XMLストリーム処理器の自動導出が可能なXML変換言語の設計

中野圭介

本稿では，XML ストリーム処理器の自動導出が可能な XML 変換言語を設計する方法を提案する．多くの XML 変換言語は木変換として実装されるため，入力される XML の木構造全体をメモリに確保する必要がある．一方，XML ストリーム処理は，こういった無駄なメモリの消費を避けるためによく使われている方法である．XML ストリーム処理では，XML 文書をストリームとして入力しながら，その木構造を完全にメモリに確保することなく，直接，変換結果をストリームとして出力するため，メモリ節減に大いに貢献する．その反面，木構造をもつ XML をストリームとして扱わなければいけないので，プログラミングが複雑になるという問題を孕んでいる．そこで，本研究では，XML ストリーム処理器の自動導出が可能で，且つ，プログラミングしやすい実用的な XML 変換言語の設計方法を提案する．

1 はじめに

XML 変換は，近年急激に増加している XML 形式のデータベースや構造化文書を共有・交換するために，非常な重要な役割を担っている．これに伴い，XML 変換を目的とする XML の木構造の扱いに適した言語が次々と誕生してきた [1][10][5]．しかしながら，これらの言語は，入力の XML 全体の木構造をメモリに展開し，その上を走査しながら変換をする形で実装されるため，単純な変換を行うだけでもメモリや実行時間が浪費されてしまう．XML は，入力段階では単純なストリームであるため，それを木構造にパースした結果をメモリに展開する動作は，単純な形式変換や情報の抽出といった多くの XML 変換に対して，無駄な動作になることが多い．

そこで，**XML ストリーム処理**と呼ばれる，XML を木構造に展開せずに直接変換するようなプログラミングが，時に応じて必要とされる．SAX [2] は XML ストリーム処理を支援する API の一つである．ストリーム処理はメモリ消費を最適化できる反面，プログラムが複雑になるという欠点がある．ストリーム処理は，通常，開始タグ，終了タグなどの XML ストリーム上のトークンを入力とする状態遷移機械の形式で書かれるため，簡単な変換に対しても，入力や出力の XML の木構造を意識する必要があり，プログラマに大きな負担がかかる．

本稿では，新しい XML 変換言語 XTiSP を提案し，この言語によるプログラムからストリーム処理器が自動導出されることを示す．これにより，プログラマは，通常の XML 変換言語のような木構造へのアクセ

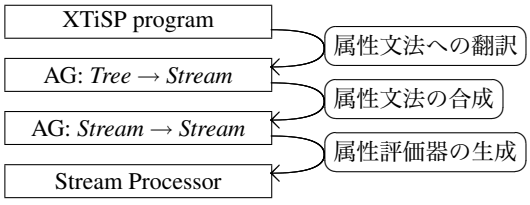


図 1 XML ストリーム処理器の自動導出の流れ

本研究の一部は文部科学省「e-Society 基盤ソフトウェアの総合開発」の委託を受けた東京大学において実施したものである．

Keisuke NAKANO, 東京大学 情報理工学系研究科, Department of Mathematical Informatics, University of Tokyo.

スをするようなプログラミングをするだけで，効率のよい XML ストリーム処理器を得ることができる．この際，プログラマは，ストリーム操作であることを意識する必要は全くない．

ストリーム処理器の自動導出には，属性文法 [11] の枠組を利用した．本研究では，次の三段階に分けて解決している（図 1）．まず，XTiSP で書かれたプログラムは，XML 木から XML ストリームへの変換を表す属性文法に自動的に翻訳される．また，得られた属性文法と，XML ストリームから対応する XML 木を生成する変換（XML パーサ）を表現する属性文法を，**記述的合成法** [9] により合成し，XML ストリームから XML ストリームへの変換を表す属性文法を得る．さらに，得られた属性文法に対して，ストリーム操作に適した属性評価方式を与えることにより，ストリーム処理器を得ることができる．

