修士学位論文

題目

記述言語 XBRL で定義された財務諸表を
計算・書式変換する言語処理系の提案と実現

指導教官
井上 克郎 教授

報告者
高尾 祐治

平成 16 年 2 月 12 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 15 年度 修士学位論文

記述言語 XBRL で定義された財務諸表を計算・書式変換する言語処理系の提案と実現

高尾 祐治

内容梗概

近年，インターネットを通じて財務諸表を公開し，決算公告を行う企業が増えてきている．企業等における財務処理を効率的に行うため，コンピュータ上で財務情報を記述するための標準的な形式である XBRL(eXtensible Business Reporting Language) の普及が進んでいる．XBRL 文書は様々な形式の文書に変換することが可能である．正確，かつ，迅速に，多様な要求に応えるための財務諸表を作成できるため，XBRL 文書を変換するという需要は非常に高い．

XBRL 文書に対する複雑な変換処理を効率的に行うには，一般的に既存のプログラミング言語によって記述されたプログラムが用いられている．しかし，既存のプログラミング言語には XBRL の持つ複雑な構造を容易に扱う手段が存在しないため，変換を行うプログラムが複雑になる傾向がある．一般に，財務諸表の編集に携わる人にプログラミングの経験が豊富にあることは珍しい．このとき，XBRL 文書を変換するために複雑なプログラミングを要求することは現実的ではない．

そこで本研究では，まず XBRL 文書処理を容易に行うためのモデルを提案する．本モデルは，XBRL 文書に記述された要素間の様々な関係や，XBRL 文書の特徴を利用し，XML のモデルとして一般的な DOM を簡略化したものである．次に，定義したモデルに基づくプログラミング言語 LMX を定義し，LMX のインタプリタと，LMX 用開発環境を実装の実装を行った．LMX は容易に習得，利用できることを目的とした構造化プログラミング言語であり，XBRL 文書のモデルを言語機能に組み込むことで，XBRL 文書の簡単な読み書きを可能とする．

既存の XML 処理系と LMX との比較検討の結果，LMX はより簡単なプログラムを記述できることが確認した．LMX を利用することにより，XBRL 文書の変換を効率よく行うことが期待できる．

主な用語

XBRL
プログラミング言語 (Programing Language)
言語処理系 (Interpreter)

目 次

1 まえがき

近年，インターネットを通じて財務諸表を公開し，決算公告を行う企業が増えてきている．しかし，コンピュータ上で財務情報を記述するための標準的な形式が存在しなかったため，従来紙媒体で行われていた財務諸表を，そのまま HTML 形式や PDF 形式など独自に変換したものを交換していた．一般に，財務諸表で使用される用語や意味は企業ごとに異なっているため，財務状態の分析を行う際，データの整合性の確認や再入力などを人の手で行う必要がある．しかし，財務諸表が電子媒体で公開されているにもかかわらず，この種の負担は軽減されておらず，非常に非効率であるといった問題があった．そこで，財務情報を記述するための標準として，XML[14] の構文を用いて財務情報を記述する言語である，XBRL(eXtensible Business Reporting Language) が XBRL International[19] によって策定された．

XBRL を用いて財務情報を記述することにより，XBRL 文書の作成から利用まで途中に人手を介することがなくなるために，財務情報の流通が大いに効率化される．また，XBRL 文書は，様々な形式に変換することが可能なため，監査機関や金融機関への提出文書の作成効率が格段に向上する．正確，かつ，迅速に多様な要求に応えるための財務諸表を作成できるため，XBRL 文書を変換するという需要は非常に高い [23].

私の所属する研究グループでは，XBRL 文書の変換を効率よく行う研究を行ってきた．[26, 27] では，XBRL 文書に記された項目間の対応関係や，値の計算式を与えることで，異なる形式の XBRL 文書を自動生成する研究を行った．その結果，XBRL 文書の変換を効率よく正確に行うことができることを確かめたが，より柔軟で複雑な変換を行うためには，プログラミング言語を使い変換処理を記述したほうがより効率的であることが分かった．

一方，XML 文書を対象としたプログラミング言語や XML 文書の相互変換に関する研究が広く行われている．例えば，XSLT[18] は制御構造や算術演算をサポートし，XBRL 文書の変換に利用することができる．しかし，XSLT には XBRL の持つ複雑な構造を簡単に扱う手段がないために，本来行いたい変換処理のプログラムに加え，XBRL 文書の構造記述したタクソノミ文書を解析するためのプログラムも書かなければならない．その上，XBRL 文書変換のために設計されたプログラミング言語ではないため，XBRL 変換において行われる処理を記述すると，プログラムが複雑になる傾向がある．また，最近注目されている XML 変換言語に XDuce[7] がある．XDuce は XML のスキーマをデータ型と見なし，生成する XML 文書の妥当性を静的に確かめる．XDuce も XSLT と同じ問題を含み，XBRL 文書のためのプログラムは複雑で難しいものとなる．一般に，財務諸表の編集に携わる人にプログラミングの経験が豊富にあることは珍しい．このとき，XBRL 文書を変換するために複雑なプログラミングを要求することは現実的ではない．

そこで本研究では，まず，XBRL 文書処理を容易に行うためのモデルを提案する．XBRL 文書には，XML 文書処理で考慮する入れ子構造による親子関係の他に，財務諸表に現れる科目間の関係に対応する，要素間の様々な関係があるという特徴がある．また，XBRL は財務情報を記述する言語であるため，一般の XML 文書と異なり財務情報を定義する要素とその値は一対一に対応するという特徴がある．XBRL 文書のモデルはこれらの特徴を利用し，XML のモデルとして一般的な DOM を簡略化したものである．

次に，定義したモデルに基づくプログラミング言語 LMX を定義し，その処理系の実装を行う．LMX は容易に習得，利用できることを目的とした構造化プログラミング言語であり，XBRL 文書のモデルを言語機能に組み込むことで，XBRL 文書への簡単なアクセスを可能としている．LMX では，XBRL 文書のモデルを DOM のラッパーとして実装するため，DOM で定義された一般の XML 文書処理も行うことができる．また，処理系では XBRL 文書処理特有の関数や入出力を行う関数を提供する．LMX とこれらの関数を利用することで，XBRL 文書の変換を効率よく行うことが期待できる．

以降，第 2 章では財務諸表とその記述言語である XBRL について説明する．第 3 章では XBRL で記述された情報を簡易に扱うための XBRL のモデルについて説明する．第 4 章では XBRL 文書処理のためのプログラミング言語の定義を行い，第 5 章では実装について説明する．第 6 章では，実際の XBRL 文書の変換事例を使い既存の処理系との比較検討を行う．第 7 章では本論文のまとめと今後の課題について述べる．

2　財務情報記述言語 XBRL

財務情報を記述するための言語である XBRL が策定されている．XBRL を利用することで，ネットワークを通じた財務情報の交換や，財務情報から財務諸表への自動整形が可能となる．本節では，XBRL の前提となる財務情報と，XBRL の歴史と現状，そして，XBRL で定義された種々の文書について簡単に説明する．

2.1　財務情報と財務諸表

企業や団体は自らの財務情報を，貸借対照表，損益計算書，キャッシュフロー計算書といった財務諸表にまとめて定期的に公表する．投資機関や銀行は，財務諸表に記述された情報を使い企業の安全性を判断し，投資を行うなど，財務諸表は経済活動にとって極めて重要な文書して扱われる．例として，商法による貸借対照表の例を図 1 に示す．貸借対照表とは，資産，負債，および，資本を対照表示することにより，企業の財政状態を明らかにする報告書である．

株式会社○○○　貸借対照表

（単位:百万円）

科目	金額	科目	金額
＜資産の部＞	8,592,822	＜負債の部＞	2,889,500
流動資産	3,620,881	流動負債	2,040,821
現預金及び有価証券	1,487,544	買掛債務	765,041
売掛債権	919,468	社債及び短期借入金	50,000
商品・製品	140,516	その他	1,225,780
その他	505,277	固定負債	848,679
固定資産	4,971,941	社債及び長期借入金	500,600
有形固定資産	1,269,042	その他	348,079
無形固定資産	1,242,883	＜資本の部＞	5,703,322
投資その他の資産	2,460,016	資本金	397,049
		資本剰余金	416,970
		利益剰余金	5,287,602
		その他	△398299
資産合計	8,592,822	負債及び資本合計	8,592,822

図 1: 貸借対照表の例

2.2　XBRL

ネットワーク環境の発展に伴い，迅速かつ効率的に財務情報を開示するため財務諸表が web 上で公開されるようになった．代表的な例として EDINET[24] が挙げられる．EDINET（Electronic Disclosure for Investors’ NETwork）とは，証券取引法に基づく有価証券報告書等の開示書類に関する電子開示システムである．EDINET は，金融庁より行政サービスの一環

として提供され，EDINET システムに提出された開示書類について，インターネット上においても閲覧を可能とするものである．

EDINET の他にも，企業が決算公告に用いた財務諸表を PDF 形式に変換し，インターネット上で公開している事例が多く見受けられる．しかし，これらの財務諸表の形式が統一されていないために，企業により使用する用語や，その意味が異なっていたり，異なる企業間の財務報告書を比較しづらいといった問題があった．これらの問題を解決するため，2000 年 7 月に XBRL 1.0[20] が XBRL International により策定された．XBRL とは，財務・経営・投資など，様々な用途に使用する情報を記述できる XML の構文を用いた言語である [23]. XBRL は，財務諸表や内部会計報告など，組織における財務情報の記述に特に適している．

XBRL には次のような利点がある [25]. 財務諸表を作成する立場の人は，財務情報を XBRL で一元管理することにより，様々な形式の財務諸表に変換ができ，各機関への提出文書の作成効率が格段に向上する．また，利用する立場の人は，利用しやすい形式に簡単に変換することができ，転記や再入力で発生するミスがなくなり，情報の分析作業に集中することができる．

我々が提案する処理系は，2001 年 12 月に公開された XBRL 2.0[20] を対象としている．以下では，XBRL 2.0 について簡単に説明する．

2.3 XBRL 文書の構成

XBRL 文書はインスタンス文書とタクソノミ文書から構成される．タクソノミ文書はタクソノミ本体 (以下，タクソノミ）と 5 種類のリンクベースから構成され，XBRL 文書の語彙と，財務報告書に記される項目間の関係を定義する．インスタンス文書にはタクソノミ文書によって定義された語彙で財務事実が記述される．図 2 に，XBRL 文書の構成を示す.

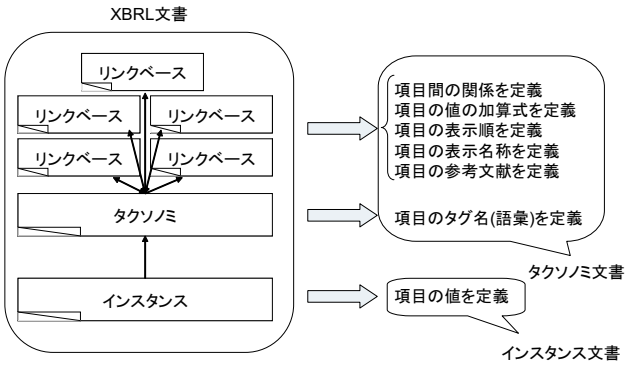

図 2: XBRL 文書の構成

2.4 インスタンス文書

インスタンス文書は，タクソノミ文書で定義された要素で財務事実を記述した XML 文書である．インスタンス文書は次の要素から構成される．

2.4.1 グループ要素

グループ要素 (group) は，インスタンス文書のルート要素である．他の要素を含む汎用のコンテナとして使うこともできる．

2.4.2 項目要素

財務事実，すなわち，所定のビジネス実体に対して，所定の期間内に報告された事実を記述する要素である．要素名はタクソノミで定義される．財務事実が数値型である項目要素には numericContext 属性が，文字列型である項目要素には nonNumericContext 属性が必ず含まれ，次で説明するコンテキスト要素への参照を指定する．

2.4.3 コンテキスト要素

コンテキスト要素は，数値コンテキスト要素 (numericContext)，非数値コンテキスト要素 (nonNumericContext) の 2 種類がある．数値コンテキストは，数値として記述された財務事実に関するメタデータや属性（期間や単位など）を提供する．非数値コンテキストは，文章として記述された財務事実に関して同様の情報を提供する．

2.4.4 インスタンス文書の構造

インスタンス文書の構造を規定するスキーマでは，項目要素が子要素を含むことを許していない．グループ要素を使って要素が複雑にネストした構造を書くことができるが，一般的にインスタンス文書は，項目要素が単純に列挙された XML 文書となっている．図 3 に，ごく簡単なインスタンス文書の例を示す．図 3 では，6 個の項目要素により財務事実が記述されている．これらの項目要素は，`c1` という id によって数値コンテキストに関係付けられている．

2.5 タクソノミ

タクソノミは，インスタンス文書の語彙 (要素名，属性，データ型) を定義する XML スキーマ [17] 文書である．タクソノミでは，インスタンス文書に記述される項目要素の定義

```
<?xml version="1.0" ?>
<xbrli:group
    xmlns:xbrli="http://www.xbrl.org/2001/instance"
    xmlns:jp-bs="http://www.xbrl-jp.org/taxonomy/jp/fr/common/bs/2003-08-31"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.xbrl-jp.org/taxonomy/jp/fr/common/bs/2003-08-31 jp-bs-2003-08-31.xsd">

  <!-- Item-Element starts here -->
  <jp-bs:Assets numericContext="c1">8592822000000</jp-bs:Assets>
  <jp-bs:CurrentAssets numericContext="c1">3620881000000</jp-bs:CurrentAssets>
  <jp-bs:FixedAssets numericContext="c1">4971941000000</jp-bs:FixedAssets>
  <jp-bs:LiabilitiesStockholdersEquity numericContext="c1">8592822000000</jp-bs:LiabilitiesStockholdersEquity>
  <jp-bs:Liabilities numericContext="c1">2889500000000</jp-bs:Liabilities>
  <jp-bs:Equity numericContext="c1">5703322000000</jp-bs:Equity>

  <!-- Context-Element starts here -->
  <xbrli:numericContext id="c1" precision="18" cwa="true">
    <xbrli:entity>
      <xbrli:identifier scheme="">SAMPLE</xbrli:identifier>
    </xbrli:entity>
    <xbrli:period>
      <xbrli:instant>2003-03-31</xbrli:instant>
    </xbrli:period>
    <xbrli:unit>
      <xbrli:measure>ISO4217:JPY</xbrli:measure>
    </xbrli:unit>
  </xbrli:numericContext>
</xbrli:group>
```

