第9講　XSLT入門「XMLでXMLを処理する？」（担当者：岩井茂樹）

9.1 DOMとXSLT

『XML入門』第10章の例題10-5の課題は，XML文書の構造変換である。サンプルのXML文書（同書，pp.375-376）では，いくつかの<indexterm>要素が誤って<title>要素の中に置かれている。これを，<title>要素の兄弟の位置に移動する変換をおこなう。このXML文書の変換をおこなうため，『XML入門』の著者Ray氏は，DOMモジュールを使うPerlスクリプトを書いた。これが下の例1［Ray, 2004: p.375］。

例1　<indexterm>要素を<title>要素の外に移動させるDOMプログラム　［Ray, 2004: p.375］

```
1:  #!/usr/bin/perl
2:  use XML::LibXML;
3:  my $parser = new XML::LibXML;
4:  my %nodeTypes = (
5:       element => 1,       attribute => 2,    text => 3,
6:       cdatasection => 4,   entityref => 5,    entitynode => 6,
7:       procinstruc => 7,    comment => 8,      document => 9
8:               );
9:  foreach my $fileName ( @ARGV ) {
10:   my $docRef;
11:   eval{ $docRef = $parser->parse_file( $fileName ); };
12:   die( "Parser error: $@" ) if( $@ );
13:   map_proc_to_elems( ¥&fix_iterms, $docRef );
14:   open( OUT, ">$fileName" ) or die( "Can't write $fileName" );
15:   print OUT $docRef->toString();
16:   close OUT;
17: }
18: sub map_proc_to_elems {
19:   my( $proc, $nodeRef ) = @_;
20:   my $nodeType = $nodeRef->nodeType;
21:   if( $nodeType == $nodeTypes{document} ) {
22:      map_proc_to_elems( $proc, $nodeRef->getDocumentElement );
23:   } elsif( $nodeType == $nodeTypes{element} ) {
24:      &$proc( $nodeRef );
25:      foreach my $childNodeRef ( $nodeRef->getChildnodes ) {
26:         map_proc_to_elems( $proc, $childNodeRef );
27:      }
28:   }
29: }
30: sub fix_iterms {
31:   my $nodeRef = shift;
32:   return unless( $nodeRef->nodeName eq 'indexterm' );
33:   my $parentNodeRef = $nodeRef->parentNode;
34:   return unless( $parentNodeRef->nodeName eq 'title' );
35:   $parentNodeRef->removeChild( $nodeRef );
36:   my $ancestorNodeRef = $parentNodeRef->parentNode;
37:   $ancestorNodeRef->insertAfter( $nodeRef, $parentNodeRef );
38: }
```

おなじ変換を実現するXSLTスタイルシートを岩井が書いてみた。それが下の例2である。

例2　<indexterm>要素を<title>要素の外に移動させるXSLTスタイルシート

```
 1:   <?xml version="1.0" encoding="UTF-8"?>
 2:   <xsl:stylesheet  version="1.0"
 3:    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 4:       <xsl:strip-space elements="*"/>
 5:       <xsl:template match="title">              <!-- template[1]-->
 6:          <xsl:copy>
 7:             <xsl:copy-of select="node()[not(self::indexterm)] | @*"/>
 8:          </xsl:copy>
 9:          <xsl:apply-templates select="indexterm"/>
10:       </xsl:template>
11:       <xsl:template match="node()">             <!--tempalte[2] -->
12:          <xsl:copy>
13:             <xsl:copy-of select="@*"/>
14:             <xsl:apply-templates/>
15:          </xsl:copy>
16:       </xsl:template>
17:  </xsl:stylesheet>
```

note：ex10-5のxml文書ではtitle要素のなかにはtext()（テキストノード）と<indexterm>要素しか出現しない。template[1]の <xsl:copy-of>命令の select 属性（第7行参照）をこのようにしておくと，title要素のなかに他のインライン要素や属性，名前空間宣言が現れるばあいでも対応できる。node()[name()!='indexterm'] と node()[not(self::indexterm)] は等価でない。前者ではindexterm要素にくわえて<?indexterm hoge?>という処理命令も除外されてしまうが，後者では indexterm要素だけを除外する。node()は，すべての種類のノードに該当するが，「子」のノードというばあい，テキストノード，要素，処理命令，コメントを含み，属性ノードと名前空間ノードは「子」に含まない。それは，「軸」が異なるからである。要素を起点とするばあい，child 軸と attribute 軸，namespace 軸は区別される。詳しくは，Kay:p.329。

『XML入門』第10章のサンプル文書（ex10_05.xml）にたいし，例1のDOMプログラムと例2のXSLTスタイルシートを適用すれば，同様の結果が得られる（厳密にいうと結果は同一でない。DOMプログラムの出力には問題がある。次節参照）。しかし，見ての通り例2のXSLTのほうがはるかに簡潔である。使われている制御文（if, elsif, foreach など）や命令の数で比較すると，XSLTスタイルシートはDOMプログラムの3分の1程度の記述量である。これほど簡潔に書けて，同じ仕事ができてしまうのは何故だろうか？　しかも，厳密に言うならば例1のDOMプログラムの変換結果には不都合な点があり，XSLTによる変換結果のほうが正しいのである。

