関数内包型 XML データ処理系 NODP の開発

原 忠司†‡　森下 淳也†　大月 一弘†　清光 英成†　織田 美樹男‡　榊原 淳‡

†神戸大学 大学院総合人間科学研究科 〒657-8501 神戸市灘区鶴甲 1-2-1

‡株式会社 メディアフュージョン 〒530-0004 大阪市北区堂島浜 2-1-29 古川大阪ビル

E-mail: †045F006F@y04.kobe-u.ac.jp, {jm, ohtsuki, kiyomitu }@kobe-u.ac.jp,
‡{hara, oda, sakaki}@mediafusion.co.jp

あらまし XML を簡便に扱うことを目的とした新たな処理系であるノードプロセッサ(NODP)を開発中である. NODP は XML 自体で記述されエレメントを基本的なデータ型として扱い，エレメント名をオペランドと見立てた前置記法を採用した言語である. 現在の一般的な XML インターフェースである DOM や SAX などとは異なり XML データ自身に対してコードを添付することにより, XML のみで完結した記述をオブジェクトとして扱えるようにすることが可能となる.

キーワード XML，DB 言語，ユーザインタフェース

1. はじめに

今日のシステム開発は開発サイクルの短縮およびシステム要件の修正要求の多様化などから，データ構造が頻繁に更新される．この場合，旧データ構造によるデータを移行させることを行う．しかし旧データが多く新データ構造への移行が困難な場合，新旧のデータ構造を混在させる対応をとる場合も多い．この場合プログラム内にそれぞれのバージョンによるデータアクセスを保持しておく必要があり，データとデータアクセスプログラムの関連付けを行う必要がある．このためデータ構造の更新に伴うプログラム修正工数が増大し迅速な対応が困難となる問題を生じる.

データ構造の更新作業については XML(eXtensible Markup Language)[1]でデータを表現し，ウェルフォームドに対応した XML データベースなどを利用することで簡略化が期待できる．しかしながら，データを実際に扱うプログラム側では依然として修正が必要であり，工数の削減が期待するほど実現できていない.

プログラムから XML を扱うためのインターフェースは W3C(World Wide Web Consortium)から勧告されている DOM(Document Object Mode)[2]や SAX(Simple Api for Xml)[3]などがある. これらは XML 文法のモデル化であり，一般化されたモデルによって XML にアクセスするため，あらゆる形の XML 文書でも操作できる利点がありながらも，構造に依存したアクセスになるためデータの構造更新の際にはプログラム側の修正も必要となる.

これに対してスキーマ言語で厳密に定義されたデータ構造に対するインターフェースを生成するためのものとして Relaxer[4]がある．データ構造に対するインターフェースを生成するため，プログラム側でデータ構造に依存するようなことは無いが，複数バージョンのデータ構造が混在する場合には，やはりデータとインターフェースの関連付けを行う必要がある.

この問題を解決するために, XML で表現されたデータに対してアクセサとなるプログラムを埋め込み，XML 文書をオブジェクトして扱うことができれば有用と考えた．そこで我々はこれらを可能とする XML データ処理系として NODP(NODe Processor)を提案する.

NODP では処理を XML 文書内に記述することによってデータの可搬性と独立性を高めている．これ対して，プログラムからの統合化されたアクセス手段を提供し，XML 文書をオブジェクト化させる．この仕組みによって，複数バージョンのデータ構造について単一のインターフェースを持つデータアクセス処理を記述することで，プログラムから単一の方法でデータへのアクセスが可能としてる．さらに，データに対して処理が埋め込まれていることによって，データの意味をより具体的に読み取りやすいものとしている.

プログラムを埋め込んだ XML データの処理系としては, 横浜ベイキットによる Xi(eXtensible It)[5]がある. Xi が XML 文書から単独のプログラムによって処理結果となる新たな XML 文書を生成するのに適した言語となっているのに対し，NODP では XML 文書自体をオブジェクト化させることによって，既存のプログラム言語からのアクセスの簡便化を図ることを目的とするため，本質的な設計方針が異なると考える.

