XSLT の逆変換

岡嶋　大介(東京大学) d-oka@is.s.u-tokyo.ac.jp
萩谷　昌己(東京大学) hagiya@is.s.u-tokyo.ac.jp

論文要旨

XSLT(eXtensible Stylesheet Language Transformations)は入力 XML 文書を別のスキーマの出力 XML 文書に変換するルールを定義する宣言的な言語である。この論文では XSLT の逆変換、すなわち XSLT と出力先のスキーマに沿った XML 文書から入力元を得る方法について述べる。

与えられた１つの XSLT は出力先のスキーマに従った XML を生成する文脈自由文法とみなすことができ、その XSLT が変換する入力 XML は、文法のどの生成規則を適用するかのプログラムとみなすことができる。すると、出力 XML から文脈自由文法に基づいた構文木を構成することで、使われた生成規則とそこに書かれた XPath から入力 XML を決定することができる。

実際には XSLT で曖昧な文法を記述することが可能であり、また使われた XPath が複雑すぎるため入力 XML を決定できない場合もあるため完全な逆変換はできない。しかし XSLT,XPath の仕様に制限を加えればこの問題は回避可能である。本論文では最後にこれらの制限を加えた実験的な逆変換の実装についても述べる。

1　背景と動機

XSLT([6])は入力 XML([5])文書を別のスキーマの出力 XML 文書に変換するルールを定義する宣言的な言語であり、1999 年 11 月に W3C の勧告になった。利用例としては、XML 文書を HTML に変換して Web ブラウザで表示できるようにすること、スキーマの更新時に既存データを変換すること、解釈するスキーマの異なるアプリケーション間の橋渡しをすることなどが考えられている。Web ベースのアプリケーションについて書かれた[3]では XML を用いてシステムを構築する際に XML のスキーマ変換を行うために XSLT に似た独自構文を用いているが、目的は XSLT と同じである。

この論文では XSLT の逆変換、すなわち XSLT と出力先のスキーマに沿った XML 文書から入力元を得る方法と、その実験的な実装について述べる。もし完全な逆変換ができると、例えば次のようなアプリケーションが可能になる。

A) GUI による XML エディタ

XML は純粋なデータなので、このままでは GUI による編集には適さない。そこで、あるスキーマに沿った XML を HTML に変換する XSLT を書けば、既存の GUI による HTML エディタを使って編集した結果を逆変換することによりもとの

XML へその編集結果が反映されるようになる。HTML を GUI で編集する研究は例えば[2]などでなされている。

B) スキーマ退化ツール

データをアップグレードするための XSLT を書けば、同時にダウングレードも可能になる。新形式のデータを旧形式のプログラムに読ませる必要がある場合に有用である。

一般には、入力 XML を一意的に定めることはできない。本研究の目標は、より正確には、可能な入力 XML のうちできるだけコンパクトで無駄な要素のないものを求めることである。

2 文脈自由文法としてのXSLT

XSLT は変換方法を記述するので一見するとプログラミング言語のようであるが、XML 出力器として捕らえると、出力先のスキーマに従った XML を生成する文脈自由文法と考えることができる。

簡単な例を挙げてこのことを見てみる。次の XSLT の断片があるとしよう。

```
<xsl:for-each select="person">
 <p><xsl:value-of select="name"/></p>
</xsl:for-each>
```

XSLT はそれ自体も XML であり、上の例の各タグは次のような意味を持っている。

- `<xsl:for-each select="person">`

  入力 XML の各 person タグについて、xsl:for-each の中身を繰り返し実行する。

- `<p>`

  <p>タグを出力する。

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

  入力 XML から name タグの中身を取得して、そのテキストを出力する。

従って、入力 XML が

```
<person><name>Tyler</name></person>
<person><name>Jack</name></person>
<person><name>Marla</name></person>
```

となっていれば、この XSLT をプロセッシングした結果

```
<p>Tyler</p>
<p>Jack</p>
<p>Marla</p>
```

という出力が得られる。

なお、この xsl:for-each、xsl:value-of の select アトリビュートの中に記述した、目標の XML ノードを特定するための文字列は XPath([7])という仕様に則っており、整数イン