本問題は，XML パーサという変換とプログラマの定義する XML 木上の変換を合成して，XML の木構造生成をしない変換を導出する問題であるので，一種の**デフォレステーション** [16] の問題と見做すことができる．デフォレステーションとは，二つ以上の変換に対し，それらの合成変換を，中間的なデータ構造を生成しない形で求めることである．属性文法を使う理由は，このようなデフォレステーションの実現に適しているためである．ある制限下の二つの属性文法に対し，記述的合成法により，その合成変換を表す一つの属性文法を得ることができる．ただし，記述的合成法の制限では，パーサを表す変換が合成できないため，[13] で拡張された合成法を利用する．

XML ストリーム処理器の自動導出への試みに対し，ここ数年で数多くの研究がなされている．[7][4][8] では，XPath フィルタに対してストリーム処理をする枠組を提供しているが，XML の変換は一切考慮されていない．また，[12] は，XML 変換言語 XQuery [18] の一部に対して，ストリーム処理器を導出することに成功したが，クエリ言語としての局面が強く，タグ名変更などの単純な変換にも対応していない．XP++ [17] は，Java の拡張言語としてストリーム処理を手軽に扱えるように試みていて，XTiSP とは全く異なる視点からプログラミングのしやすさを追及している．

```
1  <article> [
2    article/(title|author);
3    invite article/body do
4      visit .//bib do
5        <cite id="@id"> [ ]
6      done
7    done;
8    <references> [
9      invite .//bib do
10       <bib id="@id" title="@title"
11            author="@author"
12            year="@year" > [ ]
13     done
14   ]
15 ]
```

図 2 XTiSP によるプログラムの例

一方，本研究の出発点となった [14] では，プログラマは属性文法を使って変換を定義しなければならず，尚且つ，入力される XML 文書の深さが制限されているという問題があった．本研究では，この二つの問題も解消している．

本稿は，まず，第 2 節で言語 XTiSP の概略を述べる．次に，第 3 節において，XTiSP で書かれたプログラムがどのように属性文法に翻訳されるかを示す．第 4 節では，翻訳された属性文法からストリーム処理器を導くための二つの段階について述べる．また，第 5 節に，XTiSP の仕様のうち，前節までの形式化に含まれない部分に対する補足を加えた．最後に，将来課題を交えながら本稿に結論を加える．

2 言語 XTiSP の概略

ここでは，本稿が提案する言語 XTiSP について，図 2 のプログラム例を用いて概説する．このプログラムは `<bib>` 要素を `<cite>` 要素で置き換え，`<references>` 要素内で，その詳細を参照できるようにする変換を表している．XTiSP によるプログラムは，XML 木を受け取り，ノードを往き来しながら処理することで，XML 木の列を返すような変換を行う．XTiSP における変換処理は，XML 要素生成，ノード集積，反復，の三つの操作を用いて記述される．変換処理は，**カレントノード** を移動させながら行われ，処理結果は，；（セミコロン）によって連結される．カレントノードの初期値は，ルートノードとする．

XML 要素生成

XML 要素生成は，[5] に倣い，式 <*s a*>[*e*] によって表現される．ここで，*s* はタグの名前，*a* は属性，*e* は他の XTiSP コードの断片である．例えば，図 2 の 8-14 行目は，9-13 行目が返す XML 木の列を子としてもつような，`references` 要素を返すことを表している．カレントノードは，この操作によって移動しない．

ノード集積

ノード集積は，XPath 式による表現そのものを用いて記述される．この操作による返り値は，カレントノードに対し，XPath 式を満たすようなノードを，入力順に連結したような XML 木の列である．例えば，図 2 の 2 行目は，入力となる XML 木のうち，`article` の子の `title`，`author` 要素を入力順に返すことを表している．XML 要素生成と同様，カレントノードは，この操作によって移動しない．

反復

反復操作は，XPath 式とボディから成る．この操作による返り値は，XPath によって指定されるそれぞれのノードをカレントノードとしてボディを処理した結果によって定義される．以下の節では，XTiSP が扱うことのできる反復処理，**招待反復**と**訪問反復** を順に説明する．