図 3: インスタンス文書の例

を行う．さらに，インスタンス文書に記述される項目間の関係を定める，リンクベースへの参照を指定する．

2.5.1 項目要素の定義

インスタンス文書に記される，項目要素の要素名と型を定義する．XBRL 2.0 では，表 1 に示す 6 種類の型が定義されている．

表 1: 項目要素の型

型名	意味
monetaryItemType	金額型
decimalItemType	10 進数型
sharesItemType	株数型
stringItemType	文字列型
uriItemType	URI 型
dateTimeItemType	日付型

定義は次のように記述される．

```
<element name="Assets" type="xbrli:monetaryItemType"
 substitutionGroup="xbrli:item" id="jp-bs_Assets" />
```

name 属性で要素名を，type 属性で要素の型を指定する．この例では，monetaryItemType（金額型）の Assets という名前の要素を定義し，この要素は，項目要素であることと，リンクベースから Assets 要素を参照するときの id が jp-bs_Assets であることを示している．

2.5.2 リンクベースの参照

リンクベースへの参照は linkbaseRef 要素を使って行う．linkbaseRef 要素の role 属性でリンクベースの種類を，href 属性で参照先のファイル名を指定する．定義リンクを参照する例を次に示す．

```
<annotation>
 <appinfo>
  <link:linkbaseRef
   xlink:type="simple"
   xlink:arcrole="http://www.w3.org/1999/xlink/properties/linkbase"
   xlink:actuate="onRequest"
   xlink:role="http://www.xbrl.org/linkprops/linkRef/definition"
   xlink:href="jp-bs-2003-08-31_definition.xml">
   Links for definition relationship</link:linkbaseRef>
 </appinfo>
</annotation>
```

2.5.3 タクソノミの拡張

他のタクソノミで定義された項目要素を利用するために，他のタクソノミを参照することができる．参照は import 要素を使って行う．

```
<import namespace=
    "http://www.xbrl-jp.org/taxonomy/jp/fr/common/bs/2003-08-31"
 schemaLocation="jp-bs-2003-08-31.xsd" />
<import
 namespace="http://www.xbrl-jp.org/taxonomy/jp/gcd/2003-08-31"
 schemaLocation="jp-gcd-2003-08-31.xsd" />
```

この例は，XBRL Japan [21] が公開している税務申告用財務諸表のタクソノミ [22] より引用した．税務申告用財務諸表 XBRL タクソノミは，基本財務諸表タクソノミ（jp-bs-2003-08-31.xsd）と，日本共通文書タクソノミ（jp-gcd-2003-08-31.xsd）を参照している．

インスタンス文書が複数のタクソノミを参照したり，タクソノミが，さらに他のタクソノミを参照することができる．そのため，図 4 に示すように，一個のインスタンス文書に記される項目要素を，多量のタクソノミやリンクベースで定義するために解析が複雑になる場合がある．

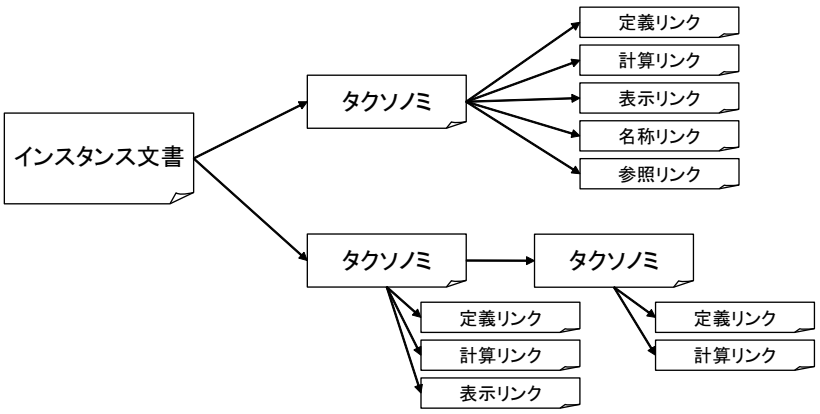

図 4: タクソノミの参照

2.6 リンクベース

リンクベースは，タクソノミで定義された項目間の関係や，各項目に対する追加情報を XLink [15] の外部リンク機能を利用して定義した文書である．リンクベースには表 2 に示す 5 種類がある．

表 2: リンクベースの種類

型名	意味
定義リンク	項目間の関係を定義
計算リンク	項目の値の加算式を定義
表示リンク	項目の表示順を定義
名称リンク	項目の名称を定義
参照リンク	項目の参考文献を定義

定義リンク，計算リンク，表示リンクは関係リンクとも呼ばれ，インスタンス文書に記された要素間の関係を定義する．これより，最も基本的な形式を持つ定義リンクを詳しく説明する．そして，そのほかの計算リンク，表示リンク，名称リンク，参照リンクは定義リンク

との違いについて説明する．

2.6.1 定義リンク

定義リンクは，インスタンス文書に記された項目間の概念上の親子関係や，異なる要素として記されているが，異なる観点から見た同一の概念であることを指定する，等価概念の関係を定義する．

定義リンクは，タクソノミで定義された項目要素を指定するためのロケータ（loc 要素）と，ロケータ同士を関係付ける定義アーク（definitionArc 要素）からなる．

```
<loc xlink:type="locator" xlink:title="Assets"
 xlink:label="jp-bs_Assets"
 xlink:href="jp-bs-2003-08-31.xsd#jp-bs_Assets"/>

<loc xlink:type="locator" xlink:title="CurrentAssets"
 xlink:label="jp-bs_CurrentAssets"
 xlink:href="jp-bs-2003-08-31.xsd#jp-bs_CurrentAssets" />

<definitionArc
 xlink:from="jp-bs_Assets"
 xlink:to="jp-bs_CurrentAssets"
 xlink:arcrole="http://www.xbrl.org/linkprops/arc/parent-child"
 xlink:title="Go to element jp-bs_CurrentAssets"
 xlink:type="arc" xlink:show="replace" xlink:actuate="onRequest" />
```

この例で現れる最初のロケータは，jp-bs-2003-08-31.xsd というファイルのタクソノミで定義された jp-bs_Assets という id が付いた項目要素（資産の部）を参照し，このロケータに jp-bs_Assets というラベルを付けている．次のロケータも同様，jp-bs-2003-08-31.xsd というファイルのタクソノミで定義された jp-bs_CurrentAssets という id が付いた項目要素（流動資産）を参照し，このロケータに jp-bs_CurrentAssets というラベルを付けている．そして，定義アークでロケータ間の関係を定義する．具体的には，definitionArc 要素の from 属性と to 属性にロケータのラベルを記し，arcrole 属性で from 属性で指定されたロケータから to 属性で指定されたロケータへの関係を定義する．全体では，資産の部は，流動資産の親であるということを表す.

図 5 に，ロケータからタクソノミへの参照，定義アークからロケータへの参照をまとめた図を示す．図中の要素が持つ属性は適宜省略している．

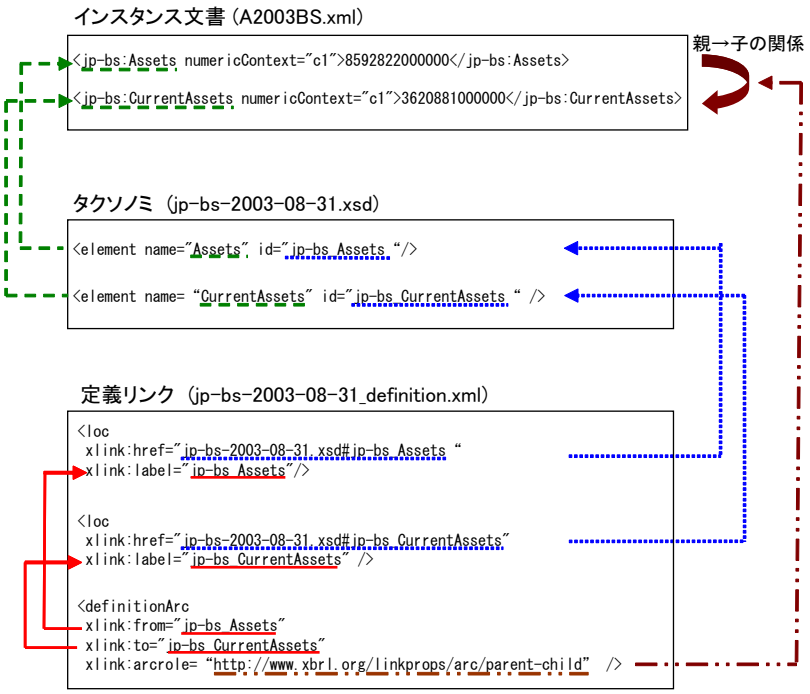

図 5: 定義リンクの参照構造

2.6.2 計算リンク

財務報告書では，親項目の値が子項目の値の和として計算される項目がある．計算リンクでは項目間の親子関係と，計算によって値を決定するという関係を定義する．さらに，親項目の値を子項目の値の和として計算する際に，子項目の値に掛ける重みの値を指定することもできる．

項目要素間の関係づけは，定義アークに似た計算アーク（calculationArc 要素）で行う．計算アークは，定義アークが持つ属性と，重みの値を記す weight 属性を持つ．

```
<calculationArc
 xlink:from="jp-bs_Assets"
 xlink:to="jp-bs_CurrentAssets"
 xlink:arcrole="http://www.xbrl.org/linkprops/arc/parent-child"
 weight="1"
 xlink:title="Go to element jp-bs_CurrentAssets"
 xlink:type="arc" xlink:show="replace" xlink:actuate="onRequest" />
```

この例では，Assets（資産の部）の値を計算するときに，Assets の子要素として CurrentAssets（流動資産）も含め，合計するときの重みを 1 とすることを示している．

2.6.3 表示リンク

表示リンクは，XBRL 文書を貸借対照表などの財務諸表の形式で表示するときの，内訳を表す項目要素間の階層構造と表示順序を指定する．項目要素間の関係付けは，定義アークに似た表示アーク（presentationArc 要素）で行う．計算アークは，定義アークが持つ属性と，表示順序を表す order 属性を持つ．

```
<presentationArc
 xlink:from="jp-bs_Assets"
 xlink:to="jp-bs_CurrentAssets"
 xlink:arcrole="http://www.xbrl.org/linkprops/arc/parent-child"
 order="1"
 xlink:title="Go to element jp-bs_CurrentAssets"
 xlink:type="arc" xlink:show="replace" xlink:actuate="onRequest" />
```

この例では，貸借対照表の形式に直すときに Assets（資産の部）の内訳として CurrentAssets（流動資産）も含め，内訳の一番最初に表示することを示している．

2.6.4 名称リンク

名称リンクは，XBRL 文書を貸借対照表などの財務諸表の形式で表示するときの，表示名を指定する．これまで説明した，定義リンク，計算リンク，表示リンクはロケータ同士を関連付けるものであるが，名称リンクは，項目要素を参照するロケータと，表示名称を定義する label 要素を関連付ける．labelArc 要素と label 要素は以下のように定義する．

```
<labelArc
 xlink:from="jp-bs_Assets"
 xlink:to="jp-bs_Assets_ja"
 xlink:arcrole="http://www.xbrl.org/linkprops/arc/element-label"
 xlink:title="Go to label jp-bs_Assets_ja"
 xlink:type="arc" xlink:show="embed" xlink:actuate="onRequest" />

<label
 xlink:type="resource"
 xlink:label="jp-bs_Assets_ja"
 xlink:title="jp-bs_Assets_ja"
 xlink:role="http://www.xbrl.org/linkprops/label/standard"
 xml:lang="ja">（資産の部）</label>
```

この例では，jp-bs_Assets で参照される項目要素，すなわち以前の例で示した Assets は，貸借対照表では（資産の部）と表示されることを表している．また，xml:lang 属性や，xlink:role

を適切に設定することにより様々な表示方法に対応できる．

2.6.5　参照リンク

参照リンクでは，項目要素の概念の根拠となる情報を指定する．参照リンクでは，情報の定義に reference 要素を用い，項目要素と reference 要素との関連付けに参照アーク（referenceArc 要素）を用いる．

```
<referenceArc
 xlink:from="jp-bs_Assets"
 xlink:to="jp-bs_Assets_REF"
 xlink:arcrole="http://www.xbrl.org/linkprops/arc/element-reference"
 xlink:title="Go to reference jp-bs_Assets_REF"
 xlink:type="arc" xlink:show="embed" xlink:actuate="onRequest" />

<reference
 xlink:type="resource"
 xlink:label="jp-bs_Assets_REF"
 xlink:title="jp-bs_Assets_REF">
  <jp-bs:name xmlns:jp-bs=
     "http://www.xbrl-jp.org/taxonomy/jp/fr/common/bs/2003-08-31"
  >財規様式第二号</jp-bs:name>
</reference>
```

この例では，Assets は財規様式第二号に拠るものであることを示している．

2.6.6　アークの上書き

タクソノミの拡張で述べた，項目要素間の関係を記すアークが複数個設定される場合や，既存のアークを取り消したい場合がある．このような場合，XBRL には，リンクベースのアークに優先度を付け，既存のアークを上書きしたり，取り消したりできる機構がある．この機構をアークの上書きという．アークの上書きのため，アークには以下の 2 つの属性が定義されている．

- priority 属性
  priority 属性は整数値を取る属性で，アークの優先度を表す．規定値は 0 である．同じ要素間のアークが複数存在する場合には，priority 属性の値を比較し，より大きい値を持つアークが優先する．仮に同じ要素間のアークが複数存在し，その priority 属性が等しい場合には，その振る舞いはアプリケーションに依存する．

- use 属性

  use 属性は required，optional，prohibited の 3 種類の値をとり，そのアークの利用に関する情報を表す.

  - optional

    リンクがトラバーサルの候補であることを表す．use 属性の規定値である.

  - required

    アークに関連付けられたデータ項目のうち一方の要素が出現すると，もう一方の要素も必ず出現しなければならないことを表す.

  - prohibited

    リンクを取り消すことを表す.

