>lactobacillus</s><i>1</i><n>6</n></arg>
  <arg><s>monocotyledon</s><i>7</i><n>20</n></arg>
</substr>
```

変換結果 substr.xml + substr.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <value>santhem</value>
  <value>lactob</value>
  <value>tyledon</value>
</result>
```

文字列から部分文字列を取り出す関数としては、substring のほかにも、

substring-before substring-after

という二つの関数があります。

substring-before は、引数として 2 個の文字列を受け取ります。そして、1 個目の文字列の左端から右に向かって 2 個目の文字列を探索して、それが発見されたならば、それよりも左側の部分文字列を取り出して、戻り値として返します。たとえば、

```
substring-before('cinematograph', 'to')
```

という関数呼び出しを評価すると、cinema という文字列が値として得られます。

substring-after は、substring-before と同じように、引数として 2 個の文字列を受け取って、1 個目の文字列の左端から右に向かって 2 個目の文字列を探索します。そして、それが発見された場合は、それよりも右側の部分文字列を取り出して、戻り値として返します。たとえば、

```
substring-after('cinematograph', 'to')
```

という関数呼び出しを評価すると、graph という文字列が値として得られます。

なお、substring-before も substring-after も、1 個目の文字列の中で 2 個目の文字列が発見されなかった場合は、戻り値として空文字列を返します。

XSLT スタイルシートの例 before.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="before">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="arg">
    <value>
      <a><xsl:copy-of select="substring-before(., '/')"/></a>
      <b><xsl:copy-of select="substring-after(., '/')"/></b>
    </value>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 before.xml

```
<?xml version="1.0"?>
<before>
  <arg>positive/negative</arg>
  <arg>real/virtual</arg>
  <arg>astronomical</arg>
</before>
```

変換結果 before.xml + before.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <value><a>positive</a><b>negative</b></value>
  <value><a>real</a><b>virtual</b></value>
  <value><a/><b/></value>
</result>
```

3.6.6 文字の置き換え

translate という関数を使うことによって、文字列の中に含まれている特定の文字を別の文字に置き換えることができます。

translate は、引数として 3 個の文字列を受け取ります。そして、1 個目の文字列に含まれているそれぞれの文字について、もしもそれが 2 個目の文字列に含まれているならば、それを 3 個目の文字列の中の文字に置き換えて、その結果としてできた文字列を戻り値として返します。2 個目の文字列を構成するそれぞれの文字と、3 個目の文字列を構成するそれぞれの文字は、同じ番号を持つもの同士が対応します。たとえば、

```
translate('ooopgokopokgoopokpogo', 'gkp', '-/:')
```

という関数呼び出しを評価すると、g が - に、k が / に、p が : に置き換わりますので、

```
ooo:-o/o:o/-oo:o/:o-o
```

という文字列が値として得られます。

3 番目の文字列の長さが 2 番目の文字列よりも短い場合、2 番目の文字列の中にある、3 番目の文字列の中に対応する文字を持たない文字は、置き換えの対象となる文字列から削除されます。たとえば、

```
translate('ooopgokopokgoopokpogo', 'gkpo', '-/:')
```

という関数呼び出しを評価すると、o という文字は削除されますので、

```
:-/:/-:/:-
```

という文字列が値として得られます。

XSLT スタイルシートの例　trans.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="trans">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="arg">
    <value><xsl:copy-of select="translate(s, f, t)"/></value>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　trans.xml

```
<?xml version="1.0"?>
<trans>
  <arg><s>narratology</s><f>aor</f> <t>246</t></arg>
  <arg><s>mathematics</s><f>mtac</f><t>357</t></arg>
  <arg><s>l o  gi   c</s><f> </f>   <t></t></arg>
  <arg><s>xx-xx/xx_xx</s><f>-/_</f> <t>:::</t></arg>
</trans>
```

変換結果　trans.xml + trans.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <value>n2662t4l4gy</value>
  <value>375he375is</value>
  <value>logic</value>
  <value>xx:xx:xx:xx</value>
</result>
```

3.7 ステップ

3.7.1 軸

第 2.2 節で説明したように、位置関係にもとづいてツリーの中にあるいくつかのノードを指示する式は、「ロケーションパス」(location path) と呼ばれます。

位置関係にもとづいてノードを指示するためには、「軸」(axis) と呼ばれるものを使う必要があります。軸というのは、何らかの方向へノードをたどっていくことによってできるノードの列

のことです。

軸は、ひとつのノードを出発点としてノードをたどることによって構成されます。軸の出発点となるノードという概念には、それをあらわす正式な言葉がないのですが、ここではそれを「起点ノード」(start node) と呼ぶことにしたいと思います。

軸は、どのような方向へノードをたどっていくのかという違いによっていくつかの種類に分類することができます。XPath 1.0 では、軸の種類として 13 個のものを定義していて、それぞれの種類に「軸名」(axis name) と呼ばれる名前を与えています。

それでは、XPath 1.0 で定義されている 13 種類の軸のうちで、主要なものをいくつか紹介しましょう。

軸名	説明
`ancestor`	起点ノードのすべての祖先を、起点ノードに近いものから順番に並べたもの。
`attribute`	起点ノードのすべての属性ノードから構成される列。並べる順番は規定されていない。
`child`	起点ノードのすべての子供を、出現する順番のとおりに並べたもの。
`descendant`	起点ノードのすべての子孫を、出現する順番のとおりに並べたもの。
`following`	起点ノードよりもうしろにあるすべてのノードを、出現する順番のとおりに並べたもの。ただし起点ノードの子孫は含まない。
`parent`	起点ノードの親だけから構成される軸。起点ノードがルートノードの場合は空の軸になる。
`preceding`	起点ノードよりも前にあるすべてのノードを、出現する順番とは逆の順番で並べたもの。ただし起点ノードの祖先は含まない。
`self`	起点ノードだけから構成される軸。

なお、13 種類の軸のうちで、自分の中に起点ノードを含んでいるものは少数派です（ここで紹介したものの中では `self` だけです)。大多数の軸は起点ノードを自分の中に含んでいない、という点に注意してください。

ノードの位置を記述するためには、どの軸を使うのかということを指定する必要があります。軸は、「軸指定子」(axis specifier) と呼ばれるものを書くことによって指定することができます。軸指定子というのは、軸名の右側にコロンコロン (`::`) を書いたもののことです。たとえば、`parent::` という軸指定子を書くことによって、`parent` 軸を指定することができます。

3.7.2 ステップの構文

軸指定子の右側に「ノードテスト」(node test) と呼ばれるものを書いて、さらにその右側に「述語」(predicate) と呼ばれるもの[2]を書いたもの、つまり、

軸指定子	ノードテスト	述語

という構文を持つものは、「ステップ」(step) と呼ばれるロケーションパスになります。ただし、軸指定子と述語は省略することができますので、ノードテストだけを書いたものも、ひとつのステップとして評価することができます。

ノードテストというのは、ノードの名前またはノードの型を指定する記述のことです。ノードテストとして要素型名を書いたとすると、軸に含まれているノードのうちで、その要素型を持つ要素ノードが指定されます。たとえば、`namako` という要素型名を持つ要素ノードは、`namako` というノードテストによって指定されます。

3.7.3 ステップの値

ステップ（ただし述語を省略したもの）を評価すると、その値として、軸指定子によって指定された軸に含まれているノードのうちで、ノードテストによって指定されたものから構成される集合が得られます。たとえば、

```
descendant::namako
```

というステップを評価すると、起点ノードの子孫になっているノードのうちで、`namako` という名前を持つものから構成される集合が値として得られます。

[2]述語については、もう少しあとのところで説明したいと思います。

ステップを単独の式として評価する場合は、文脈ノードが軸の起点ノードになります（ステップを評価するときの起点ノードについては、次の節でさらに一般的な説明をしたいと思います)。

XSLT スタイルシートの例 step.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="step">
    <xsl:copy-of select="descendant::name"/>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 step.xml

```
<?xml version="1.0"?>
<step>
  <name>Edogawa Ranpo</name>
  <a><name>Hisao Juuran</name></a>
  <b><a><name>Oguri Mushitarou</name></a></b>
  <c><b><a><name>Yokomizo Seishi</name></a></b></c>
</step>
```

変換結果 step.xml + step.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <name>Edogawa Ranpo</name>
  <name>Hisao Juuran</name>
  <name>Oguri Mushitarou</name>
  <name>Yokomizo Seishi</name>
</result>
```

3.7.4 ノードの型を指定するノードテスト

先ほど説明したように、ノードの名前またはノードの型を指定する記述は、「ノードテスト」(node test) と呼ばれます。要素型名は、その名前を持つ要素ノードを指定するノードテストで、属性名は、その名前を持つ属性ノードを指定するノードテストです。

さて、それでは、ノードの型を指定したいときは、どのようなノードテストを書けばいいのでしょうか。

ノードの型を指定したいときは、ノードテストとして、ノードの型を識別する名前の右側に一組の丸括弧 (parenthesis, ( )) を付けたものを書きます。たとえば、

`comment()`	注釈ノード
`text()`	テキストノード
`node()`	任意の型のノード

などをノードテストとして書くことができます。

なお、ルートノード、要素ノード、属性ノードについては、ほかの方法で指定することができますので、ノードテストとして記述する方法は定義されていません。

XSLT スタイルシートの例 comment.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result>
      <xsl:copy-of select="descendant::comment()"/>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 comment.xml

```
<?xml version="1.0"?>
<!-- root element -->
<comment>
  <!-- toplevel element -->
  <toplevel>
    <!-- first sentence -->
    <sentence>I am a text.</sentence>
    <!-- second sentence -->
    <sentence>I am not a cat.</sentence>
  </toplevel>
</comment>
```

変換結果 comment.xml + comment.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <!-- root element -->
  <!-- toplevel element -->
  <!-- first sentence -->
  <!-- second sentence -->
</result>
```

3.7.5 任意の名前を持つノードを指定するノードテスト

ノードテストとしてアスタリスク (asterisk, `*`) を書くと、それは、任意の名前を持つノードを指定することになります。

XSLT スタイルシートの例 anyname.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="a">
    <b><xsl:copy-of select="child::*"/></b>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 anyname.xml

```
<?xml version="1.0"?>
<anyname>
  <a><cat/><grep/></a>
  <a><cp/><ls/><mv/><rm/></a>
  <a><ping/><ftp/><telnet/></a>
</anyname>
```

変換結果 anyname.xml + anyname.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <b><cat/><grep/></b>
  <b><cp/><ls/><mv/><rm/></b>
  <b><ping/><ftp/><telnet/></b>
</result>
```

3.7.6 軸指定子の省略形

`attribute` 軸を指定する軸指定子としては、`attribute::` という形のほかに、アットマーク (at sign, `@`) という省略形を使うこともできます。ですから、たとえば、

```
attribute::width
```

というステップは、`@width` と書いても同じ意味になります。

XSLT スタイルシートの例 attrib.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="person">
    <person>
      <name><xsl:value-of select="@name"/></name>
      <born><xsl:value-of select="@born"/></born>
      <dead><xsl:value-of select="@dead"/></dead>
    </person>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 attrib.xml

```
<?xml version="1.0"?>
<attrib>
  <person name="Oda Nobunaga"       born="1534" dead="1582"/>
  <person name="Toyotomi Hideyoshi" born="1536" dead="1598"/>
  <person name="Tokugawa Ieyasu"    born="1542" dead="1616"/>
</attrib>
```

変換結果 attrib.xml + attrib.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <person>
    <name>Oda Nobunaga</name>
    <born>1534</born><dead>1582</dead>
  </person>
  <person>
    <name>Toyotomi Hideyoshi</name>
    <born>1536</born><dead>1598</dead>
  </person>
  <person>
    <name>Tokugawa Ieyasu</name>
    <born>1542</born><dead>1616</dead>
  </person>
</result>
```

child 軸を指定する child:: という軸指定子は、何も書かない、という省略形を持っています。つまり、軸指定子を省略した、ノードテストのみから構成されるステップを書くことができて、そのようなステップを評価すると、その値として、起点ノードの子供のうちで、名前または型によって指定されたノードから構成される集合が得られる、ということです。ですから、たとえば、

```
child::comment()
```

というステップは、 child:: を省略して comment() と書いても同じ意味になります。

第 2.5 節で、要素型名を式として評価すると、文脈ノードの子供のうちで、その要素型を持つすべての要素ノードから構成される集合が得られる、と説明しましたが、この場合の要素型名というのは、 child:: という軸指定子が省略されたステップのことです。

3.7.7 ステップの省略形

第 2.5 節で、ドット (dot, .) というロケーションパスを評価することによって文脈ノードを求めることができる、と説明しましたが、この場合のドットというのは、 self::node() というステップの省略形のことです。

また、起点ノードの親を求める parent::node() というステップは、ドットドット (..) という省略形を持っています。

XSLT スタイルシートの例 parent.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="child">
    <xsl:copy-of select=".."/>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 parent.xml

```
<?xml version="1.0"?>
<parent>
  <a></a>
  <b><child/></b>
  <c></c>
  <d><child/></d>
</parent>
```

変換結果 parent.xml + parent.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <b><child/></b>
  <d><child/></d>
</result>
```

3.8 ロケーションパス

3.8.1 パス演算子

ロケーションパスとは何か、ということについては、前の章で、位置関係にもとづいてツリーの中にあるいくつかのノードを指示する記述のことだと説明しました。しかし、この説明は、ロケーションパスという式の意味について述べているだけで、構文については何も述べていません。それでは、ロケーションパスというのは、いったいどのような構文を持つ式のことなのでしょうか。

ロケーションパスというのは、その構文に着目して説明すれば、1 個以上のステップを含む式、ということになります。

1 個のステップは、それ自体、ひとつのロケーションパスです。さらに、何らかの式とステップとを組み合わせることによってロケーションパスを作ることも可能です。その場合は、

式 / ステップ

というように、スラッシュ(slash, /) という文字の左側に何らかの式を書いて、右側にステップを書きます。スラッシュの左側には、構文としてはどんな式を書いてもかまわないのですが、その式は、評価したときに値としてノード集合が得られるものでないといけません。

式とステップとのあいだに書くスラッシュは、実は、「パス演算子」(path operator) と呼ばれる二項演算子です。この演算子は、「パス演算」(path operation) と呼ばれる演算をあらわしています。

パス演算というのは、次のような処理をする演算のことです。

(1) 左辺を評価する。
(2) 左辺の値として得られたノード集合を構成するそれぞれのノードについて、それを起点ノードとして右辺を評価する。
(3) 右辺を評価した結果として得られたすべてのノード集合の和集合を求める。

ステップは、単独でロケーションパスになっている場合は、文脈ノードを起点ノードとして評価されるのですが、パス演算子の右辺になっている場合は、左辺を評価することによって得られたノード集合を構成するそれぞれのノードを起点ノードとして評価されます。ですから、パス演算子の左辺の値が 7 個のノードから構成されるノード集合だったとすると、右辺のステップは 7

回評価されることになります。

具体的な例で考えてみましょう。たとえば、a/b というロケーションパス（ちなみに、これは、

```
child::a/child::b
```

というロケーションパスの軸指定子を省略したものです）を評価すると、次のような処理が実行されます。

(1) 文脈ノードの子供になっているノードのうちで a という要素型名を持つすべての要素ノードを集めて、ノード集合を作る。
(2) そのノード集合を構成するそれぞれのノードについて、その子供になっているノードのうちで b という要素型名を持つすべての要素ノードを集めて、ノード集合を作る。
(3) それらのノード集合の和集合を求める。

XSLT スタイルシートの例　pathope.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="pathope">
    <result><xsl:copy-of select="a/b"/></result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　pathope.xml

```
<?xml version="1.0"?>
<pathope>
  <a><c>red</c>    <b>lime</b> <c>blue</c>  <c>yellow</c></a>
  <a><b>magenta</b><c>cyan</c> <c>maloon</c><b>green</b> </a>
  <z><b>navy</b>   <c>olive</c><b>purple</b><c>teal</c>  </z>
  <a><b>silver</b> <b>gray</b> <c>white</c> <b>black</b> </a>
</pathope>
```

変換結果　pathope.xml + pathope.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <b>lime</b>  <b>magenta</b><b>green</b>
  <b>silver</b><b>gray</b>   <b>black</b>
</result>
```

3.8.2 絶対ロケーションパスと相対ロケーションパス

パス演算子の左辺は、省略することができます。つまり、

/ ステップ

という形のロケーションパスを書くことができる、ということです。この形のロケーションパスを評価すると、ルートノードを起点ノードとしてステップが評価されます。たとえば、

```
/asari
```

というロケーションパスを評価すると、ルートノードの子供になっている（つまりルート要素になっている）asari という要素から構成されるノード集合が値として得られます。

XSLT スタイルシートの例　absolute.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="goods">
    <goods>
      <xsl:copy-of select="item"/>
      <price>
        <xsl:copy-of select="
```

```
          floor(price * (/absolute/@taxrate + 1))"/>
      </price>
    </goods>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 absolute.xml

```
<?xml version="1.0"?>
<absolute taxrate="0.05">
  <goods><item>book</item>  <price>2230</price></goods>
  <goods><item>bread</item>  <price>155</price></goods>
  <goods><item>coffee</item>  <price>96</price></goods>
</absolute>
```

変換結果 absolute.xml + absolute.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <goods><item>book</item>  <price>2341</price></goods>
  <goods><item>bread</item>  <price>162</price></goods>
  <goods><item>coffee</item> <price>100</price></goods>
</result>
```

左辺が省略されたパス演算子で始まるロケーションパスは、「絶対ロケーションパス」(absolute location path) と呼ばれます。それに対して、何らかのステップで始まるロケーションパスは、「相対ロケーションパス」(relative location path) と呼ばれます。絶対ロケーションパスというのはルートノードを基準にしてノードの位置を指示するロケーションパスで、相対ロケーションパスというのは文脈ノードを基準にしてノードの位置を指示するロケーションパスです。

パス演算子の左辺を省略した場合は、その右辺を省略することも可能です。つまり、スラッシュというひとつの文字だけから構成されるロケーションパスを書くこともできる、ということです。第 2.2 節で紹介したスラッシュというロケーションパスは、パス演算子の左辺と右辺の両方を省略したロケーションパスのことです。ちなみに、そのときにも説明しましたが、スラッシュというロケーションパスを評価すると、ルートノードだけで構成されるノード集合が値として得られます。

3.8.3 ロケーションパスの省略形

軸のひとつに、descendant-or-self という名前のものがあります。これは、起点ノードも含めて、起点ノードのすべての子孫を、出現する順番のとおりに並べることによってできる軸です。

descendant-or-self 軸を使ってすべての子孫を求めるステップを含む、

式/descendant-or-self::node()/ステップ

という形のロケーションパスは、

式//ステップ

という省略形を書いても同じ意味になります。この省略形の中で使われている // は、演算子のひとつで、「省略パス演算子」(abbreviated path operator) と呼ばれます。

XSLT スタイルシートの例 abbpath.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:copy-of select="abbpath//name"/></result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 abbpath.xml