まず，XPath に関して，XTiSP 固有の概念を導入しておく．与えられた XPath 式 *p* に対し，あるノード *N* が *p* を満たし，*N* のどの先祖も *p* を満たさないとき，***N* は *p* を狭義的に満たす**ということにし，通常の意味で，ノードが XPath 式を満たすとき，**広義的に満たす**ということにする．例えば，`//p` は，タグ名 `p` をもつ全てのノードを表現する XPath 式であるが，入力として，XML `<a><p><p></p></p><p>text</p></a>` が与えられた場合，下線部の `p` ノードは，広義的には `//p` を満たすが，狭義的には 満たさない．

招待反復は，`invite`[*] *p* `do` *e* `done` の形で定義される．ここで，*p* は XPath 式，*e* は XTiSP プログラムである．`invite` と `invite*` の違いは，*p* を狭義的に満たすノードを処理するか，広義的に満たすノードを処理するかの違いである．招待反復では，*p* により指定されたノードの列に対し，各ノードをカレントノードとしてボディ *e* の処理を行い，それらの結果を連結したものを，全体の反復処理の結果として返す．例えば，図 2 の 9-13 行目は，XPath 式 `.//bib` を狭義的に満たすノードに対して，そのノードをカレントノードとして 10-12 行目の処理（`bib` 要素の生成）を行い，その結果を連結した結果を返す．

訪問反復は，`visit`[*] *p* `do` *e* `done` の形で定義される．訪問反復では，*p* により指定されたノード以外のカレントノード以下の構造は保存され，それを結果として返す．指定されたノードに関しては，ボディ *e* を処理した結果で置き換えられる．例えば，図 2 の 4-6 行目の `visit` 式に注目しよう．3 行目の `invite` により，カレントノードが `body` 要素になっているので，この反復は，XPath 式 `.//bib` で指定されるノードを 5 行目の処理（`cite` 要素の生成）の結果で置き換えたような `body` 要素を返す．

招待反復とは異なり，`visit` と `visit*` の違いは単純ではない．`visit*` のボディでは，特殊な変数 `children` を用いて，子ノードに対しての反復が定義できる．厳密な定義は割愛するが，例えば，右のプログラムは，全ての `a` タグを `b` タグに変更する処理を表している．

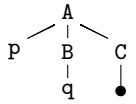
```
visit* .//a do
  <b> [ children ]
done
```

3 属性文法への翻訳

XTiSP で書かれたプログラムは，全て属性文法に翻訳される．ここでは，その方法について簡単に説明する．属性文法の詳細な定義については，[11][3] を参照されたい．本稿では，今後，「属性」は属性文法のことを表し，XML 文書内の属性に関しては「XML 属性」と表すものとする．

3.1 XML 木の表現

属性文法は文脈自由言語上の変換を定義するため，XML 木の表現として二分木表現を用いる．例えば，XML `<A>p<B>q</B><C/></A>` は，右のような木構造を定義しているが，二分木表現では，下に示したような木で表される．二分木表現では，各ノードにおいて，



$S_0 \to T_1$:
  $S_0.result^{\uparrow} = T_1.xml^{\uparrow}$
  $T_1.xml^{\downarrow} = \varepsilon$
  $T_1.under\_a^{\downarrow} = false$

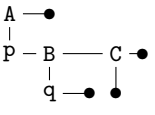
$T_0 \to$ *text* $c\ T_1$:
  $T_0.xml^{\uparrow} = c\ T_1.xml^{\uparrow}$
  $T_1.xml^{\downarrow} = T_0.xml^{\downarrow}$
  $T_1.under\_a^{\downarrow} = T_0.under\_a^{\downarrow}$