アークの上書きの例を示す.

```
(1)
<definitionArc
 xlink:from="jp-bs_Assets"
 xlink:to="jp-bs_CurrentAssets"
 xlink:arcrole="http://www.xbrl.org/linkprops/arc/parent-child"
 xlink:title="Go to element jp-bs_CurrentAssets"
 xlink:type="arc" xlink:show="replace" xlink:actuate="onRequest" />

(2)
<definitionArc
 priority="1"
 use="prohibited"
 xlink:from="jp-bs_Assets"
 xlink:to="jp-bs_CurrentAssets"
 xlink:title="Go to element jp-bs_CurrentAssets"
 xlink:arcrole="http://www.xbrl.org/linkprops/arc/parent-child"
 xlink:type="arc" xlink:show="replace" xlink:actuate="onRequest" />
```

上に記した 2 個の定義アークは，異なるタクソノミから参照される，異なるリンクベースで定義されている．(2) のアークは (1) のアークを上書きするアークである．priority 属性の規定値は 0 なので，priority 属性に 1 を，use 属性に prohibited を指定したアークを記述することで (1) のアークを取り消すことができる.

3 XBRL 文書モデル

本研究では，インスタンス文書に記述された値の読み取りや変更，もしくは，新しくインスタンス文書を作成し，その中に記述するの値の設定といった，インスタンス文書に対する操作を扱う．そのため，簡単な操作でインスタンス文書の変換を行うためには，インスタンス文書を表す理解しやすいモデルが必要である．本章では，まず，XML 文書を表すための有名なモデルである DOM について紹介する．次に，インスタンス文書の特徴を使い，XBRL 文書を簡単に扱うためのモデルを提案する．

3.1 DOM（Document Object Model）

Document Object Model（DOM）とは，W3C が策定した XML 文書を表現するためのモデル [13] である．DOM は，XML 文書に現れる要素，テキスト，属性などの文書構成要素をノードとして表し，ノードが相互に関係する木構造（DOM ツリー）として XML 文書を表現する．例えば，前節図 3 の XBRL 文書は，図 6 のように表される．図中の四角がノードを表し，ノードは親から子の方向に引かれた矢印で関連付けられる．XML 文書は，document ノードから始まる木構造として表現される．

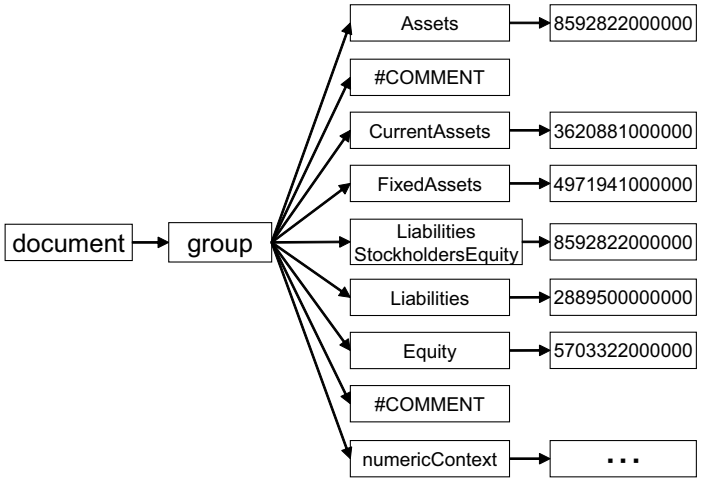

図 6: DOM ツリーによる表現

DOM は，XML の要素もテキストデータも同じノードとして表現するため，XBRL 文書処理で頻繁に行われる財務データの読み取り，書き換えを行うために煩雑な処理を行う必要がある．例として，assets 要素の値を書き換える手順を記す．

1. DOM ツリーから assets 要素ノードを探す

2. 見つかった assets 要素ノードの子ノードリストを求める

3. 子ノードリストからテキストノードを選ぶ（コメントノードなどテキストノード以外のノードが含まれる場合がある）

4. テキストノードの値を書き換える

DOM をインスタンス文書のモデルとして使うと，以上のような手順を踏む必要があり，非常に煩雑である．そこで，インスタンス文書の特徴を使い，DOM を基にした新しいモデルを考える．

3.2 インスタンス文書の特徴

インスタンス文書に記された項目要素には，子要素を持たないという特徴と，入れ子構造による関係以外の関係が定義されているという特徴がある．

一般の XML 文書では，図 7 のように，マークアップされた文字列データが子要素により分割されうる．

```
<P>
  text1
  <IMG src=“picture.jpg”>
  text2
</P>
```

図 7: 要素によるテキストの分割

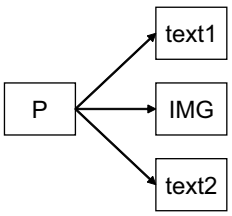

図 8: DOM ツリー

分割された文字列データやその順番も保持するため，DOM ツリーでは文字列データを，テキストノードとして表現し，要素を表す要素ノードと同等に扱う．図 7 の XML 文書から DOM ツリーを構築すると図 8 のようになる．しかし，インスタンス文書に記された項目要素は子要素を持たないということが，インスタンス文書のスキーマで規定されているために，マークアップされた文字列データは分割されない．そこで，要素ノードに文字列データ

を持たせたモデルを作ることが可能となる．要素ノードに文字列データを持たせると，項目要素が持つ値（文字列データ）を参照する際に，子要素ノードの解析をすること無しに値を参照できるため，大幅な省力化が可能となる．

また，インスタンス文書に記された要素間には，次に示す関係がある．

- 要素間の入れ子構造による親子関係
- リンクベースで定義された項目要素間の親子関係
- 項目要素からコンテキスト要素間への参照関係

DOM を使った場合，入れ子構造による親子関係しか解析しないため，リンクベースで定義された親子関係などは，利用者が解析する必要があった．そこで，リンクベースで定義された関係もモデルに取り込み，解析処理の負担軽減を図る．

3.3 提案モデル

要素を頂点とし，要素間の関係を辺とする有向グラフを XBRL 文書のモデルとして提案する．有向グラフの頂点には，要素の値（マークアップされた文字列データ）と，属性を持たせる．

図 3 のインスタンス文書の，有向グラフによる表現を図 9 に記す．ただし，簡単のために名前空間接頭辞と，`xbrli:numericContext` の子要素は省略してある．また，表 1 の様に次の親子関係がリンクベースで定義されているとする．

- `Assets`（資産合計）は，
  `CurrentAsset`（流動資産）と `FixedAssets`（固定資産）の親
- `LiabilitiesStockholdersEquity`（負債および資本合計）は，
  `Liabilities`（負債の部）と `Equity`（資本の部）の親

有向グラフによる表現では，要素のみが頂点として現れるため，頂点の数が DOM ツリーよりも少なくなり，簡単なグラフとなる．有向グラフのモデルは，DOM ツリーの要素ノードにノードの値の取得方法と他の要素へのアクセス方法を提供した DOM のラッパーであるととらえることができる．さらに，DOM の上に XBRL 文書特有の関係で要点間に辺を引くため，要素間の関係を利用した要素の参照を簡単に行うことができる．有向グラフによるモデルは，DOM ツリーより理解しやすく，操作しやすいモデルであると考えられる．

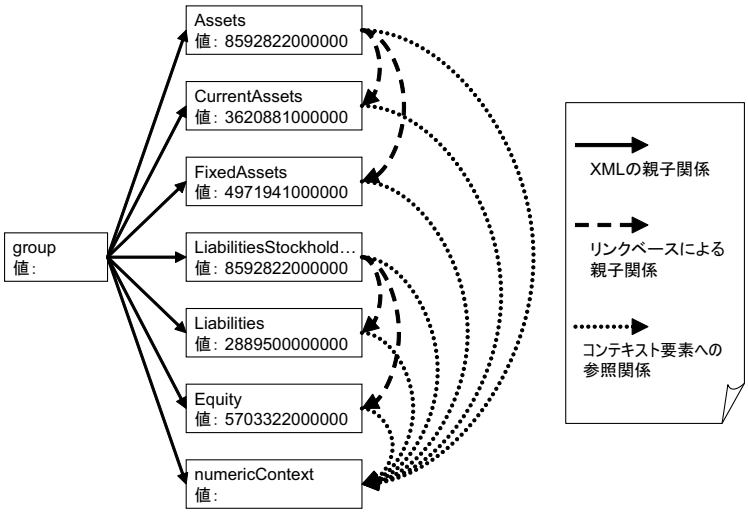

図 9: 有向グラフによる表現

3.4 モデルに対する操作

インスタンス文書への操作は次の 3 種類がある．

- 値の操作
- 属性の操作
- 構造の操作

値の操作とは，数値，又は文字列として表された財務情報に何らかの計算処理，文字列処理を施すこと，属性の操作とは，属性の追加，削除，および，値を変更すること，構造の操作とは，要素間の入れ子構造を操作することである．

インスタンス文書のモデルでは，値と属性は有向グラフの頂点が持っているため，値の操作と，属性の操作は，頂点が持っているデータを変更することで行う．構造の操作は，頂点の追加，削除，および，XML の親子関係による辺（図 9 の実線の矢印）をつなぎ変えることで行う．リンクベースで定義された親子関係（波線の矢印）を変更することは，本モデルでは考えない．コンテキスト要素への参照関係（点線の矢印）の変更は，コンテキスト属性の変更によって行い，点線の矢印をつなぎ変える操作は扱わない．

4　プログラミング言語 LMX

XBRL 文書に書かれた値を使った指標の計算や，書式変換といった XBRL 文書処理を簡単に行うことを目的とし，3 章で述べたモデルに基づく XBRL 文書操作言語，LMX （A Tiny **L**anguage for **M**anipulating **X**BRL Documents）を設計した．LMX の特徴は次のようにまとめられる．

- C 言語に似た文法を持つ構造化プログラミング言語である
- 名前や型の宣言なしで変数を使うことができる
- 3 節で述べた XBRL 文書のモデルを実装し，XBRL 文書への簡単なアクセスを可能としている
- XBRL 文書特有の処理（定義リンクによる親子関係の計算，計算リンクによる値の計算など）をサポートしている

本節では，まず，LMX の言語仕様について説明し，次に，XBRL 文書操作のために提供する機構について述べる．また，付録に LMX の文法（BNF）と言語処理系が提供する API を載せる．

4.1　言語仕様

4.1.1　データ型と変数

LMX では，名前と型の宣言なしに自由に変数を使うことができる．また，変数名の大文字小文字は区別する．変数宣言をしないため，プログラム上で型名が現れることはないが，基本型，配列変数を表す Array 型，4.1.6 節で述べる Node 構造体を表す NodeStruct 型の区別がある．さらに，基本型は，Integer 型（整数），Float 型（浮動小数点），String 型（文字列），Boolean 型（論理型）に分けられるが，基本型同士は演算中に自動で型変換されるため，型を意識してプログラムを記述する必要はない．

本言語は財務報告書を扱う言語であり，財務情報を扱う整数型は桁あふれを起こしてはならないため，整数型は任意精度の演算をサポートする．

次に示す単語群は予約語であり，変数名として使用することはできない．

予約語

```
break  case  continue  default  do  else  false  for
if  null  return  function  switch  true  while
```

4.1.2 演算子

LMX で使用できる演算子を以下に載せる．演算子の意味は，C 言語のものと同一である．

演算の過程で，型が変換される場合がある．算術二項演算子は，Integer 型同士，Float 型同士，そして，Integer 型と Float 型（Float 型と Integer 型）に対して適用できる．異なる型に対して演算が行われた場合,（例，Integer + Float）Integer 型は Float 型に変換され，演算結果は Float 型として得られる．

+演算子は，String 型に対しても適用でき，少なくとも一つのオペランドが String 型である場合は，他方のオペランドも String 型に変換され文字列の連結が行われる．

単項算術演算子

```
+, -, ++, --
```

二項算術演算子

```
+, -, *, /, %
```

代入演算子

```
=, +=, -=, <<=, >>=, *=, /=,
```

比較演算子

```
==, !=, <=, >=, <, >
```

論理演算子

```
&&, ||, !
```

三項演算子

```
? :
```

4.1.3 制御構造

if 文による選択，for 文，while 文，do while 文による繰り返しを使用できる．さらに，break 文による，ループ脱出と，continue 文によるループの再開を使用できる．statement は，C 言語同様単文，もしくは，“{”，“}” で囲まれた複文を指定できる．

expr は，真偽判定を行う式である．

if 文

```
if (expr)
    statement1
else
    statement2
```

else 節は省略可能である．

```
if (expr)
    statement
```

for 文

```
for (expr1; expr2; expr3)
    statement
```

while 文

```
while (expression)
    statement
```

do while 文

```
do {
    statement
} while (expr)
```

4.1.4 関数

スコープ

変数の有効範囲は，関数の中のみである．変数は，値が代入されたときに作成される．関数の外で作成された変数は，関数の中からも参照できるが，関数の中で作成された変数は，関数の実行が終わった後では参照できない．また，関数の外で作成された変数（変数 A）と同じ名前の変数（変数 B）を関数の中で作成すると，値の読み取り，設定は，変数 B に対して行われる．変数 B が作成される前は，変数 A の値を読みとることができる．ただし，変数 B を作成する前に，変数 A に対してインクリメント・デクリメント演算子を適用すると変数 A の値が書き換えられてしまうので，注意が必要である．

関数定義

function キーワードを使い，以下のように関数を定義する．関数には引数を渡すことができ，arglist に記述する．これらは，関数内でのみ有効なスコープを持つ．基本型は値渡し，

Array 型，NodeStruct 型は参照渡しとなる．引数を複数個の変数を使うときは，変数をコンマで区切って書く．関数では return 文を使い，呼び出し元に値を返すことができる．また，再帰呼び出しも可能である．