```
<?xml version="1.0"?>
<abbpath>
  <name>Hagio Moto</name>
```

```
  <a><name>Azuma Hideo</name></a>
  <b><a><name>Mizuki Shigeru</name></a></b>
  <c><b><a><name>Ueshiba Riichi</name></a></b></c>
</abbpath>
```

変換結果 abbpath.xml + abbpath.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <name>Hagio Moto</name>
  <name>Azuma Hideo</name>
  <name>Mizuki Shigeru</name>
  <name>Ueshiba Riichi</name>
</result>
```

3.9 述語

3.9.1 述語とは何か

式を角括弧 (square bracket, []) で囲んだもの、つまり、

[式]

という構文を持つ記述は、「述語」(predicate) と呼ばれます。たとえば、[4] や、

```
[@name = 'Miyazaki Keiko']
```

などは述語の例です。

述語が使われるのは、何らかの条件にもとづいてノード集合の部分集合を作る必要があるときです。述語は、与えられたノード集合を構成するそれぞれのノードに対して適用されて、部分集合に採用するか採用しないかを決定します。

述語の中に書かれる式は、ノードを部分集合に採用する条件をあらわしています。式の値が数値だった場合は、ノードが持っている「文脈位置」(context position) と呼ばれる番号とその値とが一致することが採用の条件になります。そして、式の値が数値ではなかった場合は、その値を真偽値に変換した結果が真になることが採用の条件です。

3.9.2 述語を含むステップ

述語は、それ自体は式ではなくて、式の一部分として書くことのできる記述です。式の構文の中で述語を書くことのできる場所は 2 か所あって、そのうちのひとつはステップの末尾です。

第 3.7 節でも説明しましたが、ステップというのは、

軸指定子 ノードテスト 述語

という構文を持つ式のことです。このように、ステップの末尾には述語を書くことができます。

ステップの中に述語を書いた場合、そのステップは次のように評価されます。

(1) 軸指定子とノードテストによって指定されたノード集合を作る。
(2) そのノード集合を構成するそれぞれのノードに対して述語を適用する。
(3) 述語によって採用されたノードだけから構成される部分集合を作って、その結果をステップの値にする。

ちなみに、述語は、ひとつのステップの中に何個でも並べて書くことが可能です。2 個以上の述語が書かれている場合は、それらの述語を、左に書かれているものから順番に使って部分集合を作っていきます。つまり、部分集合を作って、さらにその部分集合を作って、という過程を述語の個数だけ繰り返すわけです。

3.9.3 カレントノードと文脈ノード

ここで、「カレントノード」と「文脈ノード」という言葉の意味について復習しておきたいと思います。「カレントノード」(current node) というのは、命令がインスタンス化されているときに、その処理の対象になっているノードのことで、「文脈ノード」(context node) というのは、式が評価されているときに、その処理の対象になっているノードのことです。

第 2.5 節でも軽く触れましたが、命令の中に書かれた式の文脈ノードは、基本的には、その命令のカレントノードと同じものになるわけですが、同じものにならない場合もあります。

カレントノードと文脈ノードとが同じものにならない場合というのは、述語の中の式が評価される場合です。述語の中に書かれた式は、その述語が適用されたノードを文脈ノードとして評価されるのです。

それでは、例として、

```
<xsl:template match="stock">
  <xsl:copy-of select="goods[@price = 5000]"/>
</xsl:template>
```

というテンプレートルールで考えてみましょう。このテンプレートルールの中に書かれた命令は、stock 要素をカレントノードとしてインスタンス化されます。ですから、その命令の中にある、

```
goods[@price = 5000]
```

という式が評価されるときの文脈ノードも、stock 要素です。しかし、その式の中の述語の中にある、

```
@price = 5000
```

という式が評価されるときの文脈ノードは、stock 要素ではありません。この式は、述語が適用された goods 要素を文脈ノードとして評価されるのです。ですから、このテンプレートルールは、stock 要素の子供になっている goods 要素のうちで、price 属性が 5000 と等しいものから構成されるノード集合を結果ツリーに挿入することになります。

XSLT スタイルシートの例　predica.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="predica">
    <result>
      <xsl:copy-of select="person[@age >= 60]"/>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　predica.xml

```
<?xml version="1.0"?>
<predica>
  <person name="Itami Shinsuke"   age="67"/>
  <person name="Mizobuchi Keigo"  age="58"/>
  <person name="Yokota Masamichi" age="43"/>
  <person name="Nakagawa Takuo"   age="72"/>
</predica>
```

変換結果　predica.xml + predica.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <person name="Itami Shinsuke" age="67"/>
  <person name="Nakagawa Takuo" age="72"/>
</result>
```

3.9.4 文脈位置

ノード集合を構成するそれぞれのノードは、「文脈位置」(context position) と呼ばれる番号を持っています。文脈位置は、常に変化しない識別番号のようなものではなくて、どのようなノード集合を扱っているのかという状況に応じて変化する番号です。

述語の中の式を評価するとき、文脈位置は、その述語よりも左側の部分を処理することによって作られたノード集合を構成するそれぞれのノードに対して、軸の中でノードが並んでいる順番にしたがって、1 番、2 番、3 番、……と与えられます（先頭は 0 番ではなくて 1 番です）。ただし、ノードの順番が規定されていない軸（たとえば attribute 軸）では、ノードは文脈位置を持

ちません。

たとえば、述語の左側に、

```
descendant::theorem
```

という軸指定子とノードテストが書かれているとすると、その述語を評価するときの文脈位置は、起点ノードの子孫になっているそれぞれの theorem 要素に対して、文書の中で出現する順番にしたがって、1 番、2 番、3 番、……と与えられます。

述語の中の式を評価して得られた値が数値だった場合は、述語が適用されたノードの文脈位置と、述語の中の式の値とが一致することが、ノードを部分集合に採用する条件になります。たとえば、 person[4] というステップを評価すると、起点ノードの子供になっている person 要素のうちで 4 番目に出現するものだけから構成されるノード集合が値として得られます。

XSLT スタイルシートの例 position.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="position">
    <result><xsl:copy-of select="element[5]"/></result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 position.xml

```
<?xml version="1.0"?>
<position>
  <element>first</element><element>second</element>
  <element>third</element><element>fourth</element>
  <element>fifth</element><element>sixth</element>
</position>
```

変換結果 position.xml + position.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result><element>fifth</element></result>
```

3.9.5 文脈位置に関連する関数

position という関数は、文脈ノードの文脈位置を戻り値として返します（引数は何も受け取りません）。ですから、

```
[position() = 4]
```

という述語は、 [4] という述語と同じ意味になります。

XSLT スタイルシートの例 between.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="position">
    <result>
      <xsl:copy-of select="
        element[position() >= 2 and position() &lt;= 4]"/>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果 position.xml + between.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <element>second</element><element>third</element>
  <element>fourth</element>
</result>
```

last という関数は、「文脈サイズ」(context size) と呼ばれるゼロまたはプラスの整数を戻り値として返します（引数は何も受け取りません）。文脈サイズというのは、式による処理の対象

となっているノード集合を構成しているノードの個数のことです。

述語の中の式を評価しているときは、その述語の左側に書かれている部分を処理することによって作られたノード集合を構成しているノードの個数が文脈サイズになります。

文脈番号は1番から始まりますので、ノード集合の末尾にあるノードの文脈番号は、文脈サイズと一致します。ですから、[last()]という述語を書くことによって、ノード集合の末尾にあるノードを求めることができます。

XSLT スタイルシートの例 last.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="position">
    <result><xsl:copy-of select="element[last()]"/></result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果 position.xml + last.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result><element>sixth</element></result>
```

current という関数は、カレントノードだけから構成されるノード集合を戻り値として返します（引数は何も受け取りません）[3]。

カレントノードを求めたいときは、たいていの場合、文脈ノードを求めるドット (.) を書けばいいわけですが、述語の中でカレントノードが必要になった場合は、ドットではなくて current を使う必要があります。

XSLT スタイルシートの例 current.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="numbers">
    <largerequal>
      <xsl:copy-of select="n[. >= current()/@border]"/>
    </largerequal>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 current.xml

```
<?xml version="1.0"?>
<current>
  <numbers border="50">
    <n>88</n><n>50</n><n>12</n><n>41</n><n>27</n><n>64</n>
  </numbers>
  <numbers border="250">
    <n>201</n><n>440</n><n>333</n><n>250</n><n>180</n>
  </numbers>
</current>
```

変換結果 current.xml + current.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <largerequal><n>88</n><n>50</n><n>64</n></largerequal>
  <largerequal><n>440</n><n>333</n><n>250</n></largerequal>
</result>
```

[3] current は、XPath ではなくて XSLT で定義されている関数です。

3.9.6 フィルター式

式の構文の中で述語を書くことのできる場所は、2 か所あります。そのうちのひとつがステップの末尾だったわけですが、もうひとつは、「フィルター式」(filter expression) と呼ばれる式の末尾です。

フィルター式というのは、

式 述語

という構文を持つ式のことです。構文としては、フィルター式の述語の左側には、どんな式を書いてもかまいません。ただし、その式は、何らかのノード集合が値として得られるものでないといけません。

フィルター式を評価すると、述語の左側の式が評価されて、その結果として得られたノード集合を構成するそれぞれのノードに対して述語が適用されます。そして、述語によって採用されたノードだけから構成される部分集合が、フィルター式全体の値になります。たとえば、書籍をあらわす要素ノードの集合に対して book という変数名が与えられているとするとき、

```
$book[@author = 'Umberto Eco']
```

というフィルター式を評価することによって、Umberto Eco という author 属性を持っている要素ノードだけから構成される部分集合を求めることができます。

XSLT スタイルシートの例 filter.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:variable name="bases" select="/filter/bases/base"/>
  <xsl:template match="bases"/>
  <xsl:template match="rna">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="b">
    <b>
      <xsl:value-of select="
        $bases[@symbol = current()]/@name"/>
    </b>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 filter.xml

```
<?xml version="1.0"?>
<filter>
  <bases>
    <base name="uracil"   symbol="U"/>
    <base name="cytosine" symbol="C"/>
    <base name="guanine"  symbol="G"/>
    <base name="adenine"  symbol="A"/>
  </bases>
  <rna>
    <b>A</b><b>C</b><b>G</b><b>U</b><b>U</b><b>C</b>
    <b>G</b><b>U</b><b>A</b><b>C</b><b>A</b><b>U</b>
  </rna>
</filter>
```

変換結果 filter.xml + filter.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <b>adenine</b><b>cytosine</b><b>guanine</b>
  <b>uracil</b><b>uracil</b><b>cytosine</b>
  <b>guanine</b><b>uracil</b><b>adenine</b>
  <b>cytosine</b><b>adenine</b><b>uracil</b>
</result>
```

3.10 ノード集合

3.10.1 和集合

この節では、ノード集合を処理する演算や関数を紹介したいと思います。

まず最初は、和集合を求めるという演算です。

2 個の集合があるとするとき、それらのうちのどちらか一方または両方に含まれているものから構成される集合は、もとの 2 個の集合の「和集合」(union) と呼ばれます。XPath は、2 個のノード集合の和集合を求めるという演算を定義していて、その演算は「和集合演算」(union operation) と呼ばれます。和集合演算は、「和集合演算子」(union operator) と呼ばれる二項演算子によってあらわされます。

和集合演算子は、縦棒 (vertical line, |) という文字です。この演算子は、左辺と右辺の和集合を求めるという演算をあらわしています。たとえば、a|b という式を評価すると、その値として、文脈ノードの子供のうちで a または b という要素型名を持つ要素ノードから構成されるノード集合が得られます。

XSLT スタイルシートの例 union.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="union">
    <result><xsl:copy-of select="a|b"/></result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 union.xml

```
<?xml version="1.0"?>
<union>
  <a>alpha</a><b>beta</b>   <c>gamma</c>
  <a>delta</a><b>epsilon</b><c>zeta</c>
  <a>eta</a>  <b>theta</b>  <c>iota</c>
</union>
```

変換結果 union.xml + union.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <a>alpha</a><b>beta</b>
  <a>delta</a><b>epsilon</b>
  <a>eta</a>  <b>theta</b>
</result>
```

3.10.2 ノードの個数

count という関数は、引数として 1 個のノード集合を受け取って、そのノード集合を構成しているノードの個数を戻り値として返します。

XSLT スタイルシートの例 count.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="count">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="a">
    <a><xsl:copy-of select="count(b)"/></a>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 count.xml

```
<?xml version="1.0"?>
<count>
  <a><b/><b/><b/><b/><b/><b/><b/></a>
```

```
  <a><b/><b/><b/><b/><b/><b/><b/><b/><b/><b/><b/><b/><b/></a>
  <a></a>
  <a><b/><b/><b/><b/></a>
  <a><b/><b/><b/><b/><b/><b/><b/><b/><b/></a>
</count>
```

変換結果　count.xml + count.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result><a>7</a><a>12</a><a>0</a><a>4</a><a>9</a></result>
```

3.10.3　ノードの合計

sum という関数は、引数として 1 個のノード集合を受け取って、そのノード集合を構成しているそれぞれのノードを数値に変換して、それらの数値の合計を戻り値として返します。

XSLT スタイルシートの例　sum.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="sum">
    <result><xsl:copy-of select="sum(n)"/></result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　sum.xml

```
<?xml version="1.0"?>
<sum><n>50000</n><n>7000</n><n>400</n><n>90</n><n>2</n></sum>
```

変換結果　sum.xml + sum.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>57492</result>
```

3.10.4　ノードの名前

name という関数は、引数として 1 個のノード集合を受け取って、そのノード集合を構成しているノードのうちで、文書の中で最初に出現するものの名前（要素ノードの場合は要素型名、属性ノードの場合は属性名）を戻り値として返します（戻り値のデータ型は文字列です）。

引数が空集合だった場合、name は、長さが 0 の文字列を戻り値として返します。ちなみに、長さが 0 の文字列は、「空文字列」(null string) と呼ばれます。name は、引数の最初のノードがルートノードやテキストノードや注釈ノードのような、名前を持たないノードだった場合も、戻り値として空文字列を返します。

引数を何も渡さないで name を呼び出すと、name は、文脈ノードの名前を戻り値として返します。つまり、name(.) という関数呼び出しと、name() という関数呼び出しとは同じ意味になります。

XSLT スタイルシートの例　name.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="a">
    <a><xsl:copy-of select="concat('[', name(*), ']')"/></a>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　name.xml

```
<?xml version="1.0"?>
<name>
  <a><sin/></a>
```

```
  <a><cos/><tan/><log/><exp/></a>
  <a></a>
</name>
```

変換結果 name.xml + name.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <a>[sin]</a>
  <a>[cos]</a>
  <a>[]</a>
</result>
```

3.10.5 XML 文書の読み込み

document という関数は、XML 文書が格納されているファイルの URI を引数として受け取って、その XML 文書を読み込んで、それを解析してそのツリーを作って、そしてそのルートノードだけから構成されるノード集合を戻り値として返します[4]。たとえば、XSLT スタイルシートの中に書かれた、

```
document('namako.xml')
```

という関数呼び出しを評価すると、その値として、その XSLT スタイルシートと同じディレクトリにある namako.xml というファイルに格納されている XML 文書のルートノードから構成されるノード集合が得られます。

XSLT スタイルシートの例 document.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:variable name="members"
     select="document(/document/@mfile)/members/member"/>
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="work">
    <work>
      <xsl:copy-of select="date"/>
      <name>
        <xsl:value-of
          select="$members[@id = current()/id]/@name"/>
      </name>
    </work>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 members.xml

```
<?xml version="1.0"?>
<members>
  <member id="001" name="Fujiwara Hitomi"/>
  <member id="002" name="Akagi Ritsuko"/>
  <member id="003" name="Kusanagi Motoko"/>
  <member id="004" name="Kirishima Kayako"/>
</members>
```

XML 文書の例 document.xml

```
<?xml version="1.0"?>
<document mfile="members.xml">
  <work><date>1 Apr</date><id>003</id></work>
  <work><date>2 Apr</date><id>002</id></work>
  <work><date>3 Apr</date><id>004</id></work>
  <work><date>4 Apr</date><id>003</id></work>
  <work><date>5 Apr</date><id>002</id></work>
```

[4] document は、XPath ではなくて XSLT で定義されている関数です。

```
</document>
```

変換結果　document.xml + document.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <work><date>1 Apr</date><name>Kusanagi Motoko</name></work>
  <work><date>2 Apr</date><name>Akagi Ritsuko</name></work>
  <work><date>3 Apr</date><name>Kirishima Kayako</name></work>
  <work><date>4 Apr</date><name>Kusanagi Motoko</name></work>
  <work><date>5 Apr</date><name>Akagi Ritsuko</name></work>
</result>
```

3.11 属性値テンプレート

3.11.1 属性値テンプレートとは何か

属性ノードを持つ要素ノードを結果ツリーに挿入する方法のひとつは、テンプレートの中の要素の中に属性指定を書く、という方法です。たとえば、

```
<wind velocity="4.7m/s"/>
```

という要素をテンプレートの中に書くことによって、属性名が velocity で属性値が 4.7m/s という属性ノードを持つ、 wind という要素ノードを結果ツリーに挿入することができます。

さて、それでは、式の値を属性値とする属性ノードを持つ要素ノードを結果ツリーに挿入したいときは、いったいどうすればいいのでしょうか。基本的には、属性指定の中の引用符で囲まれた文字列は、そのまま属性値になりますので、そこに式を書いたとしても、式がそのまま属性値になるだけです。

そこで登場するのが「属性値テンプレート」(attribute value template) と呼ばれる記述です。この記述を属性指定の中に書くことによって、式の値を属性値にすることができるのです。

なお、「属性値テンプレート」という言葉の中には「テンプレート」という言葉が含まれていますが、「属性値テンプレート」の「テンプレート」と、 xsl:template 要素の内容として書かれるものを意味する「テンプレート」という言葉とは、意味の上での関連はまったくありません。

3.11.2 属性値テンプレートの書き方

構文の観点から言えば、属性値テンプレートというのは、対応の取れている中括弧 (curly brace, {}) を含む文字列のことです。たとえば、

```
namako{isoginchaku}umiushi{hitode}sango
```

という文字列は属性値テンプレートの一例です。

属性値テンプレートの中の中括弧を入れ子にすることはできません。たとえば、