デクスによるアクセスや条件指定、論理演算などが可能になっている。

この XSLT が出力する XML(これを非終端記号 T とする)を文脈自由文法で記述すると

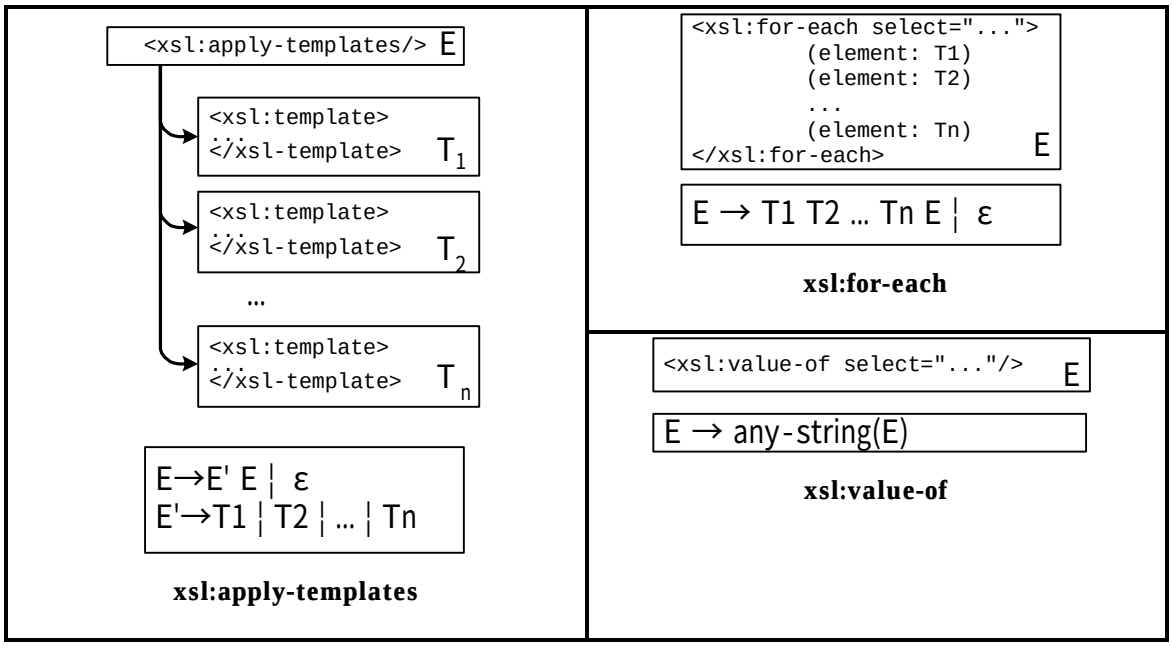
```
T := ε | PT
P := <p>any-string</p>
```

となるのは明らかである。ここで any-string は XSLT からは特定できない任意の文字列を表現する終端記号である。

すると、XSLT が処理する入力 XML は文法のどの生成規則を適用するのかを決定する役割をもっているので、一種のプログラムであると考えることができる。このとき、我々の試みている逆変換は変換後のプログラムから変換前のプログラムを探ることであり、リバースエンジニアリングの一形態であるといえる。

3　文脈自由文法の生成

まず与えられた XSLT が規定する、出力 XML を生成するような文脈自由文法を取得するところから出発する。そのために、各 XSLT エレメントのノードに非終端記号を割り当て、XSLT の局所的な形から文法の生成規則を得る方針を採用した。なお、この文法におけるトークンの種類は、開きタグ・閉じタグ・単純テキスト・アトリビュートの４種類である。主要な XSLT エレメントについて、生成規則は次のようになる。