関数定義

```
function funcname (arglist) {
    statements;
}
```

関数呼び出し

“(”，“)” 記号を使い，関数を呼び出す．

関数呼び出し

```
funcname(arg1, arg2);
```

4.1.5 配列変数

任意の型の値を格納することができる，配列変数を使用することができる．

配列変数の作成

配列変数を使用する際には，array 関数を使って配列変数の作成が必要である．配列変数の長さは array 関数の引数で与える．

配列変数の作成

```
arrayvar = array(expr);
```

配列変数の使用

“[”，“]” 記号を使い，配列の中に格納された変数を読み出す．配列のインデックスは 0 から始まる．

配列変数の使用

```
var = arrayvar[expr];
```

4.1.6 Node 構造体

Node 構造体は，XML 文書中に現れる要素を抽象化した構造体である．3.3 節で述べた有向グラフの頂点に相当する．Node 構造体は，親要素や，子要素を参照するメンバや，要素の値や属性を読み書きするためのメンバを持つ．メンバへの参照は，“.” 記号を使う．メンバ一覧を表 3 に載せる．

表 3: メンバ一覧

型	メンバ名	意味	読み書き
String 型	name	要素名	read only
String 型	localname	ローカル名	read only
String 型	uri	名前空間 URI	read only
タクソノミでの定義による	value	要素の値	read/write
NodeStruct 型	parent	親要素	read/write
Array 型	children	子要素の配列	read/write
ハッシュ	attribute	属性	read/write

name

名前空間接頭辞を含む，要素の名前を表す．name メンバは値の読み取りだけ可能で，書き換えはできない．

localname

名前空間接頭辞を含まない，要素のローカル名を表す．localname メンバは値の読み取りだけ可能で，書き換えはできない．

uri

名前空間 URI を表す．名前空間接頭辞とは異なる．名前空間接頭辞を取得したい場合は，後に述べる getPrefix 関数を用いる．uri メンバは値の読み取りだけ可能で，書き換えはできない．

value

要素の値を表す．要素の値とは，要素の開始タグと終了タグの間にあるテキストである．テキストが子要素により複数個に分割された場合は，複数のテキストを出現順に連結した文字列が返される．Node 構造体が XBRL の項目要素を表している場合は，タクソノミを解析し，適切な型（Integer 型，Float 型，String 型）として変換された値を取得できる．

また，基本型の値を代入することで，要素の値を設定できる．

parent

NodeStruct 型の変数が表している要素の，親要素を表す．親要素が存在しない場合は，parent

メンバの値は null と等しい．NodeStruct 型の値を代入することで，親要素を設定できる．ただし，同じ XML 文書にある要素間で，親に当たる要素を（親要素を順に辿って得られる要素）を親要素として設定できない．同一文書にある要素を親要素として設定した場合，移動となり，異なる文書にある要素を親要素として設定した場合は，子要素を含む要素のコピーとなる．

children

子要素の配列を表す．“[”，“]” 記号を使って，子要素一つの読み出しや，設定ができる．また，要素の配列を代入することにより子要素全てを一度に設定することができる．

parent と同様，同一文書にある要素間で children の設定をすると，要素の移動となり，異なる文書間で行った場合は，要素のコピーとなる．

attribute

属性を表す．attribute は，他のメンバと異なる記法で値の読み書きを行う．“{”，“}” 記号で属性名を指定すると，その属性名を持つ値の取得や設定ができる．また，attribute メンバを代入することで属性全てを一括して設定することもできる．

値の参照の例

```
nodename = nodevar.name;
attrvalue = nodevar.attribute{"numericContext"}
child1 = nodevar.children[0];
```

値の設定の例

```
node1.value = 10000;
node1.parent = parentnode;
node2.children[0] = childnode;
node2.children = node3.children;
node3.attribute{"nonNumericContext"} = "s1";
node3.attribute = node4.attribute;
```

LMX では，DOM ツリーの要素ノードを Node 構造体に対応付けた DOM のラッパーとして，XBRL 文書のモデルを実装する．Node 構造体のメンバーである，parent，children を使って取得した Node 構造体は，必ず要素を表すが，Node 構造体は，DOM のノードとして扱うことも可能である．すなわち，後述する DOM API を用いることで（例えば，getNodeChildren），要素ノード以外のノード（例えば，テキストノードやコメントノード）を Node 構造体に対

応付けることができる．この場合，getNodeValue や，setNodeValue 関数を使い，値を取得・変更することができる．

このような実装を採用することで，DOM で扱うことのできる汎用的な操作が可能で，かつ，XBRL 文書操作を簡単に扱うことができる．

4.2 XBRL サポート

LMX では，XBRL 文書を解析し，その結果を返す関数を用意することで，XBRL 文書を効率よく扱うための機構を提供する．XBRL 文書を解析して得られる情報には大きく分けて 3 種類ある．リンクベースを解析して得られる親子関係と表示方法の情報，インスタンス文書を解析して得られる，コンテキスト要素の情報，そして，インスタンス文書に記された，項目要素の情報である．これより，提供する関数の概略をを 3 種類に分けて説明する．関数の完全なリストは，付録に載せる．

4.2.1 リンクベースで定義された情報の取得

リンクベースで定義された，親子関係の定義，表示する際の入れ子構造と順番，表示言語に応じた表示ラベルの情報を取得する関数である．

`calculateNodeValue`

計算リンクを利用して，ノードの値を計算する．

`getDefinitionChildren`

定義リンクで定義された親子関係により，子 Node 構造体の配列を取得する．

`getDefinitionParents`

定義リンクで定義された親子関係により，親 Node 構造体の配列を取得する

`getLabel`

ラベルリンクで定義されたラベルを取得する

`getPresentationChildren`

表示リンクで定義された親子関係により，子 Node 構造体の配列を取得する．返される値は，表示リンクで定義された順番でソート済みである．

これらの関数を使うことで，実際の財務諸表に現れる項目間の親子関係で項目要素にアクセスすることができる．

4.2.2 コンテキスト要素の取得

インスタンス文書に記された項目要素を補足するための，コンテキスト要素を取得するための getContextNode 関数を提供する．さらに，項目要素に関連付けられたコンテキスト要素の Node 構造体を返す関数の他に，プログラムの書きやすさのために，コンテキスト要素を解析して得られる情報（CWA 値や，precision 値，下位要素）を取得する関数も提供する．

4.2.3 項目要素に関する情報の取得

インスタンス文書に記された項目要素の取得や，項目要素の情報の取得のための関数である．

`getItemElementsByName`
指定した名前の項目要素の配列を取得する．DOM でも同様の関数が定義されているが，この関数は，検索対象を項目要素に限定していることが特徴である．

`isItemElement`
Node が項目要素かどうか判定する．

`getItemElementType`
項目要素の型を取得する．

項目要素の型として，表 4 に示す定数を定義する．

4.3 DOM API

DOM で定義されている関数を提供する．DOM には，要素ノードやテキストノードを作成すると行ったノードの作成や，子ノードを追加・削除するといった構造の操作を行う関すが定義されている．これらの関数を用いて XML 文書の構造を操作することができる．関数のリストは，付録に載せる．

4.4 入出力

LMX では，XBRL 文書を含む XML 文書への入出力と，標準出力への出力を考えている．これらの処理を行うための関数を提供する．

表 4: 項目要素の種類を表す定数

定数名	意味
MONETARY_TYPE	金額型
SHARES_TYPE	株数型
DECIMAL_TYPE	十進小数型
STRING_TYPE	文字列型
URI_TYPE	URI 型
DATETIME_TYPE	日付時間型
NONITEM_TYPE	項目要素ではないことを表す

4.4.1 XML 文書への入出力

`openInstance`

指定したファイル名のインスタンス文書を開き，ルート要素の Node 構造体を返す．XBRL 文書でない XML 文書も同様に開くことができる．この場合は，タクソノミの解析は行われない．

`save`

ファイルに上書き保存する．

`saveAs`

ファイル名を指定して，XML 文書を保存する．

4.4.2 標準出力

`print`

標準出力に指定された変数，値を出力する．

`println`

標準出力に指定された変数，値を出力し，改行文字を出力する．

5 実装

5.1 概要

本研究では，4 節で提案したプログラミング言語 LMX のインタプリタおよび，5.4 節で述べる GUI を実装した．インタプリタおよび GUI は Java[12] で実装した．LMX のためのパーサジェネレータとして Java Compiler Compiler（JavaCC）[10] を利用し，XBRL 文書を解析するための XML パーサは Java 2 SDK バージョン 1.4 以上に同梱される JAXP[11] を利用した．また，ビルドツールとして Apache Ant[1] を利用した．

LMX のインタプリタおよび GUI をビルドするために必要な環境は次のとおりである．

- Java 2 SDK，Standard Edition バージョン 1.4 以上
- Java Compiler Compiler（JavaCC） バージョン 3.2 以上
- Apache Ant バージョン 1.5.4 以上

LMX のインタプリタおよび GUI を実行するために必要な環境は次のとおりである．

- Java 2 Runtime Environment，Standard Edition バージョン 1.4 以上

LMX のインタプリタおよび GUI の実装は，次のパッケージからなる．

- LMX.XBRL – XBRL 解析
- LMX.Parser – LMX のパーサ．AST 作成ルーチンを含む（JavaCC により生成）．
- LMX.Interpreter – LMX のインタプリタ
- LMX.GUI – GUI

これより，それぞれのパッケージごとに機能を説明する．

5.2 XBRL 解析

XBRL 解析パッケージでは，インスタンス文書を解析し，有向グラフによるアクセスのための情報や，タクソノミ，リンクベースを解析し，項目要素の情報や，項目要素間の関係，表示に関する情報を得る．インスタンス文書は，有向グラフによるモデルで DOM ツリーを利用するため DOM パーサで解析している．また，タクソノミとリンクベースは，それらに記された情報のみを得ることができればよく，DOM ツリーとして構造を保持しておく必

要はないため，利用するメモリの量と実行速度の点で有利な [5] SAX パーサ [3] で解析している．

LMX では，インスタンス文書の操作のみを対象としている．すなわち，ユーザが作成するプログラムにおいて，入出力の対象となる XBRL 文書はインスタンス文書のみである．LMX のプログラムでインスタンス文書が開かれると，まず，当該インスタンス文書の解析を行い，さらに，そのインスタンス文書が参照するタクソノミ，リンクベースの解析を行う．XBRL 文書解析の概略は以下のようになる．

1. インスタンス文書をファイルから読み出し，DOM ツリーを作成する
2. インスタンス文書が参照しているタクソノミのファイル名のリスト（タクソノミリスト）を取得する
3. タクソノミリストからタクソノミのファイル名を取り出し，解析済みでないことを確認し，SAX パーサでタクソノミの解析を行う
   3.1. element 要素が見つかれば，項目要素として登録する
   3.2. import 要素を使った他のタクソノミへの参照が見つかれば，参照先のタクソノミのファイル名を求め，タクソノミリストに登録する
   3.3. linkbaseRef 要素を使ったリンクベースへの参照が見つかれば，リンクベースの種類を判別し，ファイル名をリンクベースリストに登録する
4. 解析が終わったタクソノミをタクソノミリストから削除し，タクソノミリストが空でなければ 3. に戻る
5. リンクベースリストからリンクベースのファイル名を取り出し，SAX パーサで構文解析を行う
   5.1. loc 要素が見つかれば，ロケータから項目要素を求めるハッシュテーブルを作成する
   5.2. arc（定義アーク，計算アークなど）が見つかれば，アークリストに登録する
6. アークリストからアークを一つ取り出し，ロケータ間の関係を求め，5.1. で作成したハッシュテーブルを使い項目要素間の関係として登録する
7. 解析が終わったリンクベースをリンクベースリストから削除し，リンクベースリストが空でなければ 5. に戻る

次節より，インスタンス文書，タクソノミ，および，リンクベースの解析アルゴリズムについて詳しく説明する．

5.2.1　インスタンス文書解析アルゴリズム

インスタンス文書解析は，まずインスタンス文書をファイルから読み込み，DOM ツリーを作成する．そして，DOM ツリーをラップするためのルーチンを実装することで，DOM ツリーから，有向グラフのモデルを作り上げている．有向グラフのモデルのうち，項目要素間の関係は後述するタクソノミ解析やリンクベース解析により得られるが，コンテキスト要素の参照は DOM ツリーの解析で実装している．本節では，コンテキスト要素の参照，Node 構造体のメンバの値の取得，設定を行うアルゴリズムについて述べる．

コンテキスト要素の参照

インスタンス文書に記された項目要素には，項目要素が数値型の場合は numericContext が，項目要素が文字列型の場合は nonNumericContext 属性が付与される．また，インスタンス文書には，numericContext 要素や，nonNumericContext 要素が記されており，これらには id 属性が設定されている．項目要素は numericContext 要素または nonNumericContext 要素の id を，numericContext 属性または nonNumericContext 属性に指定することで，コンテキスト要素への参照を表す．コンテキスト要素への参照の例を図 10 に示す．図 10 では，数値型の項目要素 Assets は，c1 という id を持つ numericContext 要素を参照し，文字列型の項目要素 BalanceSheetUnit は，s1 という id を持つ nonNumericContext 要素を参照している．

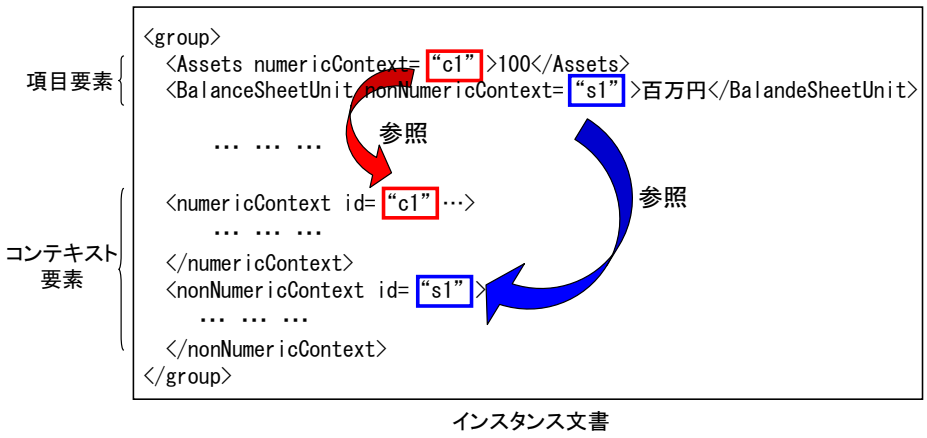

図 10: コンテキスト要素への参照

項目要素が参照するコンテキスト要素を求めるには次の手順でインスタンス文書を解析する．

1. 項目要素の型をタクソノミの解析結果から得る
2. 項目要素の型に応じて numericContext または nonNumericContext の値を取得する