例1のPerl言語で書かれたDOMプログラムによる変換処理は次のような手順になっている。

現在のnodeが要素であることを確認（第23行）→

　サブルーチン"fix_iterms"を呼びだす（第24行）

処理対象となる<indexterm>要素を発見（第32～34行）→

　処理ルーチンを呼びだしてその要素nodeのtree上の位置を変更する（削除して元の親nodeに続く位置に挿入　第35～37行）

それ以外のnodeを発見したばあいには，なにもしない

この手順をnode treeをたどりながら再帰的にくり返し（第25～26行），再帰ルーチンから復帰したところで，DOMの文書オブジェクト全体を文字列にして書きだす（第15行）。DOMを使った処理は，rootを起点として枝分かれしながら末端まですべてのnodeが樹状に連なっているXML文書の構造に即したものになる。

樹状構造の再帰的処理は，XML文書を処理するさいにほとんど必ず使われる手法である。XML文書をparseして樹状のnode集合をメモリ上に作り，rootからはじめて再帰的に処理するといった基本的な処理は，いちいちプログラムを書いて実行を指示しなくても，自動的に片づけられるはずだ。つまり，DOMプログラムのなかで，

XML文書をparseしてDOMオブジェクトとしてnode treeを生成（第11～12行）

node treeにたいする再帰処理（第25～26行）

という枠組みの部分を，もっと単純に指示する，あるいは文書処理の前提として自動的に行ってしまうことを工夫してよいはずだ。XML文書からDOMオブジェクトを生成し，直接にそれを操作するプログラムでは，これらの手順をすべて指示せねばならない。こうした決まり切った処理を，自動化，あるいは半自動化したうえで，treeの中から処理対象のnodeを見つけ出す径路（path）の特定（「パタン」と呼ぶ。XPath式にもとづく）と，それにたいする処理方法との組み合わせを用意し，この規則をXML文書に適用した変換結果を出力するような枠組みをつくれば，XMLの変換処理はもっと単純化できるはずだ。XML文書の処理に特化することで，このような単純化を実現したのがXSLTスタイルシートである。

PerlによるDOMを利用した処理では，つねに必要になる決まり切った仕事までプログラムのなかにいちいち書かねばならない。

XSLTスタイルシートでは，決まり切った仕事は自動的にかたづけてくれる。こちらでやるのは，処理対象のpath指定とその処理方法の組み合わせを用意することだけですんでしまう。

```
<xsl:stylesheet>
   <xsl:template match="/">[1]
      .....( root node についての処理)
      <xsl:apply-templates/> (このスタイルシートの再帰的適用)
   </xsl:template>

   <xsl:template match="対象1">
      .....(対象1についての処理)
      <xsl:apply-templates/> (このスタイルシートの再帰的適用)
   </xsl:template>
```

[1] match="/"で使われている単独の”/”は，root nodeを表すパターン。パターンやXPath式のなかで，”/”は pathやstepの区切りを示す演算子であるが，単独の”/”や，”/node-name”のように，XPath式の冒頭に置かれた”/” はroot nodeを表す。root node とはXML文書そのものであり，文書要素（ルート要素ともいう）から要素などの node が樹状に連なるXMLの木構造が，このroot node の上に乗っていると考えればよい。文書要素の名前が”doc”であるとすると，そのXPath式は”/doc”となる。これは「root node の直接の子としてのdocという名前の要素」という意味である。XSLTスタイルシートでは，最初にこのroot node ”/” に適用するtemplate を書くのが普通であるが，先ほどの例2ではroot node に対するtemplate は含まれていない。これは，XSLTスタイルシートを適用すれば，出力結果に自動的にroot node が作られるからである（暗黙のtemplate規則）。templateの適用を明示的にroot node から始める必要がない場合には，root node に対する template は省略してよい。ただし，XSLTスタイルシートがpull型の処理をする場合は，メインに相当する template は，match="/" という対象の指定をしておくのが普通である。 pull型の処理とは，変換元のXML文書から，特定の要素だけを変換および出力の対象とするような処理である。例3の索引作成のスタイルシートはpull型である。

```
   <xsl:template match="対象2">
      .....(対象2についての処理)
      <xsl:apply-templates/>(このスタイルシートの再帰的適用)
   </xsl:template>
   ......
</xsl:stylesheet>
```

というXSLTの基本型は，構造として単純かつ合理的である。

9.2 DOMの長所と欠点

DOMを利用した処理は，XML文書を細かく柔軟に操作できるというメリットがある。しかし，そこに落とし穴があることもある。じつは，『XML入門』の著者Ray氏が書いた例1のDOMプログラムによる変換結果には問題がある。変換元の文書と変換後の文書をよく比べてほしい（『XML入門』pp.375-376）。移動対象である<indexterm>要素の順番が入れ替わってしまっている。元の文書では

"wee creatures"を内容とする<indexterm>要素　：1番目

"woodland faeries"を内容とする<indexterm>要素：2番目

という出現の順番であった。ところが，変換後には，これが逆になっている。

XMLでは要素の出現する位置だけでなく，その順番が意味をもっている。XPathではこれを利用して，

//indexterm[1]，//indexterm[position()=2]，//indexterm[last()]

などの書式による要素の特定が可能であるし，XSLTの<xsl:number>命令を使って，文書中のすべての（あるいは特定の位置にある）<indexterm>要素に連番をふることも可能である。