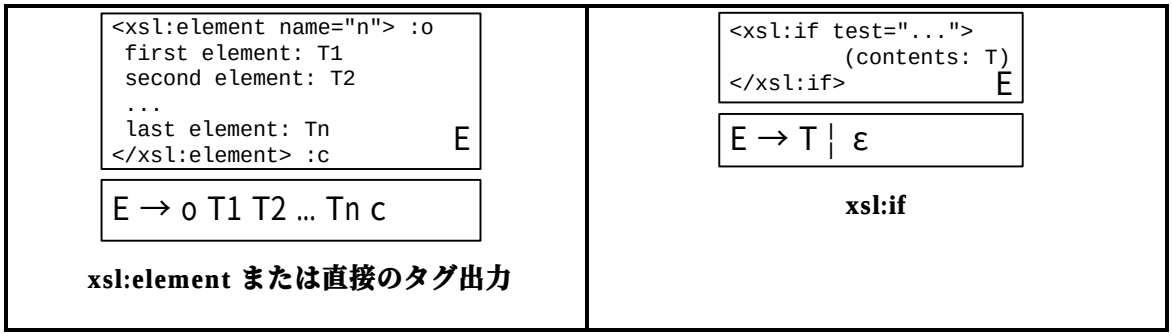


図 1 主要なXSLT 要素についての生成規則

ただし、いくつかの要素については注意を要する。まず、xsl:apply-templates によって次にどの xsl:template が実行されるかは、xsl:apply-templates の select アトリビュートと xsl:template の match アトリビュートを比較することによって候補を限定することができる。しかし上記の図および今回行った実装ではこのことを考慮せず、すべての template に対して可能性があるものとしている。このことが文法の曖昧性に影響することもありうる。

次に、xsl:value-of においては、そこが生成する文字列は任意であるので、タグを含まないすべての文字列にマッチする特殊な終端記号として any-string 記号を導入した。この記号は後に出力 XML をパースする段階で使用される。

また、XML のアトリビュートを出力するためには、

<xsl:attribute name="*attr-name*">*attr-value*</xsl:attribute>

という構文を用いるが、これは直前に出力された XML エレメントに対してアトリビュートを生成するので、XSLT エレメントの順序と出力側の生成順序が一致しない唯一の場合となっている。

このように、通常の文脈自由文法に多少の工夫を加えることが必要である。また、XSLT によっては文法が曖昧になってしまうこともあり、パーサの合成時にこのことが問題になる。

4 パーサの合成

文脈自由文法が得られた後、それをパーサに読ませるため我々は SLR によるパーサを実装した。従って、文法が曖昧でないにも関わらず出力がパースできないケースがあるが、経験的には、ふつう書かれる XSLT においては SLR で十分対応できるようである。SLR でパーサを作る方法は教科書[4]を参考にした。この段階では特に XSLT 固有の工夫はなく、次のような手順に従った。

① 文法中の生成規則に従い、LR(0)項を項集合にまとめる。
② 項集合間をトークンによって遷移するオートマトンを計算する。

③　文法の各非終端記号について FIRST と FOLLOW を計算する。

④　②と③に基づいて構文解析表を作成する。

5　出力 XML のパース

次に、逆変換したい XML をその文脈自由文法が生成する構文木として解釈しなければならない。これは、次のような構文木を求めることである。

(D) <xsl:for-each select="person">**(F)** <xsl:value-of select="name"/> </xsl:for-each> <xsl:if test="mark">**(B)** <xsl:value-of select="mark"/> </xsl:if> 	**D := FI** **F := ***any-string*** F | ε** **B := ***any-string*** | ε**

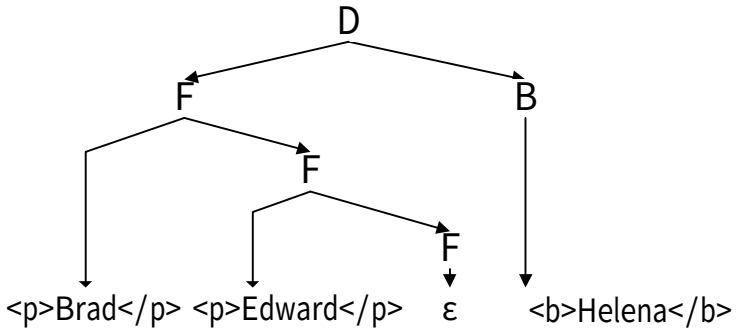


図 2　XSLT と構文木の例

このとき、本来は木構造である XML をストリームとして読まなければならず、また any-string 終端記号やアトリビュートに起因する問題を解決するために通常のパージング方法に手を加える必要がある。

そのために、出力 XML を単なるストリームではなく、パーサが次にどのようなシンボルを期待しているかの指示を受けて適切な振る舞いをさせるという方針を採用し、この方針のもとで次のような問題を解決した。

まず、xsl:value-of は単純なテキスト(ただしタグを含まない)を生成するので、文脈自由文法では終端記号列に相当するものが出力 XML 上では１個のテキストとしてしか得られない場合があるという問題がある。例えば、

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

という記述があると、出力 XML においては Name:Tyler のようになっており、これが単

一のテキストノードとしてしか得られない。これをパースするときには、パーサが次に期待するテキストは”Name:”であるという情報をストリームが受け、ストリームは部分文字列 Name:をパーサに渡し、その次のシンボルとして Tyler を xsl:value-of の値として渡す、という手順を踏む。一方、

```
<xsl:value-of select=”@name”/>さん
```