3. インスタンス文書から numericContext または nonNumericContext 要素を探す

4. 2. で得た属性の値と，3. で得た要素の id 属性の値を比べ，等しい場合，3. で得た要素を項目要素が参照するコンテキスト要素とする

Node 構造体のメンバ

次に，DOM ツリーを有向グラフのモデルとして扱うための Node 構造体について，メンバの値の取得，設定の実装方法について述べる．Node 構造体は DOM のノードのラッパーであり，DOM のノードが関連付けられている．

name，localname，uri

要素名，ローカル名，名前空間を取得するためのメンバである．これらは DOM で定義された，getName，getLocalName，getNamespaceURI 関数を呼び出し，DOM のノードから直接値を取得する．

value

要素の値を取得，設定するためのメンバである．値を取得する際は，DOM の子要素で，テキストノードであるノードの値を連結し，要素の値とする．値を設定する際は，DOM の子要素で最初に現れるテキストノードの値を指定された値にする．DOM の子要素にテキストノードが存在しない場合は，指定された値のテキストノードを作成し，子要素リストの末尾に追加する．

parent

親要素を取得，設定するためのメンバである．親要素を取得する際は，DOM ツリーを親方向に辿り，最初に現れた要素ノードを親要素とする．設定する際は，設定操作前の親要素ノードと，新しく親要素にする要素ノードが同じドキュメントにあるかどうかを調べる．同じドキュメントにある場合は，設定操作前の親要素ノードから，Node 構造体が表す子ノードを削除し，新しく親要素にするノードの子リストの末尾追加する．違うドキュメントにある場合は，DOM で定義された importNode 関数により同じドキュメントにコピーし，子リストの末尾に追加する．

children

子要素を取得，設定するためのメンバである．親要素を取得する際は，DOM の直接の子要素で，要素ノードであるものを子要素とする．設定する際は，子要素にするノードと，Node 構造体が表すノードが同じドキュメントにあるかどうかを調べる．同じドキュメントにある

場合は，DOM で定義された replaceChild 関数で新しい子要素を設定する．親要素を子要素にしようとした場合は，replaceChild がエラーとなる．違うドキュメントにある場合は，DOM で定義された importNode 関数により同じドキュメントにコピーし，子要素として設定する．

attribute

要素の属性を取得，設定するためのメンバである．DOM で定義された getAttribute，setAttribute 関数を使って，属性の取得，設定を行う．

5.2.2 タクソノミ解析アルゴリズム

タクソノミを SAX パーサで解析し，タクソノミ中に element 要素が見つかれば，項目要素として登録する．タクソノミの解析によって得られる項目要素の情報は，表 5 のようにまとめられる．

表 5: 項目要素の情報

情報	意味と用途
要素名	インスタンス文書に記述される要素の名前
名前空間 URI	項目要素が定義されたタクソノミの名前空間 URI
型	XBRL 文書で用いられるデータ型
id	リンクベースから項目要素を参照するときに使われる
SubstitutionGroup	項目要素か，タプルかを示す値
abstract	インスタンス文書に現れる要素かどうかを示す値

リンクベースの解析が簡単になるよう，名前空間 URI から項目要素が定義されたタクソノミを求めるデータベース，タクソノミと要素名から，項目要素の情報を求めるデータベース，タクソノミと id から，項目要素の情報を求めるデータベースの 3 種類のデータベースを作成し，情報を登録する．

import 要素を使った他のタクソノミへの参照が見つかれば，参照先のタクソノミのファイル名を求め，タクソノミリストに登録する．タクソノミは，import 要素でインスタンス文書のスキーマや，リンクベースのスキーマも参照するが，XBRL 文書解析には必要ないため，これらファイルは無視する．

linkbaseRef 要素を使ったリンクベースへの参照が見つかれば，リンクベースの種類を判別し，ファイル名をリンクベースリストに登録する．リンクベースの種類は，linbaseRef 要素の arcrole 属性で，定義リンク，計算リンク，表示リンク，名称リンク，参照リンクの別が判

定できる．5 種類のリンクベースのファイル名を保存するリンクベースリストも 5 種類あり，href 属性から得たリンクベースのファイル名を種類ごとにリンクベースリストに登録する．

インスタンス文書とタクソノミから参照された全てのタクソノミの解析が終わり，全ての項目要素の情報とリンクベースのファイル名を得ると，続けてリンクベースの解析に移る．

5.2.3 リンクベース解析アルゴリズム

本節では，定義リンクを解析するアルゴリズムに絞って説明する．

1. SAX パーサでリンクベースの解析を行う．
   - loc 要素（ロケータ）が見つかれば，loc 要素の属性（label，href）を解析し，href 属性が指し示す項目要素を求める．

     href 属性は，タクソノミのパスとタクソノミの中で定義された項目要素の id の値からなる．これらの情報から，項目要素の情報を得る．そして，loc 要素の label 属性の値から項目要素を求めるハッシュテーブルに登録する．
   - definitionArc 要素（定義アーク）が見つかれば，アークリストに登録する．
2. SAX パーサによる解析が終わると，アークリストに登録したアークを一つずつ調べる．

   アークは，from 属性，to 属性，arcrole 属性を持つ．from 属性，to 属性の値はロケータの label 属性を指し示している．アークは label 属性が from 属性の値と等しいロケータと，label 属性が to 属性の値と等しいロケータを arcrole 属性が示す意味で関係づける．

   アークの解析ではまず，from 属性と to 属性が指し示す項目要素をハッシュテーブルから取得する．次に，arcrole 属性の値を取得し関係の意味（親から子，子から親の別）を調べる．最後に，項目要素同士を関連づける．また，リンクの上書きの処理を行うために，priority 属性，use 属性が存在しているなら登録する．
3. 最後に項目要素同士の関係を保存する．この際にリンクの上書きについて調べる．すでに，2. で見つかった項目要素の関係が登録されている場合は，登録されている priority の値を調べ，2. で見つかった関係の priority の値よりも小さいならば，すでに登録されている関係を削除し，2. で見つかった関係を登録する．

他のリンクベース解析の場合も同様に行う．計算アークは weight 属性が，表示アークは order 属性を持つのでアークの解析結果とともに保存しておく．

5.3 LMX 言語処理系

LMX の処理系は，パーサとインタプリタからなる．LMX プログラムの解釈実行は次の順に行われる．

1. パーサがプログラムの構文解析を行い，AST を作成する
2. インタプリタが AST をトラバースして，プログラムの解釈実行をする

処理系の構成図を図 11 に示す．LMX の解釈実行においては，関数を登録する関数表と，変数を登録する変数表に情報を取得，登録しながら実行を行う．関数表は，関数名から関数本体の AST を取得するためのハッシュテーブルである．変数表は，変数名からその値を取得するためのハッシュテーブルであり，スコープを実現するため，ハッシュテーブルのリンクリストになっている．

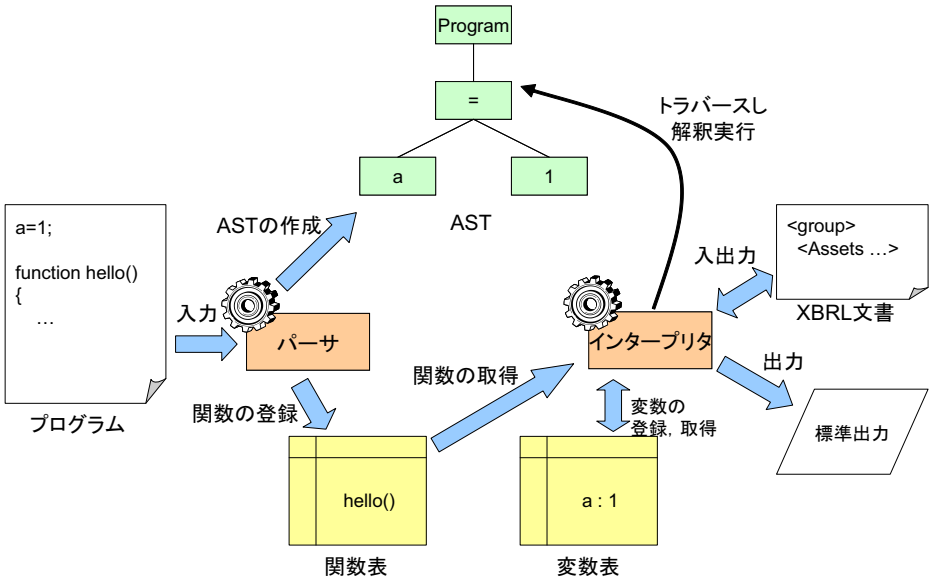

図 11: 解釈実行の構成

1. プログラムの構文解析と AST の作成

LMX のパーサは，JavaCC と JJTree を利用した．JJTree とは，構文定義ファイルを与えると AST（抽象構文木）を作成するツールであり，JavaCC に同梱されている．

入力プログラムをパーサに与えると，JJTree が AST を作成する．AST を作成する過程で関数宣言が見つかると，関数名と関数本体の AST を関数表に登録する．

2. AST をトラバースによるプログラムの解釈実行

インタプリタは Visitor デザインパターン [6] を採用し，実装した．Visitor が AST の頂点を訪れ必要な処理を実行する．

解釈実行処理では，AST をトラバースし，変数への値の代入が見つかれば変数表に値を登録し，変数への参照が見つかれば，変数表から値を取得する．制御構造は条件分岐に応じて，トラバースする AST を変更することで望みの動作を実行する．関数の呼び出しが見つかった場合は，新たなスコープに入るため，変数表に新しいハッシュテーブルを作成し，リンクリストに登録する．そして，関数表から取得した関数本体の AST に実行を移す．関数では，新しく追加されたハッシュテーブルに対して，変数の登録，参照を行う．

エラーには構文エラーと，実行時エラーがある．構文エラーは，パーサが発見する構文上の間違いによるエラーである．実行時エラーは，相互に変換できない型同士で演算を行おうとしている場合など，解釈実行の際に起こるエラーである．どちらも回復不可能なエラーのため，解釈実行は，エラーが見つかった時点で停止する．

5.4 GUI – XBRL Documents Manipulation Environment

プログラムの作成，実行，デバグといった一連の作業を支援するための GUI として，XBRL Documents Manipulation Environment を作成した．本節では，GUI が持つ機能を説明する．スクリーンショットを図 12 に，各部の名称を図 13 に示す．

メニューは，File，Edit，Script，Help メニューからなる．File メニューには，New，Open，Save，SaveAs，Exit という項目があり，それぞれ，プログラムを新規作成する，ファイルから読み込む，ファイルに保存する，名前を付けてファイルに保存する，GUI を終了するという機能を実行する．Edit メニューには，Undo，Redo，Cut，Copy，Paste，Select all という項目があり，それぞれ，元に戻す，元に戻すを取り消す，プログラム編集ペインの選択文字列の切り取り，コピー，貼り付け，そして，プログラム編集ペインの文字列を全て選択するという機能を実行する．Script メニューには Run という項目があり，この項目を選択すると，プログラム編集ペインに記述されたプログラムを実行を開始する．Help メニューには，Set font size... と About... という項目がある．Set font size... 項目を選択すると，図 14 に示すダイアログが表示され，フォントサイズの入力を促し，全てのペインのフォントサイズを指定された値に変更する．About... 項目を選択すると，図 15 に示すバージョン情報ダイアログを表示し，GUI の情報をユーザに提示する．

ツールバーからもメニューの一部の機能を呼び出せる．

プログラム編集ペインでは，実行対象となるプログラムの編集を行う．このペインでマウスの右ボタンをクリックすると，コンテキストメニューが表示され，元に戻す，元に戻すの取り消し機能や，選択文字列の切り取り，コピー，そして，文字列の貼り付け機能を実行す

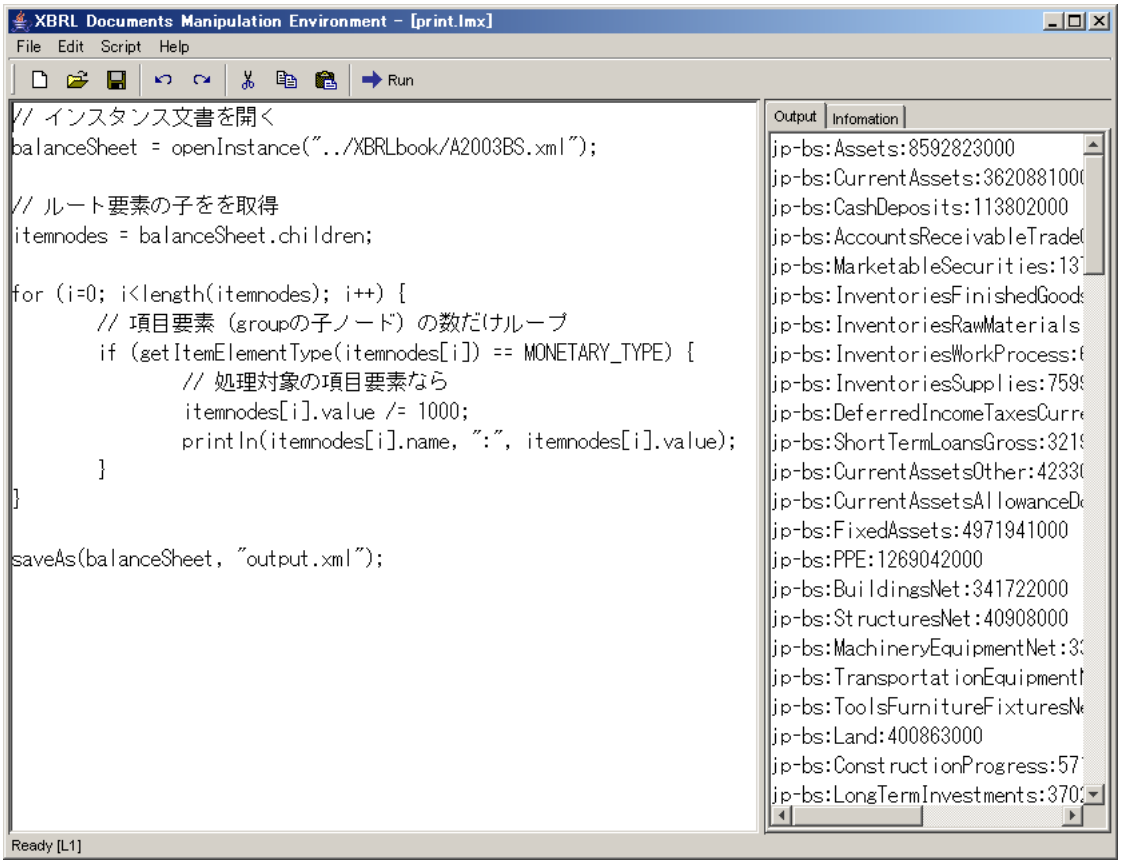

図 12: XBRL Documents Manipulation Environment

ることができる．

出力，エラー情報表示ペインには，出力，エラー情報切り替えタブの状態により，出力，もしくは，エラー情報が表示される．出力とは，print，println 関数によりプログラムが実行時に出力した文字列である．エラー情報とは，プログラムの構文エラーや，実行時エラーにより，プログラムの実行が中断された場合，その原因を示す情報である．エラー情報切り替えタブの “Output” タブが選択されている場合は出力が，“Information” タブが選択されている場合はエラー情報が表示される．このペインでマウスの右ボタンをクリックすると，コンテキストメニューが表示され，ペインに表示された文字列をファイルに保存，ペインに表示された文字列のコピー，ペイン内の文字列の全消去を行うことができる．