$T_0 \to$ *leaf* :
  $T_0.xml^{\uparrow} = T_0.xml^{\downarrow}$

$T_0 \to$ *node* $t\ T_1\ T_2$:
  IF $T_0.under\_a^{\downarrow}$ THEN
    $T_0.xml^{\uparrow} =$ <y> $T_1.xml^{\uparrow}$
    $T_1.xml^{\downarrow} =$ </y> $T_2.xml^{\uparrow}$
    $T_2.xml^{\downarrow} = T_0.xml^{\downarrow}$
  ELSE
    $T_0.xml^{\uparrow} =$ <n> $T_1.xml^{\uparrow}$
    $T_1.xml^{\downarrow} =$ </n> $T_2.xml^{\uparrow}$
    $T_2.xml^{\downarrow} = T_0.xml^{\downarrow}$
  ENDIF
  $T_1.under\_a^{\downarrow} = (t = \mathtt{a})$
  $T_2.under\_a^{\downarrow} = T_0.under\_a^{\downarrow}$

図 3 XML 変換を定義する属性文法

左の子は XML 木における子の長男ノード，右の子は XML 木における直後の弟ノードに相当する†1．二分木表現では，子の数が限られているので，属性文法が定義しやすい．XML 木の二分木表現は，導出規則 $S \to T$, $T \to$ *node* $t\ T\ T$, $T \to$ *text* $c\ T$, $T \to$ *leaf* で定義される．ここで，t は要素（タグ）名，c は文字列データに相当する．本稿では，XML 属性は省略しているが，要素名と同様に扱うことで拡張できる．



3.2 属性文法による XML 変換

図 3 は属性文法によって定義される XML 変換の一例である．この XML 変換は，親要素が a という名前を持つような要素のタグは y に改名し，その他のタグは n という名前に改名する変換を表す．属性文法は，文脈自由言語の導出規則にそれぞれ**属性規則**を与えることで定義される．各属性規則は，非終端記号に付随している属性の依存関係を等式の集合で定義している．$\uparrow$ は合成属性，$\downarrow$ は相続属性を表す†2．

本枠組で扱う属性文法は，IF による条件分岐を許しており，今の例では，$T \to$ *node* $t\ T_1\ T_2$ に与えられた属性規則は，属性 $T_0.under\_a^{\downarrow}$ の値に応じて，一部の属性は別々の定義が与えられている．属性 $under\_a^{\downarrow}$ には，親の要素名が a であるかを示す真偽値が代入され，その値は右の子（XML 木としては弟）に伝播する．大まかにいえば，$T_1.xml^{\uparrow}$ には，部分木に対する変換結果の後ろに $T_1.xml^{\downarrow}$ で渡される値が返ってくるので，$T_0.under\_a^{\downarrow}$ が真の時は，その部分結果を <y> と </y> で挟んだ結果が親に伝播され，偽の時はその部分結果を <n> と </n> で挟んだ結果が親に伝播される．

$S_0 \to T_1$:
  $T_1.p1^{\downarrow} = true$
  $T_1.p2^{\downarrow} = false$

$T_0 \to$ *node* $t\ T_1\ T_2$:
  $T_1.p1^{\downarrow} = false$
  $T_2.p1^{\downarrow} = T_0.p1^{\downarrow}$
  $T_1.p2^{\downarrow} = T_0.p2^{\downarrow}\ ||\ (T_0.p1^{\downarrow}\&\&(t = \mathtt{a}))$
  $T_2.p2^{\downarrow} = T_0.p2^{\downarrow}$
  $T_0.q1^{\uparrow} = (t = \mathtt{c})\ ||\ T_2.q1^{\uparrow}$

図 4 XPath を表現する属性文法の一部

3.3 XPath 部分の翻訳

ノード集積に用いられる XPath は後述の `invite*` による反復の翻訳と同様に処理されるため，ここでは，反復表現の定義に用いられる XPath について属性文法に翻訳することを考える．XTiSP のプログラムに記述される XPath は相対パスであるが，カレントノードを静的に解析できるため，絶対パスのみを翻訳すれば十分である．また，絶対 XPath に含まれる後方参照はすべて除去できるため [15]，XTiSP では後方参照を含まない絶対 XPath のみを対象とする．与えられた XPath 式に対し，その部分式ごとに属性を対応させることにより，属性文法への翻訳ができる．`[ ]` で囲まれた述部に含まれる部分式は合成属性，それ以外の部分式には相続属性を対応させる．ここでは，詳細な定義は省略し，例を用いて説明する．