例題では，変換後に<indexterm>要素の順番が逆になっているが，その後につづくと想定される処理（文章全体から<indexterm>要素だけを抽出して索引をつくる）にとって不都合はないであろう。このため，Ray氏は手を抜いてプログラムを単純にしたものと推測される。しかし，要素の順番が変わってしまうと不都合な場合もある。要素を別の位置に移動する汎用のプログラムでは，このような順序を無視した構造変換は許されない。意図せずに順番を変えるような変換プログラムは缺陥品である。

例1のDOMプログラムでは，何故，移動した<indexterm>要素の順番が逆になるのか。DOM操作の手順を理解すれば，原因を発見するのは容易である（問題は第37行にある）。正しい順番に出力するためには，もう少し複雑な処理をする必要がある。

一方，岩井が書いたXSLTスタイルシートを適用すると，移動後の<indexterm>要素の出現順が乱れることはない。意図的に出現順を逆にするためには，例2のtemplate[2]はもっと複雑になる。

DOMを直接に操作するPerlスクリプトでは，正しい結果を得るために，処理を複雑にしなければならない。一方，XSLTスタイルシートでは，変換の論理どおりに書いておけば正しい結果が得られる。XML文書の変換という処理について，どちらが好ましいであろうか。

Perlはひろく使われ，DOMを使った操作は柔軟である。この利点は大きい。しかし，『XML入門』の著者Ray氏がこの例題のばあいに目指したような単純化（手抜きと言うべきか）をしようすると，望ましい結果が得られないこともある。

一方，XSLTを利用するとXML文書の変換は簡潔に書けて，落とし穴も少ない。しかし，XSLTは汎用言語ではないので，XML文書にXSLTスタイルシートを適用することしかできそうになく，融通がきかない（じっさいにはXSLTでXML文書でない形式の文書を処理することも可能である。詳しくは，Michael Kay著『XSLTバイブル』2002年，pp.627-629を参照）。

しかし，考えてみてほしい。面倒を覚悟してタグや属性をつけ，文書をXMLの形式にするのは，それによって文書の構成要素に意味をもたせ，計算機が文書の構造や意味に即した処理を実行するのを容易にするためだった。不定形なプレイン・テキストであれば，柔軟性にとんだPerlを使って処理するのがよい。しかしわざわざ手間をかけてXML文書にしたものは，それ専門の処理の枠組みを用意すれば，XML文書の利点を活かして，処理方法の指示を簡潔にすることができるし，誤りの発生しにくい堅固なプログラムを書くことが可能になる。

自動車整備工場にいけば，スパナやドライバのような汎用工具のほかに，それぞれの部品や車種に合わせた専用工具がある。汎用工具だけで片づけることも不可能ではないが，工夫が必要になるし，手順が面倒になる。テキパキとやるにはやはり専用工具。「餅は餅屋」というではないか。XSLTは専用工具をつかう自動ロボットつきの整備工場のようなものだ。XML文書と，処理したいnodeの径路指定（XPath式による）とその処理方法の組み合わせ（つまりXSLTスタイルシート）を工場にわたせば，あとは自動ロボットが処理結果を持ってきてくれる。工場のなかでどのように仕事をするか，その手順をいちいち指示する必要はない。手順を知る必要もない。仕事の質が高ければブラックボックスはありがたいのだ。

XSLTのスタイルシートをXML文書に適用して，変換結果を出力する「XSLTエンジン」とよばれる処理プログラム（Java言語ではclassとして提供され，C言語ではlibraryとして提供される）はいくつかあるが，いずれも完成度は高い。

プログラムのなかにブラックボックスがあるのは嫌だ，という御仁はDOMやSAXを使って低レベルの処理まで自分で指示するプログラムを書けばよい。しかし，XML文書の利点を活かして，効率よくしかも美しく処理をするには，XSLTを利用するのがよい[2]。

9.3 索引作成のためのXSLTスタイルシート

例2のスタイルシートは，XML文書の構造を少しだけ変換するものであったが，元のXML文書から必要な部分だけを抽出して，まったく異なった構造のXML文書やテキスト文書，HTML文書などを出力することもできる。さきほどのサンプルデータにはindextermという名前の要素があり，さらに<indexterm>要素のなかに，<primary>要素，あるいは<primary>要素と<secondary>要素の両方が現れている。この<indexterm>要素は，索引として抽出すべき語句を示すための標識であろう。サンプルデータを対象として，索引をつくる作業をおこなうXSLTスタイルシートを書いてみよう。それが例3である。サンプルデータにはページ番号がないので，かわりに各索引項目にchapterとsectionの複合連番と<indexterm>要素の通番を自動的につけ，さらにリーダをつけて見やすくするなどの細工をほどこすことにする。

スタイルシートの全体は，2つのテンプレート規則からできている。template[1]がいわばメインプログラムであり，template[2]はtemplate[1]から呼ばれている（第6行）。

template[1]では，<index></index>という開始タグ・終了タグの組が全体の枠組みをつくる（第5, 7行）。

---

[2] もっとも，XSLTのスタイルが美しいという点には，反論が予想される。CやPerlのような記述の省力化を徹底したプログラミング言語に慣れた人にとって，すべての命令や制御文がXMLの要素の形式をとり，処理の対象や判定条件をその要素の属性として記述するXSLTスタイルシートは，ごたごたとして繁瑣だ，美しくないと感じるかもしれない（これはXML文書全般にも言えること）。しかし，処理対象であるXML文書が，タグをつけることによって意味づけられ，ただ一つの文書要素から枝分かれする構造になっていることを考えれば，それを処理するプログラムが，木構造上の各要素への径路（path）表現と処理内容のペアを並べ，文書順に処理をおこなうというXSLTの処理モデルは，対象と手段とがみごとに適合しているという点でやはり美しい。ついでに雑言をひとつ。Perl言語では，if文のあとにつづく else if を elsif と書くが，このような省力化は美しくない。普通のテキストは自然言語で書かれている。テキストを処理することが主たる目的であるPerl のような言語が，このような不自然なキーワードを使うとは！