ステータスバーには，プログラムを実行している時は，Running という文字列が，実行していないときは Ready という文字列が表示され，利用者は GUI の状態を把握することができる．また，状態表示の右にはプログラム編集ペインのカーソルがある行番号が表示され，エラーが起きた箇所を容易に知ることができる．

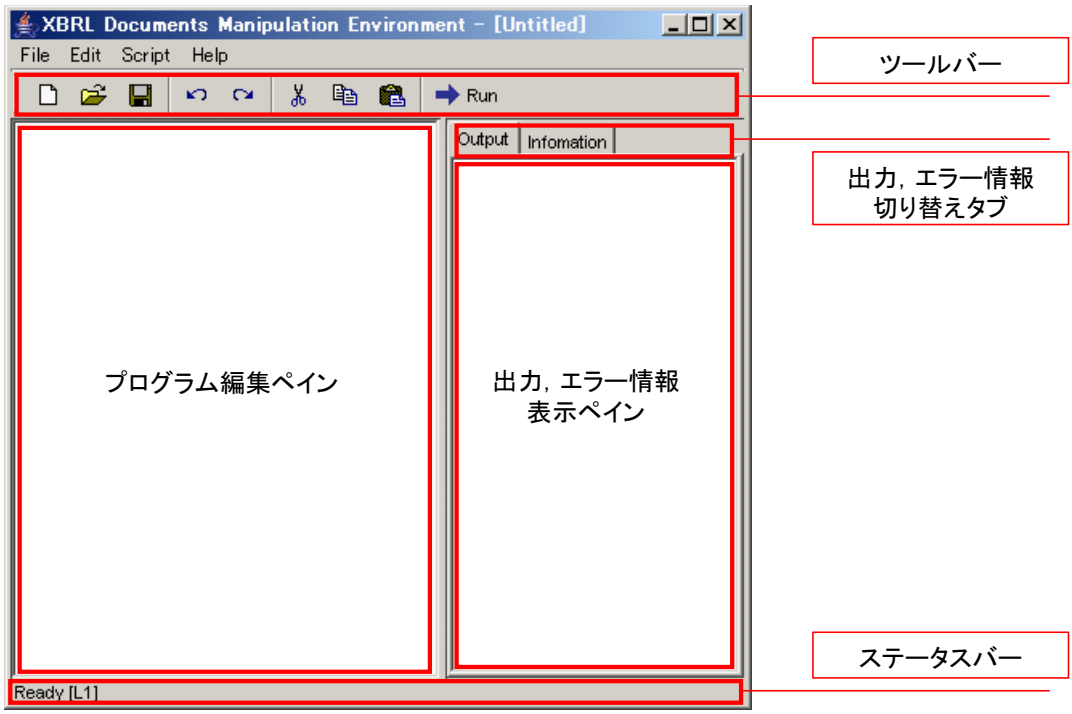

図 13: 各部の名称

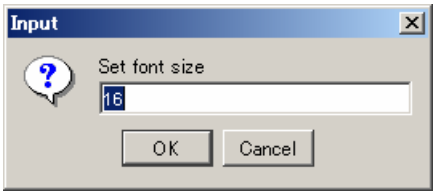

図 14: フォントサイズ入力ダイアログ

図 15: バージョン情報ダイアログ

6 事例研究

頻繁に行われる XBRL 文書処理を LMX と既存の言語で行い，両者のプログラムを比較検討する．これより，一般的によく行われる XBRL 文書処理である，次の 2 種類の処理について考える．

1. 単位の変換
2. フォーマットの変換

6.1 単位の変換

p この事例では，100 万円単位でかかれた情報を，1000 円単位でかかれた情報に直す処理を考える．具体的には，全ての項目要素の値について 1000 を掛けるという計算を行う．この処理は金融機関などで頻繁に行われる．

計算処理が主になるので，汎用のスクリプト言語である ECMAScript [4] との比較を行う．ECMAScript とは，ECMA（European Computer Manufacturer Association）が策定した，一般に JavaScript[9] として知られている言語仕様である．

ECMAScript から XML 文書を操作する方法として，一般的に DOM が使われている [2]. 本節では，ECMAScript による DOM を使った操作と，3.3 節で述べた，XBRL 文書のモデルを使った操作とを比較する．ただし，公平のために，LMX で使うことのできる XBRL 文書の処理関数を ECMAScript でも利用できることを仮定している．

```
1  // インスタンス文書を開く
2  balanceSheet = openInstance("A2003BS.xml");
3
4  // ルート要素の子ををを取得
5  itemnodes = balanceSheet.children;
6
7  for (i=0; i<length(itemnodes); i++) {
8      // 項目要素（groupの子ノード）の数だけループ
9      if (getItemElementType(itemnodes[i]) == MONETARY_TYPE) {
10         // 処理対象の項目要素なら
11         itemnodes[i].value *= 1000;
12     }
13 }
14
15 saveAs("output.xml");
```

図 16: LMX によるプログラム

LMX によるプログラムを図 16 に，ECMAScript によるプログラムを図 17 に載せる．説明のために行番号を付けているが，実際のプログラムに行番号は必要ない．ECMAScript の

```
 1  objXML = WScript.CreateObject("MSXML.DOMDocument");
 2  objXML.load("A2003BS.xml");
 3
 4  // ルート要素を取得
 5  groupnode = objXML.getElementsByTagName("xbrli:group");
 6  itemnodes = groupnode[0].childNodes;
 7
 8  // group要素の子ノードの数だけループ
 9  for (i=0; i<itemnodes.length; i++) {
10      if (itemnodes[i].nodeType == 1 /* Node.ELEMENT_NODE */) {
11          // ノードが要素だったら
12
13          if (getItemElementType(itemnodes[i]) != MONETARY_ITEM_TYPE) {
14              // 処理対象の項目要素でなければ，次のループ
15              continue
16          }
17
18          itemchildren = itemnodes[i].childNodes;
19          for (j=0; j<itemchildren.length; j++) {
20              // 項目要素の子ノードの数だけループ
21              if (itemchildren[j].nodeType == 3) { // Node.TEXT_NODE
22                  // 項目要素の子供がテキストノードだったら
23                  itemchildren[j].nodeValue *= 1000;
24                  break;
25              }
26          }
27      }
28  }
29
30  objXML.save("output.xml");
```

図 17: ECMAScript によるプログラム

広く使われている実装として，Microsoft による JScript[8] を選んだため，XBRL 文書のファイルとの入出力は JScript 特有の記法となっている．

LMX を使った場合に行うべき手順は次のとおりである．まず，XBRL 文書を開き，ルート要素の子要素を取得する（5 行目）．そして，子要素それぞれについて繰り返し処理を行い，要素が項目要素ならば，その値を書き換える（7〜13 行）．最後に XBRL 文書をファイルに保存する．

対して，ECMAScript を使った DOM 操作を行う場合は次のような手順となる．まずは，XBRL 文書を開き，ルート要素である gruop 要素を取得する（5 行目）．次に項目要素を処理するために，group 要素の子ノードの数だけ繰り返し処理を行う（9〜28 行）．group 要素の子ノードが要素ならば（10 行目），項目要素かどうかを確かめその子ノードを取得する（18 行目）．そして，その子ノードの中からテキストノードを見つけ，値の計算を行う（23 行目）．

この処理量の差は，プログラム中の for 文によるループの数，if 文による分岐の数に如実に表れており，ECMAScript 版は LMX 版の 2 倍のネストの深さになっている．このような

差が現れた原因は，ECMAScript が扱うモデルである DOM と，3 節で述べた XBRL 文書のモデルの差にある．有向グラフで表現された XBRL 文書では，グラフ中のノードは DOM とは異なり必ず要素となる．そのため，ノードが要素かそうでないかという判定処理を省くことができる．同様に，要素の値もノードのメンバとして読み書きできるため，子ノードからテキストノードを選び出すという処理を省くことができる．このような XBRL 文書のモデルの扱いやすさにより，同一の処理を少ないコード量で記述することが可能になったと考えることができる．

6.2 書式変換

XBRL 文書に書かれた情報を，HTML で記述された文書に書式変換する処理を考える．XBRL は HTML などの他の XML で記述される形式に変換されることを想定して設計されており，書式の変換処理は頻繁に行われる．この事例では，XML の書式変換に用いられる XSLT との比較検討を行う．XSLT とは，XML によって記述された文書を他の XML 文書に変換するための簡易言語である．

この事例では，貸借対照表を記述した XBRL 文書を HTML の表に変換する．貸借対照表は，資産の部，負債の部，資本の部から構成されるが，この例では，資産の部と，その内訳を適切にインデントして表示することを考える．表示のための手順は次のようになる．資産の部は，Assets 要素として XBRL 文書に記述されるため，まず，Assets 要素を取得する．次に，表示リンクで定義された子要素を取得し，インデントして出力する．要素に表示リンクで定義された子要素がある限り，繰り返しインデントして出力する．このような再帰的な処理を行うことで，XBRL 文書を HTML 文書に変換することができる．

LMX によるプログラムを図 18 に，XSLT によるプログラムを図 19 に載せる．6.1 の場合と同様，LMX で使うことのできる XBRL 文書の処理関数を XSLT でも利用できることを仮定している．同様に，説明のために行番号を付けているが，実際のプログラムは行番号は必要ない．また，これらのプログラムの入力となる XBRL 文書の例を図 20 に，生成された HTML 文書を図 21 に，そして，HTML 文書を web ブラウザで表示させた例を図 22 に載せる．

LMX によるプログラムでは，2 行目で Assets 要素を取得し，Assets 要素が見つかった数だけ書式変換を開始する（5〜8 行のループ）．書式変換は formatXBRL 関数で行う．15 行目で，名称リンクで定義された項目要素の表示名を取得する．この例では日本語（“ja”）の表示名を取得している．17〜21 行目では HTML として出力する．24 行目で，表示リンクで定義された子要素を取得し，子要素の数だけ formatXBRL 関数を再帰呼び出しする．再帰的な処理を行うことで，Assets 要素の子要素全てを出力する．

```
1  balanceSheet = openInstance("A2003BS.xml");
2  assets = getItemElementsByName(balanceSheet, "Assets");
3
4  println("<TABLE>");
5  for (i=0; i<length(assets); i++) {
6      // Assets要素を起点に表示を開始する
7      formatXBRL(assets[i], 0);
8  }
9  println("</TABLE>");
10
11 // 表示リンクによる子要素をHTMLの表で出力する
12 function formatXBRL(node, indent)
13 {
14     // ラベルの取得
15     label = getLabel(node, "ja");
16
17     print("<TR>");
18     print("<TD style='padding-left:", indent, "px'>");
19     print(label, "</TD>");
20     print("<TD align='right'>", node.value, "</TD>");
21     println("</TR>");
22
23     // 子要素の数だけ再帰呼び出し
24     children = getPresentationChildren(node);
25     for (i=0; i<length(children); i++) {
26         formatXBRL(children[i], indent+20);
27     }
28 }
```

図 18: LMX によるプログラム

XSLT は処理対象の XML ファイルを先頭から読み，要素名とマッチするテンプレートを適用するという考え方で処理を進める．この例では，Assets 要素が現れたら 7〜14 行のテンプレートにマッチし，HTML に変換するためのテンプレート printchild を引数付きで適用する．printchild（17〜37 行）では 21〜28 行目で HTML の出力を行う．23 行目では，getLabel 関数を呼び出し名称リンクで定義された表示名を出力する．31〜37 行目では getPresentationChildren 関数を呼び出し表示リンクで定義された子要素を取得し，その数だけ printchild を再び適用する．再帰的にテンプレートを適用することで，Assets 要素の子要素を全て出力する．

構造化プログラミング言語として設計した LMX と，関数型言語の側面を持つ XSLT は，プログラミングの考え方が異なるために行数などによる単純な比較はできない．本節では，プログラムの書きやすさ，読みやすさについて論ずる．XSLT はマッチするノードの特定に XPath[16] を用いる．そのため，XSLT のプログラミングには XSLT の習得に加え，XPath の習得が必要であり，多くの前提知識を必要としている．また，現在選択中のノードを表すコンテキストノードの概念（図 19 中の “.”）など構造化プログラミング言語にはない概念を理解する必要がある．それに対し，LMX では，既存の構造化プログラミング言語に慣れた人が簡単に使えるよう設計してある．そのため，LMX を使うことで変換プログラムを作成す

```
1  <?xml version="1.0"?>
2  <xsl:stylesheet version="1.0"
3   xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:jp-bs=
4   "http://www.xbrl-jp.org/taxonomy/jp/fr/common/bs/2003-08-31">
5
6  <!-- Assets要素を見つけたら処理開始 -->
7  <xsl:template match="ja-bs:Assets">
8    <TABLE>
9      <xsl:call-template name="printchild">
10       <xsl:with-param name="node" select="." />
11       <xsl:with-param name="indent" select="0" />
12     </xsl:call-template>
13   </TABLE>
14 </xsl:template>
15
16 <!-- 表示リンクによる子要素を出力するためのテンプレート -->
17 <xsl:template name="printchild">
18 <xsl:param name="node" />
19 <xsl:param name="indent" />
20
21   <TR>
22     <TD style="padding-left:{$indent}pt">
23       <xsl:value-of select="getLabel($node)"/>
24     </TD>
25     <TD align="right">
26       <xsl:value-of select="$node/text()"/>
27     </TD>
28   </TR>
29
30   <!-- 子要素の数だけ再帰呼び出し -->
31   <xsl:for-each select="getPresentationChildren($node)">
32     <xsl:call-template name="printchild">
33       <xsl:with-param name="node" select="." />
34       <xsl:with-param name="indent" select="$indent+20" />
35     </xsl:call-template>
36   </xsl:for-each>
37 </xsl:template>
38
39 </xsl:stylesheet>
```