`/a//b[c]` によって表される XPath 式 p は，XPath の記法として省略せずに書けば，

$$\underbrace{\underbrace{\texttt{/child}::\texttt{a/descendant}::\texttt{b}}_{p1^{\downarrow}}}_{p2^{\downarrow}}\underbrace{\texttt{[child::c]}}_{q1^{\uparrow}}$$

である．下部の括弧で示される $p1^{\downarrow}$，$p2^{\downarrow}$，$q1^{\uparrow}$ は，各部分式†3に対応づけられた属性を表している．これらの属性の値は図 4 のような属性規則によって定義される．$p1^{\downarrow}$ は，そのノードがルートの子であるか否かの真偽値を示し，$p2^{\downarrow}$ は，そのノードがルートの子の

†1 p や q のような文字列ノードでは，一つだけ子を持ち，直後の弟ノードに対応している．
†2 $xml^{\uparrow}$，$xml^{\downarrow}$ は全く別の属性である．
†3 変則的な部分式のとり方だが，属性文法への翻訳の都合である．

a 要素の子孫であるか否かの真偽値を示し，$q1^{\uparrow}$ は，そのノードに続く兄弟に c 要素を含むか否かの真偽値を示す．あるノードが p を満たすかどうかで異なる属性定義をしたい場合，$T_0 \to node\ t\ T_1\ T_2$ に付随する属性規則として，

$$\mathsf{IF}\ T_0.p2^{\downarrow}\&\&(t=\mathtt{b})\&\&T_1.q1^{\uparrow}$$
$$\mathsf{THEN}\ e_1\ \mathsf{ELSE}\ e_2\ \mathsf{ENDIF}$$

という形の規則を用いて表現できる．

3.4 反復部分の翻訳

全ての反復構造に対し，合成属性と相続属性が一つずつが対応づけられていて，合成属性は部分的な結果を返すために使われ，相続属性は結果に続くストリームを与えるために使われる．

invite や visit といった狭義的反復構造には，$T_0 \to node\ t\ T_1\ T_2$ に付随する属性規則として，右のような形の規則で定義される．ここで，π_p は，反復構造に与えられた XPath 式 p を前節のような方法で翻訳した式とする．属性 $xml^{\uparrow}$ や $xml^{\downarrow}$ には結果が蓄積される．大まかにいえば，XPath を満たすノードでは先頭に (A) の値を追加し，それ以外のノードでは子の結果を (B) の値と (C) の値で挟んだ結果を追加している．追加される値は，それぞれ，反復構造のボディによって定義される†4．分岐 THEN で，$T_1.xml^{\uparrow}$ が参照されていないことは，狭義的反復構造を考えていることに起因する．

IF π_p THEN
  $T_0.xml^{\uparrow}$ = (A) $T_2.xml^{\uparrow}$
  $T_2.xml^{\downarrow}$ = $T_0.xml^{\downarrow}$
ELSE
  $T_0.xml^{\uparrow}$ = (B) $T_1.xml^{\uparrow}$
  $T_1.xml^{\downarrow}$ = (C) $T_2.xml^{\uparrow}$
  $T_2.xml^{\downarrow}$ = $T_0.xml^{\downarrow}$
ENDIF

XTiSP の式 `invite` p `do <a>[] done` に対しては，(A) の部分に `<a></a>` が代入され，(B) と (C) の部分には空文字列が代入されることにより，翻訳が成立する．また，`visit` p `do <a>[] done` に対しては，(A) の部分に `<a></a>` が代入され，(B) と (C) の部分に `<`t`>` と `</`t`>` がそれぞれ代入されることにより，翻訳が成立する．

広義的反復構造に関しては割愛するが，上記とほぼ同様の方法で翻訳が実現できる．

†4 ボディが反復構造である場合は，その反復構造に対応した合成属性が，(A) とそれに続く属性の部分に置換される．

4 ストリーム処理器の導出

前節で得られた属性文法は，属性文法の合成により，XML ストリームから XML ストリームへの変換を表す属性文法を得ることができ，その評価器を与えることで，XML ストリーム処理器が導出できる．