2. NODP の設計方針

NODP は次のような設計方針に基づいている

1) データの可搬性と独立性を重視する
2) プログラムからのアクセスを統合化する
3) XML に対してシームレスに融合する

1)は，処理を XML 文章に直接記述し，XML 文書を一つのオブジェクトと見なすことにより，データの可搬性と独立性を高める．これにより，特定のデータ構造に依存した処理をデータ内に記述することが可能となる．

これに対して統合化されたアクセス方法を提供することで, XML 文書内に記述された処理に対してさまざまなプログラムからのアクセスを可能とする 2)の方針を採用した．これによりデータを利用するプログラムはデータアクセスのためのコードを記述せずとも良くなる．また 1)の方針と合わせて複数バージョンのデータ構造が混在する状況であってもデータ構造とアクセス方法の関連付けを意識する必要がなくなる．つまり，プログラム側からの視点では動的に型付けされたデータに対するアクセスという感覚で操作できることになる．さらに複数のシステム間でデータ連絡を行う場合などに，それぞれのシステムに対する共通化されたデータアクセス方法を提供することができ，データの可搬性はさらに高まると考えられる．

データに対する処理を XML 文書に埋め込むことで, XML データの可読性が低下する危険性が生じる. これを回避するために処理の記述は XML にうまく融合し, データ構造への影響が少ない状態でなされる必要がある. このため 3)の方針を採用した. XML のデータ構造への影響をおさえ，さらにデータに対する処理が記述されることで，データの意味を汲み取りやすい表現となるようにする．

これらの設計方針を元に言語を設計することで，XML で表現されたデータへのアクセスが簡略化でき，データの可搬性を高める仕組みができると考える．

次の章では，この設計方針を元に本研究で設計・実装中である NODP の構造を説明していく．

3. NODP の構造

3.1. 言語概要

NODP 文法の基本的な考え方は, XML のエレメントノードを 1 つの S 式としてみなすことにある．Lisp などで S 式はネストしたリストとして表現されているが, これは視点を変えればツリー構造ととらえることもできる．NODP 文法はエレメント名をオペランドとして認識する前置記法言語として記述され，エレメントノードが保持する子エレメントノードはオペランドに対する引数リストとみなし，これらを末尾再帰的に評価することにより目的とする結果を取り出すように設計されている．

言い方を変えれば，NODP の文法は XML の表記方法に合わせた Lisp の方言の実装と言えなくもないが, 基本的なデータ型としてエレメントノードを扱うようにしているなど, XML を操作するための言語として特化してある部分もあり，その意味では新たな言語と言える．

NODP が想定している使われ方は, XML で表現されたデータに対して，振る舞いとなるロジックを添付することにより, XML 文書をオブジェクトとして扱えるようにすることである．ここで言う振る舞いは単純に他のプログラム言語から, XML データの部分にアクセスするためのアクセサでもかまわないし，任意のビジネスロジックを記述し，それだけで単体のプログラムとして機能するものでもかまわない．なんらかのロジックを介して結果としてエレメントノードを返す物を想定している．ただし，NODP が扱う XML として想定しているのはデータ構造の表現形式としての XML であり，マークアップされた文書としての XML については今回あまり考えていない．このため，ミックスドコンテントなどのように, XML でデータを表現するための手段として冗長と感じられる要素については今回検討を見送っている．

3.2. 式

NODP はエレメント名をオペランドと見なし，エレメントノード自体を式とみなす一種の前置記法を採用している．NODP 自体はエレメントノードを操作する言語であるので，NODP で記述されたプログラム自身をデータと見なし NODP 自体を特殊化させる言語を構築することができる．

オペランドは，名前空間 nodp を持つエレメント名として記述される．

```
<nodp:OPERAND>
  演算対象
</nodp:OPERAND>
```

この例では XML のネームスペース宣言は省略している．式はネストして記述することも可能である．

```
<nodp:OPERAND-1>
  <nodp:OPERAND-2>
    <nodp:OPERAND-3/>
  </nodp:OPERAND-2>
```

</nodp:OPERAND-1>

この場合，オペランドは末尾再帰的に評価されていくことになる．この例の場合，まず最初に OPERAND-3 が評価され，その結果を持って OPERAND-2 の評価を行う．同様に OPERAND-2 の結果でもって OPERAND-1 の評価が行われこれが最終的な結果として返されることになる．

nodp 以外の名前空間を持ったエレメントノードも式として扱う．この場合，式の評価結果として返される値は自身のエレメント自体である．

```
<foo>data</foo>

  →<foo>data</foo>
```

ただし，この場合も，エレメントノードに属する子エレメントノードも全て末尾再帰的に評価されるため，nodp の名前空間を持たないエレメントノードの子孫エレメントノードに名前空間 nodp に属するエレメントノードが存在した場合，NODP のプログラムとして評価され，その結果が当該エレメントノードに置き換えられることになる．

```
<hoge>
  <nodp:add>
    <foo>100</foo>
    <bar>200</bar>
  </nodp:add>
<hoge>

  →<hoge>
      <foo>300</foo>
    </hoge>
```