のように、xsl:value-of の次にテキストが来ているときには、出力 XML 中のテキストノードから部分文字列「さん」を検索し、その直前までを xsl:value-of の値とする。しかしこの手法によっても、xsl:value-of が連続して登場する場合はテキストの切れ目を判定することが不可能であるのでバージングは失敗するなどの問題は残っている。

この時点で、文法上 any-string だったシンボルの値を決定することができ、出力 XML を構文木にまとめることができる。

一方、アトリビュートについては、出力 XML のトークン順序の変更が必要になる。通常は XSLT によって出力される XML は inorder であるからこの順序でパースするのだが、アトリビュート出力が XSLT に書かれていた場合は次のトークンとして出力 XML 側のアトリビュート値をパーサに与える。そのアトリビュート値がパースできたら本来の順序に戻って処理を継続する。

6　入力 XML の合成

そして逆変換の最終段階へすすむ。まず出力 XML をパースして得られた構文木から、文法のどの生成規則が適用されたのかの情報を取得する。また、構文木に現れる非終端記号は XSLT エレメントと対応付けられているので、生成規則が適用されたときの XPath も取得する。両者の情報をあわせると、入力 XML を合成することができる。

例えば、<xsl:value-of select=”name”/>に相当する出力側のテキストが Tyler であることが分かっているとき、新しく name タグを作りその中のテキストを Tyler に設定するといった操作である。さらに、もし XPath に条件指定が含まれていれば、入力 XML もその条件を満たすように合成される。例えば

```
<xsl:value-of select=”person[@country='Japan'][2]/name”/>
```

の意味は、country アトリビュートが Japan であるような２番目の要素の name タグのテキストであるから、

```
<person country=”Japan”/>
<person country=”Japan”><name>Tyler</name></person>
```

といった形で入力の合成が得られる。

我々の実装では、使うことのできる XPath を次の文法が生成できる形に制限している。

```
XPath := Step | Step / XPath
Step := AxisSpecifier NodeTest (Predicate)*
AxisSpecifier := @ | ε
NodeTest := string
Predicate := [XPath = string] | [ number ]
```

[number]の形式の predicate は、直前までの XPath で抽出されたノードの集合のうち number 番目を取り出す役割を持っている。ただし、[number]形式の Predicate は１つの Step の Predicate 列において最後にしか現れることができないものとする。

完全な Xpath の仕様では、Predicate の中に論理演算や変数の参照などかなり複雑な機能を埋め込むことができるが、これらのサポートは行わなかった。また、サブツリー全体を対象とする//という記号もあるが、これを使うと検索結果から木の深さの情報が失われるため逆変換が原理的にできなくなるという点も注意を要する。

以下に、入力 XML 合成のためのアルゴリズムを示す。まず、ノードの順序つき集合に対し、１つの Step に適合する部分集合を返す関数 FindStep を導入する。この関数は自分自身を再帰的に呼び出し、Step の最後の Predicate に合うものを抽出して返す。

```
function FindStep(nodeset, Step)
{
  //We represent this Step as N[P1][P2]...[Pn].
  if(n==0) //with no predicates
    return children whose name is N and whose parent is an element in nodeset.
  else
  {
    x = FindStep(nodeset, N[P1][P2]...[Pn-1])
    if(Pn is of the form path=Literal)
    {
     r = φ
     for each i in x
     {
       if(the first element of FindStep({i}[1], path) equals Literal)
       add i to r
     }
     return r
  }
  else // Pn is a positioned predicate [n]
     return {nth element of x}
  }
}
```

Algorithm 6- 1 Function FindStep

次に、関数 MakeStep を導入する。これは FindStep と同様にノードの集合と Step を受け取り、その Step を満たすようなノードを１つ新たに作成して返す。ただし、MakeStep を実行する時点ではその Step を満たすようなノードは存在しないこと、Step の Predicate