4.1 XML パーサとの合成

XML パーサは，属性文法を使って表現できる [14]．具体的な定義は省略するが，兄弟の変換結果を蓄積するスタック型の合成属性と部分木の変換結果を保存する合成属性により定義される．この属性文法と前節で得られた属性文法を，属性文法の合成により，XML ストリームから XML ストリームへの変換を表す属性文法を得る．

属性文法の合成は，記述的合成法 [9] がよく知られているが，パーサを表す属性文法は，スタック構造の扱いを必要とするため，拡張された記述的合成法 [13] を用いる．この方法では，条件分岐を含むような属性文法の合成は示していないが，[6] での拡張方法も用いることで適用が可能になる．

4.2 属性評価器

XTiSP では，ストリーム処理器の導出を目的とした属性評価器を提案した．基本的には，[14] で導入したような，依存グラフに基づく解析で評価器を生成するが，XTiSP により生成される属性文法はスタック構造を含むため，この解析を静的に行うことが困難である．そこで，XTiSP で実装される属性評価器では，ストリームのイベントを読む毎に，対応する属性規則に従って属性評価を行い，先のイベントの結果を必要とする合成属性の部分は，それを変数として抽象化しておく．また，結果となる属性の値の頭部に連結されている値は，後に続くイベントにより変更されることがないため，そのまま出力することができ，この結果，入力しながら出力を行うというストリーム処理が可能になる．

5 XTiSP の特徴とその可能性

前節までに述べた通り，XTiSP によって定義された全てのプログラムは，XML ストリーム処理器に自動的に導出される．ここでは，XTiSP の表現力や，導出されるストリーム処理器の計算量などに関して，問題点や解決の可能性に関して述べる．

5.1 XTiSP の表現力

付録 A で紹介しているように，条件分岐や原始関数を含むようなプログラムも XTiSP によって表現が可能である．XML の標準的な変換言語である XSLT [1] は，いくつかテンプレートを XPath などに基づいて呼ぶことで定義しているが，XTiSP ではこれを `invite*` などで書き換えることができるため，一部の XSLT のようなスタイルのプログラムも変換が可能である．

しかしながら，XTiSP は XPath のサポートに関して危惧される課題がある．図 2 の 2 行目に示されるようなノード集積は，入力を最後まで読まなければ出力が進まないため，ストリーム処理の弊害となる．これは，ストリーム処理器が，`title` 要素や `author` 要素が，`body` 要素の後に現れないことを知らないために起こる現象である．このため，XTiSP では，p_1 `before` p_2 という文法を用意しており，「XPath p_1 を満たすノードを，XPath p_2 を満たすノードが出現するまで集積する」というプログラムを実現している．今後は XML 型スキーマなどを参照するによって，こういった記述を省略できるように改良する予定である．

5.2 ストリーム処理の計算量

XTiSP で行われるストリーム処理に必要となる計算量は，属性の個数と入力される文書の深さに依存する．

属性の個数は有限であるため，各イベントを読む毎に，抽象化された変数の個数や保持する値の個数が増加することはないが，XTiSP では，一部の属性にスタック構造を許しているため，保持する値自体がスタック構造を持つ場合に入力される文書の深さに比例して増加する可能性がある．

また，属性の個数は，XPath の翻訳部分以外では，XTiSP に現れるプログラムの部分式の個数の約二倍だけ存在するが，入力イベント毎に全属性の評価をする必要はないため，必ずしもその個数が計算量にはさほど反映されない．ただし，XPath が前方参照を含むような場合には，[15] による除去法が必要になるが，その結果，最悪の場合，式の大きさが指数関数的に増大することがあるため，これによる計算量への影響は大きい．

6 おわりに

本稿では，新しい XML 変換言語 XTiSP を提案し，その言語で書かれたプログラムが属性文法に翻訳できることを利用して，ストリーム処理器を自動導出できることを示した．また，言語 XTiSP は現在実装中であり，完成次第，ベンチマークをとるなどして，有効性を確証していく予定である．