```
namako{isoginchaku{umiushi}hitode}sango
```

という文字列は、正しい属性値テンプレートではありません。

3.11.3 属性値テンプレートのインスタンス化

属性値テンプレートは、何らかの文字列を求めるという動作をあらわしています。属性値テンプレートがあらわしている動作を実行することを、属性値テンプレートを「インスタンス化する」(instantiate) と言います。

属性値テンプレートは、次のような手順でインスタンス化されます。

(1) 中括弧で囲まれた内部にある文字列を式とみなして評価する。
(2) その式の値を文字列に変換する。
(3) 中括弧で囲まれた部分（中括弧も含む）を、その文字列に置き換える。

ですから、たとえば、34 という数値に n という変数名が与えられているとするときに、

```
{$n} * 2 = {$n * 2}
```

という属性値テンプレートをインスタンス化したとすると、その結果として、

```
34 * 2 = 68
```

という文字列が得られます。

3.11.4 属性値テンプレートを書くことができる場所

基本的には、属性値テンプレートを書くことができる場所は、テンプレートの中の属性指定の中の引用符のあいだです。属性値テンプレートをその場所に書いておくと、そのテンプレートがインスタンス化されるときに、その中の属性値テンプレートもインスタンス化されて、その結果が属性値になります。たとえば、

```
<value add="{300 + 40}" mul="{300 * 40}"/>
```

というテンプレートをインスタンス化すると、その結果として、

```
<value add="340" mul="12000"/>
```

という要素が得られます。

テンプレートの中と言っても、命令の中の属性指定の中には、ほとんどの場合、属性値テンプレートを書くことはできません。たとえば、

```
<xsl:copy-of select="{$exp}"/>
```

というように、xsl:copy-of 要素の select 属性に対して属性値テンプレートを使って属性値を与える、ということはできません。

XSLT スタイルシートの例 atttemp.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="twonum">
    <twonum multiply="{@a} * {@b} = {@a * @b}" />
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 atttemp.xml

```
<?xml version="1.0"?>
<atttemp>
  <twonum a="38"  b="22"/>
  <twonum a="400" b="-300"/>
  <twonum a="66"  b="0.001"/>
</atttemp>
```

変換結果 atttemp.xml + atttemp.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <twonum multiply="38 * 22 = 836"/>
  <twonum multiply="400 * -300 = -120000"/>
  <twonum multiply="66 * 0.001 = 0.066"/>
</result>
```

3.12 パターン

3.12.1 式とパターンとの関係

この節ではパターンについて説明していきたいと思うのですが、まず最初に強調しておかないといけないのは、式とパターンとはあくまで別のものだということです。式というのは何らかの動作をあらわしている記述のことですが、それに対して、パターンというのはノードに関する条件をあらわしている記述です。また、式は XPath で定義されているのですが、それに対して、パターンは XSLT で定義されています。

しかし、式とパターンとのあいだには、密接な関係があります。パターンを書くための規則は、式を書くための規則の部分集合です。値としてノード集合が得られる式は、ほとんどすべて、パターンとして使うことも可能です。

ノードとパターンとが一致するかどうかは、そのパターンを式として解釈したときに得られる値によって決定されます。XSLT 1.0 は、それについて次のように述べています。

> ノードがパターンと一致するのは、そのノードが、候補となるいくつかの文脈に対してそのパターンを式として評価した結果のひとつである場合である。候補となる文脈というのは、その文脈ノードが、照合の対象となっているか、またはその先祖のひとつであるような文脈のことである。

かなり理解しにくい文章ですが、これをもう少し噛み砕くと、次のようになります。

> ノード N がパターン P と一致するというのは、N を含む N の先祖のそれぞれを文脈ノードとして、P を式として評価したときに、それによって得られたノード集合のうちのどれかに N が要素として含まれているということである。

つまり、ノード N がパターン P と一致するかどうかを調べたいときは、ノード N から始まってルートノードで終わる先祖の列の中で、それを文脈ノードとして P を評価したときに得られる値の中に N が含まれているものを探せばいい、ということです。含まれているノードが見付かったならば一致するということで、見付からなかったならば一致しないということです。

3.12.2 パターンで使うことのできる軸

値としてノード集合が得られる式は、パターンとして使うこともできるわけですが、これには例外もあります。

例外のひとつは、使うことのできる軸に関するものです。XPath 1.0 は 13 種類の軸を定義していますが、それらのうちでパターンの中で使うことができるのは `child` と `attribute` の 2 種類だけです。

なお、式の場合と同じように、パターンでも、`attribute::` という軸指定子の代わりとして、アットマーク (at sign, `@`) という省略形を使うことができます。また、軸指定子を何も書かなければ、それは `child` 軸を指定したものとみなされる、という点についても式とパターンとは共通です。

XPath 1.0 では起点ノードの子孫から構成される `descendant` という軸が定義されていますが、この軸もパターンでは使うことができません。しかし、省略パス演算子 (`//`) を使うことはパターンでも可能ですので、それを使うことによって、特定の先祖を持つ子孫と一致するパターンを書くことができます。たとえば、

```
a//z
```

というパターンは、`a` というノードを先祖として持っている `z` という子孫のノードと一致します。

XSLT スタイルシートの例　`descend.xsl`

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="x//person">
    <x-person><xsl:value-of select="."/></x-person>
  </xsl:template>
  <xsl:template match="y//person">
    <y-person><xsl:value-of select="."/></y-person>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　`descend.xml`

```
<?xml version="1.0"?>
<descend>
  <x>
    <person>Morishita Hikari</person>
    <a><person>Takebe Noriko</person></a>
    <a><b><person>Kawasaki Motoko</person></b></a>
  </x>
  <y>
```

```
    <person>Hiranuma Kanae</person>
    <a><person>Yoshizawa Takako</person></a>
    <a><b><person>Nomura Saeko</person></b></a>
  </y>
</descend>
```

変換結果　descend.xml + descend.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <x-person>Morishita Hikari</x-person>
  <x-person>Takebe Noriko</x-person>
  <x-person>Kawasaki Motoko</x-person>
  <y-person>Hiranuma Kanae</y-person>
  <y-person>Yoshizawa Takako</y-person>
  <y-person>Nomura Saeko</y-person>
</result>
```

第 4 章　命令

4.1　ノードの挿入

4.1.1　命令についての復習

この章では、さまざまな命令について説明していきたいと思います。

命令とは何か、ということについては、すでに第 2.4 節で説明していますが、この章を始めるに当たって、まず、そのときに説明したことについて復習しておくことにしましょう。

「命令」(instruction) というのは、テンプレートの中に書かれる要素のうちで、自分自身を結果ツリーに挿入するということ以外の動作をあらわしているもののことです。XSLT プロセッサーは、テンプレートをインスタンス化するときに、その中に命令が含まれていた場合、その命令をそのままの形で結果ツリーに挿入するのではなくて、その命令があらわしている動作を実行します。

命令があらわしている動作を実行することを、その命令を「インスタンス化する」(instantiate) と言います。

テンプレートの中にある要素のうちで、命令ではないもの、つまり結果ツリーにそのまま挿入されるものは、「リテラル結果要素」(literal result element) と呼ばれます。

命令は、命令を作るための要素型を使って作られた要素です。XSLT 1.0 は、命令を作るために使うことのできる要素型として、18 個の要素型を定義しています。

第 2 章と第 3 章では、命令を作るための要素型として、

xsl:copy-of	式の値を結果ツリーに挿入する。
xsl:value-of	式の値を文字列に変換した結果を結果ツリーに挿入する。
xsl:apply-templates	テンプレートルールを適用する。
xsl:variable	データに名前を与える。

という四つのものを紹介しました。

4.1.2　要素ノードの挿入

この節では、結果ツリーにノードを挿入する命令を作るための要素型について説明したいと思います。まず、そのひとつ目として、xsl:element という要素型を紹介しましょう。この要素型から作られた命令は、要素ノードを結果ツリーに挿入するという動作をあらわします。

要素ノードを結果ツリーに挿入したいときは、普通、挿入したい要素ノードをそのままの形でテンプレートの中に書きます。つまり、リテラル結果要素を書けばいいわけです。しかし、リテラル結果要素では目的を果たすことができない場合もあります。たとえば、要素型の名前が実行時に決まるような要素ノードは、リテラル結果要素では書くことができません。そのような要素ノードを記述するためには、xsl:element 命令を使う必要があるのです。

xsl:element 命令は、自分の内容をテンプレートとしてインスタンス化して、その結果を内容とする要素ノードを生成して、それを結果ツリーに挿入します。生成される要素ノードの要

素型名としては、xsl:element 命令が持っている name という属性の属性値が使われます。たとえば、

```
<xsl:element name="namako">847032</xsl:element>
```

という xsl:element 命令をインスタンス化すると、

```
<namako>847032</namako>
```

という要素ノードが結果ツリーに挿入されます。

name 属性に属性値を設定する属性指定の中には、属性値テンプレートを書くこともできます。属性値テンプレートを書いた場合は、それをインスタンス化した結果が要素型名になります。たとえば、elename という変数名が umiushi という文字列に与えられているとするときに、

```
<xsl:element name="{$elename}">590116</xsl:element>
```

という xsl:element 命令をインスタンス化すると、

```
<umiushi>590116</umiushi>
```

という要素ノードが結果ツリーに挿入されます。

XSLT スタイルシートの例　inselem.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="material">
    <xsl:element name="{@elename}">
      <xsl:value-of select="@content"/>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　inselem.xml

```
<?xml version="1.0"?>
<inselem>
  <material elename="title"  content="Neuromancer"/>
  <material elename="author" content="William Gibson"/>
  <material elename="year"   content="1984"/>
</inselem>
```

変換結果　inselem.xml + inselem.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <title>Neuromancer</title>
  <author>William Gibson</author>
  <year>1984</year>
</result>
```

4.1.3 属性ノードの挿入

リテラル結果要素を書くことによって要素ノードを結果ツリーに挿入する場合は、そのリテラル結果要素の中に属性指定を書くことによって、属性ノードを要素ノードに持たせることができます。しかし、xsl:element 命令を使って要素ノードを生成する場合、属性指定を書くことによってその要素ノードに属性ノードを持たせることはできません。また、属性ノードの名前を実行時に決めることも、属性指定を書くという方法では不可能です。

xsl:element 命令によって生成される要素ノードに属性ノードを持たせたいときや、属性ノードの名前を実行時に決めたいときは、xsl:attribute という要素型から作られた命令を使う必要があります。

xsl:attribute 命令は、自分の内容をテンプレートとしてインスタンス化して、その結果を属性値とする属性ノードを生成して、自分の親にそれを持たせます（親が xsl:element 命令の場合は、親が生成した要素ノードに属性ノードを持たせます）。生成される属性ノードの属性名

としては、xsl:attribute 命令が持っている name という属性の属性値が使われます（属性値テンプレートを使うこともできます）。たとえば、attname という変数名が id という文字列に与えられているとするときに、

```
<member>
  <xsl:attribute name="{$attname}">810227</xsl:attribute>
  Hagio Moto
</member>
```

というテンプレートをインスタンス化すると、

```
<member id="810227">Hagio Moto</member>
```

という要素ノードが結果ツリーに挿入されます。

XSLT スタイルシートの例　insattr.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="material">
    <xsl:element name="{@elename}">
      <xsl:attribute name="{@attname}">
        <xsl:value-of select="@value"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　insattr.xml

```
<?xml version="1.0"?>
<insattr>
  <material elename="line"   attname="length" value="7cm"/>
  <material elename="circle" attname="radius" value="4cm"/>
  <material elename="fill"   attname="color"  value="green"/>
</insattr>
```

変換結果　insattr.xml + insattr.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <line length="7cm"/>
  <circle radius="4cm"/>
  <fill color="green"/>
</result>
```

4.1.4　属性ノードのコピー

属性ノードを新しく生成するのではなくて、ソースツリーの中にある属性ノードをそのまま結果ツリーにコピーしたいときは、その属性ノードを持たせたい要素ノードの子供として、xsl:copy-of 命令を書きます。たとえば、

```
<member><xsl:copy-of select="@*"/></member>
```

というテンプレートをインスタンス化することによって、カレントノードが持っているすべての属性ノードを member という要素にコピーして、その結果を結果ツリーに挿入することができます。

XSLT スタイルシートの例　copyatt.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
```

```
  <xsl:template match="person">
    <member>
      <xsl:copy-of select="@*"/>
      <xsl:value-of select="."/>
    </member>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 copyatt.xml

```
<?xml version="1.0"?>
<copyatt>
  <person birth="1982/10/03">Kobayashi Kiyomi</person>
  <person weight="72.4kg">Yoshioka Haruka</person>
  <person height="176cm">Mihara Shigeo</person>
</copyatt>
```

変換結果 copyatt.xml + copyatt.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <member birth="1982/10/03">Kobayashi Kiyomi</member>
  <member weight="72.4kg">Yoshioka Haruka</member>
  <member height="176cm">Mihara Shigeo</member>
</result>
```

4.1.5 注釈ノードの挿入

注釈ノードを結果ツリーに挿入したい、というときは、いったいどうすればいいのでしょうか。テンプレートの中に注釈をそのまま書いたとしても、XSLT プロセッサーはその注釈を無視してテンプレートをインスタンス化しますので、その注釈が注釈ノードとして結果ツリーに挿入されることはありません。

注釈ノードを結果ツリーに挿入するためには、xsl:comment という要素型から作られた命令を使う必要があります。

xsl:comment 命令は、自分の内容をテンプレートとしてインスタンス化して、その結果を内容とする注釈ノードを生成して、それを結果ツリーに挿入します。たとえば、

```
<xsl:comment>last modified at 16:11 03 Jan 2005</xsl:comment>
```

という命令をインスタンス化すると、

```
<!--last modified at 16:11 03 Jan 2005-->
```

という注釈ノードが結果ツリーに挿入されます。

XSLT スタイルシートの例 inscomm.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="comment">
    <xsl:comment><xsl:value-of select="."/></xsl:comment>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 inscomm.xml

```
<?xml version="1.0"?>
<inscomm>
  <comment>I am a comment.</comment>
  <comment>see http://www.w3.org/ for details.</comment>
  <comment>last modified at 21:38, 25 Nov 1706</comment>
</inscomm>
```

変換結果 inscomm.xml + inscomm.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <!--I am a comment.-->
  <!--see http://www.w3.org/ for details.-->
  <!--last modified at 21:38, 25 Nov 1706-->
</result>
```

4.2　選択

4.2.1　選択の基礎

この節では、「選択」と呼ばれる動作をあらわす命令について説明したいと思います。

日常的な文脈では、「選択」(selection) という言葉は、ただ単に「選び出すこと」という意味で使われていますが、プログラミングに関連する文脈では、「いくつかの動作の中からどれかひとつを選び出して実行するという動作」という、少し限定された意味で使われます。

選び出される対象となる動作は、「選択肢」(alternative) と呼ばれます。選択肢は、何かが成り立っているかどうかという判断にもとづいて選び出されます。成り立っているかどうかという判断の対象は、「条件」(condition) と呼ばれます。ひとつの条件に対しては、選択肢がひとつだけ対応付けられます。そして、条件が成り立っているならば、その条件に対応付けられている選択肢が実行されます。

ひとつの選択は、条件と選択肢のペアをいくつか並べることによって構成されます。それは、それぞれの条件が成り立っているかどうかを順番に判断していって、成り立っている条件が発見されたならば、それに対応付けられている選択肢を実行する、という動作になります。なお、必要ならば、どの条件も成り立っていなかった場合に実行される選択肢を選択の末尾に置くこともできます。

選択というもののイメージを日本語で記述すると、

もしも c_1 ならば　a_1 を実行して、
そうでなくて c_2 ならば　a_2 を実行して、
そうでなくて c_3 ならば　a_3 を実行して、
⋮　⋮
そうでなくて c_n ならば　a_n を実行して、
そうでなければ　a_{n+1} を実行する。

という感じになります。

4.2.2　xsl:choose 命令

XSLT では、選択は、xsl:choose という要素型から作られた命令を書くことによって記述することができます。

xsl:choose 命令の子供にすることができるのは、xsl:when と xsl:otherwise という二つの要素型のどちらかから作られた要素だけです。

xsl:when 要素は、条件と選択肢のペアを記述するためのものです。xsl:choose 命令は、かならず、この要素を 1 個以上、子供として持っていないといけません。

xsl:otherwise 要素は、どの条件も成り立っていなかった場合に実行される選択肢を記述するためのものです。xsl:choose 命令は、この要素を 0 個または 1 個だけ子供として持つことができます。また、この要素は、xsl:choose 命令の末尾の子供でないといけません。

条件は、xsl:when 要素が持っている、test という属性の属性値として記述します。この属性値は、xsl:choose 命令がインスタンス化されるときに、式として評価されます。得られた値は真偽値に変換されて、その結果が真ならば、条件が成り立っていると判断されて、偽ならば、条件が成り立っていないと判断されます。

選択肢は、xsl:when 要素または xsl:otherwise 要素の内容として記述します。選択肢が選択された場合、それはテンプレートとしてインスタンス化されます。

xsl:choose 命令がインスタンス化されると、その子供になっているそれぞれの xsl:when 要素について、文書の中での順番のとおりに test 属性の属性値が評価されていきます。そして、成

り立っている条件が発見されたならば、その条件に対応する選択肢がインスタンス化されます。たとえば、

```
<xsl:choose>
  <xsl:when test="false()">one</xsl:when>
  <xsl:when test="false()">two</xsl:when>
  <xsl:when test="true()">three</xsl:when>
</xsl:choose>
```

という xsl:choose 命令をインスタンス化すると、three というテキストノードが結果ツリーに挿入されます。

ひとつの選択肢をインスタンス化すると、xsl:choose 命令のインスタンス化は、そこで終了します。インスタンス化された選択肢よりもうしろにある選択肢は、たとえ条件が成り立っていたとしても、決してインスタンス化されません。たとえば、

```
<xsl:choose>
  <xsl:when test="true()">one</xsl:when>
  <xsl:when test="true()">two</xsl:when>
</xsl:choose>
```

という xsl:choose 命令をインスタンス化した場合の結果は one だけです。two はインスタンス化されません。

xsl:otherwise 要素が存在する場合は、どの条件も成り立っていなかったときに、その要素の中の選択肢がインスタンス化されます。たとえば、

```
<xsl:choose>
  <xsl:when test="false()">one</xsl:when>
  <xsl:when test="false()">two</xsl:when>
  <xsl:otherwise>many</xsl:otherwise>
</xsl:choose>
```

という xsl:choose 命令をインスタンス化すると、many という結果が得られます。

XSLT スタイルシートの例　choose.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="threenum">
    <threenum>
      <xsl:copy-of select="@*"/>
      <xsl:attribute name="same">
        <xsl:choose>
          <xsl:when test="@a=@b and @b=@c">a=b=c</xsl:when>
          <xsl:when test="@a=@b">a=b</xsl:when>
          <xsl:when test="@a=@c">a=c</xsl:when>
          <xsl:when test="@b=@c">b=c</xsl:when>
          <xsl:otherwise>nothing</xsl:otherwise>
        </xsl:choose>
      </xsl:attribute>
    </threenum>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　choose.xml