例3 索引を作るXSLTスタイルシート

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <xsl:stylesheet version="1.1" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3:   <xsl:output method="xml" encoding="UTF-8" indent="yes"/>
4:   <xsl:template match="/">                          <!--    template[1]-->
5:      <index>
6:         <xsl:apply-templates
   select="chapter/indexterm/*|chapter/section/indexterm/*">
7:            <xsl:sort select="."/>
8:         </xsl:apply-templates>
9:      </index>
10:  </xsl:template>

11:  <xsl:template match="primary | secondary">     <!--    template[2]-->
          <!-- 出現位置のchapter, sectionの連番を変数にセット-->
12:     <xsl:variable name="pos">
13:        <xsl:number format="1.1." level="multiple" count="chapter | section"/>
14:     </xsl:variable>
          <!-- 親の<indexterm>要素の通番を変数にセット-->
15:     <xsl:variable name="num">
16:        <xsl:number format="1" level="any" count="indexterm"/>
17:     </xsl:variable>
          <!-- <i>要素の出力 -->
18:     <i>
19:        <xsl:value-of select="concat(.,substring('.....................',
                                 string-length(.)))"/>
20:        <xsl:value-of select="$pos"/>
21:        <xsl:value-of select="concat('(',$num,')')"/>
22:     </i>
23:  </xsl:template>

24:  </xsl:stylesheet>
```

これが変換結果のXML文書の文書要素（ルート要素）になる。<index>要素の子には，template[2]を適用することによって得られるノード集合をソートして並べたものとなる（第6〜8行）。

template[2]をみると，ローカルな変数posとnumの値を設定してから，<i>要素を作る（第18, 22行）。<i>要素の内容を出力するのが第19〜21行である。この構造が読みとれれば，メインプログラムに相当するtemplate[1]が出力しようとしているのは，

```
<index>
    <i>...</i>
    <i>...</i>
    ...
</index>
```

変数posとnumの値を設定する部分には，<xsl:number>要素が現れる（第13, 16行）。これはたいへんに便利な命令で，単純な数値に書式を設定したり，count属性に指定するnodeの出現順にもとづく番号を生成したりすることができる。

第19行目のselect属性は,

```
concat(.,substring('.....................', string-length(.)))
```

という式が値である。concat() 関数は",″で区切られた複数の文字列（node や変数でもよい）を引数として受け取り，それを結合した文字列を返す。引数の冒頭に，".″という不思議な記号があらわれる。これは文脈node を表す式である。文脈node とは，通常，現在templateが処理しているnode（カレントノード）と一致する。ただし，node をより細かく特定するための述語（XPath式のなかでは[]で囲まれた部分が述語）のなかでは，現在テストされている node が文脈node となる。このため，術語の中ではカレントノードと文脈ノードは一致しないことがある。".″は，″self::node()″の省略表現である。また，"..″という式は，親の node （″parent::node()″ と同義 ）を表す。concat() など文字列関数の引数のなかで ".″ が使われると，それはカレントノードの子であるテキストノード，およびカレントノードの子孫の要素の子であるテキストノードをすべて結合した文字列をあらわす。例3の第19行目の ".″ は，template[2]が現在処理している<primary>要素もしくは<secondary>要素を意味するが，自動的に要素の内容の文字列に変換される。

第19行目の concat() 関数の引数のなかでは，substring() 関数が使われている。これは，見出し語の長さに応じて，適切な長さのリーダ（".....″が一般的）を切り出すために使われている。substring() 関数は3つの引数をとる。

substring(″元の文字列″, (切り出しの開始位置), (切り出す文字数))

3番目の引数が省略されると，切り出し開始位置より後ろの部分文字列を返す。

この例3のように，文字列や部分的な node tree を値とする変数を使ったり，templateをサブルーチンのように呼びだしたり，文字列関数を使ったりすると，複雑な変換や新たな要素，属性の生成などをおこなうことができる。例3はきわめて単純なXML文書を出力するだけだが，XSLTが用意する制御文や関数をつかうと，複雑な処理をすることも可能となる。また，例3のスタイルシートに小さな変更を加えれば，例5のようにテキスト形式で結果を出力することもできる[3]。

また，XSLTスタイルシートは，適用対象の要素ごとに複数のtemplateを並べるという形式をとる。もちろん，<xsl:template match="/">...</xsl:template>というroot node に対するtemplateの中だけですべての処理をおこなうことも可能であるが，そのようなスタイルシートでは処理の見通しが利きにくくなる。複数のtemplateを並べるというのは，プログラムをモジュー

例4 XSLTスタイルシートが出力した索引のXML文書

```
?xml version="1.0" encoding="UTF-8"?>
<index>
   <i>little people.........1.1.(4)</i>
   <i>magical folk..........1.1.(5)</i>
   <i>sprites...............1.(3)</i>
   <i>wee creatures.........1.(1)</i>
   <i>woodland..............1.(3)</i>
   <i>woodland faeries......1.(2)</i>
</index>
```

例5 XSLTスタイルシートが出力した索引のテキスト文書