今回提案した XTiSP には，XDuce [10] などで議論されてるような XML 型スキーマに関しての情報には対応していないので，この点に関しては，将来課題として，研究を進めていきたい．また，他の XML 変換言語に対しても，本稿で提案したようなストリーム処理器の導出方法が適用できないかについても検討していきたい．

謝辞

本研究に関して，今回の発表を薦めてくださった武市正人先生をはじめとする e-Society プロジェクト PSD のメンバーに感謝します．また，貴重なご意見をくださった匿名の査読者の方々に謝意を表します．

参考文献

[1] XSL Transformations (XSLT) Version 2.0 W3C Working Draft. `http://www.w3.org/TR/xslt20/`, 2003.

[2] SAX 2.0: Simple API for XML `http://www.saxproject.org`, 2000.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *The VLDB Journal*, pages 53–64, 2000.

[5] V. Benzaken, G. Castagna, and A. Frisch. Cduce: An XML-Centric General-Purpose Language. In *Proceedings*

of the 8^{th} *ACM SIGPLAN ICFP*, Uppsala, 2003.

[6] J. Boyland and S. L. Graham. Composing tree attributions. In *Proceedings of the* 21^{th} *ACM SIGPLAN-SIGACT POPL*, Portland, 1994.

[7] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *ICDE*, 2002.

[8] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *ICDE*, 2002.

[9] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19 of *SIGPLAN Notices*, pages 157–170, June 1984.

[10] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2), May 2003.

[11] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[12] B. Ludäscher, P. Mukhopadhayn, and Y. Papakonstantinou. A transducer-based XML query processor. In *Proceedings of 28th VLDB, Hongkong*, August 2002.

[13] K. Nakano, Composing stack-attributed tree transducers, Tech. Rep. METR–2004–01, Department of Mathematical Informatics, University of Tokyo, Japan (2004). Composing stack-attirbuted tree transducers.

[14] K. Nakano and S. Nishimura. Deriving event-based document transformers from tree-based specifications. In *Workshop on Language Descriptions, Tools and Applications*, volume 44 of *ENTCS*. Elsevier Science, 2001.

[15] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proc.@ of the EDBT Workshop on XML Data Management (XMLDM)*, volume 2490 of *LNCS*, pages 109–127. Springer, 2002.

[16] P. Wadler. Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *LNCS*, pages 344–358. Springer Verlag, 1988.

[17] XP++: XML processing plus plus. `http://www.alphaworks.ibm.com/tech/`.

[18] XQuery 1.0: An XML query language. `http://www.w3.org/TR/xquery/`.

A XTiSP の文法

言語 XTiSP によるプログラムは，以下の文法における *xtisp* によって定義される．

xtisp = *valexp*
| `if` *valexp* `then` *xtisp* `else` *xtisp* `endif`
| `<`*valexp*`>[`*xtisp*`]` | *xtisp*`;`*xtisp*
| *xpath* [`before` *xpath*]
| `invite` *xpath* `do` *xtisp* `done`
| `invite*` *xpath* `do` *xtisp* `done`
| `visit` *xpath* `do` *xtisp* `done`
| `visit*` *xpath* `do` *E*[`children`] `done`

valexp = *string* | *boolean* | `value(`*xpath*`)`
| *f*(*valexp*, ⋯, *valexp*)

xtisp は，文字列や真偽値などを表す *valexp*，条件分岐を表す `if`，要素を生成する `< >[ ]`，結果を連結する `;`，XPath によりノードを集積する *xpath*，XPath により反復を行う `invite`[`*`] と `visit`[`*`] から構成される．それぞれの意味に関しては，第 2 節で述べた通りである．また，*xpath* には第 5 節で述べたような `before` 構文の使用が可能である．*valexp* には，操作するための様々な関数 *f* が用意されている．文字列や真偽値を操作する関数であれば，任意の関数をこの枠組みに加えることができる．また，関数 `value` は，タグ名や属性などの文字列を取り出すために用いる．*string* や *boolean* は，文字列定数，真偽値定数に相当する．