リラックスのしかた

村田　真

$Id: howToRELAX.sdoc 1.9 2000/08/26 03:12:38 murata Exp $

目 次

第I部

第一部: RELAX Core

第1章　STEP 0: はじめに

$Id: step0.sdoc 1.10 2000/08/06 08:45:41 murata Exp $

STEP 0 は，RELAX について大まかに説明し，このチュートリアルの読み方を説明します.

1.1　RELAX の概要

RELAX は，XML ベースの言語を記述するための仕様です．たとえば，XHTML 1.0 は XML ベースの言語ですから，RELAX で記述することができます．RELAX による記述を RELAX 文法といいます.

1.1.1　DTD との比較

従来用いられてきた DTD(Document Type Definition) と比べて，RELAX は次のような特徴を備えています.

- XML 文書として記述できる
- 豊富なデータ型を XML Schema Part 2 から借りている
- 名前空間を扱うことができる

1.1.2　RELAX プロセッサ

RELAX プロセッサによって，XML 文書を RELAX 文法と照合することができます．RELAX プロセッサへの入力は，XML 文書と RELAX 文法です．正確には，RELAX プロセッサは XML 文書や RELAX 文法を直接扱うのではなく，XML プロセッサがこれらを処理したあとの出力を扱います.

XML 文書がこの RELAX 文法に照らして合法かどうかを RELAX プロセッサは報告します．RELAX プロセッサは，他にも診断メッセージを出すことがあります．RELAX プロセッサには，これ以外の出力はありません.

1.1.3 RELAXの構成

RELAX は，RELAX Core と RELAX Namespace の二つから成ります．RELAX Core は，一つの名前空間にある要素とその属性を扱います．RELAX Core は，XML Schema Part 2[1]からデータ型を借用しています．RELAX Namespace は，RELAX Core で書かれた複数のモジュールを組み合わせ，複数の名前空間を扱います．現時点では，このチュートリアルは RELAX Core だけを説明しています．

1.1.4 このチュートリアルについて

このチュートリアルは出来るだけ簡単に書かれています．この STEP は例外ですが，他ではふんだんに例を用い，できるだけ具体的に説明しています．

STEP 1 から 10 までは，RELAX Core を扱います．どの STEP まで読んでも，それなりの理解が得られます．STEP 1 だけを読んで RELAX を使い始めることは十分可能です．STEP 6 まで読めば，DTD でできる範囲の機能はすべて修得できます．すべての RELAX プロセッサは，この範囲の機能をサポートします．STEP 7 から 10 は RELAX の独自の機能を説明します．ただし，これらの機能を実装しない RELAX プロセッサも認められています．

説明を簡単にするため，STEP 1, 2 にはわざと不正確な説明をしている点がいくつかあります．正確な説明は，STEP 7, 8 にあります．

1.2 まとめ

RELAX はとても簡単です．いままで DTD を使っていた人ならすぐに使えますし，使っていなかった人にもすぐ修得できます．ぜひ RELAX を使ってみてください．RELAX!

[1] http://www.w3.org/TR/xmlschema-2/

第2章　STEP 1: XML DTDからの移行（パラメタ実体なし）

$Id: step1.sdoc 1.13 2000/08/06 08:47:44 murata Exp $

STEP 1 は，DTD をちょっと知っている人なら，すぐに移行できる範囲の機能です．また，DTD2RELAX コンバータが生成する範囲でもあります．

2.1　モジュールの例

RELAX の感じをつかむため，一つの DTD を RELAX のモジュールとして表現してみます．

まず DTD を示します．`title` 要素の number 属性は整数だけに制限したいのですが，DTD では出来ません．

```
<!ELEMENT doc   (title, para*)>

<!ELEMENT para  (#PCDATA | em)*>

<!ELEMENT title (#PCDATA | em)*>

<!ELEMENT em    (#PCDATA)>

<!ATTLIST para
  class   NMTOKEN #IMPLIED
>

<!ATTLIST title
  class   NMTOKEN #IMPLIED
  number  CDATA   #REQUIRED
>
```

つぎに RELAX モジュールを示します．`number` 属性は整数であると指定されています．

```
<module
      moduleVersion="1.2"
      relaxCoreVersion="1.0"
      targetNamespace=""
```

```
      xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="doc"/>
  </interface>

  <elementRule role="doc">
    <sequence>
      <ref label="title"/>
      <ref label="para" occurs="*"/>
    </sequence>
  </elementRule>

  <elementRule role="para">
    <mixed>
      <ref label="em" occurs="*"/>
    </mixed>
  </elementRule>

  <elementRule role="title">
    <mixed>
      <ref label="em" occurs="*"/>
    </mixed>
  </elementRule>

  <elementRule role="em" type="string"/>

  <tag name="doc"/>

  <tag name="para">
    <attribute name="class" type="NMTOKEN"/>
  </tag>

  <tag name="title">
    <attribute name="class" type="NMTOKEN"/>
    <attribute name="number" required="true" type="integer"/>
  </tag>

  <tag name="em"/>

</module>
```

以下の章では，この例で示した構文を説明していきます.