```
little people.........1.1.(4)
magical folk..........1.1.(5)
sprites...............1.(3)
wee creatures.........1.(1)
woodland..............1.(3)
woodland faeries......1.(2)
```

[3] 出力結果としてテキスト文書を得る場合でも，途中の処理はXMLのnode（要素や属性）の形式で行うべきである。例5の結果を出すには，例3のスタイルシートのtemplate[2]とtemplate[3]は触らずに，template[1]から<xsl:element name="index"></xsl:element >と，<xsl:element name="i"></xsl:element>をコメントアウトすればよい。そうすると，<xsl:value-of select="concat(.,@position)"/>という命令は，何のタグもつかないテキストを出力に複写することになる。さらに簡略な方法もある。例3のスタイルシートの第3行目に見えるmethod属性の値を"text"に設定するだけで，タグのつかないテキストを出力させることができる。ただし，この場合には，すべての項目が改行されずに出力される。あらかじめ，出力される<i>要素のあとに，
（改行の文字参照）を挿入するようにtemplateに手を加えておけばよい。

ル化することに他ならない。モジュール化を意識して書いたスタイルシートでは，対象とするXML文書の構造が変更されるなどしても，関係するtemplateを見つけだして，その部分だけを変更することで対処できる可能性が高い。例えば，例3のスタイルシートを，例2のスタイルシートで処理する前のXML文書，つまり<title>要素の中にも<indexterm>が現れるようなXML文書でも正しく処理できるようにするためには，例3のスタイルシートの第6行，

select="chapter/indexterm/*|chapter/section/indexterm/*">

とある部分を，例えば

select="//indexterm/* ">

に書き換えるだけで済んでしまう[4]。

9.4 XSLTにおける「変数」

例3の索引を抽出して加工するスタイルシートには，

<xsl:variable name=”変数名”>変数の値となるnode集合</xsl:variable>

という形式[5]で，変数を宣言し，それに値を設定するという手続きが含まれている。しかし，このXSLTで使われる変数は，一般の手続き型プログラミング言語の変数とは性格が異なっている。XSLTでは，いったん変数の値を設定すると，スタイルシートの中でその値を変更できないという厳しい制約が課される。つまり，変数の宣言と同時にその値が設定された後で，a = b, a=100, a=”文字列”のような変数の代入文を使ってその値を変更することが許されないわけである。これは，通常の手続き型言語では考えられない制約条件である。for文で使われる繰り返し制御のための変数（iterator）も許されない。したがって，XSLTの<xsl:for-each>文は何らかのnode集合を対象として，そのnodeをひとつずつ処理し，nodeが無くなればループを抜けるというという制御しかできないわけである[6]。変数とはいうものの，値を変更することは不可という条件を課せられたならば，CやPascalのような手続き型言語で多用されるプログラミング技法のほとんどは，大幅な書き換えを迫られるであろう。XSLTが取りつきにくいのは，いささか特異な書法とともに，この「変えられない変数」という制約条件のためであるように思われる。しかし，XSLTに習熟すると，変数の値を処理の途中で変更できなくても，多段処理（二つのことを一度に処理せず，中間の結果置き場の変数をつかうなど）や再帰処理の駆使によって，そうとうに複雑な処理も可能となる。

プログラムの途中で変数の値が変更されないという制約条件には，小さからぬ利点がある。XML文書では，要素の順番が意味をもつ。したがってその処理は文書順におこなうのが普通である。マルチプロセッサなどで並行処理をおこなうことのできる計算機上では，例えば対象となるXML文書の並行処理をおこない，それぞれの処理結果を正しく並べて全体的な結果を出力するようにすれば，高速化が実現できるだろう。処理の途中でグローバルな変数の値が変更される可能性があり，その変更によって処理が分岐したり，異なる結果を導いたりするプログラミング言語であれば，こうした並行処理によって問題が生じることもあり得る。しかし，XSLTのように変数の値の変更を認めないという原則の下では，まずグローバルな変数の値を決定

---

[4] 詳細なパス指定をするには，select=”chapter/indexterm/*｜chpater/section/*｜chapter/title/indexterm/*｜chapter/section/title/indexterm/*”>と書いてもよいが，<indexterm>がさまざまな要素のなかに現れる可能性があるときには，select=”//indexterm/*”>とするのが簡便である。これは select=”descendant::indexterm/child::*” と同義である。ただし，処理対象のXML文書が長大な場合，”//indexterm”という形式のpath指定は処理時間を増大させる。<indexterm>要素が凡ゆる要素のなかに出現する可能性があれば，凡ゆる要素の中を走査して<indexterm>要素を探そうとするからだ。出現位置が限定されている場合には，限定されたpathを指示するほうが好ましい。

[5] <xsl:variable name=”変数名” select=”値”/>という変数宣言の形式もある。

[6] 決められた回数だけループを回すという処理（他の言語ではfor文やwhile文でおこなう）は，XSLTでは再帰テンプレートを用いて実現できる。再帰処理に慣れてしまえば，for文やwhile文が無くてもプログラムは書けてしまう。

してしまえば，あとは並行処理をしても安全だということになる。また，スタイルシートに含まれるtemplate 実行の順番についても，XSLTエンジンが最適化のために順番を変えることの自由度も高まる。

9.5 HTMLへの変換

HTMLはSGMLから生まれた。XMLの母胎もSGMLである。XMLとHTMLが兄弟だという関係を知れば，Web上でXMLを利用できそうだし，それは簡単かも知れないという豫想を懐くにちがいない。XML文書をHTML文書に変換したり，XML文書の構造をかえて必要な情報だけを取りだしそれをHTMLに加工したりすることは，Web上でXML文書を利用するさいの基本的技術である。はじめに紹介したPerl言語は，クライアント側からの要求に応じてサーバで動的にHTMLを生成して送り返すCGIというサーバ側の仕組みのなかで，もっともよく使われている言語であろう。CGIのPerlスクリプトの多くは，print文でHTMLの開始タグ，内容，終了タグを随時に（時にはバラバラで）出力するように書いてある[7]。データベースなど，まったく異なったデータ源からHTMLを生成する場合には，こうした手順になるのも止むを得ないが，XML文書がデータ源であるならば，Perlで変換・加工してHTMLを出力するという方法よりも，はるかに賢い方法がある。それはXSLTスタイルシートを使う方法である。

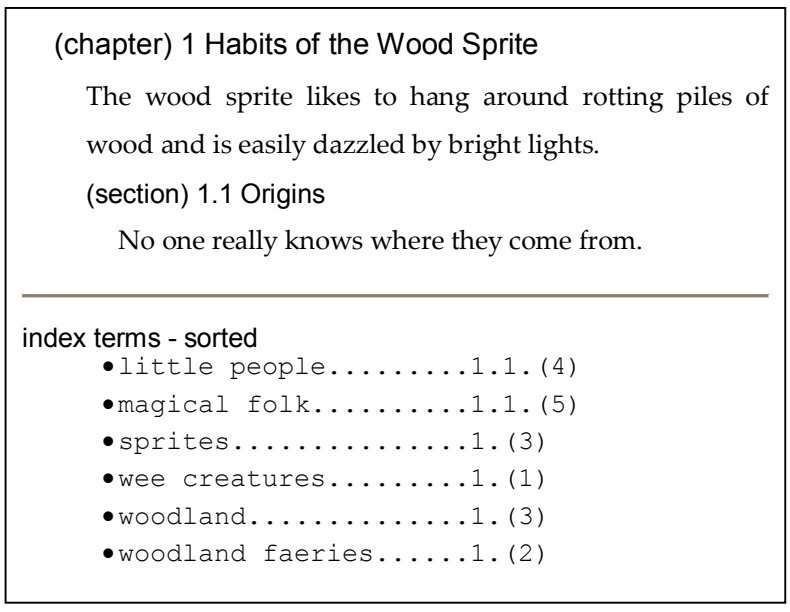
XML文書にXSLTスタイルシートを適用してHTML文書としてブラウザに表示させるには，二つの選択肢がある。一つは，クライアントにXML文書とXSLTスタイルシートを送り，ブラウザに変換の仕事をさせる方法である。これをクライアントサイドの変換という。もう一つはサーバでXML文書にスタイルシートを適用し，変換の結果をHTMLとしてクライアントに返す方法である。これをサーバサイドの変換という。

(1) クライアントサイドの変換（静的な変換）

XML文書のなかには「処理命令」という特殊なnodeを含めることができる。クライアントサイドの変換はこれを利用して実現する。第1節で例としてとりあげたサンプルのXML文書に，例2のXSLTスタイルシートを結びつけて，クライアントのブラウザが前者に後者を適用して変換するように指示する「処理命令」は，