図 19: XSLT によるプログラム

る手間を削減できると考えられる．

6.3 考察

6.1 節，6.2 節で，既存のプログラミング言語と LMX を比較し，LMX はより簡単に分かりやすくプログラムを記述できることを確認した．XBRL の有向グラフによるモデルを導入したことで，既存の DOM よりも直感的にプログラムを記述することができたと考えられる．さらに，C 言語などの既存のプログラミング言語に近い構文や概念を採用することで，言語習得にかかる手間も省くことができると考えられる．

また，LMX の利用方法としては，XBRL 文書の計算や，書式変換の他に，XBRL を学習

```
<?xml version="1.0" ?>
<xbrli:group
    xmlns:xbrli="http://www.xbrl.org/2001/instance"
    xmlns:jp-bs="http://www.xbrl-jp.org/taxonomy/jp/fr/common/bs/2003-08-31"
    xmlns:jp-gcd="http://www.xbrl-jp.org/taxonomy/jp/gcd/2003-08-31"
    xsi:schemaLocation="http://www.xbrl-jp.org/taxonomy/jp/fr/common/bs/2003-08-31
                        jp-bs-2003-08-31.xsd
                        http://www.xbrl-jp.org/taxonomy/jp/gcd/2003-08-31
                        jp-gcd-2003-08-31.xsd">
  <jp-gcd:EntityName xml:lang="ja" nonNumericContext="s1">A株式会社</jp-gcd:EntityName>
  <jp-gcd:EntityName xml:lang="en" nonNumericContext="s1">A Company</jp-gcd:EntityName>
  <jp-bs:BalanceSheetDate nonNumericContext="s1">2003-03-31</jp-bs:BalanceSheetDate>
  <jp-bs:BalanceSheetUnit xml:lang="ja" nonNumericContext="s1">百万円</jp-bs:BalanceSheetUnit>
  <jp-bs:BalanceSheetUnit xml:lang="en" nonNumericContext="s1">Million Yen</jp-bs:BalanceSheetUnit>
  <jp-bs:Assets numericContext="c1">8592823000000</jp-bs:Assets>
  <jp-bs:CurrentAssets numericContext="c1">3620881000000</jp-bs:CurrentAssets>
  <jp-bs:CashDeposits numericContext="c1">113802000000</jp-bs:CashDeposits>
  <jp-bs:AccountsReceivableTradeGross numericContext="c1">919468000000</jp-bs:AccountsReceivableTradeGross>
  <jp-bs:MarketableSecurities numericContext="c1">1373742000000</jp-bs:MarketableSecurities>
  <jp-bs:InventoriesFinishedGoods numericContext="c1">140516000000</jp-bs:InventoriesFinishedGoods>
  <jp-bs:InventoriesRawMaterials numericContext="c1">13807000000</jp-bs:InventoriesRawMaterials>
  <jp-bs:InventoriesWorkProcess numericContext="c1">64881000000</jp-bs:InventoriesWorkProcess>
  <jp-bs:InventoriesSupplies numericContext="c1">7599000000</jp-bs:InventoriesSupplies>

    . . .
```

図 20: 変換に使う XBRL 文書

している人が XBRL 文書の内容を解析し理解を深めるという使い方が想定できる．また，XBRL のためのシステムを設計開発しているエンジニアも，LMX を使うことで対象としている XBRL 文書を容易に読み書きし，システムの開発に役立てることができると考えられる．

```
<TABLE>
<TR><TD style="padding-left:0px">資産合計</TD><TD align="right">8592823000000</TD></TR>
<TR><TD style="padding-left:20px">流動資産合計</TD><TD align="right">3620881000000</TD></TR>
<TR><TD style="padding-left:40px">現金及び預金</TD><TD align="right">113802000000</TD></TR>
<TR><TD style="padding-left:40px">売掛金</TD><TD align="right">919468000000</TD></TR>
<TR><TD style="padding-left:40px">有価証券</TD><TD align="right">1373742000000</TD></TR>
<TR><TD style="padding-left:40px">製品</TD><TD align="right">140516000000</TD></TR>
<TR><TD style="padding-left:40px">原材料</TD><TD align="right">13807000000</TD></TR>
<TR><TD style="padding-left:40px">仕掛品</TD><TD align="right">64881000000</TD></TR>
<TR><TD style="padding-left:40px">貯蔵品</TD><TD align="right">7599000000</TD></TR>
<TR><TD style="padding-left:40px">繰延税金資産</TD><TD align="right">250469000000</TD></TR>
<TR><TD style="padding-left:40px">短期貸付金</TD><TD align="right">321986000000</TD></TR>
<TR><TD style="padding-left:40px">その他の流動資産</TD><TD align="right">423307000000</TD></TR>
<TR><TD style="padding-left:40px">貸倒引当金</TD><TD align="right">-8700000000</TD></TR>
<TR><TD style="padding-left:20px">固定資産合計</TD><TD align="right">4971941000000</TD></TR>
<TR><TD style="padding-left:40px">有形固定資産合計</TD><TD align="right">1269042000000</TD></TR>
<TR><TD style="padding-left:60px">建物(純額)</TD><TD align="right">341722000000</TD></TR>

   . . .

</TABLE>
```

図 21: 生成された HTML の例

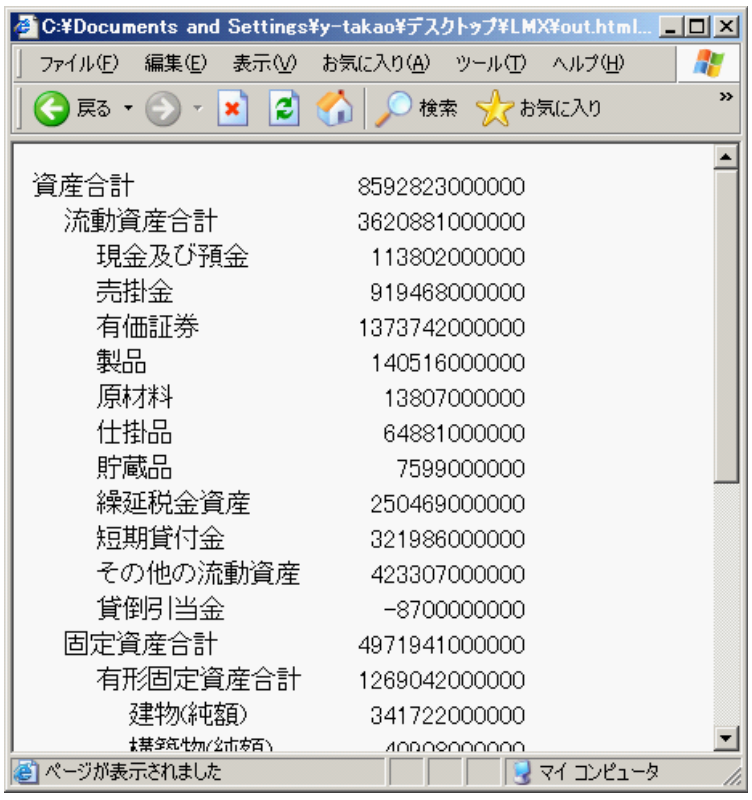

図 22: ウェブブラウザでの表示例

7 まとめ

本研究では，XBRL 文書の構造を利用し DOM を簡略化した，理解しやすい XBRL 文書のモデルを提案した．そして，XBRL 文書に記述された値の計算や，書式変換のための処理を簡単に記述するため，XBRL 文書のモデルを利用したプログラミング言語 LMX を提案した．さらに，プログラミング言語のインタプリタと，GUI として開発環境を実装した．既存の XML 処理系と LMX との比較検討の結果，LMX はより簡単なプログラムを記述できることを確かめた．

また今後の課題として，次が考えられる．

- プログラムの型チェック
  XBRL では，本研究で利用した項目間の関係の他に，財務情報を扱うための型が定義してある．財務情報の型を利用し，プログラム上で演算の型チェックを行う拡張が考えられる．

- GUI を用いたプログラムの自動生成
  プログラムを記述しなくても XBRL 文書の計算，変換を行うため，GUI を操作することでプログラムを自動生成するシステムの開発が考えられる．

謝辞

本研究の全過程を通して，常に適切なご指導および御助言を賜りました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に心より深く感謝致します.

本論文を作成するに当たり，逐次適切なご指導およびご助言を賜りました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻 楠本 真二 助教授に心から感謝致します.

本論文を作成するに当たり，適切なご指導およびご助言を賜りました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 助手に心から感謝致します.

本論文の作成において，適切なご助言を頂きました 株式会社日立製作所 湯浦 克彦 氏に心から感謝致します.

本論文の作成において，適切なご助言を頂きました 大阪市立大学大学院経営学研究科 坂上 学 助教授に心から感謝致します.

最後に，その他様々なご指導，御助言等を頂いた 大阪大学大学院情報科学研究科コンピュータサイエンス専攻　井上研究室の皆様に深く感謝致します.

参考文献

[1] The Apache Software Foundation, “Apache Ant”, http://ant.apache.org/

[2] David Flanagan, “JavaScript: The Definitive Guide, 4th Edition”, O’Reilly, 2001.

[3] David Megginson, “SAX”, http://www.saxproject.org/

[4] Ecma International, “ECMAScript Language Specification”, http://www.ecma-international.org/publications/standards/Ecma-262.htm

[5] Elliotte Rusty Harold, W. Scott Means 著, 瀬尾 明志 訳, “XML クイックリファレンス 第 2 版”, オライリー・ジャパン, 2002.

[6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides 著, 本位田 真一, 吉田 和樹 訳, “オブジェクト指向における再利用のためのデザインパターン 改訂版”, ソフトバンクパブリッシング, 1999.

[7] Haruo Hosoya and Benjamin C. Pierce. “XDuce: A typed XML processing language”. ACM Transactions on Internet Technology, 3(2):117-148, 2003.

[8] Microsoft Corporation, “Windows Script”, http://msdn.microsoft.com/scripting/

[9] Netscape Communications Corporation, “JavaScript 1.1 Language Speccation”, http://www.netscape.com/eng/javascript/index.html

[10] sreeni, “Java Compiler Compiler [tm] (JavaCC [tm]) - The Java Parser Generator”, https://javacc.dev.java.net/

[11] Sun microsystems Inc., “Java API for XML Processing (JAXP)”, http://java.sun.com/xml/jaxp

[12] Sun Microsystems Inc., “Java Technology”, http://java.sun.com/

[13] The World Wide Web Consortium, “Document Object Model (DOM)”, http://www.w3.org/DOM

[14] The World Wide Web Consortium, “Extensible Markup Language (XML)”, http://www.w3.org/XML

[15] The World Wide Web Consortium, “XML Linkink Language (XLink)”, http://www.w3.org/TR/xlink

[16] The World Wide Web Consortium, “XML Path Language (XPath)”, http://www.w3.org/TR/xpath

[17] The World Wide Web Consortium, “XML Schema”, http://www.w3.org/XML/Schema

[18] The World Wide Web Consortium, “XSL Transformations (XSLT)”, http://www.w3.org/TR/xslt

[19] XBRL International, “XBRL”, http://www.xbrl.org/

[20] XBRL International, “XBRL Specifications”, http://www.xbrl.org/resourcecenter/specifications.asp

[21] XBRL Japan, “XBRL Japan”, http://www.xbrl-jp.org/

[22] XBRL Japan, “税務申告用財務諸表 XBRL タクソノミ”, http://www.xbrl-jp.org/taxonomy/2003/jp-ta-2003-08-31.zip

[23] XBRL Japan マーケット・アンド・コミュニケーション（“マーコム”）委員会, “XBRL FACT BOOK”, 2003.

[24] 金融庁, “EDINET 証券取引法に基づく有価証券報告書等の開示書類に関する電子開示システム”, http://info.edinet.go.jp/

[25] 坂上 学, 白田 佳子著, XBRL Japan 監修, “XBRL による財務諸表作成マニュアル”, 日本経済新聞社, 2003.

[26] 高尾 祐治, 渡辺 貴史, 松下 誠, 井上 克郎, 湯浦 克彦, “項目間の対応関係を用いた XBRL 財務報告書自動変換手法の提案”, ソフトウェア・シンポジウム 2003 論文集 pp.149-158, 2003.

[27] 渡辺 貴史, “項目間の対応関係を用いた XBRL 財務報告書自動変換ツールの試作”, 大阪大学 基礎工学部情報科学科 特別研究報告, 2003.

付録

A. API リファレンス

B. 文法定義（BNF）

A　API リファレンス

これより，LMX 処理系が提供する API の一覧を載せる．LMX で用いる型には，基本型，配列変数を表す Array 型，Node 構造体を表す NodeStruct 型があるが，API リファレンスでは参考のために，基本型のうち，Integer 型（整数），Float 型（浮動小数点），String 型（文字列），Boolean 型（論理型）の区別まで記載する．なお，戻り値の型の指定がない関数は戻り値を返さない関数である．

A.1　XBRL 処理

`Float calculateNodeValue(NodeStruct, Boolean)`

計算リンクを利用して子から値を計算する．
第一引数　計算の起点となる Node 構造体
第二引数　計算リンクで定義された子を再帰的に辿って値を計算するかどうか．false の場合は，計算リンクで定義された子の値を計算するのみ．

`Array getDefinitionChildren(NodeStruct)`

定義リンクで定義された親子関係にある子要素を計算し，子の Node 構造体の配列を返す．
第一引数　計算の起点となる Node 構造体

`Array getDefinitionParents(NodeStruct)`

定義リンクで定義された親子関係にある親要素を計算し，親の Node 構造体の配列を返す．
第一引数　計算の起点となる Node 構造体

`Array getItemElementsByName(NodeStruct, String)`

指定した名前の項目要素の配列（Node 構造体の配列）を返す．
第一引数　同じ文書に含まれる Node 構造体
第二引数　要素名

`Boolean isItemElement(NodeStruct)`

Node 構造体が項目要素かどうか判定する.
第一引数　判定対象の Node 構造体

`Integer getItemElementType(NodeStruct)`

Node 構造体が項目要素の場合に，その型（monetary / shares / decimal / string / uri / datetime）を取得する．型を表す定数として，MONETARY_TYPE，SHARES_TYPE，DECIMAL_TYPE，STRING_TYPE，URI_TYPE，DATETIME_TYPE を利用できる.
第一引数　判定対象の Node 構造体

`String getLabel(NodeStruct, String)`

名称リンクで定義された表示名を取得する.
第一引数　判定対象の Node 構造体第一引数　言語．省略された場合は “en” が指定されたものと見なす.