3.3. データ型

NODP で扱うデータは XML の各構文要素を構成するものとなる．ただし，言語全体においての一貫性を保つため，全ての構文要素を言語でプリミティブに扱うデータ型とすることはしない．

NODP でプリミティブに扱うデータ型として以下のものがある．

- アトム
- リスト
- 真偽値
- プロシージャ

これらのものを以下に定義していく

3.3.1. アトム

テキストノードを 1 つのみ持つエレメントノードを最も基本的なデータ型として扱う．これにはテキストノードを含む子ノードを持たないエレメントノードも含まれる．具体的には次のようなものとなる．

```
<foo>data</foo>
```

これをアトムと呼ぶ．NODP ではこれを最小のデータ構造として扱うため，テキストノードを直接的に扱うことはしない．プログラムから特定のテキストノードを操作する必要がある場合などがあると思われるが，特定のプロシージャにおける特異なパターンとし，NODP でプリミティブに扱うデータ構造として扱うことはしない．同様の理由でアトリビュートも直接扱うことはしない．

一般的な言語では文字列型や数値型などのデータ型が存在するが，NODP ではアトムがこれの代わりとなる．つまり演算の種類に応じてアトムが保持するデータを適時変換して扱うことになる．これについては XPath におけるデータの扱いに準じた設計となっている．数値演算を行う場合には数値に変換し，文字列演算を行う場合には文字列として扱う．この際に何らかの理由で変換が成功しなかった場合にはエラーとして処理することになる．またテキストノードを含まないアトムの場合，その値を NULL と見なす．

3.3.2. リスト

アトムであるテキストノードを 1 つのみ持つエレメントノードを複数保持したエレメントノードを，NODP ではリストとして扱う．具体的には次のようなものとなる．

```
<hoge>
  <foo>data</foo>
  <bar>100</bar>
</hoge>
```

また，リストはネストすることもできる

```
<geho>
  <hoge>
    <foo>data1</foo>
    <bar>100</bar>
  </hoge>
  <hoge>
    <foo>data2</foo>
```

```
    <bar>200</bar>
  </hoge>
</geho>
```

つまり，NODP では XML のツリー構造をネストしたリスト構造として扱うことになる．

3.3.3. 真偽値

条件演算の結果などの真偽値もプリミティブなデータ型として扱う．ただし，NODP では全てのデータ型はエレメントノードとして扱う決まりとしたため，これに対するエレメント表現が必要となる．真偽値のそれぞれの実体は実行環境上にスタティックに生成されたオブジェクトとして存在するが，それに対する参照という形で名前空間 nodp を持つオペランドとして表現することになる．つまり

```
<nodp:true/>
<nodp:false/>
```

と表現される．このことから真偽値はあくまで条件演算の結果の評価という形でしか利用できず，他のデータ型と見なして演算した場合，アトムの評価の仕方に習ってその値は NULL となる．

3.3.4. プロシージャ

NODP では任意の式の集まりに対して単一のオブジェクトとして扱うことを可能としている．このオブジェクトをプロシージャという．プロシージャは任意の個数の引数を取る一つ以上の式の集まりとなる．

プロシージャはユーザが作成するプログラム上において lambda 式によって実行時に生成することができる．また，NODP のプリミティブなオペランドについてもその実行実体はプロシージャとして実装される．

Lamda 関数などでプロシージャが生成される以上これもプログラムから直接操作できる必要があるため，これを基本的なデータ型として扱えるようにする．

これについても，NODP ではデータはエレメントノードとして表現可能である必要があるため，プロシージャについても同様に名前空間 nodp を持つオペランドとして表現することにする．

```
<nodp:procedure reference=”REFERENCE”/>
```

reference アトリビュートには，NODP 実行系でプロシージャオブジェクトを特定できる識別子が設定されることになる．この識別子は関数名やオペランド名とは異なり、言語処理系内部でのオブジェクト実体の参照となっていることに注意する必要がある。

プロシージャはこの形式で表現可能であるが，これは外部表現としての表記であり，これに対してなんらかの演算を行うことは基本的にできない．これをアトムとして扱う場合，その値は NULL となる．

3.4. 変数

NODP において変数は名前空間 nodp を持つ var という識別子に関連付けられたプロシージャとして存在する．このプロシージャは変数に関する挙動を司り，言語全体における挙動の一貫性を保っている．変数は次のように記述される．

```
<nodp:var name=”変数名”/>
```