```
<?xml-stylesheet type="text/xsl" href="ex2.xsl"?>
```

である。この処理命令は，XML文書のXML宣言の直後に書いておくという決まりになっている。ただし，この例2のXSLTスタイルシートは変換後のXML文書を出力するためのものだから， HTMLにするには別のスタイルシートが必要である。元のXML文書では，本文中に<indexterm>要素が散在しているが，HTMLとして出力するさいには，<indexterm>要素を取り除いたうえで，末尾にリストアップするなどの工夫を施す必要があろう。ここでは，ブラウザで表示すると，右図のように表示され



[7] Perl でDOMをつかったCGIプログラムを書けば，結果のHTML文書をDOMオブジェクトとして構築し，最後にそれを文字列にシリアライズ（tree構造から直線的なバイト列への変換）してクライアントに送り返すという綺麗なプログラムが書けるはずである。しかし，これは面倒なことらしく，Perl を使ったCGIプログラムではHTMLのタグや内容のテキストを print 文で随時に出力することが普通である。このようなソースは読みにくい。

るHTML文書に変換してみよう。

変換後のHTML文書の骨格は，

<xsl:template match="/">

というtemplateの中に記述する。具体的には，下の例6のようになる。

このスタイルシートには<xsl:要素名>のようにxsl:という接頭辞がついたXSLTの要素と，接頭辞がないHTMLの要素が含まれている。XSLTの要素は元のXML文書にたいする変換をおこなうためのものであるが，xsl:の接頭辞がない要素（リテラル結果要素と呼ぶ）は，この場合，HTMLの要素として相対的な位置を保ちながらそのまま出力される。出力されるHTMLの要素に混じって，変換をおこなうXSLTの命令などが記述されていると見ることもできる。XSLTスタイルシートは，このような「穴埋め」テンプレートの方法に近い記述ができるので，変換後のHTML文書の構造を読みとることは容易である。

<head>要素のなかには，タイトルのほか，CSSの規格に基づくスタイルシートが記述してある。これはchapterとsection，paraという本文の内容をそれぞれブロック要素 に変換したものに適用される書式と，indextermのリストに適用される書式を設定するためである。

例6 HTML文書に変換するスタイルシートの骨格

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <xsl:stylesheet version="1.1" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3: <xsl:output method="html" version="4.0" encoding="UTF-8" indent="yes"
4: doctype-public="-//W3C//DTD HTML 4.01 Transitional//EN"
5: doctype-system="http://www.w3.org/TR/html4/loose.dtd" />
6:       <xsl:strip-space elements="*"/>
7:       <xsl:template match="/">
8:          <html>
9:             <head>
10:               <title>「東アジア人文情報学サマーセミナー 第9講　サンプル</title>
11:               <style type="text/css">
12:                  div {margin-left : 1em}
13:                  ul {margin-left : 4em;font-size : small }
14:               </style>
15:            </head>
16:            <body>
17:               <xsl:apply-templates select="*[not(self::indexterm)]|text()"/>
18:               <hr/>
19:               <h3>index terms  -  sorted</h3>
20:               <ul>
21:                  <xsl:apply-templates select="//indexterm/*">
22:                     <xsl:sort select="."/>
23:                  </xsl:apply-templates>
24:               </ul>
25:            </body>
26:         </html>
27:      </xsl:template>
～
～       以下各要素に対するtemplateを記述
～    </xsl:stylesheet>
```