```
<?xml version="1.0"?>
<choose>
  <threenum a="29" b="60" c="60"/>
  <threenum a="72" b="38" c="72"/>
  <threenum a="18" b="18" c="81"/>
  <threenum a="53" b="53" c="53"/>
  <threenum a="41" b="88" c="20"/>
</choose>
```

変換結果　choose.xml + choose.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <threenum a="29" b="60" c="60" same="b=c"/>
  <threenum a="72" b="38" c="72" same="a=c"/>
  <threenum a="18" b="18" c="81" same="a=b"/>
  <threenum a="53" b="53" c="53" same="a=b=c"/>
  <threenum a="41" b="88" c="20" same="nothing"/>
</result>
```

4.2.3　xsl:if 命令

選択を記述する命令を作るための XSLT の要素型としては、xsl:choose のほかに、xsl:if というものもあります。この要素型は、

条件が成り立っているならば動作を実行して、そうでなければ何もしない。

という単純な形の選択を記述するためのものです。

xsl:if 命令を使って選択を記述したいときは、test という属性の属性値として条件を記述して、内容として選択肢を記述します。つまり、xsl:if 命令というのは、ひとつの xsl:when 要素を命令として独立させたような構造になっているわけです。

xsl:if 命令をインスタンス化すると、test 属性の属性値が式として評価されて、その値が真偽値に変換されます。そして、その結果が真ならば命令の内容がテンプレートとしてインスタンス化されて、偽ならば何もインスタンス化されません。たとえば、

```
<xsl:if test="@a=0">zero</xsl:if>
```

という xsl:if 命令は、@a=0 という条件が真ならば zero というテキストノードを結果ツリーに挿入して、偽ならば何もしない、という動作をあらわしています。

XSLT スタイルシートの例　if.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="time">
    <time>
      <xsl:if test="@minutes >= 60">
        <xsl:attribute name="hours">
          <xsl:copy-of select="floor(@minutes div 60)"/>
        </xsl:attribute>
      </xsl:if>
      <xsl:if test="@minutes mod 60 != 0">
        <xsl:attribute name="minutes">
          <xsl:copy-of select="@minutes mod 60"/>
        </xsl:attribute>
      </xsl:if>
      <xsl:if test="@minutes = 0">
        <xsl:copy-of select="@minutes"/>
      </xsl:if>
    </time>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　if.xml

```
<?xml version="1.0"?>
<if>
  <time minutes="222"/>
  <time minutes="300"/>
  <time  minutes="27"/>
  <time   minutes="0"/>
</if>
```

変換結果 if.xml + if.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <time hours="3" minutes="42"/>
  <time hours="5"             />
  <time           minutes="27"/>
  <time            minutes="0"/>
</result>
```

4.3 繰り返し

4.3.1 繰り返しの基礎

同じ動作を何回も実行するという動作は、「繰り返し」(iteration) と呼ばれます。

XSLT では、繰り返しは、xsl:for-each という要素型から作られた命令を書くことによって記述することができます。

また、第 2.5 節で紹介した、xsl:apply-templates という要素型から作られた命令も、テンプレートルールを適用するという動作をいくつかのノードに対して実行する、という繰り返しを記述するためのものだと考えることができます。

4.3.2 xsl:for-each 命令

xsl:for-each という要素型から作られた命令は、ひとつのノード集合を構成するそれぞれのノードをカレントノードにして同じ動作を実行する、という繰り返しをあらわします。

xsl:for-each 命令を使って繰り返しを記述するためには、ノード集合を求めるための式と、実行の対象となるテンプレートを記述する必要があります。

ノード集合を求めるための式は、xsl:for-each 命令が持っている select という属性に、属性値として設定します。この式は、xsl:for-each 命令がインスタンス化されたときに、1 回だけ評価されます。

実行の対象となるテンプレートは、xsl:for-each 命令の子供として書きます。このテンプレートは、select 属性に設定された式の値として得られたノード集合を構成するそれぞれのノードについて、それをカレントノードとしてインスタンス化されます。

XSLT スタイルシートの例 foreach.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="person">
    <person>
      <xsl:for-each select="@*">
        <xsl:element name="{name()}">
          <xsl:value-of select="."/>
        </xsl:element>
      </xsl:for-each>
    </person>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 iterate.xml

```
<?xml version="1.0"?>
<iterate>
  <person name="Yoshimitsu" vocation="pilot" age="33"/>
  <person name="Eri" birth="73/05/12" hobby="gardening"/>
  <person name="Susumu" weight="63kg" height="174cm"/>
</iterate>
```

変換結果 iterate.xml + foreach.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<result>
  <person>
    <name>Yoshimitsu</name>
    <vocation>pilot</vocation>
    <age>33</age>
  </person>
  <person>
    <name>Eri</name>
    <birth>73/05/12</birth>
    <hobby>gardening</hobby>
  </person>
  <person>
    <name>Susumu</name>
    <weight>63kg</weight>
    <height>174cm</height>
  </person>
</result>
```

4.3.3　xsl:apply-templates 命令の select 属性

xsl:apply-templates 命令も、xsl:for-each 命令と同じように、何らかのノード集合を構成するそれぞれのノードに対して同じ動作を実行する、という繰り返しを記述するための命令です。

xsl:apply-templates 命令が繰り返しの対象とするノード集合は、デフォルトではカレントノードのすべての子供から構成される集合ですが、それ以外のノード集合を繰り返しの対象にすることも可能です。

繰り返しの対象となるノード集合を指定したいときは、xsl:for-each 命令の場合と同じように、select という属性に式を設定する属性指定を書きます。そうすると、その式の値として得られたノード集合が、繰り返しの対象になります。たとえば、

```
<xsl:apply-templates select="@*"/>
```

という命令は、カレントノードのすべての属性ノードに対してテンプレートルールを適用するという繰り返しを実行します。

属性ノードに適用されるテンプレートルールを書くためには、属性ノードと一致するパターンを書くことが必要になります。属性ノードと一致するパターンは、

attribute::属性名

と書くか、または、

@属性名

という省略形を書きます。たとえば、@width というパターンは、width という名前の属性ノードと一致します。属性名の代わりにアスタリスク (asterisk, *) を書くと、そのパターンは、任意の名前を持つ属性ノードと一致することになります。

先ほどの foreach.xsl という XSLT スタイルシートでは、person 要素のすべての属性ノードに対する繰り返しを、xsl:for-each 命令を使って記述していましたが、次の XSLT スタイルシートでは、その繰り返しを、xsl:apply-templates 命令を使って記述しています。

XSLT スタイルシートの例　apptemp.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="person">
    <person><xsl:apply-templates select="@*"/></person>
  </xsl:template>
  <xsl:template match="@*">
    <xsl:element name="{name()}">
      <xsl:value-of select="."/>
    </xsl:element>
```

```
  </xsl:template>
</xsl:stylesheet>
```

4.3.4 ソート

データの列を構成するそれぞれのデータを、その一部分の大小関係にもとづいて並べ替える、という処理のことを、「ソート」(sorting) と呼びます。大小関係の判定に使われるデータの一部分は、「ソートキー」(sort key) と呼ばれます。

ソートキーがもっとも小さいデータが先頭になっていて、列のうしろへ行くほどソートキーが大きくなっていく、という順番のことを「昇順」(ascending order) と言います。それとは逆に、先頭がもっとも大きくて、うしろへ行くほど小さくなっていく、という順番は、「降順」(descending order) と呼ばれます。

大小関係は、数値と数値とのあいだだけではなくて、文字列と文字列とのあいだにも存在します。文字列の大小関係は、「辞書式順序」(lexicographical order) と呼ばれる規則によって決定されます。辞書式順序というのは、辞書を作るときに見出し語を並べる順番を決めるための規則のことです。辞書式順序にしたがって並べられた文字列は、前にあるものほど小さくて、うしろにあるものほど大きいとみなされます。

1 回のソートは、1 個以上の任意の個数のソートキーを使って実行することができます。ただし、それらのソートキーのあいだでは、優先される順位が決まっている必要があります。順位が低いソートキーは、順位が高いソートキーが同一だったいくつかのデータをさらに細かく並べ替えるために使われます。

4.3.5 xsl:sort 要素

xsl:for-each 命令または xsl:apply-templates 命令を使って繰り返しを実行した場合、ノード集合を構成するそれぞれのノードは、デフォルトでは、XML 文書の中で出現する順番のとおりに処理されていきます（ただし、属性ノードに関しては、処理の順番は不定です。つまり、ひとつの要素が持っている属性ノードのそれぞれが処理される順番は、文書の中で並んでいる順番のとおりになるとは限りません）。

繰り返しの中で処理されるノードの順番は、それらをソートすることによって変更することができます。ノードをソートしたいときは、 xsl:sort という要素型から作られた要素を使います（これは命令ではありません）。

ひとつの xsl:sort 要素は、ひとつのソートキーについての記述です。ですから、複数のソートキーを使ってノードをソートしたいときは、複数の xsl:sort 要素を書く必要があります。その場合、それらの xsl:sort 要素は、前にあるものほど優先順位が高くなります。

xsl:sort 要素は、内容は常に空です。ソートキーとその扱い方は、次のような属性に対して属性値として設定します。

select	ソートキーを求める式。この式は、ソートの対象となるそれぞれのノードを文脈ノードとして評価される。デフォルトは string(.) 、つまり文脈ノードを文字列に変換したものがソートキーになる。
data-type	ソートキーのデータ型。 text を設定すると文字列とみなされ、 number を設定すると数値とみなされる。デフォルトは文字列。
order	ソートの順序。 ascending を設定すると昇順でソートされ、 descending を設定すると降順でソートされる。デフォルトは昇順。
case-order	大文字と小文字のどちらを前にするか。 upper-first を設定すると大文字が前になり、 lower-first を設定すると小文字が前になる。デフォルトは言語によって異なる。

たとえば、

```
<xsl:sort select="@score"
          data-type="number"
          order="descending"/>
```

という xsl:sort 要素を書くことによって、 score という属性の属性値をソートキーにして、それを数値とみなして、それぞれのノードを降順にソートすることができます。

なお、 xsl:sort 要素の属性のうちで select 以外のものについては、属性値テンプレートを使って属性値を設定することができます。

xsl:for-each 命令によって処理されるノードをソートしたいときは、何個かの xsl:sort 要素を、その命令の子供として書きます。ただし、xsl:sort 要素は、繰り返される動作をあらわすテンプレートよりも前に書かないといけません。

XSLT スタイルシートの例　sortfor.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="sort">
    <result>
      <xsl:for-each select="person">
        <xsl:sort select="@score"
                  data-type="number"
                  order="descending"/>
        <xsl:copy-of select="."/>
      </xsl:for-each>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　sort.xml

```
<?xml version="1.0"?>
<sort>
  <person name="Takaoka Tomoko"      score="87"/>
  <person name="Yoshizawa Natsumi"  score="833"/>
  <person name="Kitaoka Marina"    score="1264"/>
  <person name="Matsushima Hitomi"   score="96"/>
  <person name="Suzuki Rika"       score="1868"/>
  <person name="Furukawa Akina"     score="122"/>
</sort>
```

変換結果　sort.xml + sortfor.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <person name="Suzuki Rika"       score="1868"/>
  <person name="Kitaoka Marina"    score="1264"/>
  <person name="Yoshizawa Natsumi"  score="833"/>
  <person name="Furukawa Akina"     score="122"/>
  <person name="Matsushima Hitomi"   score="96"/>
  <person name="Takaoka Tomoko"      score="87"/>
</result>
```

xsl:apply-templates 命令によってテンプレートを適用するノードをソートしたいときも、何個かの xsl:sort 要素を、その命令の子供として書きます。

xsl:apply-templates 命令は、普通は空要素タグを使って作るのですが、xsl:sort 要素をその子供として書く場合は、開始タグと終了タグのペアを使って作ることになります。

次の XSLT スタイルシートは、上の sortfor.xsl では xsl:for-each 命令で記述されていた繰り返しを、xsl:apply-templates 命令を使って書き換えたものです。

XSLT スタイルシートの例　sortapp.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="sort">
    <result>
      <xsl:apply-templates>
        <xsl:sort select="@score"
                  data-type="number"
                  order="descending"/>
      </xsl:apply-templates>
    </result>
  </xsl:template>
  <xsl:template match="person">
```

```
    <xsl:copy-of select="."/>
  </xsl:template>
</xsl:stylesheet>
```

4.4 番号付け

4.4.1 番号付けの基礎

XSLT では、ノードに対して番号を与えて、その番号をテキストノードとして結果ツリーに挿入する、という動作のことを「番号付け」(numbering) と呼びます。

番号付けは、xsl:number という要素型から作られた命令を書くことによって記述することができます。

4.4.2 xsl:number 命令

xsl:number 命令は、常に空要素です。また、この命令の属性指定は、すべて省略が可能です。したがって、

```
<xsl:number/>
```

というのが、もっとも単純な xsl:number 命令ということになります。この単純な命令は、カレントノードの兄弟になっているノードの中での順番をあらわす自然数をカレントノードに与えて、その自然数を 10 進数に変換した結果をテキストノードとして結果ツリーに挿入します。

XSLT スタイルシートの例　number.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="person">
    <person>
      <no><xsl:number/></no>
      <name><xsl:value-of select="."/></name>
    </person>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　number.xml

```
<?xml version="1.0"?>
<number>
  <person>Hasegawa Akina</person>
  <person>Matsushima Rie</person>
  <person>Kuroki Kanako</person>
  <person>Yamaguchi Haruka</person>
  <person>Tanaka Satomi</person>
  <person>Nishida Sawako</person>
</number>
```

変換結果　number.xml + number.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <person><no>1</no><name>Hasegawa Akina</name></person>
  <person><no>2</no><name>Matsushima Rie</name></person>
  <person><no>3</no><name>Kuroki Kanako</name></person>
  <person><no>4</no><name>Yamaguchi Haruka</name></person>
  <person><no>5</no><name>Tanaka Satomi</name></person>
  <person><no>6</no><name>Nishida Sawako</name></person>
</result>
```

4.4.3 番号を付ける対象を決定するパターン

xsl:number 命令が持っている count という属性にパターンを設定しておくと、そのパターンに一致するノードだけを数えたときの番号をカレントノードに与えます。

count 属性にパターンを設定しなかった場合のデフォルトは、カレントノードと同じ種類のノードと一致するパターンです。ただし、カレントノードが名前を持っている場合は、それと同じ名前を持つノードと一致するパターンになります。

XSLT スタイルシートの例　count.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="x-person | y-person">
    <person>
      <no><xsl:number count="*"/>-<xsl:number/></no>
      <name><xsl:value-of select="."/></name>
    </person>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　count.xml

```
<?xml version="1.0"?>
<count>
  <x-person>Takigawa Yurika</x-person>
  <x-person>Hatanaka Marina</x-person>
  <y-person>Matsuoka Sakiko</y-person>
  <x-person>Kitamura Risako</x-person>
  <y-person>Nagamine Haruka</y-person>
  <x-person>Watanuki Tomomi</x-person>
  <y-person>Sakagami Noriko</y-person>
</count>
```

変換結果　count.xml + count.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <person><no>1-1</no><name>Takigawa Yurika</name></person>
  <person><no>2-2</no><name>Hatanaka Marina</name></person>
  <person><no>3-1</no><name>Matsuoka Sakiko</name></person>
  <person><no>4-3</no><name>Kitamura Risako</name></person>
  <person><no>5-2</no><name>Nagamine Haruka</name></person>
  <person><no>6-4</no><name>Watanuki Tomomi</name></person>
  <person><no>7-3</no><name>Sakagami Noriko</name></person>
</result>
```

4.4.4　自然数から文字列への変換

xsl:number 命令は、デフォルトでは、カレントノードに番号として与えた自然数を、アラビア数字による 10 進数に変換して、その結果として得られた文字列を結果ツリーに挿入します。

自然数をどのような文字列に変換するかということは、format という属性に値を設定することによって制御することができます。format 属性に設定される文字列は、「フォーマット文字列」(format string) と呼ばれます。

フォーマット文字列は、「トークン」(token) と呼ばれる文字列を並べたものです。トークンは二つの種類に分類することができます。ひとつは「フォーマットトークン」(format token) と呼ばれ、もうひとつは「句読点トークン」(separator token) と呼ばれます。

フォーマットトークンというのは、自然数をどのような文字列に変換するのかということをあらわす文字列で、表 4.1 のようなものがあります。

また、何個かのゼロ (0) の右側に 1 個の 1 を書いたものをフォーマットトークンとして使うこともできます。その場合、自然数は、左側の余った部分にゼロが充填された、フォーマットトークンと同じかまたはそれよりも長い 10 進数に変換されます。たとえば、 001 というフォーマッ

フォーマットトークン	変換結果
1	1、2、3、4、5、6、7、8、9、10、11、12、···
A	A、B、C、D、···、X、Y、Z、AA、AB、AC、AD、···
a	a、b、c、d、···、x、y、z、aa、ab、ac、ad、···
i	i、ii、iii、iv、v、vi、vii、viii、ix、x、···
I	I、II、III、IV、V、VI、VII、VIII、IX、X、···

表 4.1: フォーマットトークン

トトークンを使うことによって、自然数を、

001、002、003、···、009、010、···、999、1000、···

という文字列に変換することができます。

句読点トークンというのは、英数字以外の文字から構成される文字列のことです。句読点トークンは、結果ツリーに挿入される文字列の中にそのまま出現することになります。

フォーマット文字列は、フォーマットトークンと句読点トークンとを交互に並べることによってできる文字列です。たとえば、((001)) というのはフォーマット文字列の一例です。このフォーマット文字列を使った、

```
<xsl:number format="((001))"/>
```

という xsl:number 命令は、カレントノードの番号が 37 だとすると、((037)) という文字列を結果ツリーに挿入します。

XSLT スタイルシートの例 format.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="a">
    <a>
      <xsl:number format="[i]"/><xsl:value-of select="."/>
    </a>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 format.xml

```
<?xml version="1.0"?>
<format>
  <a>love</a><a>rain</a><a>node</a><a>tree</a><a>cube</a>
  <a>book</a><a>root</a><a>food</a><a>pool</a><a>cool</a>
  <a>pure</a><a>link</a><a>warm</a><a>line</a><a>blue</a>
</format>
```

変換結果 format.xml + format.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <a>[i]love</a><a>[ii]rain</a><a>[iii]node</a>
  <a>[iv]tree</a><a>[v]cube</a><a>[vi]book</a>
  <a>[vii]root</a><a>[viii]food</a><a>[ix]pool</a>
  <a>[x]cool</a><a>[xi]pure</a><a>[xii]link</a>
  <a>[xiii]warm</a><a>[xiv]line</a><a>[xv]blue</a>
</result>
```

4.4.5 level 属性

xsl:number 命令は、level という属性を持っています。この属性に値を設定することによって、カレントノードに対してどのように番号を与えるのかということを制御することができま