または

```
<nodp:var name=”変数名”>
  データ
</nodp:var>
```

1 つ目の例は変数の参照を表し，2 つ目の例は変数に対する代入を表している．var は変数の挙動を司るプロシージャであるため，これとは別に変数自体を特定するためにアトリビュート name によって識別子を指定している．

変数はプロシージャにおいてローカルな名前空間を持ち，プロシージャが関連付けられる関数名などの識別子とは別個に管理される．

変数は使用に際して宣言などは必要ないが，データの実体が割り付けられていない変数を参照した場合にはエラーとなる．

NODP において変数はデータ型を持たない．データ型は値自体が保持する動的な型システムを採用している．このため，変数には先に定義した NODP で扱えるデータ型全てを割り当てることができる．

変数に対する代入を行う式において，変数に割り付けられるデータ部分に式を指定することもできる．この場合やはり末尾再帰的に処理されることになり，実際に変数に割り付けられる値は式を評価した結果となる．

3.5. lambda

lambda はプロシージャを生成するための関数である．lambda は 2 つの引数を取り，1 つは生成するプロシージャに対する仮引数リストであり，もう 1 つはプロシージャが実行すべき式のあつまりになる．

```
<nodp:lambda>
  <nodp:arg>
    仮引数リスト
  </nodp:arg>
  <nodp:begin>
    式
  <nodp:begin>
</nodp:lambda
```

関数 arg は仮引数リストに関する処理を司るプロシージャがバインドされた関数である．仮引数リストにはプロシージャの実行すべき式の中で，引数を操作するための変数が指定される．

```
<nodp:arg>
  <nodp:var name=”arg1”/>
  <nodp:var name=”arg2”>
    <foo>100</foo>
  </nodp:var>
</nodp:arg>
```

変数 arg2 には代入式が記述されているが，これはデフォルト引数となる．プロシージャが実行される前に，プロシージャに引き渡された引数を順にこれらの変数に割り付けられていき，これらの変数が存在するローカル環境を構成し，この環境下でプロシージャを実行することによって引数の受け渡しを実現する．

プロシージャが実行すべき処理は関数 begin はブロックを表し，この中にプロシージャに必要な処理を記述する．

Lambda によって生成されたプロシージャは，その時点で実行可能なオブジェクトとして存在しているため，関数 call を利用することでこれを起動させることができる．

```
<nodp:call>
  <nodp:lambda/>
  引数リスト
</nodp:call>
```

もしくは

```
<nodp:var name=”func”>
  <nodp:lambda/>
</nodp:var>
<nodp:call>
  <nodp:var name=”func”/>
  引数リスト
</nodp:call>
```

引数リストにはプロシージャに引き渡す引数を記述する．このように，lambda で生成したプロシージャを直接起動したり，一旦変数にバインドされたものを起動することもできる．

3.6. 関数

lambda によって生成されたプロシージャは関数 call によって起動することもできるが，あたかも NODP のプリミティブな関数のように起動させるようにすることもできる．この仕組みを関数と呼ぶ．

関数はプロシージャに対してグローバルなスコープでユニークな識別子を割り付け，割り付けられた以降のプログラム全域からアクセス可能とする．

関数は次のように記述される．

```
<nodp:defun name=”関数名”>
  プロシージャ
</nodp:defun>
```

defun はアトリビュート name によって示された識別子に対して引き渡されたプロシージャを割り付ける．NODP のプリミティブな関数についても，内部的には defun と同様の仕組みによって特定の識別子への割り付けを行っている．このためユーザは必要があれば，defun によって既存の識別子に対して任意の処理を行うプロシージャを割り付けることにより，NODP 自体の挙動を変更することもできる．

4. 評価

現在，一般的にプログラム言語から XML を操作するために使用されるインターフェースは DOM である．NODP の言語について，XML データの操作が簡便に扱えるようになったかどうかを評価するため，同じ目的のプログラムを DOM を使って作成した場合と，NODP を利用した場合との比較で行う．

評価に際して使用した環境は NODP については本研究で実装を行っている NODP 処理系を用いた．DOM についてはプログラム言語には Ruby[5]を，DOM には Ruby に標準で添付されている REXML[6]を利用した．

プログラムを開発する際のインターフェースを決定する上で重要なファクターとして，実行効率のよさもあげられるが，NODP の実装は現段階で実験的なものであり，ベンチマークを取る段階に至っていないため，今回はこれを無視する．

4.1. ケース 1

XML で表現されたデータについての演算を行う. 例で使用する XML はある個人の成績についてのデータが格納されているとし，次のような構造と仮定する.

```
<root>
  <Person>
    <math>80</math>
    <english>60</english>
    <science>70</science>
  </Person>
</root>
```

ここから合計点を算出することを目的とする.