2.2 module 要素

RELAX 文法は，いくつかのモジュールを組み合わせたものです．名前空間が一つだけで，それほど大規模でない文法なら，一つのモジュールがそのまま RELAX 文法になります．モジュールは**module 要素**によって表現されます.

```
<module
      moduleVersion="1.2"
      relaxCoreVersion="1.0"
      targetNamespace=""
      xmlns="http://www.xml.gr.jp/xmlns/relaxCore">
  ...
</module>
```

moduleVersion 属性は，このモジュールのバージョンを表します．この例では，”1.2”です．

relaxCoreVersion 属性は，RELAX Core 自体のバージョンを表します．現在は必ず”1.0”です．

targetNamespace 属性は，このモジュールが扱う名前空間を指定します．この例では，””を指定しています．

RELAX Core のための名前空間名は，http://www.xml.gr.jp/xmlns/relaxCore です．

2.3 interface 要素

module の先頭はinterface 要素が入ります．一つのモジュールにはinterface 要素が一つだけ存在します．

```
<module
      moduleVersion="1.2"
      relaxCoreVersion="1.0"
      targetNamespace=""
      xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    ...
  </interface>
  ...
</module>
```

2.3.1 export 要素

interface 要素の中にはexport 要素がいくつか入ります．

```
<export label="foo"/>
```

export 要素のlabel 属性の値は，ルートになりうる要素型です．export 要素は複数あっても構いません．

以下の例は，要素型 foo と bar がルートになりうることを表しています．

```
<interface>
  <export label="foo"/>
  <export label="bar"/>
</interface>
```

2.4 要素型宣言

XML でいう要素型宣言 (<!ELEMENT ...>) は，elementRule 要素によって表現されます．elementRule のrole 属性は，要素型の名前を指定します．elementRule は，interface の後にいくつでも書くことができます．

```
<elementRule role="element-type-name">
  ...hedge model...
</elementRule>
```

elememementRule 要素は，**生け垣モデル**を持つことができます．**生け垣**とは，要素（子孫要素も含む）と文字データの並びのことです．生け垣モデルとは，どんな生け垣が許されるかを表す制約条件です．

生け垣モデルには，要素生け垣モデル，データ型参照，混在生け垣モデルがあります．

2.4.1 要素生け垣モデル

要素生け垣モデルは，empty，ref，choice，sequence，none 要素とoccurs 属性によって表現されます．要素生け垣モデルは，子供要素の並びとして何を許容するかを表現します．また，これらの要素の間に適当に空白を挿入することができます．

2.4.2 empty 要素

empty 要素は，空の内容を表します．

次のelementRule を考えます．

```
<elementRule role="foo">
  <empty/>
</elementRule>
```

このelementRuleは，foo要素の内容が空であることを表しています．開始タグの直後に終了タグを置いても，空要素タグを用いても構いません．

```
<foo/>
```

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

XMLのEMPTYとは異なり，開始タグと終了タグの間に空白だけを置くことも許されます．

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

emptyは，後述するchoiceやsequenceの中で使用することができます．このように拡張する理由は，STEP 2[1]で明らかになります．なお，XMLのEMPTYとまったく同じ機能が必要なら，STEP 3[2]にあるデータ型emptyStringを使って下さい．

以下では，要素型foo，foo1，foo2は，emptyを生け垣モデルとして持つelementRuleによって宣言されているものとみなして説明します．

2.4.3 ref要素

ref要素は，要素型を参照します．例えば，<ref label=”foo”/>は，fooという要素型を参照します．

次のelementRuleを考えます．

```
<elementRule role="bar">
  <ref label="foo"/>
</elementRule>
```

このelementRuleは，bar要素の内容はfoo要素であることを示しています．例えば，次のbar要素はこのelementRuleに従っています．

```
<bar><foo/></bar>
```

[1] ./step2.html
[2] step3.html

foo 要素の前後に空白が入っても構いません.

```
<bar>
  <foo/>
</bar>
```

ref 要素は，occurs 属性を持つことができます．値は，”*”, ”+”, ”?”のどれかで，それぞれ 0 回以上の繰り返し，1 回以上の繰り返し，0 回または 1 回の繰り返しを意味します.

occurs 属性に”?”を指定した例を示します.

```
<elementRule role="bar">
  <ref label="foo" occurs="?"/>
</elementRule>
```

このelementRule は，bar 要素の内容はfoo 要素または空であることを示しています.

```
<bar><foo/></bar>
```

```
<bar></bar>
```

foo 要素の前後に空白が入っても構いません．また，空の場合にも空白を入れることができます.

```
<bar>
  <foo/>
</bar>
```

```
<bar>
</bar>
```

2.4.4 choice要素

choice 要素は選択 (XML にある"|") を表します．choice 要素の子供は要素生け垣モデルです．choice もoccurs 属性を持つことが出来ます．

choice を用いたelementRule の例を示します．

```
<elementRule role="bar">
  <choice occurs="+">
    <ref label="foo1"/>
    <ref label="foo2"/>
  </choice>
</elementRule>
```

このelementRule は，bar 要素の内容がfoo1 要素またはfoo2 要素の 1 回以上の繰り返しであることを示しています．

```
<bar><foo2/></bar>
```

```
<bar>
  <foo2/>
</bar>
```

```
<bar>
  <foo1/>
  <foo2/>
  <foo1/>
</bar>
```

2.4.5 sequence要素

sequence 要素は並び (XML にある",") を表します．sequence 要素の子供は要素生け垣モデルです．sequence もoccurs 属性を持つことが出来ます．

sequence を用いたelementRule の例を示します．

```
<elementRule role="bar">
  <sequence occurs="?">
    <ref label="foo1"/>
    <ref label="foo2"/>
  </sequence>
</elementRule>
```

このelementRuleは，barの内容が，foo1要素とfoo2要素の並び，または空であることを示しています.

```
<bar><foo1/><foo2/></bar>
```

```
<bar>
  <foo1/>
  <foo2/></bar>
```

```
<bar/>
```

```
<bar></bar>
```

```
<bar>
  </bar>
```

2.4.6 noneの要素

noneの要素は，RELAXではじめて導入されたもので，何ともマッチしないという生け垣モデルです.

```
<elementRule role="bar">
  <none/>
</elementRule>
```

このelementRuleは，barの内容としてはなにも許されないことを示しています．noneを導入する理由は，STEP 2[3]で明らかになります.