す。この属性に設定することができるのは、

```
single  multiple  any
```

という三つの文字列のうちのいずれかです。

single は、カレントノードの兄弟になっているノードの中での順番をあらわす自然数をカレントノードに与えるという意味です。

level 属性のデフォルトは single です。つまり、level 属性に値を設定する属性指定を省略すると、single が設定されているとみなされます。

4.4.6 階層ごとの番号付け

xsl:number 命令の level 属性に multiple を設定することによって、複数の階層の中での位置をあらわす番号の列をカレントノードに与えることができます。つまり、書籍を構成するそれぞれの項に対して、章の番号、節の番号、項の番号という 3 個の番号から構成される列を与える、というようなことができるわけです。

階層ごとの番号から構成される列をカレントノードに与えるためには、level 属性に multiple を設定するだけではなくて、さらに、count 属性に対して、それぞれの階層を構成する要素ノードと一致するパターンを設定する必要があります。つまり、それぞれの階層にある要素の要素型名を和集合演算子で結合したものを設定するわけです。たとえば、書籍の章が chapter 要素、節が section 要素、項が subsection 要素で書かれているとするとすれば、

```
count="chapter | section | subsection"
```

という属性指定を書くことによって、それぞれの項に対して、章の番号と節の番号と項の番号から構成される列を与えることができます。

カレントノードに対して、1 個の番号ではなくて、2 個以上の番号の列が与えられている場合、その番号の列を文字列に変換するためには、番号の個数と同じ個数のフォーマットトークンを含むフォーマット文字列を format 属性に設定する必要があります。たとえば、3 個の番号から構成される列を文字列に変換したいときは、1.1.1 というようなフォーマット文字列を設定します。そうすることによって、4、7、3 という番号の列は、4.7.3 という文字列に変換されます。

XSLT スタイルシートの例 multi.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="a">
    <a>
      <h><xsl:number/></h>
      <xsl:apply-templates/>
    </a>
  </xsl:template>
  <xsl:template match="b">
    <b>
      <h>
        <xsl:number
            level="multiple" count="a | b" format="1.1"/>
      </h>
      <xsl:apply-templates/>
    </b>
  </xsl:template>
  <xsl:template match="c">
    <c>
      <xsl:number
          level="multiple" count="a | b | c" format="1.1.1"/>
    </c>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 multi.xml

```
<?xml version="1.0"?>
<multi>
  <a>
    <b><c/><c/><c/></b><b><c/><c/><c/></b><b><c/><c/><c/></b>
  </a>
  <a>
    <b><c/><c/><c/></b><b><c/><c/><c/></b><b><c/><c/><c/></b>
  </a>
  <a>
    <b><c/><c/><c/></b><b><c/><c/><c/></b><b><c/><c/><c/></b>
  </a>
</multi>
```

変換結果　multi.xml + multi.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <a><h>1</h>
    <b><h>1.1</h><c>1.1.1</c><c>1.1.2</c><c>1.1.3</c></b>
    <b><h>1.2</h><c>1.2.1</c><c>1.2.2</c><c>1.2.3</c></b>
    <b><h>1.3</h><c>1.3.1</c><c>1.3.2</c><c>1.3.3</c></b>
  </a>
  <a><h>2</h>
    <b><h>2.1</h><c>2.1.1</c><c>2.1.2</c><c>2.1.3</c></b>
    <b><h>2.2</h><c>2.2.1</c><c>2.2.2</c><c>2.2.3</c></b>
    <b><h>2.3</h><c>2.3.1</c><c>2.3.2</c><c>2.3.3</c></b>
  </a>
  <a><h>3</h>
    <b><h>3.1</h><c>3.1.1</c><c>3.1.2</c><c>3.1.3</c></b>
    <b><h>3.2</h><c>3.2.1</c><c>3.2.2</c><c>3.2.3</c></b>
    <b><h>3.3</h><c>3.3.1</c><c>3.3.2</c><c>3.3.3</c></b>
  </a>
</result>
```

4.4.7　階層構造とは無関係な番号付け

xsl:number 命令の level 属性に any を設定することによって、階層構造とはまったく無関係に、カレントノードに番号を与えることができます。つまり、書籍の中に書かれている図や表や注釈などのそれぞれに対して一連の番号を与える、というようなことができるわけです。

階層構造とは無関係な番号付けをする場合でも、しばしば、大きな要素が変わったときに番号をふたたび 1 から始めたい、ということがあります。たとえば、項や節とは無関係に注釈に番号を付けたいけれども、章が変わったところで 1 に戻したい、というような場合です。

any で付ける番号を、何らかの要素が始まる場所で 1 に戻したい、というときは、その要素と一致するパターンを from という属性に設定します。たとえば、

```
<xsl:number level="any" from="chapter"/>
```

という xsl:number 命令によって与えられる番号は、 chapter という要素が始まる場所で、1 に戻って再出発します。

XSLT スタイルシートの例　any.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="a">
    <a><xsl:apply-templates/></a>
  </xsl:template>
  <xsl:template match="b">
    <b><xsl:apply-templates/></b>
  </xsl:template>
  <xsl:template match="x">
    <x><xsl:number/></x>
  </xsl:template>
```

```
  <xsl:template match="y">
    <y><xsl:number level="any" from="a"/></y>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　any.xml

```
<?xml version="1.0"?>
<any>
  <a>
    <b><x/><y/></b><b><x/><y/></b><b><x/><y/></b>
    <x/><y/><x/><y/><x/><y/>
  </a>
  <x/><y/><x/><y/><x/><y/>
  <a>
    <b><x/><y/></b><b><x/><y/></b><b><x/><y/></b>
    <x/><y/><x/><y/><x/><y/>
  </a>
</any>
```

変換結果　any.xml + any.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <a>
    <b><x>1</x><y>1</y></b>
    <b><x>1</x><y>2</y></b>
    <b><x>1</x><y>3</y></b>
    <x>1</x><y>4</y><x>2</x><y>5</y><x>3</x><y>6</y>
  </a>
  <x>1</x><y>7</y><x>2</x><y>8</y><x>3</x><y>9</y>
  <a>
    <b><x>1</x><y>1</y></b>
    <b><x>1</x><y>2</y></b>
    <b><x>1</x><y>3</y></b>
    <x>1</x><y>4</y><x>2</x><y>5</y><x>3</x><y>6</y>
  </a>
</result>
```

第 5 章　名前付きテンプレート

5.1　名前付きテンプレートの基礎

5.1.1　名前付きテンプレートとは何か

XSLT は、テンプレートに名前を付けることができる、という機能を持っています。テンプレートに名前を付けるための要素、または名前が与えられているテンプレートは、「名前付きテンプレート」(named template) と呼ばれます。

名前付きテンプレートを書いておくと、その名前を指定することによって、それをインスタンス化することができるようになります。名前を指定することによってテンプレートをインスタンス化することを、テンプレートを「呼び出す」(call) と言います。

5.1.2　名前付きテンプレートの書き方

テンプレートに名前を付けたいときは、xsl:template という要素型の要素を、XSLT スタイルシートのトップレベル要素として書きます。つまり、名前付きテンプレートは、テンプレートルールと同じ要素型を使って書くわけです。

テンプレートルールを書く場合は、xsl:template 要素が持っている match という属性に、テンプレートを適用するノードと一致するパターンを設定しないといけなかったわけですが、名前付きテンプレートを書く場合は、match にパターンを設定する必要はありません。その代わりに、name という属性に、テンプレートに与える名前を設定する必要があります。

xsl:template 要素の name 属性に名前を設定して、その要素の子供としてテンプレートを書

くことによって、そのテンプレートにその名前を与えることができます。たとえば、

```
<xsl:template name="hello">
  <message>Hello, world.</message>
</xsl:template>
```

という xsl:template 要素を書くことによって、

```
<message>Hello, world.</message>
```

というテンプレートに対して hello という名前を与えることができます。

match にパターンを設定して、さらに name に名前を設定した xsl:template 要素を書くことも可能です。そのような xsl:template 要素は、テンプレートルールとしてノードに適用することもできますし、名前を使って呼び出すこともできます。

なお、名前付きテンプレートの中から結果ツリーにノードを挿入することを、ノードを「返す」(return) と言うこともあります。

5.1.3 名前付きテンプレートの呼び出し

名前付きテンプレートを呼び出したいときは、xsl:call-template という要素型から作られた命令を書きます。

xsl:call-template 命令は、name という属性を持っています。この属性にテンプレートの名前を設定することによって、そのテンプレートを呼び出すことができます。たとえば、

```
<xsl:call-template name="hello"/>
```

という xsl:call-template 命令をインスタンス化すると、hello という名前を持つテンプレートが呼び出されます。

XSLT スタイルシートの例　call.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="hello">
    <message>Hello, world.</message>
  </xsl:template>
  <xsl:template match="/">
    <result><xsl:call-template name="hello"/></result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果　empty.xml + call.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result><message>Hello, world.</message></result>
```

5.1.4 名前付きテンプレートのカレントノード

xsl:call-template 命令によって呼び出された名前付きテンプレートがインスタンス化されるときのカレントノードは、それを呼び出した xsl:call-template 命令のカレントノードと同じです。

XSLT スタイルシートの例　callcur.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="current-member">
    <member><xsl:value-of select="."/></member>
  </xsl:template>
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="person">
    <xsl:call-template name="current-member"/>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　callcur.xml

```
<?xml version="1.0"?>
<callcur>
  <person>Yoshinaga Miyuki</person>
  <person>Hasegawa Tetsuko</person>
  <person>Natsume Hitomi</person>
</callcur>
```

変換結果　callcur.xml + callcur.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <member>Yoshinaga Miyuki</member>
  <member>Hasegawa Tetsuko</member>
  <member>Natsume Hitomi</member>
</result>
```

5.2　パラメーター

5.2.1　引数とパラメーター

名前付きテンプレートを呼び出すとき、その名前付きテンプレートにデータを渡す、ということができます。名前付きテンプレートに渡されるデータは、「引数」(argument) と呼ばれます。

名前付きテンプレートに引数を渡すことができるようにするためには、その名前付きテンプレートを書くときに、自分が受け取った引数に与える変数名を宣言しておく必要があります。引数に与える変数名として宣言された名前は、「パラメーター」(parameter) と呼ばれます。

5.2.2　パラメーターの宣言

パラメーターは変数名の一種ですが、それを宣言するのは xsl:variable 要素ではありません。パラメーターを宣言したいときは、xsl:param という要素型から作られる要素を書く必要があります。

xsl:param 要素は、xsl:variable 要素と同じように、name と select という属性を持っています。name 属性に名前を設定すると、その名前がパラメーターとして宣言されます。なお、パラメーターの値は、名前付きテンプレートが呼び出された時点で与えられますので、xsl:param 要素の select 属性に式を設定する属性指定は、書かなくてもかまいません。たとえば、

```
<xsl:param name="namako"/>
```

という xsl:param 要素を書くことによって、namako というパラメーターを宣言することができます。

引数を受け取る名前付きテンプレートを作りたいとき、言い換えれば、パラメーターを持つ名前付きテンプレートを作りたいときは、xsl:template 要素の子供として xsl:param 要素を書きます。ただし、xsl:param 要素は、xsl:template 要素のほかの子供たちよりも前に書く必要があります。たとえば、

```
<xsl:template name="square">
  <xsl:param name="a"/>
  <xsl:copy-of select="$a * $a"/>
</xsl:template>
```

という名前付きテンプレートは、a というパラメーターに引数を受け取って、その引数の 2 乗を結果ツリーに挿入します。

5.2.3　パラメーターに値を与える要素

名前付きテンプレートを呼び出して、それが持っているパラメーターに値を与えたいときは、xsl:with-param という要素型から作られた要素を、xsl:call-template 命令の子供として書きます。

xsl:with-param 要素は、name と select という属性を持っています。name 属性には名前付きテンプレートが持っているパラメーターを設定して、select 属性には式を設定します。そうすることによって、name 属性に設定されたパラメーターに、select 属性に設定された式の値を与えることができます。たとえば、

```
    <xsl:call-template name="square">
      <xsl:with-param name="a" select="30"/>
    </xsl:call-template>
```

という xsl:call-template 命令を書くことによって、square という名前を持つテンプレートを呼び出したとすると、そのテンプレートが持っている a というパラメーターに 30 という数値が与えられます。

XSLT スタイルシートの例　param.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="square">
    <xsl:param name="a"/>
    <xsl:copy-of select="$a * $a"/>
  </xsl:template>
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="n">
    <s>
      <xsl:call-template name="square">
        <xsl:with-param name="a" select="."/>
      </xsl:call-template>
    </s>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　param.xml

```
<?xml version="1.0"?>
<param>
  <n>7</n><n>30</n><n>100</n><n>0.5</n><n>1</n><n>0</n>
</param>
```

変換結果　param.xml + param.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <s>49</s><s>900</s><s>10000</s><s>0.25</s><s>1</s><s>0</s>
</result>
```

xsl:with-param 要素は、select 属性に設定された式の値を引数にするわけですが、その属性に式を設定する属性指定は、省略することもできます。select の属性指定が省略された場合は、xsl:with-param 要素の子供が引数になります。

5.2.4 パラメーターのデフォルト値

名前付きテンプレートを呼び出すときに xsl:with-param 要素によってパラメーターに値を与えなかったとしても、エラーにはならないで、パラメーターには何らかの値が与えられます。xsl:with-param 要素によって値が与えられなかった場合に、その代わりとしてパラメーターに与えられる値は、そのパラメーターの「デフォルト値」(default value) と呼ばれます。

パラメーターのデフォルト値は、次のような規則にしたがって決定されます。

- xsl:param 要素の select 属性に属性値として式が設定されている場合は、その式の値がパラメーターのデフォルト値になります。
- select 属性に属性値が設定されていなくて、xsl:param 要素に子供がある場合は、その子供がパラメーターのデフォルト値になります。
- select 属性に属性値が設定されていなくて、xsl:param 要素に子供がない場合は、空文字列がパラメーターのデフォルト値になります。

XSLT スタイルシートの例　default.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
```

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="add-mark">
    <xsl:param name="item"/>
    <xsl:param name="marker" select="'*'"/>
    <xsl:copy-of select="concat($marker, ' ', $item)"/>
  </xsl:template>
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="item">
    <item>
      <xsl:choose>
        <xsl:when test="@marker">
          <xsl:call-template name="add-mark">
            <xsl:with-param name="item" select="."/>
            <xsl:with-param name="marker" select="@marker"/>
          </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
          <xsl:call-template name="add-mark">
            <xsl:with-param name="item" select="."/>
          </xsl:call-template>
        </xsl:otherwise>
      </xsl:choose>
    </item>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　default.xml

```
<?xml version="1.0"?>
<default>
  <item>hotategai</item>
  <item>hamaguri</item>
  <item marker="?">namako</item>
  <item>sazae</item>
</default>
```

変換結果　default.xml + default.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <item>* hotategai</item>
  <item>* hamaguri</item>
  <item>? namako</item>
  <item>* sazae</item>
</result>
```

5.3　再帰

5.3.1　再帰の基礎

世の中にあるさまざまなものごとの中には、再帰的な構造を持っているものがいくつもあります。「再帰的な」(recursive) 構造というのは、その一部分が全体の構造と同じになっているような構造のことです。たとえば、2 枚の鏡を向かい合わせにしたときにそこに映し出される映像は、再帰的な構造を持っています。

「再帰的な」という単語は形容動詞（英単語の recursive は形容詞）ですが、「再帰する」(recurse) という動詞や、「再帰」(recursion) という名詞もしばしば使われます。「再帰する」は、全体と同じ構造を一部分として使う、という動作を意味する動詞で、「再帰」は、再帰すること、または構造が再帰的であることを意味する名詞です。

5.3.2　再帰呼び出し

名前付きテンプレートを呼び出す xsl:call-template 命令は、名前付きテンプレートの中に書くことも可能です。そうすることによって、名前付きテンプレートが別の名前付きテンプレー

トを呼び出す、ということができます。

さらに、名前付きテンプレートは、自分自身を呼び出すことも可能です。自分の名前を name 属性に設定した xsl:call-template 命令を自分の中に書いておくと、その命令は、自分自身を呼び出すことになります。

自分自身を呼び出すというのは、一部分の処理をするために全体と同じ動作をするということですから、それは再帰の一種です。ですから、自分自身を呼び出すことは、「再帰呼び出し」(recursive call) と呼ばれます。再帰呼び出しを使うことによって、再帰的な構造を持っているものを処理することができます。

再帰的な構造を持っているものを処理する例として、n が 0 またはプラスの整数だとするとき、長さが n のアスタリスクの列を作る、という処理について考えてみましょう。

長さが n のアスタリスクの列は、1 個のアスタリスクと、長さが $n-1$ のアスタリスクの列とを連結したものだと考えることができます。長さが $n-1$ のアスタリスクの列という部分は、全体と同じ構造になっていますので、これは再帰的な構造です。

長さが n のアスタリスクの列は、再帰呼び出しを使うことによって作ることができます。つまり、再帰呼び出しを使って長さが $n-1$ のアスタリスクの列を作って、それと 1 個のアスタリスクとを連結すれば、長さが n のアスタリスクの列ができるというわけです。

5.3.3 基底

再帰的な構造は、基本的には、一部分として全体があるという構造が無限に続くことになるわけですが、場合によってはどこかでそれがストップすることもあります。再帰的な構造が無限に続いていない場合、そのもっとも内側にあるもののことを、その再帰の「基底」(basis) と呼びます。

XSLT で再帰呼び出しを使う場合も、基底に関する記述がないと、その再帰は無限に続くことになってしまいます。ですから、再帰呼び出しを使う場合は、かならず、基底の場合とそうでない場合との選択を書く必要があります。

再帰を使って長さが n のアスタリスクの列を作るという処理でも、基底の場合とそうでない場合との選択が必要です。この処理では、長さが 0 のアスタリスクの列、つまり空文字列が基底ですので、n が 0 の場合は再帰呼び出しをしないで空文字列を返して、そうでない場合は再帰呼び出しをする、という選択を書く必要があります。

XSLT スタイルシートの例 recurse.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="asterisksequence">
    <xsl:param name="n"/>
    <xsl:choose>
      <xsl:when test="$n > 0">
        <xsl:variable name="subsequence">
          <xsl:call-template name="asterisksequence">
            <xsl:with-param name="n" select="$n - 1"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:copy-of select="concat('*', $subsequence)"/>
      </xsl:when>
      <xsl:otherwise></xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="n">
    <s>
      <xsl:call-template name="asterisksequence">
        <xsl:with-param name="n" select="."/>
      </xsl:call-template>
    </s>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　recurse.xml

```
<?xml version="1.0"?>
<recurse>
  <n>7</n><n>33</n><n>0</n><n>12</n>
</recurse>
```

変換結果　recurse.xml + recurse.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <s>*******</s>
  <s>**********************************</s>
  <s/>
  <s>************</s>
</result>
```