[1] {i} は i のみから成る集合を意味する

はすべて Path=Literal の形式（すなわち、[number]の形式の Predicate は存在しない）という前提をおいている。

```
function MakeStep(nodeset, Step)
{
  //We assume that FindStep(nodeset, Step) returns the empty set.
  //We represent this Step as N[P1][P2]... [Pn-1] [path=Literal].
  if(n==0) //When this step has no predicates
    append an element with its name N to the first element of nodeset and return it
  else
  {
    x = FindStep(nodeset, N[P1]...[Pn-1])
    if(x==φ) x = {MakeStep(nodeset, N[P1]...[Pn-1])}
    //Now we have obtained nodes that meet P1 ,..., Pn-1.
    for each i in x
    {
      v = FindStep({i}, path)
      if(v==φ)
        set the value of MakeStep({i},path) to Literal and return i
    }
    //When existing nodes cannot meet the last predicate,
    append an element e with its name N to the first element of nodeset.
    for each Pi in P1...Pn-1
      set the value of MakeStep({e}, the path of Pi) to Literal of Pi
  }
}
```

Algorithm 6- 2 Function MakeStep

次の MakePositionedStep は、最後の Predicate が[number]形式であることを除いて MakeStep と同じである。[number]は Step の Predicate 列の中で最後にしか現れることができないという制約を置いていることに注意されたい。

```
//We assume that FindStep(nodeset, Step) returns the empty set.
function MakePositionedStep(nodeset, Step)
{
  //We represent this Step as N[P1][P2]... [Pn-1] [pos].
  x = FindStep(nodeset, N[P1][P2]... [Pn-1])
  if((pos+1) > |x|)
  {
    repeat (pos+1)-|x| times
    {
      append an element e with name N to the first element of nodeset
      for each Pi
      {
        set the value of MakeStep({e}, path of Pi) to Literal of Pi
       append e to x as last element
      }
    }
    return the posth element of x.
  }
  }
```

Algorithm 6- 3 Function MakePositionedStep

以上までで、1 Step を処理することができるようになったので、XPath を処理する関数へ移る。これが次の SolvePath 関数である。

```
function SolvePath(node, Step1/Step2/.../StepN)
{
  t = (N==1)? {node} : SolvePath(node, Step1/.../StepN-1)
  if(FindStep(t, StepN)==φ)
  {
    if(StepN is a positioned predicate)
      return MakePositionedStep(t, StepN)
    else
      return MakeStep(t, StepN)
  }
  else
    return the first element of FindStep(t, StepN)
}
```

Algorithm 6- 4 Function SolveStep

入力 XML を合成するには、最初に空の XML から出発し、出力 XML の構文木をトラバースしながらノードを合成する必要が生じるたびに SolvePath を呼んでいく。合成の必要が生じるのは次の場合である。合成時には、通常の XSLT プロセッシングと同様にコンテキストノードに着目し、SolvePath 関数の引数として用いる。

- <xsl:value-of select="*path*"/>

SolvePath(context, *path*)で得られたノードの値を、この xsl:value-of に対応するテキストの値とする。

- <xsl:for-each select="*path*">

構文木を見ることで、この for-each で何回繰り返しが起こったかは分かっているので、まず SolvePath(context, *path*[1][2])によって 1 回目の繰り返しに相当するノードを作り、この新要素を新しい context にして xsl:for-each の子要素を処理する。次に SolvePath(context, *path*[2])によって２回目の繰り返しに相当するノードを作り同様の処理を行う。こうして必要な回数だけ繰り返しをすればよい。

- <xsl:apply-templates select="*path*">

xsl:for-each と似ているが、どの template が次に実行されるかが繰り返しごとに異なる場合がある点が異なる。コンテキストノードが変わるのは for-each と apply-templates の２つだけである。

- <xsl:if test="*path*">, <xsl:when test="*path*">

条件が成立したのかはどうかは出力 XML の構文木を見ることで分かる。条件が成立した場合は、SolvePath(context, *path*)を実行する。そうでなければ何も実行しない。

7　この逆変換手法の限界

しかし実際には次のような理由により完全な逆変換は不可能である。

A. 入力 XML から出力 XML へ渡らなかった(即ち、XSLT のプロセッシングの過程でアクセスされなかった)情報はそもそも復元が不可能。
B. XSLT で曖昧な文脈自由文法を記述することが可能である。
C. xsl:value-of の連続など、復元が原理的にできない構文も存在する。
D. 使われた XPath が入力 XML を合成するための十分な情報を持っていなかったり、複雑すぎたりする場合がある。

理由 A により、我々の目的は、変換すると出力 XML を与えるような入力 XML のうちできるだけ単純なものを求めるということへ修正された。B は、可能な解を複数個提示するような実装であれば回避できるが、応用上は困ることもあると思われる。