[3] ./step2.html

2.4.7 データ型参照

elementRule のtype 属性によって，データ型を参照する内容モデルを書くことができます．文書中に書かれた文字列は，参照されたデータ型と照合されます．指定できるのは，XML Schema Part 2 で導入されたデータ型の名前か，RELAX が独自に導入したデータ型の名前です．データ型については STEP 3[4]で説明します．

type を用いたelementRule の例を示します．

```
<elementRule role="bar" type="integer"/>
```

このelementRule は，bar 要素の内容は整数を表現する文字列であることを示します．

```
<bar>10</bar>
```

前後に空白を挿入することはできません．すなわち，次の例は許されません．

```
<bar>
  10
</bar>
```

2.4.8 混在生け垣モデル

mixed 要素は，XML にある混在生け垣モデル (#PCDATA|a|b|...|z)*を大幅に拡張したものです．

<mixed> と</mixed> の間には，要素生け垣モデルが書けます．要素生け垣モデルを直接書いたときは，要素の間には空白だけが挿入できたことを思い出して下さい．mixed で囲むことにより，空白だけではなく，どんな文字でも挿入できるようになります．

XML の (#PCDATA | foo1| foo2)*を表現するには，次のように書きます．

```
<elementRule role="bar">
  <mixed>
    <choice occurs="*">
      <ref label="foo1"/>
      <ref label="foo2"/>
    </choice>
  </mixed>
</elementRule>
```

[4] step3.html

このmixed要素に含まれるchoice要素は，foo1とfoo2の0個以上の繰り返しとマッチします．mixedの効果により，これらの要素の間に任意の文字が許されます．したがって，XMLの混在内容モデル(#PCDATA | foo1| foo2)*と等価になります．

生け垣モデル(#PCDATA)を表現するには，二つの方法があります．一つは，データ型stringをtype属性で指定する方法です．もう一つは，mixed要素のなかに，空のラベル列だけとマッチする要素生け垣モデルを記述する方法です．次にその例を示します．

```
<elementRule role="bar" type="string"/>
```

```
<elementRule role="bar">
  <mixed>
    <empty/>
  </mixed>
</elementRule>
```

より進んだ例として，次のelementRuleを考えます．

```
<elementRule role="bar">
  <mixed>
    <sequence>
      <ref label="foo1"/>
      <ref label="foo2"/>
    </sequence>
  </mixed>
</elementRule>
```

<foo1/>と<foo2/>の並びは，mixedの中のsequence要素にマッチします．したがって，次の例はこのelementRuleにより許されています．

```
<bar>Murata<foo1/>Makoto<foo2/>IUJ</bar>
```

次の例のようにCDATAセクションや文字参照が現れても構いません．

```
<bar><![CDATA[Murata]]><foo1/>Mako&#x74;&#x6F;<foo2/>IUJ</bar>
```

2.5 属性リスト宣言

XML でいう属性リスト宣言 (<!ATTLIST ...>) は，tag 要素によって表現されます.

```
<tag name="element-type-name">
  ...list of attribute declarations...
</tag>
```

tag 要素は，子供要素としてattribute 要素をいくつか持つことができます.

```
<tag name="element-type-name">
  <attribute ... />
  <attribute ... />
</tag>
```

一つのattribute 要素は，一つの属性を宣言します．attribute 要素の例を下に示します.

```
<attribute name="age" required="true" type="integer"/>
```

name **属性**の値は，宣言される属性名です．この例では age です.

required **属性**の値としてtrue が指定されれば，この属性は省略できません．required 属性が指定されなければ省略できます．この例では指定されていますから省略できません.

type **属性**は，データ型名を指定します．type 属性が省略された場合は，string というデータ型（任意の文字列）が指定されたものとみなされます.

このattribute 要素だけからなるtag の例を考えます.

```
<tag name="bar">
  <attribute name="age" required="true" type="integer"/>
</tag>
```

次の開始タグはこのtag に従っています.

```
<bar age="39">
```

次の二つの開始タグは従っていません．一つ目の例はage 属性を省いていますし，二つ目の例は値が整数ではありません.

```
<bar>
```

```
<bar age="bu huo">
<!-- "bu huo" means forty years in Chinese.  In Japan,
     it is pronounced as "FUWAKU". -->
```

DTD では，属性を持たない要素型については，属性リスト宣言を書く必要がありませんでした．しかし，RELAX では，属性を持たない場合も空の`tag`を書く必要があります．たとえば要素型`bar`が属性を持たない場合は次のように書きます.

```
<tag name="bar"/>
```

2.6 まとめ

ここまで読んでいただければ，ただちに RELAX を使いはじめることができます．不便を感じなければ，STEP 2 以降の内容を読む必要はありません．ぜひ使ってみて下さい．RELAX!