5.4　数値の処理

5.4.1　階乗

数値に関する概念のうちには、再帰的な構造を持っているものがけっこうたくさんあります。この節では、数値に関する再帰的な概念をいくつか紹介したいと思います。

まず最初に紹介するのは、「階乗」(factorial) という概念です。

n が自然数だとするとき、n から 1 までの整数をすべて乗算した結果、つまり、

$$n \times (n-1) \times (n-2) \times \cdots \times 1$$

という計算の結果のことを、n の「階乗」(factorial) と呼んで、$n!$ と書きあらわします。ただし、$0!$ は 1 だと定義します。

たとえば、$5!$ は、

$$5 \times 4 \times 3 \times 2 \times 1$$

という計算をすればいいわけですから、120 ということになります。

階乗というのは再帰的な構造を持っている概念ですので、$n!$ は、

$$\begin{cases} & 0! = 1 \\ n \geq 1 \text{ ならば} & n! = n \times (n-1)! \end{cases}$$

というように再帰的に定義することができます。

XSLT スタイルシートの例　fact.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="factorial">
    <xsl:param name="n"/>
    <xsl:choose>
      <xsl:when test="$n > 0">
        <xsl:variable name="f1">
          <xsl:call-template name="factorial">
            <xsl:with-param name="n" select="$n - 1"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:copy-of select="$f1 * $n"/>
      </xsl:when>
      <xsl:otherwise>1</xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  <xsl:template match="n">
    <result>
      <xsl:value-of select="concat(., '! = ')"/>
      <xsl:call-template name="factorial">
        <xsl:with-param name="n" select="."/>
      </xsl:call-template>
    </result>
```

```
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　fact.xml

```
<?xml version="1.0"?>
<n>7</n>
```

変換結果　fact.xml + fact.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>7! = 5040</result>
```

5.4.2 フィボナッチ数列

第 0 項と第 1 項が 1 で、第 2 項以降はその直前の 2 項を足し算した結果である、という数列のことを、「フィボナッチ数列」(Fibonacci sequence) と言います。フィボナッチ数列の第 0 項から第 12 項までを表にすると、次のようになります。

n	0	1	2	3	4	5	6	7	8	9	10	11	12
第 n 項	1	1	2	3	5	8	13	21	34	55	89	144	233

フィボナッチ数列というのは再帰的な構造を持っている概念ですので、その第 n 項 (F_n) は、

$$\begin{cases} & F_0 = 1 \\ & F_1 = 1 \\ n \geq 2 \text{ ならば} & F_n = F_{n-2} + F_{n-1} \end{cases}$$

というように再帰的に定義することができます。

XSLT スタイルシートの例　fibona.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="fibonacci">
    <xsl:param name="n"/>
    <xsl:choose>
      <xsl:when test="$n = 0">1</xsl:when>
      <xsl:when test="$n = 1">1</xsl:when>
      <xsl:when test="$n >= 2">
        <xsl:variable name="f2">
          <xsl:call-template name="fibonacci">
            <xsl:with-param name="n" select="$n - 2"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:variable name="f1">
          <xsl:call-template name="fibonacci">
            <xsl:with-param name="n" select="$n - 1"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:copy-of select="$f2 + $f1"/>
      </xsl:when>
      <xsl:otherwise>0</xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  <xsl:template match="n">
    <result>
      <xsl:value-of select="concat('F(', ., ') = ')"/>
      <xsl:call-template name="fibonacci">
        <xsl:with-param name="n" select="."/>
      </xsl:call-template>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　fibona.xml

```
<?xml version="1.0"?>
<n>20</n>
```

変換結果　fibona.xml + fibona.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>F(20) = 10946</result>
```

5.4.3　最大公約数

n がプラスの整数で、m が 0 またはプラスの整数だとするとき、n と m の両方に共通する約数のうちで最大のもののことを、n と m の「最大公約数」(greatest common measure, GCM) と呼びます（m が 0 の場合は、n と m の最大公約数は n だと定義します)。たとえば、100 と 36 の最大公約数は 4 です。

n と m の最大公約数は、「ユークリッドの互除法」(Euclidean algorithm) と呼ばれる次のような再帰的な手順を実行することによって求めることができます。

- m が 0 ならば、n が、n と m の最大公約数である。
- m が 0 よりも大きいならば、n を m で除算したときのあまりを求めて、その結果を r とする。そして、m と r の最大公約数を求めれば、その結果が n と m の最大公約数である。

XSLT スタイルシートの例　gcm.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="gcm">
    <xsl:param name="n"/>
    <xsl:param name="m"/>
    <xsl:choose>
      <xsl:when test="$m = 0">
        <xsl:copy-of select="$n"/>
      </xsl:when>
      <xsl:when test="$m > 0">
        <xsl:call-template name="gcm">
          <xsl:with-param name="n" select="$m"/>
          <xsl:with-param name="m" select="$n mod $m"/>
        </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>0</xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  <xsl:template match="twonum">
    <result>
      <xsl:value-of
          select="concat('gcm(', n, ', ', m, ') = ')"/>
      <xsl:call-template name="gcm">
        <xsl:with-param name="n" select="n"/>
        <xsl:with-param name="m" select="m"/>
      </xsl:call-template>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　gcm.xml

```
<?xml version="1.0"?>
<twonum><n>294</n><m>54</m></twonum>
```

変換結果　gcm.xml + gcm.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>gcm(294, 54) = 6</result>
```

5.4.4 数値から 2 進数への変換

XSLT では、数値を 10 進数以外の記数法による文字列に変換したいときは、そのための名前付きテンプレートを書く必要があります。そこで、そのような名前付きテンプレートの例として、数値を 2 進数に変換する名前付きテンプレートを書いてみましょう。

数値を 2 進数に変換したいときは、まずそれを、2 で除算したときの商とあまりに分解します。そして、商とあまりのそれぞれを 2 進数に変換して、商の 2 進数の右側にあまりの 2 進数を連結します。

商を 2 進数に変換するためには再帰呼び出しを使う必要がありますが、あまりは 0 か 1 のどちらかですから、あまりの 2 進数への変換は、それをそのまま文字列に変換すればいいだけです。

XSLT スタイルシートの例　num2bin.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="number-to-binary">
    <xsl:param name="n"/>
    <xsl:choose>
      <xsl:when test="$n = 0">0</xsl:when>
      <xsl:when test="$n = 1">1</xsl:when>
      <xsl:when test="$n >= 2">
        <xsl:variable name="head">
          <xsl:call-template name="number-to-binary">
            <xsl:with-param name="n"
                select="floor($n div 2)"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:copy-of select="concat($head, $n mod 2)"/>
      </xsl:when>
      <xsl:otherwise>0</xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  <xsl:template match="n">
    <result>
      <d><xsl:value-of select="."/></d>
      <b>
        <xsl:call-template name="number-to-binary">
          <xsl:with-param name="n" select="."/>
        </xsl:call-template>
      </b>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　num2bin.xml

```
<?xml version="1.0"?>
<n>587</n>
```

変換結果　num2bin.xml + num2bin.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result><d>587</d><b>1001001011</b></result>
```

5.5 文字列の処理

5.5.1 部分文字列の置き換え

XSLT では、第 3.6 節で紹介した XPath の関数を組み合わせることによって、文字列に対するさまざまな処理を実行することができます。しかし、ただ単に関数を組み合わせるだけでは実現することのできない処理もあります。そのような処理を実現するためには、名前付きテンプレートを書く必要があります。この節では、文字列を処理する名前付きテンプレートをいくつか書いてみることにしたいと思います。

まず最初に、文字列の中に含まれている部分文字列を別の文字列に置き換える、replace とい

う名前付きテンプレートを書いてみましょう。

replace は、s 、sub 、rep という三つのパラメーターのそれぞれに文字列を受け取って、s の中に部分文字列として含まれているすべての sub を rep に置き換えます。たとえば、s に、

```
namamuginamagomenamatamago
```

という文字列を、sub に ma を、rep に zoza を渡して replace を呼び出すと、その結果として、

```
nazozamuginazozagomenazozatazozago
```

という文字列が得られます。

replace は、次のような手順を実行することによって、文字列 s に含まれている部分文字列 sub を文字列 rep に置き換えます。

(1) contains を使って、s の中に部分文字列として sub が含まれているかどうかを調べる。含まれているならば次の手順を実行して、含まれていないならば s をそのまま返して終了する。

(2) substring-before と substring-after を使って、s を、発見された sub の左側と右側に分解する。

(3) 右側にはまだ sub が含まれている可能性があるので、再帰呼び出しを使ってそれを置き換える。

(4) concat を使って、左側、rep 、そして右側を再帰的に処理した結果を、この順番で連結して返す。

XSLT スタイルシートの例　replace.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="replace">
    <xsl:param name="s"/>
    <xsl:param name="sub"/>
    <xsl:param name="rep"/>
    <xsl:choose>
      <xsl:when test="contains($s, $sub)">
        <xsl:variable name="left"
            select="substring-before($s, $sub)"/>
        <xsl:variable name="right">
          <xsl:call-template name="replace">
            <xsl:with-param name="s"
                select="substring-after($s, $sub)"/>
            <xsl:with-param name="sub" select="$sub"/>
            <xsl:with-param name="rep" select="$rep"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:copy-of select="concat($left, $rep, $right)"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:copy-of select="$s"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  <xsl:template match="replace">
    <result>
      <xsl:call-template name="replace">
        <xsl:with-param name="s" select="s"/>
        <xsl:with-param name="sub" select="sub"/>
        <xsl:with-param name="rep" select="rep"/>
      </xsl:call-template>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　replace.xml

```
<?xml version="1.0"?>
<replace>
```

```
  <s>mmmxyzmmxyzmmmmxyzmm</s><sub>xyz</sub><rep>1234</rep>
</replace>
```

変換結果　replace.xml + replace.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>mmm1234mm1234mmmm1234mm</result>
```

5.5.2　文字列の分解

文字列をいくつかの部分文字列に分解して、それぞれの部分を子供とする要素ノードを生成して、それらの要素ノードから構成されるノード集合を返す、split という名前付きテンプレートを書いてみましょう。

split は、s、sep、name という三つのパラメーターを持っています。s は分解される文字列、sep は部分文字列を区切っている文字列、name は部分文字列を子供とする要素ノードの要素型名です。たとえば、s に、

```
8527,604,2119,33,4040
```

という文字列を、sep に , を、name に n を渡して split を呼び出すと、その結果として、

```
<n>8527</n><n>604</n><n>2119</n><n>33</n><n>4040</n>
```

というノード集合が得られます。

split は、次のような手順を実行することによって、文字列 s を区切り文字列 sub で分解して、それぞれの部分文字列を子供とする要素型名 name の要素ノードを生成します。

(1) contains を使って、s の中に部分文字列として sep が含まれているかどうかを調べる。含まれているならば次の手順を実行して、含まれていないならば、s を子供とする要素型名 name の要素ノードを生成して終了する。

(2) substring-before と substring-after を使って、s を、発見された sep の左側と右側に分解する。

(3) sep の左側を子供とする要素型名 name の要素ノードを生成する。

(4) sep の右側には、まだ sep が含まれている可能性があるので、再帰呼び出しを使って右側を処理する。

XSLT スタイルシートの例　split.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="split">
    <xsl:param name="s"/>
    <xsl:param name="sep"/>
    <xsl:param name="name"/>
    <xsl:choose>
      <xsl:when test="contains($s, $sep)">
        <xsl:element name="{$name}">
          <xsl:copy-of select="substring-before($s, $sep)"/>
        </xsl:element>
        <xsl:call-template name="split">
          <xsl:with-param name="s"
              select="substring-after($s, $sep)"/>
          <xsl:with-param name="sep" select="$sep"/>
          <xsl:with-param name="name" select="$name"/>
        </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
        <xsl:element name="{$name}">
          <xsl:copy-of select="$s"/>
        </xsl:element>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  <xsl:template match="split">
```

```
    <result>
      <xsl:call-template name="split">
        <xsl:with-param name="s" select="s"/>
        <xsl:with-param name="sep" select="sep"/>
        <xsl:with-param name="name" select="name"/>
      </xsl:call-template>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　split.xml

```
<?xml version="1.0"?>
<split>
  <s>SMDL:ABC:LilyPond:MusicXML:GUIDO</s>
  <sep>:</sep><name>lang</name>
</split>
```

変換結果　split.xml + split.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <lang>SMDL</lang><lang>ABC</lang><lang>LilyPond</lang>
  <lang>MusicXML</lang><lang>GUIDO</lang>
</result>
```

5.5.3　2 進数から数値への変換

前の節で、数値を 2 進数に変換する名前付きテンプレートを書きましたが、今度はそれとは逆に、2 進数を数値に変換する名前付きテンプレートを書いてみましょう。

2 進数を数値に変換したいときは、まずそれを、右端の桁と、その左側の 2 進数に分解します。そして、左側の 2 進数と右端の桁をそれぞれ数値に変換して、左側の数値を 2 倍した結果と右端の桁の数値とを加算します。

左側の 2 進数を数値に変換するためには再帰呼び出しを使う必要がありますが、右端の桁は、0 か 1 のどちらかですから、再帰呼び出しを使わなくても数値に変換することができます。

XSLT スタイルシートの例　bin2num.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="binary-to-number">
    <xsl:param name="b"/>
    <xsl:variable name="length" select="string-length($b)"/>
    <xsl:choose>
      <xsl:when test="$length >= 1">
        <xsl:variable name="tail"
            select="substring($b, $length, 1)"/>
        <xsl:variable name="tailnum">
          <xsl:choose>
            <xsl:when test="$tail = '0'">0</xsl:when>
            <xsl:when test="$tail = '1'">1</xsl:when>
            <xsl:otherwise>NaN</xsl:otherwise>
          </xsl:choose>
        </xsl:variable>
        <xsl:choose>
          <xsl:when test="$length >= 2">
            <xsl:variable name="headnum">
              <xsl:call-template name="binary-to-number">
                <xsl:with-param name="b"
                    select="substring($b, 1, $length - 1)"/>
              </xsl:call-template>
            </xsl:variable>
            <xsl:copy-of select="$headnum * 2 + $tailnum"/>
          </xsl:when>
          <xsl:otherwise>
            <xsl:copy-of select="$tailnum"/>
```

```
          </xsl:otherwise>
        </xsl:choose>
      </xsl:when>
      <xsl:otherwise>NaN</xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  <xsl:template match="n">
    <result>
      <b><xsl:value-of select="."/></b>
      <d>
        <xsl:call-template name="binary-to-number">
          <xsl:with-param name="b" select="."/>
        </xsl:call-template>
      </d>
    </result>
  </xsl:template>
  <xsl:template match="b">
    <n>
    </n>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　bin2num.xml

```
<?xml version="1.0"?>
<n>1001001011</n>
```

変換結果　bin2num.xml + bin2num.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result><b>1001001011</b><d>587</d></result>
```

5.6 ノード集合の処理

5.6.1 ノード集合の処理の基礎

第 4.3 節で説明したように、ノード集合を構成しているそれぞれのノードに対して何らかの処理を実行したいときは、普通、xsl:for-each 命令または xsl:apply-templates 命令を使います。しかし、ノード集合に対するどのような処理でも、繰り返しの命令を使うことによって実現することができる、とまでは言えません。繰り返しの命令では実現することのできない処理を書きたいときは、再帰を使う必要があります。

この節では、再帰を使ってノード集合を処理する名前付きテンプレートをいくつか書いてみたいと思います。

再帰を使ってノード集合を処理するためには、与えられたノード集合を、ひとつのノードと、それ以外の残りのノードの集合とに分解することが必要になります。この分解は、1 番目のノードを [1] という述語で取り出して、1 番目以外のノードから構成される集合を、

```
[position() != 1]
```

という述語で作ることによって実現することができます。

5.6.2 ノードの積

ノード集合を構成しているそれぞれのノードを数値に変換して、それらの合計を求めたいというときは、第 3.10 節で説明したように sum という関数を使えばいいわけですが、ノード集合の合計ではなくて積を求めたいというときは、そのための名前付きテンプレートを再帰を使って書く必要があります。

再帰を使ってノードの積を求めたいときは、まずノード集合を、1 番目のノードと、1 番目以外のノードの集合とに分解します。次に、1 番目以外のノードの積を再帰を使って求めます。そして最後に、1 番目のノードと、残りのノードの積との積を求めて返します。

この処理の基底はノード集合が空集合だった場合で、その場合は積として 1 を返します。

ちなみに、真偽値が必要とされる文脈では、空集合は偽に変換されて、空ではないノード集合は真に変換されます。ですから、ノード集合を求める式は、そのままで、ノード集合が空ではな

いという条件をあらわすことになります。

XSLT スタイルシートの例　product.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="product">
    <xsl:param name="nodeset"/>
    <xsl:choose>
      <xsl:when test="$nodeset">
        <xsl:variable name="first" select="$nodeset[1]"/>
        <xsl:variable name="product-of-rest">
          <xsl:call-template name="product">
            <xsl:with-param name="nodeset"
                select="$nodeset[position() != 1]"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:copy-of select="$first * $product-of-rest"/>
      </xsl:when>
      <xsl:otherwise>1</xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  <xsl:template match="product">
    <result>
      <xsl:call-template name="product">
        <xsl:with-param name="nodeset" select="n"/>
      </xsl:call-template>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　product.xml

```
<?xml version="1.0"?>
<product>
  <n>3</n><n>7</n><n>2</n><n>4</n><n>5</n><n>10</n><n>6</n>
</product>
```

変換結果　product.xml + product.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>50400</result>
```

5.6.3　最長ノード

ノード集合を構成しているそれぞれのノードのうちで、文字列に変換したときの長さがもっとも長いものを求める、という処理について考えてみましょう。この処理は、ノードの長さをソートキーにしてノード集合を降順にソートして、その最初のノードを求める、という方法でも実現することができますが、この方法には無駄な処理が含まれていますので、再帰を使うほうが適切です。

再帰を使って最長ノードを求めたいときは、まずノード集合を、1 番目のノードと、1 番目以外のノードの集合とに分解します。次に、1 番目以外のノードの集合のうちでもっとも長いものを再帰を使って求めます。そして最後に、1 番目のノードの長さと、残りのノードの集合のうちでもっとも長いものの長さとを比較して、長いほうを結果として返します。

この処理の基底は、ノード集合に含まれているノードが 1 個だけしかない場合です。その場合は、そのノードがもっとも長いわけですから、そのノードをそのまま返します。また、引数が空集合だった場合に選択される、空集合を返すという選択肢も書いておく必要があります。

XSLT スタイルシートの例　longest.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="longest">
    <xsl:param name="nodeset"/>
    <xsl:variable name="nodecount" select="count($nodeset)"/>
```