4.1.1. DOM による記述

成績が格納されている個々のエレメントを取得し，これらの値を数値に変換した上で加算する処理を記述することで，目的を実現することができる．処理対象の XML は変数 target_xml に格納されているものとすると，次のようなコードとなる.

```
root = REXML::Document.new(target_xml)
Person = root.elements[“Person”]
summary = 0
Person.elements.each{|course|
  summary += course.get_text.value.to_i
}
```

4.1.2. NODP による記述

NODP ではエレメントノードを直接演算することが可能であるため，成績を格納しているエレメントを加算関数に引き渡すことによって目的を実現することができる.

```
<root>
  <Person>
    <nodp:add>
      <math>80</math>
      <english>60</english>
      <science>70</science>
    </nodp:add>
  </Person>
</root>
```

4.1.3. ケース 1 の評価

NODP によるコードの方が簡便なコードと考えることができる．気になる点としては，NODP のコードが記述されることにより，元のデータ構造が変わってしまっていることが上げられる．これを避けるためには加算を行うロジックを関数として外に出してしまうことが考えられる.

しかし，エレメントノードに対する処理を記述する上で，NODP は DOM に比べてシンプルな記述ですむと言える.

4.2. ケース 2

XML で表現された任意の部分のデータを取得することを目的とする．例で使用する XML データ構造は次のものとする.

```
<root>
  <Person>
    <name>原 忠司</name>
    <address>神戸市灘区鶴甲 1-2-1</address>
  </Person>
</root>
```

このデータから名前を取得する.

4.2.1. DOM による記述

REXML は Level-3 に準拠しているため，DOM 上で XPath によるアクセスを記述することが可能である．よって，パースによって生成されたドキュメントノードに対して必要とするノードのパスで指定した XPath を記述することで目的を実現することができる.

処理対象の XML は変数 target_xml に格納されているものとすると，次のようなコードとなる.

```
root = REXML::Document.new(target_xml)
name = root.elements[“Person/name”]
```

4.2.2. NODP による記述

NODP でも XPath によってエレメントノードを取り出す関数が存在する．関数 XPath は引数としてパス式を値として持つアトムをとり，指定されたパス式を評価した結果エレメントを返す．これらのことを行う関数を処理対象の XML に記述することにより，目的を実現することが可能となる．今回は元のデータ構造からの変更を抑えるため，ロジック自体を関数化して記述することにした.

```
<root>
  <program>
    <nodp:defun name=”get_name”>
      <nodp:lambda>
```

```
        <nodp:arg/>
        <nodp:begin>
          <nodp:xpath>
            <foo>/root/Person/name</foo>
          </nodp:xpath>
        </nodp:begin>
      <nodp:lambda>
    </nodp:defun>
  <program>
  <Person>
    <name>原 忠司</name>
    <address>神戸市灘区鶴甲 1-2-1</address>
  </Person>
</root>
```

これに対して，name エレメントを必要とするプログラムから次の関数を呼び出せばよいことになる．

```
<nodp:get_name/>
```

これによって目的とするエレメントを取得することができる．

4.2.3. ケース 2 の評価

関数化するためのコードが冗長なため，これを見る限り DOM を利用したコードの方が簡便なコードとなっているが，本質的な部分においては同等の処理が行われていることが分かる．このことから双方の間での差異はない，もしくは DOM の方が簡便なコードと考えられる.しかしながら,プログラムが完成した後に，データ構造が次のように変更なった場合 NODP の利点を見ることができる．

```
<root>
  <Person>
    <name>
      <first>忠司</first>
      <last>原</last>
    </name>
    <address>神戸市灘区鶴甲 1-2-1</address>
  </Person>
</root>
```

プログラムが完成した後にデータ構造の変更は実際のシステム開発において頻繁に起こることである．このとき，DOM を利用したプログラムの場合，プログラム側の修正が必須となる．今回の場合，2 つに分かれたエレメントからそれぞれデータを抜き出し，つなげ合わせることが必要となる．

```
root = REXML::Document.new(target_xml)
first = root.elements[“person/name/first/text()”]
last = root.elements[“person/name/last/text()”]
name = last + first
```