第3章　STEP 2: XML DTDからの移行（パラメタ実体あり）

$Id: step2.sdoc 1.10 2000/11/01 13:41:12 murata Exp $

STEP 2 では，同じことを何度も繰り返して記述する代わりに，一回だけ記述しておいたものを繰り返して参照するための機構が加わります．XML にあるパラメタ実体に相当します．

3.1　要素内容モデルの中で用いられるパラメタ実体

生け垣モデルを一回だけ記述し，それを繰り返して参照するための機構が**hedgeRule 要素**です．DTD にあるパラメタ実体のうち，内容モデルで使われるものに相当します．

3.1.1　概要

hedgeRule 要素の構文を次に示します．foo はパラメタ実体の名前です．

```
<hedgeRule label="foo">
  ...element content model...
</hedgeRule>
```

このように定義したhedgeRule を参照するには，<hedgeRef label="foo"/> と書きます．このhedgeRef 要素は，hedgeRule の中で指定された要素生け垣モデルで置き換えられます．

以下の例では，要素型doc のためのelementRule の生け垣モデルからhedgeRule を参照しています．このelementRule は，STEP 1[1]の先頭にあるモジュールにあったものを書き換えて，title 以外の部分をhedgeRule で記述したものです．

```
<hedgeRule label="doc.body">
  <ref label="para" occurs="*"/>
</hedgeRule>
```

[1] ./step1.html

```
<elementRule role="doc">
  <sequence>
    <ref label="title"/>
    <hedgeRef label="doc.body"/>
  </sequence>
</elementRule>
```

doc.body への参照は次のように展開されます.

```
<elementRule role="doc">
  <sequence>
    <ref label="title"/>
    <ref label="para" occurs="*"/>
  </sequence>
</elementRule>
```

この例では，elementRule の中からhedgeRule を参照しましたが，hedgeRule の中からも同様に可能です.

3.1.2 使用できる生け垣モデル

hedgeRule の中には，要素生け垣モデルだけが書けます．データ型参照や混在生け垣モデルは許されません．たとえば，以下の二つはどれも許されません.

```
<hedgeRule label="mixed.param">
  <mixed>
    <choice occurs="*">
      <ref label="em"/>
      <ref label="strong"/>
    <choice>
  </mixed>
</hedgeRule>

<hedgeRule label="string.param" type="string"/>
```

hedgeRef と混在生け垣モデルを併用するときは，hedgeRule の中でmixed を使うのではなく，hedgeRef をmixed で括ってelementRule の中に書きます．その例を次に示します．混在生け垣モデルはphrase を参照しており，phrase はhedgeRule で記述されています.

```
<hedgeRule label="phrase">
  <choice>
    <ref label="em"/>
    <ref label="strong"/>
  <choice>
</hedgeRule>

<elementRule role="p">
  <mixed>
    <hedgeRef label="phrase" occurs="*"/>
  </mixed>
</elementRule>
```

3.1.3 occurs 属性

hedgeRef 要素はoccurs 属性を持つことができますし，hedgeRule の中に書かれる要素生け垣モデルもoccurs 属性を持つことができます．次の例では，両方にoccurs 属性が指定されています．

```
<hedgeRule label="bar">
  <sequence occurs="+" >
    <ref label="foo1"/>
    <ref label="foo2"/>
  </sequence>
</hedgeRule>

<elementRule role="foo">
  <hedgeRef label="bar" occurs="*"/>
</elementRule>
```

この例を DTD で表現すれば，パラメタ実体の展開がどう行われるべきかは明らかです．

```
<!ENTITY % bar "(foo1, foo2)+">
<!-- original --> <!ELEMENT foo (%bar;)*>
<!-- expanded --> <!ELEMENT foo ((foo1, foo2)+)*>
```

上の例を展開した結果を次に示します．展開のときに，一つしか子供を持たないchoice 要素が導入されていることに注意して下さい．これは，hedgeRef 要素にあったoccurs="*"を引き継いでいます．

```
<elementRule role="foo">
  <choice occurs="*">
    <sequence occurs="+" >
      <ref label="foo1"/>
```

```
      <ref label="foo2"/>
    </sequence>
  </choice>
</elementRule>
```

3.1.4 hedgeRef とhedgeRule の順番

DTD にあるパラメタ実体と違い，hedgeRef で参照する前にhedgeRule を書く必要はありません．たとえば，次の記述はエラーではありません．

```
<elementRule role="doc">
  <sequence>
    <ref label="title"/>
    <hedgeRef label="doc.body"/>
  </sequence>
</elementRule>

<hedgeRule label="doc.body">
  <ref label="para" occurs="*"/>
</hedgeRule>
```

3.1.5 自分自身を参照しないこと

自分自身を直接または間接に参照するようなhedgeRule を書いてはいけません．次の例では，bar のための生け垣モデルの中でbar を参照していますからエラーになります．

```
<hedgeRule label="bar">
  <choice>
    <ref label="title"/>
    <hedgeRef label="bar" occurs="*"/>
  </choice>
</hedgeRule>
```

次の例では，bar1 のための生け垣モデルの中でbar2 を参照しており，bar2 のための生け垣モデルの中でbar1 を参照しています．したがって，エラーになります．

```
<hedgeRule label="bar1">
  <hedgeRef label="bar2" occurs="*"/>
</hedgeRule>

<hedgeRule label="bar2">
```

```
  <choice>
    <ref label="title"/>
    <hedgeRef label="bar1"/>
  </choice>
</hedgeRule>
```

3.1.6 empty の用途

STEP 1[2] で述べたempty は，主にhedgeRule の中で使います．以下に例を示します．

```
<hedgeRule label="frontMatter">
  <empty/>
</hedgeRule>

<elementRule role="section">
  <sequence>
    <ref label="title"/>
    <hedgeRef label="frontMatter"/>
    <ref label="para" occurs="*"/>
  </sequence>
</elementRule>
```

このモジュールを再利用する人は，frontMatter の記述をカスタマイズすることによって，section の構造を変更できます．

3.1.7 none の用途

同じく STEP 1[3] で述べたnone もhedgeRule の中で使います．以下に，使用例を示します．

```
<hedgeRule label="local-block-class">
  <none/>
</hedgeRule>

<hedgeRule label="block-class">
  <choice>
    <ref label="para"/>
    <ref label="fig"/>
    <hedgeRef label="local-black-class"/>
  </choice>
</hedgeRule>
```

このモジュールを再利用する人は，local-block-class の記述をカスタマイズすることによって，block-class の内容を変更できます．

[2] ./step1.html
[3] ./step1.html

3.2 属性リスト宣言の中で用いられるパラメタ実体

属性宣言をいくつかまとめて一回だけ記述し，それを繰り返して参照するための機構が**attPool 要素**です．DTD にあるパラメタ実体のうち，属性リスト宣言で使われるものに相当します.