```
    <xsl:choose>
      <xsl:when test="$nodecount = 0"></xsl:when>
      <xsl:when test="$nodecount = 1">
        <xsl:copy-of select="$nodeset"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:variable name="first" select="$nodeset[1]"/>
        <xsl:variable name="longest-of-rest">
          <xsl:call-template name="longest">
            <xsl:with-param name="nodeset"
                select="$nodeset[position() != 1]"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:choose>
          <xsl:when test="string-length($first) >
                          string-length($longest-of-rest)">
            <xsl:copy-of select="$first"/>
          </xsl:when>
          <xsl:otherwise>
            <xsl:copy-of select="$longest-of-rest"/>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  <xsl:template match="longest">
    <result>
      <xsl:call-template name="longest">
        <xsl:with-param name="nodeset" select="string"/>
      </xsl:call-template>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　longest.xml

```
<?xml version="1.0"?>
<longest>
  <string>I don't know what I should do.</string>
  <string>Is the invitation still open?</string>
  <string>The sky is clear tonight.</string>
  <string>I have only one dollar right now.</string>
  <string>I promise it will never happen again.</string>
  <string>That's what I want to know.</string>
</longest>
```

変換結果　longest.xml + longest.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <string>I promise it will never happen again.</string>
</result>
```

第 6 章　シリアライズ

6.1　シリアライズの基礎

6.1.1　シリアライズについての復習

この章では、シリアライズを制御する方法について説明したいと思います。

第 2.1 節で説明したように、ツリーを文字列に変換することを、ツリーを「シリアライズする」(serialize) と言います。XSLT プロセッサーは、変換が終わったのち、結果ツリーをシリアライズして、その結果として得られた文字列を出力します。

6.1.2 xsl:output 要素

XSLT プロセッサーによる結果ツリーのシリアライズは、xsl:output という要素型を持つ要素をトップレベル要素として書くことによって、さまざまな制御をすることができます。なお、この要素は常に空要素です。

6.1.3 出力メソッド

XSLT プロセッサーが出力する文書の形式は、「出力メソッド」(output method) と呼ばれる文字列によってあらわされます。XSLT 1.0 では、次の三つの出力メソッドが定義されています。

xml	XML 文書
html	HTML 文書
text	プレーンテキスト

シリアライズの結果として作られる文字列の形式は、xsl:output 要素が持っている method という属性に対して出力メソッドを設定することによって指定することができます。たとえば、

```
<xsl:output method="text"/>
```

という xsl:output 要素を書くことによって、結果ツリーをシリアライズする形式としてプレーンテキストを指定することができます。

6.2 XML 文書へのシリアライズ

6.2.1 符号化方式の指定

xsl:output 要素の method 属性のデフォルトは、基本的には xml です。ですから、XML 文書を XML 文書に変換する XSLT スタイルシートを書く場合、xsl:output 要素を書くことによって文書の形式を指定する必要はありません。

しかし、XML 文書を XML 文書に変換する場合でも、xsl:output 要素を書く必要が生じることがあります。たとえば、結果ツリーのシリアライズに使う符号化方式を指定したいときは、xsl:output 要素を書く必要があります。

xsl:output 要素は、encoding という属性を持っています。この属性に対して符号化方式の名前を設定すると、その符号化方式を使って結果ツリーがシリアライズされます。たとえば、

```
<xsl:output encoding="EUC-JP"/>
```

という xsl:output 要素を書くことによって、符号化方式として EUC–JP を指定することができます。

また、encoding 属性に符号化方式の名前を設定すると、シリアライズで作られる XML 文書の XML 宣言の中で、符号化方式の名前としてその名前が使われます。

XSLT スタイルシートの例　eucjp.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output encoding="EUC-JP"/>
  <xsl:template match="/">
    <result>Good morning.</result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果　empty.xml + eucjp.xsl

```
<?xml version="1.0" encoding="EUC-JP"?>
<result>Good morning.</result>
```

6.2.2 文書型宣言の挿入

XSLT プロセッサーは、結果ツリーを XML 文書にシリアライズする場合、デフォルトではその中に文書型宣言を挿入しません。しかし、xsl:output 要素を書くことによって、シリアライズの結果の中に文書型宣言を挿入することも可能です。

xsl:output 要素は、doctype-system という属性を持っています。この属性に属性値を設定する属性指定を書くと、結果ツリーを XML 文書にシリアライズするときに、その中に文書型宣言が挿入されます。そして、doctype-system 属性に設定された属性値は、システム識別子として文書型宣言の中に挿入されます。

XSLT スタイルシートの例　doctype.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output doctype-system="result.dtd"/>
  <xsl:template match="/">
    <result>Good morning.</result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果　empty.xml + doctype.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE result SYSTEM "result.dtd">
<result>Good morning.</result>
```

6.2.3 公開識別子の挿入

XML 文書に文書型宣言を挿入するときに、公開識別子をその文書型宣言の中に挿入することも可能です。

xsl:output 要素は、doctype-public という属性を持っています。この属性に属性値を設定する属性指定を書くと、その属性値が公開識別子として文書型宣言の中に挿入されます。

XSLT スタイルシートの例　public.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output
      doctype-system="http://www.example.org/result.dtd"
      doctype-public="-//example//DTD result 1.0//EN"/>
  <xsl:template match="/">
    <result>Good morning.</result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果　empty.xml + public.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE result PUBLIC "-//example//DTD result 1.0//EN"
    "http://www.example.org/result.dtd">
<result>Good morning.</result>
```

6.3 HTML 文書へのシリアライズ

6.3.1 method 属性のデフォルト

xsl:output 要素の method 属性に属性値を設定する属性指定が書かれていない場合、結果ツリーは XML 文書にシリアライズされます。ただし、これには例外があって、結果ツリーが XML 文書ではなくて HTML 文書にシリアライズされる場合もあります。

method 属性に属性値が設定されていないときに結果ツリーが HTML 文書にシリアライズされるのは、結果ツリーが次の三つの条件をすべて満足している場合です。

- ルートノードが少なくとも 1 個の要素ノードを子供として持っている。
- ルートノードが子供として持っている 1 個目の要素の要素型名が html である（大文字と小文字は区別しない）。
- ルートノードが子供として持っている 1 個目の要素よりも前に、ホワイトスペース以外の文字を含んでいるテキストノードがない。

ですから、XML 文書を HTML 文書に変換する XSLT スタイルシートを書く場合、これらの条件をすべて満足しているならば、 xsl:output 要素の method 属性に属性値を設定する属性指定は、書かなくてもかまいません。

6.3.2　HTML 文書の文書型宣言

XSLT プロセッサーは、結果ツリーを XML 文書にシリアライズする場合と同じように、結果ツリーを HTML 文書にシリアライズする場合も、デフォルトではその中に文書型宣言を挿入しません。

HTML 文書に文書型宣言を挿入する方法と、文書型宣言に公開識別子を挿入する方法は、XML 文書にそれらを挿入する方法と同じです。すなわち、 doctype-system 属性に属性値を設定する属性指定を書けば、HTML 文書に文書型宣言が挿入されて、その属性値がシステム識別子になります。そして、 doctype-public 属性に属性値を設定する属性指定を書けば、その属性値が公開識別子として文書型宣言の中に挿入されます。

XSLT スタイルシートの例　html.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output
      doctype-system="http://www.w3.org/TR/html4/strict.dtd"
      doctype-public="-//W3C//DTD HTML 4.01//EN"/>
  <xsl:template match="/article">
    <html>
      <head>
        <title><xsl:value-of select="heading"/></title>
      </head>
      <body>
        <h1><xsl:value-of select="heading"/></h1>
        <xsl:apply-templates select="par"/>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="par">
    <p><xsl:value-of select="."/></p>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　article.xml

```
<?xml version="1.0"?>
<article>
  <heading>Mona Lisa Was Stolen</heading>
  <par>Someone stoled Mona Lisa from Louvre Musium.</par>
  <par>How did the thief steal the Mona Lisa?</par>
</article>
```

変換結果　article.xml + html.xsl

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type"
        content="text/html; charset=UTF-8">
    <title>Mona Lisa Was Stolen</title>
  </head>
  <body>
    <h1>Mona Lisa Was Stolen</h1>
    <p>Someone stoled Mona Lisa from Louvre Musium.</p>
    <p>How did the thief steal the Mona Lisa?</p>
  </body>
</html>
```

6.4 プレーンテキストへのシリアライズ

6.4.1 プレーンテキストとは何か

マークアップをまったく含んでいないテキスト、またはマークアップとしてホワイトスペースだけを使って書かれたテキストは、「プレーンテキスト」(plain text) と呼ばれます。たとえば、メールの本文は、たいていの場合、プレーンテキストです[1]。

6.4.2 プレーンテキストを指定する出力メソッド

xsl:output 要素の method 属性に text という出力メソッドを設定すると、XSLT プロセッサーは、結果ツリーに含まれているノードのうちのテキストノードだけをシリアライズして、それ以外のすべてのノードを無視します。ですから、結果ツリーはプレーンテキストにシリアライズされることになります。

XSLT スタイルシートの例　plain.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="plain">
    <xsl:copy-of select="."/>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　plain.xml

```
<?xml version="1.0"?>
<plain>
  <person>Nakatani Hitomi</person>
  <person>Asada Naoko</person>
  <person>Kitamura Kanako</person>
</plain>
```

変換結果　plain.xml + plain.xsl

```
  Nakatani Hitomi
  Asada Naoko
  Kitamura Kanako
```

6.4.3 その他のマークアップ言語

マークアップ言語は、XML 応用言語と HTML だけではありません。それら以外にも、troff、LaTeX、RD など、さまざまなマークアップ言語があります。

XSLT は、XML 文書を XML 応用言語でも HTML でもないマークアップ言語の文書へ変換するという処理をサポートする機能を、何も定義していません。ですから、XML 文書をその他のマークアップ言語の文書へ変換する XSLT スタイルシートを書く場合は、xsl:output 要素の method 属性に出力メソッドとして text を設定して、結果ツリーをプレーンテキストにシリアライズする必要があります。その場合、その他のマークアップ言語のマークアップは、テキストノードとして結果ツリーに挿入することになります。

6.5 ホワイトスペースノード

6.5.1 ホワイトスペースノードとは何か

第 1.2 節で説明したように、空白 (space)、タブ (tab)、復帰 (carriage return)、改行 (line feed) という 4 種類の文字のことを、総称して「ホワイトスペース」(whitespace) と呼びます。XML 文書では、ホワイトスペースは主として、文書を人間にとって読みやすいレイアウトにするために使われます。

[1] ちなみに、メールのヘッダーは、コロンなどの文字をマークアップとして使っています。

ホワイトスペースだけから構成されているテキストノードは、「ホワイトスペースノード」(whitespace node) と呼ばれます。たとえば、

```
<namako>
  <umiushi/>
</namako>
```

という要素をツリーに変換したとすると、namako という要素型の要素ノードは、1 個の改行と 2 個の空白から構成されるテキストノード、umiushi という要素型の要素ノード、1 個の改行から構成されるテキストノード、という 3 個の子供を持つことになるわけですが、それらの 2 個のテキストノードは、ホワイトスペースだけから構成されていますので、ホワイトスペースノードだということになります。同じように、

```
<hitode>&#xA;</hitode>
```

という要素をツリーに変換したとすると、hitode という要素型の要素ノードは 1 個の改行から構成されるテキストノードを子供として持つことになりますが、そのテキストノードもやはりホワイトスペースノードです。

6.5.2　ホワイトスペースノードの削除

XML 文書をプレーンテキストに変換する場合は、XML 文書の中に含まれている空白や改行などのホワイトスペースをどのように扱うかということが重要な問題となります。ホワイトスペースノードをソースツリーから削除することが必要になる場合もありますし、逆に温存することが必要になる場合もあります。

xsl:strip-space という要素型の要素をトップレベル要素として書くことによって、ソースツリーからホワイトスペースノードを削除することができます（この要素は常に空要素です）。

xsl:strip-space 要素を使ってソースツリーからホワイトスペースノードを削除するためには、その要素が持っている elements という属性に対して属性値を設定する必要があります。この属性に設定する属性値は、ホワイトスペースノードを削除する対象となる要素ノードの要素型名を、ホワイトスペースで区切って並べたものです。たとえば、

```
<xsl:strip-space elements="earth sun moon"/>
```

という xsl:strip-space 要素を書いたとすると、earth または sun または moon という要素型名の要素ノードの子供になっているホワイトスペースノードが削除されます。また、

```
<xsl:strip-space elements="*"/>
```

というように、elements 属性にアスタリスク (asterisk, *) を設定すると、ソースツリーに含まれているすべての要素ノードについて、その子供になっているホワイトスペースノードが削除されます。

XSLT スタイルシートの例　strip.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:strip-space elements="space beta"/>
  <xsl:template match="space">
    <xsl:copy-of select="."/>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　space.xml

```
<?xml version="1.0"?>
<space>
  <person>Tanaka Motoko</person><alpha>&#xA;</alpha>
  <person>Suzuki Saya</person><beta>&#xA;</beta>
  <person>Nakatani Yoshiko</person><gamma>&#xA;</gamma>
</space>
```

変換結果　space.xml + strip.xsl

```
Tanaka Motoko
```

```
Suzuki SayaNakatani Yoshiko
```

6.5.3 ホワイトスペースノードの温存

xsl:preserve-space という要素型の要素をトップレベル要素として書くことによって、ホワイトスペースノードを削除するのではなくて温存することもできます。ホワイトスペースノードを温存するというのは XSLT プロセッサーのデフォルトの動作ですので、ソースツリーにあるすべてのホワイトスペースノードを温存したいという場合、この要素を書く必要はありません。この要素型の要素が使われるのは、ほとんどすべてのホワイトスペースノードを削除したいけれども、特定の要素ノードについてはホワイトスペースノードを温存しておきたい、という場合です。

xsl:preserve-space 要素の使い方は、xsl:strip-space 要素の使い方に似ています。それが持っている elements という属性に、ホワイトスペースで区切って要素型名を並べたものを設定すると、それらの要素型名で指定された要素ノードの子供になっているホワイトスペースノードが温存されます。たとえば、

```
<xsl:strip-space elements="*"/>
<xsl:preserve-space elements="sun moon"/>
```

という二つの要素を書いたとすると、sun または moon の子供になっているホワイトスペースノードだけを残して、それ以外のホワイトスペースノードが削除されます。

XSLT スタイルシートの例　preserve.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:strip-space elements="*"/>
  <xsl:preserve-space elements="beta gamma"/>
  <xsl:template match="space">
    <xsl:copy-of select="."/>
  </xsl:template>
</xsl:stylesheet>
```

変換結果　space.xml + preserve.xsl

```
Tanaka MotokoSuzuki Saya
Nakatani Yoshiko
```

6.6 ホワイトスペースノードの挿入

6.6.1 スタイルシートツリーのホワイトスペースノード

XSLT プロセッサーは、XML 文書からソースツリーを作るとき、デフォルトではホワイトスペースノードを温存します。

それに対して、XSLT スタイルシートからスタイルシートツリーを作るときは、ホワイトスペースノードをそこから削除します。ただし、ホワイトスペースノードではないテキストノード、つまりホワイトスペース以外の文字を含んでいるテキストノードからホワイトスペースを削除するということはありません。

次の XSLT スタイルシートは、XSLT スタイルシートの中のホワイトスペースはどのように扱われるのか、ということを確認するためのものです。

XSLT スタイルシートの例　space.xsl

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="/">
    <s>Moriguchi Maiko</s>@
    <s>Shinoda Yuka</s>
    <s>Nakayama Sayaka</s>
    <s>&#xA;</s>
    <s>Hirano Rumiko</s>
  </xsl:template>
</xsl:stylesheet>
```

変換結果 empty.xml + space.xsl

```
Moriguchi Maiko@
    Shinoda YukaNakayama SayakaHirano Rumiko
```

6.6.2 xsl:text 命令

スタイルシートツリーからすべてのホワイトスペースノードが削除されるとするならば、ホワイトスペースノードを結果ツリーに挿入したいときは、いったいどうすればいいのでしょうか。

実は、「ホワイトスペースノードはスタイルシートツリーから削除される」という規則には例外があります。その例外を利用すれば、ホワイトスペースノードを結果ツリーに挿入することが可能になります。

XSLT には xsl:text という要素型があって、その要素型から作られた要素は命令になります。その命令をインスタンス化すると、その内容として書かれた文字列がテキストノードとして結果ツリーに挿入されます。たとえば、

```
<xsl:text>I am a text node.</xsl:text>
```

という xsl:text 命令をインスタンス化すると、I am a text node. というテキストノードが結果ツリーに挿入されます。

xsl:text 命令の子供は、たとえそれがホワイトスペースノードだとしても、スタイルシートツリーから削除されることはありません。つまりそれは、「ホワイトスペースノードはスタイルシートツリーから削除される」という規則の例外なのです。ですから、ホワイトスペースだけを内容とする xsl:text 命令を書くことによって、ホワイトスペースノードを結果ツリーに挿入することができます。たとえば、

```
<xsl:text> </xsl:text>
```

という xsl:text 命令は、1 個の空白から構成されるホワイトスペースノードを結果ツリーに挿入します。同じように、

```
<xsl:text>&#xA;</xsl:text>
```

という xsl:text 命令は、1 個の改行から構成されるホワイトスペースノードを結果ツリーに挿入します。

また、xsl:text 命令は、スタイルシートを読みやすくレイアウトするために書かれているホワイトスペースと、結果ツリーに挿入される文字列とがつながってしまわないようにしたい、というときにも使うことができます。たとえば、

```
<s>namako</s>@
<s>hitode</s>
```

というテンプレートをインスタンス化すると、アットマークの右側の改行は、テキストノードの一部分として結果ツリーに挿入されますが、

```
<s>namako</s><xsl:text>@</xsl:text>
<s>hitode</s>
```

というテンプレートをインスタンス化した場合、xsl:text 要素の右側の改行はホワイトスペースノードになりますので、スタイルシートツリーから削除されます。

XSLT スタイルシートの例 text.xsl

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="/">
    <s>Komori Hiroko</s><xsl:text>@</xsl:text>
    <s>Moroboshi Natsuko</s><xsl:text>&#xA;</xsl:text>
    <s>Shibata Haruna</s><xsl:text>&#xA;</xsl:text>
    <s>Yoshizawa Tomoyo</s><xsl:text>&#xA;</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

変換結果 empty.xml + text.xsl

```
Komori Hiroko@Moroboshi Natsuko
Shibata Haruna
Yoshizawa Tomoyo
```

6.6.3 CSV 文書への変換

マークアップとしてコンマ (comma, ,) を使って書かれた文書は、「CSV 文書」(CSV document) と呼ばれます（CSV は comma separated value の略称です)。