NODP においても同様の修正が必要となるが，XML データ上のプログラムの記述を変更することで対応することが可能となる．

```
<root>
  <program>
    <nodp:defun name=”get_name”>
      <nodp:arg/>
      <nodp:begin>
        <nodp:string-concat>
          <nodp:xpath>
            <foo>/root/Person/name/last</foo>
          </nodp:xpath>
          <nodp:xpath>
            <foo>/root/Person/name/first</foo>
          </nodp:xpath>
        </nodp:string-concat>
      </nodp:begin>
    </nodp:defun>
  <program>
  <Person>
    <name>
      <first>忠司</first>
      <last>原</last>
    </name>
    <address>神戸市灘区鶴甲 1-2-1</address>
  </Person>
</root>
```

一方, name エレメントを必要とするプログラム側では何も修正は必要ない．これが有効に機能するケースとしては，過去データなど複数バージョンのデータ構造が平行して存在する場合などに, 有利となる. NODP では XML のデータに対して NODP プログラムによってアクセサを記述することにより，処理対象の XML データをあたかもオブジェクトのように扱うことができる．このアクセサは XML データと結合して存在しているため，複数のバージョンのデータ構造が平行して存在したとしても，アクセサのインターフェースが変更にならない限り，データを必要とするユーザプログラム側の修正が必要なくなり，またバージョン間の

差異も考慮する必要がなくなる.

また，アクセサはデータ構造に対してのみ依存しているため，NODP でのアクセサの記述はデータ構造の設計の際にまとめて行うことができるため，多少の記述性の冗長化は無視できる要素であるとも考えられる.

5. まとめ

本稿では関数内包型 XML データ処理系である NODP について述べた.

現状の NODP は言語の一貫性，および論理の妥当性を検証するために，記述が冗長になる傾向がある．これはプログラムを記述する際の容易性を鑑みた場合，まだまだ目的を達成できているとは言えない.

だが, XML が仮想的にオブジェクトとして扱えるようになることで，メリットがあるケースが想定できたことで，NODP の言語仕様を設計する上で，方針を決定する際の指針が見え始めたことも事実である.

また，例えば XML データベースなどに組み込むことにより，アクセサとなる関数の結果に対してインデックスを張るなどことでより効果的なインデックス設計に利用できる可能性や，Web 開発などで頻発する仕様変更の多さに容易に対応できるシステムの設計など，さまざまな利用シーンが考えられる.

現状における大きな問題点と言える，記述の冗長性についても，さらに利用される状況を具体的に想定できるようになれば，それに合わせたシンタックスシュガーを導入するなど，解決のめどをつけることも可能である.

NODP はその設計に際して Lisp の方言である Scheme[7]を参考にしている．しかしながら今回は XML を効率的に操作するための表現方法の妥当性の検証を主目的としており，Lisp の全ての機能を XML で表現することではないため, Scheme の全ての機能を再現していない．あくまで構文の表現方法や式の解釈などの参考程度にとどまる．しかしながら，Scheme は優れた設計であり，本言語も最終的にはこれと同等の機能を持たせ，さらには Scheme プログラムから NODP プログラムへのコンバート等も視野に入れた言語設計を行う予定である．その上で XML のエレメントを処理する上でのルールを可能な限り一般化させながら言語仕様を拡張していき，機能の充実を図りながら，より具体的な利用のされ方を想定し，それに合わせたシンタックスの検討を行っていくことで, XML 操作言語としてより実用的なものにしていくことを考えている.

文　献

[1] W3C，“Extensible Markup Language(XML)1.0(Third Edition)”，http://www.w3.org/TR/REC-xml/，February 2004

[2] W3C DOM IG，“Document Object Model(DOM)”，http://www.w3.org/DOM/，January 2005

[3] David Brownell，SAX2，O’Reilly，January 2002

[4] Tomoharu Asami，“Relaxer Reference Manual”，http://www.relaxer.org/doc/refman/1.0/html/refman_en.html，December 2003

[5] 横浜ベイキット，“Extend it(Xi) Version 1.1.1 仕様書(改訂版)“，http://dock.baykit.org/xi13?Spec&l=jp，December 2004

[6] まつもとゆきひろ，オブジェクト指向スクリプト言語 Ruby，http://www.ruby-lang.org/ja/

[7] Sean Russell，REXML，http://www.germane-software.com/software/rexml/

[8] Marc feeley，Univesite de Montreal，“The Revised R6RS Status Report”，http://www.schemers.org/Documents/Standards/Charter/2004-10-13.pdf，October 2004