3.2.1 概要

attPool 要素の構文を下に示します．foo はパラメタ実体の名前です.

```
<attPool role="foo">
  ...attribute definitions...
</attPool>
```

このように定義したattPool を参照するには，属性定義の並びの先頭に<ref role="foo"/> と書きます．このref 要素は，attPool の中で指定された属性定義の並びで置き換えられます.

以下の例では，要素型title のためのtag 要素からattPool を参照しています．このtag は，STEP 1 の先頭にあるモジュールにあったものを書き直したものです．多くの要素型に共通するrole 属性が，common.att というattPool に記述されています.

```
<attPool role="common.att">
  <attribute name="class" type="NMTOKEN"/>
</attPool>

<tag name="title">
  <ref role="common.att"/>
  <attribute name="number" required="true" type="integer"/>
</tag>
```

ref 要素は次のように展開されます.

```
<tag name="title">
  <attribute name="class" type="NMTOKEN"/>
  <attribute name="number" required="true" type="integer"/>
</tag>
```

この例では，tag の中からattPool を参照しましたが，attPool の中からも同様に可能です.

3.2.2 ref とattPool の順番

ref で参照する前にattPool を書く必要はありません．たとえば，次の記述はエラーではありません．

```
<tag name="title">
  <ref role="common.att"/>
  <attribute name="number" required="true" type="integer"/>
</tag>

<attPool role="common.att">
  <attribute name="role" type="NMTOKEN"/>
</attPool>
```

3.2.3 複数の ref 要素

一つのtag またはattPool の中に，複数のref 要素を書くことができます．以下に，一つのattPool のなかで複数のref 要素を用いた例を示します．必須の属性をcommon-req.att にまとめ，必須ではない属性をcommon-opt.att にまとめています．この二つをcommon.att を記述するattPool から参照しています．

```
<attPool role="common.att">
  <ref role="common-req.att"/>
  <ref role="common-opt.att"/>
</attPool>

<attPool role="common-req.att">
  <attribute name="role" type="NMTOKEN" required="true"/>
</attPool>

<attPool role="common-opt.att">
  <attribute name="id" type="NMTOKEN"/>
</attPool>
```

3.2.4 自分自身を参照しないこと

hedgeRule のときと同様に，自分自身を直接的または間接的に参照するのはエラーです．たとえば，次の例はエラーです．

```
<attPool role="bar1">
  <ref role="bar2"/>
  <attribute name="id" type="NMTOKEN"/>
```

```
</attPool>

<attPool role="bar2">
  <ref role="bar1"/>
</attPool>
```

3.3 まとめ

STEP 2 までで，XML の DTD で出来ることはだいたい出来ます．ぜひ使ってみて下さい．RELAX!

第4章　STEP 3: データ型

$Id: step3.sdoc 1.10 2000/08/26 03:14:38 murata Exp $

STEP 3 では，データ型について説明します.

4.1　XML Schema Part 2 のデータ型

XML Schema Part 2[1]は，多くの組み込み済みデータ型を導入しています. それらは，XML Schema 以外の仕様からも利用できるように配慮されています. RELAX は，これらの組み込み済みデータ型をすべて引き継いでいます.

XML Schema Part 2 の組み込み済みデータ型には，XML の DTD から借りたものと，新規に導入されたものがあります. 次に，XML の DTD から借りたものの一覧を示します.

- NMTOKEN
- NMTOKENS
- ID
- IDREF
- IDREFS
- ENTITY
- ENTITIES
- NOTATION

次に，XML Schema Part 2 で新たに導入した組み込み済みデータ型の一覧を示します.

- string
- boolean
- float

[1] http://www.w3.org/TR/xmlschema-2/

- double
- decimal
- timeDuration
- recurringDuration
- binary
- uriReference
- QName
- language
- Name
- NCName
- integer
- nonPositiveInteger
- negativeInteger
- long
- int
- short
- byte
- nonNegativeInteger
- unsignedLong
- unsignedInt
- unsignedShort
- unsignedByte
- positiveInteger
- timeInstant
- time
- timePeriod

- date
- month
- year
- century
- recurringDate
- recurringDay

XML Schema Part 2 では，これらのデータ型を指定するとき，範囲などの制限をつけることができます．RELAX でも同様です．ただし，XML Schema Part 2 とは異なり，ユーザがデータ型を定義することは出来ません．

4.2 RELAX 独自のデータ型

RELAX で独自に導入したデータ型は`none` と`emptyString` の二つです．

4.2.1 none

空のデータ型です．どんな文字列もこのデータ型に属することはありません．RELAX では，none を用いて属性の禁止を指定します．次の例では，class という属性が禁止されています．`none` を導入する理由は，STEP 8 で明らかになります．

```
<tag name="p">
  <attribute name="class" type="none"/>
</tag>
```

したがって，次の開始タグは許されません．

```
  <p class="foo">
```

4.2.2 emptyString

空の文字列からなるデータ型です．DTD の`EMPTY` との互換性があります．

```
<elementRule role="em" type="emptyString"/>
```

このelementRuleは，次の二つに限って許しています．との間に空白が入ることは許されません．

```
<em/>
```

```
<em></em>
```