また D は XPath の仕様に制限を加えることで回避可能であるが、逆変換が困難なケースは多数存在する。実際、前章で示した可能な XPath の構文は XPath の正規の仕様からすればかなり縮小している。

また不等式を使った条件分岐では入力 XML の値の範囲を特定することしかできない。person[@score < 20]は score アトリビュートが 20 未満である person エレメントを選ぶ XPath だが、この情報を元に入力を合成しようとしても score アトリビュートの値を決定することはできない。

だが、そのような XPath を含む XSLT を逆変換する必要が生ずることは現実的にはあまりないと思われる。

8　実装

我々はこれらの制限を加えた XSLT/XPath を解釈する実験的な逆変換の実験的な実装を行った。この実装は C++言語と Microsoft 製の XML パーサを用いている。

パフォーマンスは次のようになった。比較的単純な XSLT に対して、3KB,10KB,30KB の出力 XML それぞれについての逆変換の結果である。

出力 XML のサイズ	3KB	10KB	30KB
CFG 作成時間	10ms	10ms	10ms
出力 XML をパースする時間	0.02sec	0.14sec	0.80sec
入力 XML を合成する時間	0.17sec	2.7sec	23.6sec
逆変換にかかった合計時間	0.19sec	2.8sec	24.4sec
順方向の変換(参考用)	0.01sec	0.03sec	0.11sec

これを見ると、入力 XML を合成するのにかかる時間が XML のサイズとともに激しく

[2] これは path の最後の Step に[1]という Predicate を付加することを意味する

増大していることが分かる。この理由は、入力 XML を合成する過程で 1 要素を追加する際、合成途中の XML を見ながら追加する位置を決めているためであり、実装上の工夫により改善可能である。アルゴリズム自体に問題があって遅いというわけではない。

次ページはサンプルプログラムの動作しているところである。

9 結論

今回の実装は、入力 XML に複数の形式が可能である場合にその両方を試してみることはせず、判断不能として停止してしまうようになっている。複数の形式が可能になるのは例えば文脈自由文法が曖昧である場合、あるいは XPath が複雑すぎて内容を特定できない場合である。今回は入力 XML のスキーマ情報（DTD など）は一切使っていないので、これも合わせればより正確な変換ができると思われる。

曖昧な文法については、パーサの能力を向上させることにより解決できる。今回はこの

文脈自由文法用 SLR パーサを書いた。しかし、出力 XML をストリームとして読む際に、パーサが次に期待しているシンボルについての情報を見るという処理が必要であるので既存の文脈自由文法パーサを流用することは難しいかもしれない。

複雑な XPath については、入力 XML を XPath から得られた制約条件の集合から合成していく手法に改めることでサポート範囲を広げることができる可能性がある。今回の実装では、１個の XPath から位置が特定できる場合についてしかサポートしていない。

今回サポートできている XSLT,XPath の仕様は全体からするとかなりの制限を受けているが、通常書かれる XSLT で逆変換に意味のあるようなものについてはある程度の能力が既にあると思われる。しかし、一般には逆変換は変換元が一意に決定できないのであればあまり有効ではないであろう。今後の課題としては、変換元が一意に定まるために必要な条件の分析が最も大きいと思われる。

なお、本研究は第一人者の卒業研究[1]として行われた。卒業論文にはより詳しい説明がなされている。

10 参考文献

[1] 岡嶋　大介 , “Inverse Transformation of XSLT”,
東京大学理学部情報科学科卒業論文, 2000.

[2] Stephane Bonhomme, Cecile Roisin,
“Interactively Restructuring HTML Documents”,
Fifth International World Wide Web Conference, 1996,
http://www5conf.inria.fr/fich_html/papers/P16/Overview.html

[3] 八木下 和代,　浦本 直彦,　田村 健人,丸山 宏
XML を用いた Web アプリケーションの構築法, 第 2 回プログラミングおよび応用のシステムに関するワークショップ(SPA'99), 1999

[4] Alfred V. Aho, Ravi Sethi, Jeffrey D.Ullman, 原田賢一(訳), コンパイラ 原理・技法・ツール, サイエンス社、1990, 第４章　構文解析

[5] W3C, Extensible Markup Language(XML), 1998, http://www.w3c.org/TR/xml-rec/

[6] W3C, XSL Transformations Version 1.0, 1999, http://www.w3c.org/TR/xslt/

[7] W3C, XML Path Language Version 1.0, 1999, http://www.w3c.org/TR/xpath/