`Array getPresentationChildren(NodeStruct)`

表示リンクで定義された子要素を計算し，子の Node 構造体の配列を返す．配列は，表示リンクで指定された順番でソートされている．第一引数　判定対象の Node 構造体

A.2 入出力

`NodeStruct openInstance(String)`

インスタンス文書を開き，ルート要素の Node 構造体を返す．XBRL 文書でない XML 文書も同様に開くことができる．この場合は，タクソノミの解析は行われない.
第一引数　インスタンス文書のファイル名

`print(...)`

標準出力に引数で指定された値を出力する．引数の数に制限はない.

`println(...)`

標準出力に引数で指定された値を出力し，最後に改行文字を出力する．引数の数に制限はない.

`save(NodeStruct)`

文書に関連付けられたファイルに上書き保存する.
第一引数　保存する文書に含まれる Node 構造体

`saveAs(NodeStruct, String)`

名前をつけてファイルを保存する.
第一引数　保存する文書に含まれる Node 構造体
第一引数　ファイル名

A.3 XML 文書操作

`NodeStruct appendChild(NodeStruct, NodeStruct)`

子リストの末尾に追加する．返り値は，追加されたノード.
第一引数　追加される Node 構造体 (親)
第二引数　子として追加する Node 構造体

`NodeStruct cloneNode(NodeStruct, Boolean)`

Node 構造体の複製を返す.
第一引数　コピーする Node 構造体
第二引数　再帰的にコピーするかどうか

`NodeStruct createCDATASection(NodeStruct, String)`

指定された文字列を値として持つ CDATASection を作成する.
第一引数　同じ XML 文書に属する Node 構造体
第二引数　CDATASection に含まれるデータ

`NodeStruct createComment(NodeStruct, String)`

指定された文字列を持つ Comment を作成する.

第一引数　同じ XML 文書に属する Node 構造体

第二引数　 Comment に含まれるデータ

`NodeStruct createDocumentFragment(NodeStruct)`

空の DocumentFragment を生成する.

第一引数　同じ XML 文書に属する Node 構造体

`NodeStruct createElement(NodeStruct, String)`

指定された名前の要素（Node 構造体）を作成する.

第一引数　同じ XML 文書に属する Node 構造体

第二引数　要素名

`NodeStruct createElementNS(NodeStruct, String, String)`

所定の修飾名と名前空間 URI を持つ要素（Node 構造体）を作成する.

第一引数　同じ XML 文書に属する Node 構造体

第二引数　名前空間 URI

第三引数　要素名 (接頭辞を含めた名前)

`NodeStruct createEntityReference(NodeStruct, String)`

EntityReference を作成する.

第一引数　同じ XML 文書に属する Node 構造体

第二引数　名前

`NodeStruct createProcessingInstruction(NodeStruct, String, String)`

指定された名前およびデータを持つ ProcessingInstruction を作成する.

第一引数　同じ XML 文書に属する Node 構造体

第二引数　名前

第三引数　データ

`NodeStruct createTextNode(NodeStruct, String)`

指定された文字列を持つ Text を作成する.

第一引数　同じ XML 文書に属する Node 構造体

第二引数　データ

`String getAttribute(NodeStruct, String)`

名前を指定して属性値を取得する

第一引数　対象となる Node 構造体

第二引数　属性の名前

`String getAttributeNS(NodeStruct, String, String)`

名前と名前空間 URI を指定して属性値を取得する.

第一引数　対象となる Node 構造体

第二引数　名前空間 URI

第三引数　属性の名前

`Array getChildNodes(NodeStruct)`

指定された Node 構造体の子をすべて含む配列を返す.

第一引数　処理対象の Node 構造体

`NodeStruct getDoctype(NodeStruct)`

文書に関連付けられた文書タイプ宣言を返す.

第一引数　対象となる Node 構造体

`NodeStruct getDocumentElement(NodeStruct)`

文書のルート要素になっている Node 構造体を返す.

第一引数　対象となる Node 構造体

`Array getElementsByTagName(NodeStruct, String)`

指定された要素名を持つ Node 構造体の配列を返す.

第一引数　検索基準となる Node 構造体

第二引数　要素名

```
Array getElementsByTagNameNS(NodeStruct, String, String)
```

指定された要素名を持つ Node 構造体の配列を返す.

第一引数　検索基準とな Node 構造体

第二引数　名前空間 URI

第三引数　要素名

```
NodeStruct getFirstChild(NodeStruct)
```

最初の子ノードを返す.

第一引数　対象となる Node 構造体

```
NodeStruct getLastChild(NodeStruct)
```

最後の子ノードを返す.

第一引数　対象となる Node 構造体

```
String getLocalName(NodeStruct)
```

修飾名のローカル部分を返す.

第一引数　対象となる Node 構造体

```
String getNamespaceURI(NodeStruct)
```

名前空間 URI を返す.

第一引数　対象となる Node 構造体

```
NodeStruct getNextSibling(NodeStruct)
```

直後の兄弟ノードを返す.

第一引数　対象となる Node 構造体

`String getNodeName(NodeStruct)`

ノード名を返す.

第一引数　対象となる Node 構造体

`Integer getNodeType(NodeStruct)`

ノードの型を表すコードを返す.

第一引数　対象となる Node 構造体

`String getNodeValue(NodeStruct)`

ノードの値を返す.

第一引数　対象となる Node 構造体

`NodeStruct getOwnerDocument(NodeStruct)`

ノードに関連付けられた Document を返す.

第一引数　対象となる Node 構造体

`NodeStruct getParentNode(NodeStruct)`

親ノードを返す.

第一引数　対象となる Node 構造体

`String getPrefix(NodeStruct)`

ノードの名前空間前置修飾子を返す.

第一引数　対象となる Node 構造体

`NodeStruct getPreviousSibling(NodeStruct)`

直前の兄弟ノードを返す.

第一引数　対象となる Node 構造体

`Boolean hasAttribute(NodeStruct, String)`

指定した名前の属性があるかどうか調べる.

第一引数　対象となる Node 構造体

第二引数　属性名

`Boolean hasAttributeNS(NodeStruct, String, String)`

名前空間 URI と名前を指定して属性があるかどうか調べる.

第一引数　対象となる Node 構造体

第二引数　名前空間 URI

第三引数　属性名

`Boolean hasAttributes(NodeStruct)`

属性があるかどうか調べる.

第一引数　対象となる Node 構造体

`Boolean hasChildNodes(NodeStruct)`

子ノードがあるかどうか調べる.

第一引数　対象となる Node 構造体

`NodeStruct importNode(NodeStruct, NodeStruct, Boolean)`

他の文書から現在の文書へノードをインポートする.

第一引数　インポート先の文書に属する NodeStruct

第二引数　インポートする NodeStruct

第三引数　再帰的インポートするか

`NodeStruct insertBefore(NodeStruct, NodeStruct, NodeStruct)`

既存の子ノード refChild の前に newChild ノードを挿入する

第一引数　この NodeStruct

第二引数　新しい子 Node 構造体（newChild）

第三引数　既存の子 Node 構造体（refChild）

`NodeStruct newDocument(String)`

新しい XML 文書を作って，ルートの Node 構造体を返す.

第一引数　要素名

`removeAttribute(NodeStruct, String)`

名前を指定して属性を削除する.

第一引数　要素を保持する Node 構造体

第二引数　削除する属性の名前

`removeAttributeNS(NodeStruct, String, String)`

ローカル名と名前空間 URI を指定して属性を削除する.

第一引数　要素を保持する Node 構造体

第二引数　名前空間 URI

第三引数　削除する属性の名前（ローカル名）

`NodeStruct removeChild(NodeStruct, NodeStruct)`

子リストから第二引数で指定する子ノードを削除し、この子ノードを返す.

第一引数　対象となる Node 構造体

第二引数　削除する Node 構造体

`NodeStruct replaceChild(NodeStruct, NodeStruct, NodeStruct)`

子リストの中の子ノード oldChild を newChild で置き換え， oldChild ノードを返す.

第一引数　対象となる Node 構造体

第二引数　新しい子 Node 構造体（newChild）

第三引数　置き換える子 Node 構造体（oldChild）

`setAttribute(NodeStruct, String, String/Integer/Float)`

属性に値を設定する.
第一引数　対象となる Node 構造体
第二引数　属性の名前
第三引数　属性の値

`setAttributeNS(NodeStruct, String, String, String/Integer/Float)`

名前空間 URI を指定して属性に値を設定する.
第一引数　対象となる Node 構造体
第二引数　名前空間 URI
第三引数　属性の名前
第四引数　属性の値

`setNodeValue(NodeStruct, String/Integer/Float)`

要素に値を設定する.
第一引数　対象となる Node 構造体
第二引数　新しい値

`String setPrefix(NodeStruct, String)`

ノードの名前空間前置修飾子を返す.
第一引数　対象となる Node 構造体
第二引数　 prefix

A.4　その他

`Array array(Integer)`

指定された長さを持つ配列変数を作る.
第一引数　配列変数の長さ

`Integer length(Array)`

配列変数の長さを求める.
第一引数　配列変数

`Integer toInteger(Integer/String/Float/Boolean)`

整数型に変換する．

`Integer toFloat(Integer/String/Float/Boolean)`

浮動小数点型に変換する．

`Integer toString(Integer/String/Float/Boolean)`

文字列型に変換する．

`Integer toBoolean(Integer/String/Float/Boolean)`

論理型に変換する．Integer 型，Float 型は，0 なら false，0 以外は true となる．
String 型は，文字列の長さが 0 なら false，0 以上ならば true となる．

B　文法定義（BNF）

```
<Program> ::= { <FunctionDeclaration> | <Statement> }

<FunctionDeclaration> ::= "function" <IDENTIFIER> "(" [<ArgumentDecList>] ")"
                          <Block>

<ArgumentDecList> ::= <IDENTIFIER> {"," <IDENTIFIER>}


<Statement> ::= <Block> |
                <EmptyStatement> |
                <StatementExpression> ";" |
                <IfStatement> |
                <WhileStatement> |
                <DoStatement> |
                <ForStatement> |
                <BreakStatement> |
                <ContinueStatement> |
                <ReturnStatement>

<Block> ::= "{" { <Statement> } "}"

<EmptyStatement> ::= ";"

<StatementExpression> ::= <Assignment> | <UnaryExpression>

<IfStatement> ::= "if" "(" <Expression> ")" <Statement> ["else" <Statement>]

<WhileStatement> ::= "while" "(" <Expression> ")" <Statement>

<DoStatement> ::= "do" <Statement> "while" "(" <Expression> ")" ";"

<ForStatement> ::= "for" "(" [ <ForInit> ] ";"
                             [ <Expression> ] ";"
                             [ <ForUpdate> ] ")"
                   <Statement>

<ForInit> ::= <StatementExpressionList>

<ForUpdate> ::= <StatementExpressionList>

<StatementExpressionList> ::= <StatementExpression>
                              { "," <StatementExpression>}

<BreakStatement> ::= "break" ";"

<ContinueStatement> ::= "continue" ";"

<ReturnStatement> ::= "return" [ <Expression> ] ";"

<Assignment> ::= <Variable> <AssignmentOperator> <Expression>

<AssignmentOperator> ::= "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>="
```

```
<Expression> ::= <Assignment> | <ConditionalExpression>

<ConditionalExpression> ::= <ConditionalOrExpression>
                            ["?" <Expression> ":" <ConditionalExpression>]

<ConditionalOrExpression> ::= <ConditionalAndExpression>
                              {"||" <ConditionalAndExpression>}

<ConditionalAndExpression> ::= <EqualityExpression>
                               {"&&" <EqualityExpression>}

<EqualityExpression> ::= <RelationalExpression>
                         [ ("==" | "!=") <RelationalExpression>]

<RelationalExpression> ::= <ShiftExpression>
                           [ ("<" | ">" | "<=" | ">=") <ShiftExpression>]

<ShiftExpression> ::= <AdditiveExpression>
                      { ("<<" | ">>" ) <AdditiveExpression> }

<AdditiveExpression> ::= <MultiplicativeExpression>
                         { ("+" | "-") <MultiplicativeExpression> }

<MultiplicativeExpression> ::= <UnaryExpression>
                               { ("*" | "/" | "%") <UnaryExpression> }

<UnaryExpression> ::=   "+" <UnaryExpression>
                      | "-" <UnaryExpression>
                      | "!" <UnaryExpression>
                      | "++" <PrimaryExpression>
                      | "--" <PrimaryExpression>
                      | <PostfixExpression>

<PostfixExpression> ::= <PrimaryExpression> [ "++" | "--" ]

<PrimaryExpression> ::= <Literal> | "(" <Expression> ")" | <VariableFunctionStruct>

<Literal> ::= <INTERGER_LITERAL> | <FLOATING_POINT_LITERAL> | <STRING_LITERAL>
            | "true" | "false" | "null"

<VariableFunctionStruct> ::= <VariableFunction> { "." <VariableFunction> }

<VariableFunction> ::= <IDENTIFIER> [<PrimarySuffix>]

<PrimarySuffix> ::= "[" <Expression>"]" | "(" <ArgumentList> ")"

<ArgumentList> ::= <Expression> { "," <Expression> }

<IDENTIFIER> ::= ("A-Z" | "a-z" | "_") {"A-Z" | "a-z" | "_" | "0-9"}
```