4.3 付加条件

XML Schema Part 2と同様に，RELAXでもデータ型に制限を加えることができます．たとえば，integerに「15以上，65以下」という制限を加えることができます．制限を示す構文も，XML Schema Part 2と同様です．

4.3.1 elementRuleの場合

elementRuleが参照しているデータ型に制限を加えるには，elementRuleに子要素を追加します．

以下の例では，要素型ageの生け垣モデルはintegerへの参照です．要素minInclusiveとmaxInclusiveはそれぞれ最小値と最大値に関する追加条件を表しています．したがって，age要素の内容は，18から65までの整数を表す文字列になります．

```
<elementRule role="age" type="integer">
  <minInclusive value="18"/>
  <maxInclusive value="65"/>
</elementRule>
```

age要素に，文字列”20”を内容として含むことができます．

```
<age>20</age>
```

しかし，文字列”11”は駄目です．

```
<age>11</age>
```

4.3.2 attributeの場合

attributeが参照しているデータ型に制限を加えるには，attributeに子要素を追加します.

以下の例では，employeeのsex属性は，manかwomanのどちらかであると指定しています．ここで，enumerationは許される値を指定する付加条件です.

```
<tag name="employee">
  <attribute name="sex" type="NMTOKEN">
    <enumeration value="man"/>
    <enumeration value="woman"/>
  </attribute>
</tag>
```

sex属性は文字列"man"を持つことが出来ます.

```
<employee sex="man"/>
```

しかし，文字列"foo"は駄目です.

```
<employee sex="foo"/>
```

4.4 まとめ

STEP 3までもRELAXは十分使いでがあると思います．RELAX!

第5章　STEP 4: 注釈

$Id: step4.sdoc 1.12 2000/11/01 13:43:29 murata Exp $

DTD を説明する資料を作るのは大変重要な作業です．DTD は単に構文を定義するだけですから，自然言語による大量の注釈によって意味を説明することが必要になります．XML にあるコメントは使えますが，RELAX モジュールを構文解析してから表示するブラウザはコメントを無視してしまいます．

STEP 4 は，モジュールに注釈を入れるための機構について説明します．これらは要素と属性を用いていますから，RELAX モジュールを構文解析するブラウザも，ユーザに分かりやすい形で注釈を表示することが出来ます．

5.1　annotation 要素

注釈を挿入するためのトップレベルの要素が，annotation 要素です．annotation 要素が入れられるのは，次の箇所です．

- interface 要素の前に一個だけ
- export 要素の中に一個だけ
- elementRule 要素の最初の子として一個だけ
- hedgeRule 要素の最初の子として一個だけ
- tag 要素の最初の子として一個だけ
- attPool 要素の最初の子として一個だけ
- attribute 要素の最初の子として一個だけ
- include 要素の子要素として一個だけ
- element 要素の最初の子として一個だけ
- div 要素の最初の子として一個だけ

elementRule 要素の最初の子として，注釈を使用した例を以下に示します．注釈の内容は省略しています．

```
<elementRule role="para">
  <annotation> ... </annotation>
  <mixed>
    <ref label="fnote" occurs="*"/>
  </mixed>
</elementRule>
```

annotation 要素は，子要素としてdocumentation とappinfo をいくつでも持つことができます.

5.1.1 documentation 要素

自然言語による説明を表現するための要素がdocumentation 要素です．RELAX Namespace が制定されていない現時点は，テキストデータしか入れられません.

さきに示した例にdocumentation を付加したものを次に示します.

```
<elementRule role="para">
  <annotation>
    <documentation>This is a paragraph.</documentation>
  </annotation>
  <mixed>
    <ref label="fnote" occurs="*"/>
  </mixed>
</elementRule>
```

documentation 要素がsource 属性を持つ場合は，説明を参照するURI が属性値です．この場合は，documentation 要素の内容は使われません．モジュールを表示するツールは，リンクを利用した表示を提供します.

```
<elementRule role="para">
  <annotation>
    <documentation source="http://www.xml.gr.jp/relax/"/>
  </annotation>
  <mixed>
    <ref label="fnote" occurs="*"/>
  </mixed>
</elementRule>
```

documentation 要素にxml:lang 属性が指定されたときは，documentation 要素の内容が，どの自然言語で書かれているかを示します.

つぎの例では，xml:lang の値として”en”が指定されています.

```
<elementRule role="para">
  <annotation>
    <documentation xml:lang="en">This is a paragraph.</documentation>
  </annotation>
  <mixed>
    <ref label="fnote" occurs="*"/>
  </mixed>
</elementRule>
```

5.1.2 appinfo 要素

文書と RELAX モジュールを照合する検証プログラム以外にも，RELAX モジュールを操作するプログラムはいくらでも存在し得ます．たとえば，モジュールからデータベースのスキーマを生成するプログラムです．そのようなプログラムが利用するための隠し情報を表現するのが appinfo 要素です．RELAX Namespace が制定されていない現時点は，テキストデータしか入れられません．

```
<elementRule role="foo" type="integer">
  <annotation><appinfo>default:1</appinfo></annotation>
</elementRule>
```

appinfo 要素が source 属性を持つ場合は，隠し情報を参照する URL が属性値です．この場合は，appinfo 要素の内容は使われません．

5.2 div 要素

複数の elementRule, hedgeRule, tag, attPool を一つにまとめて注釈をつけたいことがあります．そのために用意されているのが div 要素です．

div 要素は，module 要素の中に，elementRule, hedgeRule, tag, attPool と同レベルに置きます．div 要素の中にさらに div 要素を置くこともできます．div 要素の中には，elementRule, hedgeRule, tag, attPool, div を置くことができます．