これまでに説明したことを応用すれば、XML 文書を CSV 文書へ変換する XSLT スタイルシートを書くことも可能です。

XSLT スタイルシートの例　csv.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:strip-space elements="*"/>
  <xsl:template match="row">
    <xsl:apply-templates/><xsl:text>&#xA;</xsl:text>
  </xsl:template>
  <xsl:template match="col[position() != last()]">
    <xsl:value-of select="."/><xsl:text>,</xsl:text>
  </xsl:template>
  <xsl:template match="col[last()]">
    <xsl:value-of select="."/>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　table.xml

```
<?xml version="1.0"?>
<table>
  <row>
    <col>001</col><col>Marina</col><col>female</col>
    <col>AB</col><col>Gemini</col><col>Aomori</col>
  </row>
  <row>
    <col>002</col><col>Hitoshi</col><col>male</col>
    <col>B</col><col>Aries</col><col>Kagoshima</col>
  </row>
  <row>
    <col>003</col><col>Sayuri</col><col>female</col>
    <col>O</col><col>Scorpio</col><col>Wakayama</col>
  </row>
</table>
```

変換結果　table.xml + csv.xsl

```
001,Marina,female,AB,Gemini,Aomori
002,Hitoshi,male,B,Aries,Kagoshima
003,Sayuri,female,O,Scorpio,Wakayama
```

第 7 章　その他のトップレベル要素

7.1 xsl:include 要素と xsl:import 要素

7.1.1 インクルードとインポートの基礎

文書の中に、その文書とは別のファイルに格納されている文書を組み込むことを、文書を「インクルードする」(include) と言います。

プログラミング言語やマークアップ言語の多くは、文書をインクルードすることができるという機能を持っています。この機能を使うことによって、いくつかの文書に共通する部分をひとつのファイルに入れておいて、その部分をそれぞれの文書で共有する、ということが可能になり

ます。

「インポートする」(import) というのも、インクルードとほとんど同じ動作です。それらのあいだに違いが発生するのは、組み込み元の文書と組み込まれる文書とで定義の競合がある場合です。

インクルードの場合は、定義が競合していたとしても、それらのあいだに優先順位の差は与えられません。それに対してインポートの場合は、インポート元の文書の定義のほうに、インポートされる文書の定義よりも高い優先順位が与えられます。つまり、インポートされる文書の中の定義は、インポート元の文書の中の定義によって覆い隠されるわけです。

ですから、文書をインクルードするのではなくてインポートすることによって、インポートされる文書の中にある定義に対してインポート元の文書で変更を加えて、その結果を利用する、ということが可能になります。

ちなみに、インポートされる文書の中の定義をインポート元の文書の中の定義によって覆い隠すことを、定義を「オーバーライドする」(override) と言います。

7.1.2 `xsl:include` 要素

XSLT では、`xsl:include` という要素型から作られた要素をトップレベル要素として書くことによって、XSLT スタイルシートをインクルードすることができます。なお、この要素は常に空要素です。

`xsl:include` 要素は、`href` という属性を持っています。この属性には、インクルードする対象となる XSLT スタイルシートが格納されているファイルを指定する URI を設定します。

XSLT スタイルシートのトップレベル要素として `xsl:include` 要素が書かれていた場合、XSLT プロセッサーは、`href` 属性の属性値で指定されたファイルから XSLT スタイルシートを読み込んで、その中のトップレベル要素を、`xsl:include` 要素が書かれている場所にインクルードします。

XSLT スタイルシートの例　`mulstr.xsl`

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="multiply-string">
    <xsl:param name="s"/>
    <xsl:param name="n"/>
    <xsl:choose>
      <xsl:when test="$n > 0">
        <xsl:variable name="substring">
          <xsl:call-template name="multiply-string">
            <xsl:with-param name="s" select="$s"/>
            <xsl:with-param name="n" select="$n - 1"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:copy-of select="concat($s, $substring)"/>
      </xsl:when>
      <xsl:otherwise></xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

XSLT スタイルシートの例　`include.xsl`

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:include href="mulstr.xsl"/>
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="sn">
    <s>
      <xsl:call-template name="multiply-string">
        <xsl:with-param name="s" select="s"/>
        <xsl:with-param name="n" select="n"/>
      </xsl:call-template>
```

```
    </s>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　strnum.xml

```
<?xml version="1.0"?>
<strnum>
  <sn><s>x</s><n>12</n></sn>
  <sn><s>----+</s><n>10</n></sn>
  <sn><s>I'm sorry. </s><n>4</n></sn>
</strnum>
```

変換結果　strnum.xml + include.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <s>xxxxxxxxxxxx</s>
  <s>----+----+----+----+----+----+----+----+----+----+</s>
  <s>I'm sorry. I'm sorry. I'm sorry. I'm sorry. </s>
</result>
```

7.1.3 xsl:import 要素

XSLT では、xsl:import という要素型から作られた要素をトップレベル要素として書くことによって、XSLT スタイルシートをインポートすることができます。なお、この要素は常に空要素です。また、この要素は、他の要素型のトップレベル要素よりも前に書かないといけません。

xsl:import 要素も、xsl:include 要素と同じように href という属性を持っていて、この属性に設定された URI が、インポートする対象となる XSLT スタイルシートが格納されているファイルを指定することになります。

XSLT スタイルシートの例　profile.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:variable name="name" select="'Maehata Misako'"/>
  <xsl:variable name="age" select="28"/>
  <xsl:variable name="mail" select="'misako@example.com'"/>
</xsl:stylesheet>
```

XSLT スタイルシートの例　import.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:import href="profile.xsl"/>
  <xsl:variable name="mail" select="'top secret'"/>
  <xsl:template match="/">
    <result>
      <name><xsl:copy-of select="$name"/></name>
      <age><xsl:copy-of select="$age"/></age>
      <mail><xsl:copy-of select="$mail"/></mail>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

変換結果　empty.xml + import.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <name>Maehata Misako</name>
  <age>28</age>
  <mail>top secret</mail>
</result>
```

7.2 xsl:key 要素

7.2.1 キーとは何か

XSLT では、あらかじめ「キー」(key) と呼ばれるものを宣言しておくことによって、特定の文字列で XML 文書を検索するという処理を簡単に記述することができるようになります。また、キーを使った文字列の検索は、おそらく、それ以外の方法による文字列の検索よりも高速に実行されます（この点については、XSLT プロセッサーの実装に依存します)。

キーというのは、検索によって探し出す対象となるテキストノードや属性値を求めるために使われる式のことだと考えることができます。

7.2.2 キーの宣言

キーを使うためには、あらかじめそれを宣言しておく必要があります。キーは、xsl:key という要素型の要素をトップレベル要素として書くことによって宣言することができます。なお、この要素は常に空要素です。

xsl:key 要素は、name、match、use という三つの属性を持っていて、それらの属性値には次のような意味があります。

name	キーに与える名前。
match	キーを評価するときの文脈ノードとなるノードと一致するパターン。
use	キーとして宣言される式。

たとえば、

```
<xsl:key name="blood-key" match="person" use="blood"/>
```

という xsl:key 要素を書くことによって、person というパターンと一致したノードを文脈ノードとして評価される blood という式をキーとして宣言して、それに対して blood-key という名前を与えることができます。

なお、xsl:key 要素の三つの属性に属性値を設定する属性指定は、三つともかならず書かないといけません。

7.2.3 キーによるノードの検索

キーを使ってノードを検索したいときは、key という関数を呼び出します[1]。

key 関数は 2 個の引数を受け取ります。引数の 1 個目はキーの名前（文字列）で、2 個目は検索したい文字列です。そして key 関数は、検索の結果をノード集合として返します。

key 関数は、xsl:key 要素の match 属性に設定されたパターンと一致したノードごとに、そのノードを文脈ノードとしてキーを評価します。そして、キーの値を文字列に変換した結果と 2 個目の引数とを比較して、一致しているならば、そのときの文脈ノードを、結果となるノード集合に追加します。たとえば、

```
<xsl:key name="blood-key" match="person" use="blood"/>
```

という xsl:key 要素によって blood-key というキーが宣言されているときに、

```
key('blood-key', 'AB')
```

という関数呼び出しを評価したとしましょう。そうすると、呼び出された key 関数は、person というパターンと一致したノードのそれぞれを文脈ノードとして blood を評価して、それを文字列に変換した結果が AB と一致したときの文脈ノードから構成されるノード集合を戻り値として返します。つまり、AB というテキストノードを持つ blood 要素を子供とする person 要素の集合が得られることになります。

ちなみに、それと同じノード集合は、キーを使わずに、

```
//person[blood = 'AB']
```

という式を使って求めることも可能です。

XSLT スタイルシートの例　key.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
```

[1] key は、XPath ではなくて XSLT で定義されている関数です。

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:key name="blood-key" match="person" use="blood"/>
  <xsl:template match="/">
    <result>
      <xsl:copy-of select="key('blood-key', 'AB')"/>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 key.xml

```
<?xml version="1.0"?>
<key>
  <person><name>Tomoko</name><blood>A</blood></person>
  <person><name>Haruka</name><blood>O</blood></person>
  <person><name>Kazuko</name><blood>AB</blood></person>
  <person><name>Noriko</name><blood>B</blood></person>
  <person><name>Masayo</name><blood>AB</blood></person>
  <person><name>Sarina</name><blood>O</blood></person>
</key>
```

変換結果 key.xml + key.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <person><name>Kazuko</name><blood>AB</blood></person>
  <person><name>Masayo</name><blood>AB</blood></person>
</result>
```

key 関数の 2 個目の引数は、文字列以外のデータでもかまいません。2 個目の引数がノード集合だった場合は、それを構成しているそれぞれのノードを文字列に変換して、それぞれの文字列を検索することによって得られた結果の和集合を戻り値として返します。そして、2 個目の引数が文字列でもノード集合でもなかった場合は、それを文字列に変換した結果を検索します。

7.2.4 キーによる参照

番号と正式名称とを対応付ける XML 文書を参照することによって、番号に対応する正式名称を求める、というような処理をする場合にも、キーを使うことができます。

次の XSLT スタイルシートは、著者の番号と氏名とを対応付ける XML 文書を参照することによって、著者の番号と書名から構成される XML 文書を、著者の名前と書名から構成される XML 文書へ変換します。

XSLT スタイルシートの例 refauth.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:key name="id-key" match="author" use="@id"/>
  <xsl:variable name="authors"
      select="document(/bibliography/@authfile)"/>
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="book">
    <xsl:variable name="aid" select="@aid"/>
    <xsl:variable name="title" select="@title"/>
    <xsl:for-each select="$authors">
      <book author="{key('id-key', $aid)/@name}"
            title="{$title}"/>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例 biblio.xml

```
<?xml version="1.0"?>
<bibliography authfile="authors.xml">
```

```
  <book aid="037" title="The Bathroom"/>
  <book aid="006" title="Magellanic Clouds"/>
  <book aid="218" title="When I Was Young"/>
  <book aid="037" title="You Saw Nothing"/>
  <book aid="006" title="Time and Mind"/>
  <book aid="037" title="Humen Science"/>
</bibliography>
```

XML 文書の例　authors.xml

```
<?xml version="1.0"?>
<authors>
  <author id="006" name="Matsuoka Hiroko"/>
  <author id="037" name="Nakamura Tomomi"/>
  <author id="218" name="Tominaga Yoshie"/>
</authors>
```

変換結果　biblio.xml + refauth.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <book author="Nakamura Tomomi" title="The Bathroom"/>
  <book author="Matsuoka Hiroko" title="Magellanic Clouds"/>
  <book author="Tominaga Yoshie" title="When I Was Young"/>
  <book author="Nakamura Tomomi" title="You Saw Nothing"/>
  <book author="Matsuoka Hiroko" title="Time and Mind"/>
  <book author="Nakamura Tomomi" title="Humen Science"/>
</result>
```

上の XSLT スタイルシートの中で使われている xsl:for-each 命令は、繰り返しを目的とするものではなくて、文脈ノードの切り替えを目的とするものです。key 関数は文脈ノードを含んでいる文書しか検索しませんので、document 関数を使って読み込んだ文書の中を key 関数に検索させるためには、このように文脈ノードをその文書へ切り替える必要があります。

7.3　xsl:attribute-set 要素

7.3.1　属性集合の基礎

属性ノードから構成される集合は、「属性集合」(attribute set) と呼ばれます。

XSLT は、属性集合を生成するテンプレートに名前を与えておいて、その名前を使って属性集合を生成する、ということができる機能を持っています。属性集合を生成するために何度も同じテンプレートが利用される場合は、そのテンプレートに名前を付けておけば、それを利用したいときはそれを名前で参照すればいいということになりますので、とても便利です。

なお、属性集合を生成するテンプレートで名前が与えられているもののことを、「名前付き属性テンプレート」(named attribute template) と呼ぶことにします。

7.3.2　名前付き属性テンプレートの定義

属性集合を生成するテンプレートに名前を与えたいとき、つまり名前付き属性テンプレートを定義したいときは、xsl:attribute-set という要素型から作られた要素をトップレベル要素として書きます。

xsl:attribute-set 要素は、自分の子供として書かれたテンプレートに名前を与えます。ただし、子供として書くことができるのは xsl:atrribute 命令だけです（孫としては、どんなテンプレートを書いてもかまいません)。

テンプレートに与える名前は、xsl:attribute-set 要素が持っている name という属性に設定します。たとえば、

```
<xsl:attribute-set name="bookattr">
  <xsl:attribute name="author">Natsume Nanako</xsl:attribute>
  <xsl:attribute name="title">Blue Building</xsl:attribute>
  <xsl:attribute name="publisher">Kontonsha</xsl:attribute>
</xsl:attribute-set>
```

という xsl:attribute-set 要素を書くことによって、その子供として書かれたテンプレートに

対して、bookattr という名前を与えることができます。

7.3.3 名前付き属性テンプレートのインスタンス化

名前付き属性テンプレートを名前で参照することによってインスタンス化するための方法は、二つあります。

ひとつは、xsl:use-attribute-sets という名前の属性に属性値を設定する属性指定をリテラル結果要素の開始タグまたは空要素タグの中に書くという方法です。属性値として設定するのは、インスタンス化したい名前付き属性テンプレートの名前をホワイトスペースで区切って並べたものです。たとえば、

```
<book xsl:use-attribute-sets="bookattr orderattr"/>
```

というリテラル結果要素をインスタンス化することによって、bookattr と orderattr という二つの名前付き属性テンプレートをインスタンス化して、その結果を属性ノードとして持つ、book という要素型名の要素ノードを結果ツリーに挿入することができます。

もうひとつの方法は、XSLT の一部の要素が持っている use-attribute-sets という属性に、名前付き属性テンプレートの名前をホワイトスペースで区切って並べたものを設定する、というものです。

xsl:element は、use-attribute-sets 属性を持っている要素のひとつです。その属性に名前の列を設定しておくと、その名前で指定された名前付き属性テンプレートをインスタンス化することによって生成された属性集合が、xsl:element 命令によって生成される要素ノードに与えられます。たとえば、

```
<xsl:element name="book"
    use-attribute-sets="bookattr orderattr"/>
```

という xsl:element 命令をインスタンス化することによって、上に書いたリテラル結果要素と同じ要素ノードを結果ツリーに挿入することができます。

xsl:attribute-set 要素も use-attribute-sets 属性を持っていますので、その属性に名前の列を設定することによって、別のところで定義されている名前付き属性テンプレートを一部分として含んでいるような名前付き属性テンプレートを定義することができます。

XSLT スタイルシートの例　attrset.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:attribute-set name="personattr">
    <xsl:attribute name="number"><xsl:number/></xsl:attribute>
    <xsl:attribute name="name">
      <xsl:value-of select="."/>
    </xsl:attribute>
  </xsl:attribute-set>
  <xsl:template match="/">
    <result><xsl:apply-templates/></result>
  </xsl:template>
  <xsl:template match="person">
    <person xsl:use-attribute-sets="personattr"/>
  </xsl:template>
</xsl:stylesheet>
```

XML 文書の例　attrset.xml

```
<?xml version="1.0"?>
<attrset>
  <person>Kurahashi Mayumi</person>
  <person>Abe Tomoko</person>
  <person>Kurimoto Yukiko</person>
  <person>Komatsu Sayaka</person>
</attrset>
```

変換結果　attrset.xml + attrset.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
```

```
  <person number="1" name="Kurahashi Mayumi"/>
  <person number="2" name="Abe Tomoko"/>
  <person number="3" name="Kurimoto Yukiko"/>
  <person number="4" name="Komatsu Sayaka"/>
</result>
```

参考文献

[DuCharme,1998] Bob DuCharme, *XML: The Annotated Specification*, Prentice Hall, 1998, ISBN 0-13-082676-6. 邦訳（白根健司）、『XML 技術リファレンス——W3C 正式仕様の完全解説——』、ピアソン・エデュケーション、1999、ISBN 4-89471-126-5。

[Harold,2002] Elliotte Rusty Harold and W. Scott Means, *XML in a Nutshell, Second Edition*, O'Reilly & Associates, 2002, ISBN 0-596-00292-0.

[Holzner,2001] Steven Holzner, *Inside XSLT*, New Riders, 2001, ISBN 0-7357-1136-4. 邦訳（株式会社クイック）、『XSLT 実践ガイド——XSLT スタイルシートによる XML 文書の活用法——』、アスキー、2002、ISBN 4-7561-4065-3。

[Kay,2001] Michael Kay, *XSLT Programmer's Reference, 2nd Edition*, Wrox Press, 2001, ISBN 0-7645-4381-4. 邦訳（イデアコラボレーションズ株式会社）、『XSLT バイブル』、インプレス、2002、ISBN 4-8443-1594-3。

[KySS,2002] PROJECT KySS、ビスケット株式会社、『XSLT+XPath 実践マスター』、ソフトバンクパブリッシング、2002、ISBN 4-7973-1823-6。

[XML,2004] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler and François Yergeau (eds.), "Extensible Markup Language (XML) 1.0 (Third Edition)", World Wide Web Consortium, 2004.

[XPath,1999] James Clark and Steve DeRose (eds.), "XML Path Language (XPath) Version 1.0", World Wide Web Consortium, 1999.

[XSLT,1999] James Clark (ed.), "XSL Transformations (XSLT) Version 1.0", World Wide Web Consortium, 1999.

[高橋,2001] 高橋麻奈、『やさしい XML』、ソフトバンクパブリッシング、2001、ISBN 4-7973-1413-3。

[中山,2001] 中山幹敏、奥井康弘、吉田稔、村上泰介、『改訂版・標準 XML 完全解説』、上巻、技術評論社、2004、ISBN 4-7741-1186-4。

[福内,2003] 福内かおり、木村達哉、『XML 技術者認定試験・XML マスターラーニングブック・ベーシック編』、技術評論社、2003、ISBN 4-7741-1751-X。

[屋内,2002] 屋内恭輔、『XML がわかる本』、毎日コミュニケーションズ、2002、ISBN 4-8399-0762-5。

索引