第17行で<indexterm>要素以外の要素とtext node にこのスタイルシートを再帰的に適用する指示をしている。これによって， <indexterm>要素を除去した本文が書式化されてこの位置に出力される。本文の出力が終わると，水平線<hr/>がひかれ，その後に<h3>index terms - sorted</h3>という小見出しを置き，これに続けて<indexterm>要素をソートされたリストとして出力するようになっている。

元のXML文書の<chapter>，< section>，<para> という階層的な本文の要素は，すべてHTMLの 要素に変換する。そのためのtemplateは次頁の例7のようになる。class="{name()}"は「テンプレート属性値」の書式を用いて， 要素のclass属性の値にそれぞれ元の要素名を設定することを指示している。

例7 〈chapter〉, 〈 section〉, 〈para〉 をHTMLの〈div〉要素に変換するtemplate

```
28:   <xsl:template match="chapter | section | para">
29:      <div class="{name()}">
30:         <xsl:apply-templates select="*[not(self::indexterm)]|text()"/>
31:      </div>
32:   </xsl:template>
```

元の<title>要素を変換するにさいしては，やや込み入った細工を施している。

① タイトルがchapter のものか，section のものかによって見出の文字の大きさを変える。

② タイトルの文字列の前に，階層的な連番をつける。

③ 連番の前に，(chapter)，(section) というヘッダを加える。

これを実現するtemplateは例8である。

例8 〈title〉要素に番号などを附加し，書式つきで出力するtemplate

```
33:   <xsl:template match="title">
34:      <xsl:variable name="depth" select="count(ancestor::*)+1"/>
35:      <xsl:variable name="num">
36:         <xsl:number format="1.1" count="chapter | section" level="multiple"/>
37:         <xsl:text>  </xsl:text>
38:      </xsl:variable>
39:      <xsl:element name="h{$depth}">
40:         (<xsl:value-of select="name(parent::*)"/>)
41:         <xsl:text>  </xsl:text>
42:         <xsl:value-of select="$num"/>
43:         <xsl:copy-of select="text()"/>
44:      </xsl:element>
45:   </xsl:template>
```

タイトルの文字列は，chapterであれば<h2>，sectionであれば<h3>という要素によって書式化したいが，これを実現しているのが，要素をつくる

<xsl:element name="h{$depth}">

というXSLT命令である。要素名が<h2>になるか<h3>になるかは，変数depthの値によって決まる。この変数depthの値は第34～39行で設定している。そのさい，現在処理対象となっている<title>要素の親がchapterであるかsectionであるかという条件判定にもとづいて"2"あるいは"3"という値を設定するようになっている。

個々の<indexterm>を変換して出力するさいにも，加工が必要であろう。

① 個々の indexterm が出現する位置をつけ加える（ページ番号がないので章節番号と通番で代用）

② primary のほか secondary の語彙をもつ場合に，適切な処置をする。

これを実現するtemplateが例9である。ここでは，secondary の語彙はprimary の語彙と同列のあつかいとして，独立した見出し項目にするようになっている。第60行に見えるリーダの長さの計算などは，例3のtemplate と同じである。secondary の語彙を独立の見出項目とせずに，primary の語彙のあとに適当な区切りをいれて続けるという変換をすることも考えられる。また，secondary の語彙を，独立の見出項目とすると同時に，primary の語彙の附加的な説明としても表示するような変換もあろう。これらを実現するには，例9を少し書き換えればよい。XSLTに慣れた人は，試してみると良い。それぞれの場合に，例9の第52行目をの match 属性の値をどうしたらよいかという点と，<primary>要素がカレントノードになっている時に，それと組になっている<secondary>要素を操作するには，following-sibling::secondaryという軸指定子をともなった径路指定の式が使えるという点を押さえておけば，答えは簡単である。