STEP 1 に示したモジュールに div を使用して注釈を加えたものを下に示します．

```
<module
      moduleVersion="1.2"
      relaxCoreVersion="1.0"
      targetNamespace=""
      xmlns="http://www.xml.gr.jp/xmlns/relaxCore">
```

```
  <interface>
    <export label="doc"/>
  </interface>

  <div>
    <annotation>
      <documentation>The root node</documentation>
    </annotation>

    <elementRule role="doc">
      <sequence>
        <ref label="title"/>
        <ref label="para" occurs="*"/>
      </sequence>
    </elementRule>

    <tag name="doc"/>

  </div>

  <div>
    <annotation>
      <documentation>Paragraphs</documentation>
    </annotation>

    <elementRule role="para">
      <mixed>
        <ref label="em" occurs="*"/>
      </mixed>
    </elementRule>

    <tag name="para">
      <attribute name="class" type="NMTOKEN"/>
    </tag>

  </div>

  <elementRule role="title">
    <mixed>
      <ref label="em" occurs="*"/>
    </mixed>
  </elementRule>

  <tag name="title">
    <attribute name="class" type="NMTOKEN"/>
    <attribute name="number" required="true" type="integer"/>
  </tag>

  <elementRule role="em" type="string"/>
  <tag name="em"/>

</module>
```

5.3 まとめ

STEP 4 で，モジュールの説明を書くのが容易になりました．RELAX!

第6章　STEP 5: 大きすぎるモジュールの分割

$Id: step5.sdoc 1.7 2000/04/14 12:40:02 murata Exp $

大きすぎるモジュールは管理するのが大変です．STEP 5 は，モジュールをいくつかのファイルに分割して管理するための機能を説明します.

6.1　モジュール分割はなぜ必要か

200 の要素型をもつ DTD で RELAX で表現すると仮定します．なお，この程度の大きさの DTD はいくつも存在します．RELAX では，各要素型ごとに elementRule と tag がそれぞれ必要とします．一つが平均 3 行としても 1200 行になります．文書化を積極的に行えば 3000 行ぐらいの長さになることも考えられます．これは，一つのファイルに格納するにはいささか長すぎます.

DTD のときは，外部パラメタ実体を用いて，大規模な DTD を分割して管理していました．RELAX でも，大規模なモジュールを分割するための機構が必要になります.

6.2　include 要素

RELAX では，他のモジュールを include 要素によって参照することができます．include 要素は参照されたモジュールの本体によって置き換えられます.

include 要素の使用例を考えてみます．まず，インクルードされる側のモジュールをつぎに示します.

```
<module
    moduleVersion="1.2"
    relaxCoreVersion="1.0"
    targetNamespace=""
    xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface/>
```

```
  <elementRule role="bar" type="emptyString"/>

  <tag name="bar"/>

</module>
```

このモジュールには，bar に関するelementRule とtag 要素があります．interface 要素は空要素です．このモジュールはbar.rlx に格納されているものとします．

次に，このモジュールを参照して取り込む側のモジュールを示します．

```
<module
    moduleVersion="1.2"
    relaxCoreVersion="1.0"
    targetNamespace=""
    xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="foo"/>
  </interface>

  <elementRule role="foo">
    <ref label="bar"/>
  </elementRule>

  <tag name="foo"/>

  <include moduleLocation="bar.rlx" />

</module>
```

このモジュールには，foo に関するelementRule とtag が記述されています．モジュールの最後にあるinclude 要素は，moduleLocation 属性でbar.rlx を指定しています．

include 要素は，参照されているモジュールの本体（module 要素の内容のうち，interface 以降）によって置き換えられます．この例では，つぎのように置き換えられます．

```
<module
    moduleVersion="1.2"
    relaxCoreVersion="1.0"
    targetNamespace=""
    xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="foo"/>
  </interface>
```

```
  <elementRule role="foo">
    <ref label="bar"/>
  </elementRule>

  <tag name="foo"/>

  <elementRule role="bar" type="emptyString"/>

  <tag name="bar"/>

</module>
```

6.3 interfaceが空でない場合

前節の例では，参照される側のモジュールのinterface要素は空でした．次のように，interface要素に子要素を追加してみます．

```
<module
    moduleVersion="1.2"
    relaxCoreVersion="1.0"
    targetNamespace=""
    xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="bar"/>
  </interface>

  <elementRule role="bar" type="emptyString"/>

  <tag name="bar"/>

</module>
```

この場合には，インクルードされる側のモジュールにあるinterface要素の内容が，インクルードする側のモジュールのinterface要素に追加されます．この例では次のようになります．

```
<module
    moduleVersion="1.2"
    relaxCoreVersion="1.0"
    targetNamespace=""
    xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="foo"/>
    <export label="bar"/>
  </interface>
```

```
  <elementRule role="foo">
    <ref label="bar"/>
  </elementRule>

  <tag name="foo"/>

  <elementRule role="bar" type="emptyString"/>

  <tag name="bar"/>

</module>
```

6.4 まとめ

STEP 5 で，大規模なモジュールも管理できるようになりました．RELAX!