例9 <indexterm>要素に出現場所の番号を附加し，リストアイテム要素として出力するtemplate

```
46:    <xsl:template match="primary | secondary">
47:      <xsl:variable name="pos">
48:         <xsl:number format="1.1." level="multiple" count="chapter | section"/>
49:      </xsl:variable>
50:      <xsl:variable name="num">
51:         <xsl:number format="1" level="any" count="indexterm"/>
52:      </xsl:variable>
53:      <li>
54:         <xsl:value-of select="concat(.,substring('.....................',
                                  string-length(.)))"/>
55:         <xsl:value-of select="$pos"/>
56:         <xsl:value-of select="concat('(',$num,')')"/>
57:      </li>
58:    </xsl:template>
```

ここで取りあげた変換元のXML文書はサンプルの短い文書であるから，XSLTによるHTMLへの変換の有用性は感じにくい。しかし，通常は相当な篇幅となる現実の作品や論文をXMLでマークアップした文書に，ひとつひとつ手作業でHTMLのタグを付けることの手間を考えるならば，少し頭をひねる必要はあるものの，XSLTスタイルシートを書いて自動変換する価値はきわめて大きい。別に，元の世祖クビライの詔令をXML文書化し（本文中に多数の<indexterm>要素を含む），それブラウザで閲覧するためのXSLTスタイルシートのサンプルを用意したので，こちらも試してほしい。

(2) サーバサイドの変換（動的な変換）

Web技術として見た場合，XML文書にXSLTスタイルシートを結びつけて，クライアントサイドで変換をおこなうことに，それほど利点があるわけではない。確かにサーバの負荷は小さくなる。しかし，サーバの負荷を考慮するならば，あらかじめHTMLに変換したファイルをサーバに置いておけばよい。これにたいし，サーバ側でクライアントからの要求に応じて動的にHTML文書を生成してクライアントに送り返す方法には利点がある。それは，クライアントから受けとったリクエストを処理してXSLTに動的なパラメータとして

渡し，それにもとづく変換結果をHTML文書としてクライアントに送り返すことが可能になるからである。サーバのOS環境におうじてXSLTエンジンを選択すれば，リクエストを処理し，XML文書にXSLTスタイルシートを適用することは，CGIによって制御することも可能であるし，Java言語で書かれたサーブレットによって実現することも可能である。こうした技術の詳細については省略するが，Apache Project によって開発されているTomcat と呼ばれるサーブレットエンジン（Javaで書かれ，HTMLサーバとしての機能も兼ねる）は，種々のOS上にインストールして使うことが可能であり，Javaのプラットホームである j2se のバージョン1.4以上には，XMLの parse や XSLTによる変換機能を提供するクラスが附属している。これらを利用すれば，XML文書にたいしサーバサイドのXSLT変換をおこなうことで，Webアプリケーションを作ることはさほど難しくない。

ここでは，その実例として岩井が作った，『元典章』の検索閲覧システムを紹介しよう。このシステムでは，入り口のWebページが静的なHTML文書であるのを除き，クライアントからの要求にしたがって返送される目録や検索結果のリスト，および本文などは，すべてXML文書にXSLTを適用することによって生成される。『元典章』は全70巻に近い13世紀の法制行政文書集である。その本文の条項をXML文書として用意してある（大きさは約3MB）。このシステムは，人文科学研究所のWebサーバ上で稼働しているので，どのような機能があるか，実際に接続して試してもらうのが手っとり早い。URLは，

http://www.zinbun.kyoto-u.ac.jp:8080/

本文の校訂が未了であるので，今のところアクセス制限をかけている。上記アドレスに接続すると，ユーザIDとパスワードの入力を求められるので，ID：yuandz　Password: yuandz55 を入力するとアクセスできる。 本文は，縦書きで表示するようにスタイル指定してあるが，この縦書きのスタイル指定どおりに縦書き表示可能であるのは，Internet Explorer だけのようであり，Mozilla や Netscape では横書きで表示される。句読点表示の有無を切り替えたり，個々の文書の内部構成を示したり，元刊本の文書画像を表示したりする機能を実現している。

おわりに

XMLやXSLT，サーブレットなどの技術が普及した今日では，それらを利用して実用的な検索・閲覧システムを作ることは，いささかのプログラミングの経験さえあれば，専門家でなくても可能である。Unicodeに含まれる漢字の拡張によって，コンピュータ上の漢字が足らないという問題も克服されつつある。

技術的な環境が向上したなかで，依然として最重要の課題は，確かな校訂をほどこしたテキストをつくることである。文字レベルの正確さを期すことはもとより，句読を正しく切ることも専門家にしかできない。中国の公文書は官府間の往復文書を重層的に引用するのが普通である。『元典章』は編纂の過程で適宜節略して引用するため，引用された往復文書の切れ目がどこにあるのか，注意深く解読しないと解らないこともある。こうした校訂作業をおこなうために，現在，人文科学研究所では，金文京教授と岩井が中心となって共同研究班を組織し，『元典章』の会読をおこなう研究班を組織している。入力を急いだこともあるが，研究班での会読をへてみると，苦労して作成した『元典章』電子本文には文字の誤りや句読の不適切な箇所がなお多いことが判明した。電子本文をXML文書化したり，それを検索・閲覧する仕組みを作ったりすることも簡単ではない。しかし，質の高い電子本文を用意することは，さらに重要であり，はるかに難しい。中国学を専門とする我々が努力を傾注すべきことは，じつはこの所にある，というのが最近の私の感想である。