第7章　STEP 6: デフォルト値，実体，記法

$Id: step6.sdoc 1.9 2000/11/01 13:45:32 murata Exp $

DTD にある機能のうち，まだ扱っていないのは，デフォルト値，実体，記法です．STEP 6 では，これらをどう扱うかを示します．

7.1　RELAX で導入しない理由

デフォルト値，実体，記法は RELAX にはありません．これらを RELAX で導入しないのは，既存の XML パーサを使い続けるためです．

もし，RELAX でこれらの機能を導入したとします．たとえば，`default` 属性を attribute 要素に導入して，属性のデフォルトを指定できるようにします．しかし，既存のパーサは XML 文書の解析時に RELAX モジュールをいっさい見てくれません．したがって，default 属性もとうぜん使いません．実体や記法を宣言するための構文を RELAX に導入しても，やはり同じことです．

これらの機能を RELAX で導入するには，RELAX 専用の XML パーサを新たに作るしか方法はありません．RELAX に基づいて XML 文書を作るユーザは，もちろんこの RELAX 専用 XML パーサを使う必要があります．それだけでなく，このようにして作成された XML 文書を受け取るユーザにも，RELAX 専用 XML パーサに乗り換えてもらうことが必要になります．これはあまり現実的ではありません．

7.2　DTD と RELAX の併用

では，デフォルト値，実体，記法はまったく使えないのでしょうか．いいえ，RELAX と DTD を併用すれば，これらの機能を使うことができます．

以下に，DTD を持つ XML 文書を示します．

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE person [
<!ATTLIST person
        bloodType CDATA "A">
```

```
]>
<person/>
```

この文書と照合する RELAX モジュールを示します.

```
<module
      moduleVersion="1.0"
      relaxCoreVersion="1.0"
      xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="person"/>
  </interface>

  <elementRule role="person">
    <empty/>
  </elementRule>

  <tag name="person">
    <attribute name="bloodType">
      <enumeration value="O"/>
      <enumeration value="A"/>
      <enumeration value="B"/>
      <enumeration value="AB"/>
    </attribute>
  </tag>
</module>
```

この例では，DTD の側でデフォルト値”A”を指定しています．XML パーサは，これを利用してくれます．RELAX モジュールに照らしてこの XML 文書を検証することも問題なく可能です．”A”が属性値として指定された場合とまったく同様にして検証は行われます.

同様に，実体や記法も DTD で記述することができます．まず，解析対象実体の例を示します.

```
<?xml version="1.0"?>
<!DOCTYPE doc [
<!ENTITY foo "This is a pen">
]>
<doc>
  <para>&foo;</para>
</doc>
```

この文書は，次の RELAX モジュールに従っています.

```
<module
      moduleVersion="1.0"
      relaxCoreVersion="1.0"
      xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="doc"/>
  </interface>

  <elementRule role="doc">
    <ref label="para" occurs="*"/>
  </elementRule>

  <elementRule role="para" type="string"/>

  <tag name="doc"/>

  <tag name="para"/>

</module>
```

つぎに，解析対象外実体と記法を用いた例を示します.

```
<?xml version="1.0"?>
<!DOCTYPE doc [

<!NOTATION eps          PUBLIC
"-//ISBN 0-7923-9432-1::Graphic Notation//NOTATION Adobe Systems
Encapsulated Postscript//EN">

<!ENTITY logo_eps SYSTEM "logo.eps" NDATA eps>

<!ELEMENT doc EMPTY>

<!ATTLIST doc logo ENTITY #IMPLIED>
]>
<doc logo="logo_eps"/>
```

この文書も，次の RELAX モジュールに従っています.

```
<module
      moduleVersion="1.0"
      relaxCoreVersion="1.0"
      xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="doc"/>
  </interface>

  <elementRule role="doc" type="emptyString"/>
```

```
  <tag name="doc">
    <attribute name="logo" type="ENTITY"/>
  </tag>

</module>
```

7.3 やはり使うべきではない

前節で示したように，DTD と RELAX を併用すれば，デフォルト値，実体，記法を使うことは可能です．しかし，やはり使わないに越したことはありません．

デフォルト値については，アプリケーションプログラムの側で対応することができます．属性が省略されたときの処理として記述すればいいわけです．属性が省略されたとき，あらかじめ決めた値を追加するような XSLT スクリプトを書くこともできます．

外部解析対象実体と解析対象外実体については，リンクを使いましょう．リンクのほうが WWW にふさわしい方法です．

内部解析対象実体については，使ってかまいません．”<”のように，内部解析対象実体 (例えば<) を用いて表現するのがもっとも自然なものもあります．

なお，DTD にデフォルト値，実体，記法を記述しても，期待と異なる結果になることがあります．これは，XML パーサによっては外部 DTD サブセットまたは外部パラメタ実体にある宣言を読まないためです．ここで示した例は，すべて内部サブセットで宣言しているので，予想外の結果になることはありません．

7.4 まとめ

STEP 6 までで，DTD から乗り換えるには十分以上の機能があります．この範囲なら，DTD と相互に変換しても，データ型に関する情報以外は欠落しません．将来，XML Schema に変換することも容易に可能でしょう．RELAX!

第8章　STEP 7: elementRule とhedgeRule再訪

$Id: step7.sdoc 1.13 2000/08/26 08:37:11 murata Exp $

STEP 6 までのelementRule, hedgeRule の説明は，DTD を知っている人がすぐに理解できる範囲だけを扱っています．実際にはもっと一般化された考え方が RELAX には存在しています．

8.1　elementRule とラベル

elementRule は，label属性を持つことができます．ここでは，まずlabel属性の目的を説明し，次に機構を説明します．

8.1.1　出現位置によって異なる内容モデル

同じ名前のタグであっても出現位置によって内容モデルを変えたいということがしばしばあります．たとえば，章の中にある段落，表の中の段落は，どれも段落には違いありません．しかし，段落の中に許されるものは微妙に異なります．たとえば，章にある段落は脚注を含んでもかまいませんが，表の中の段落は文字だけに制限したいかもしれません．

```
<!-- This example is legal. -->
<section>
  <para>This paragraph can contain footnotes <footnote>This
        is a footnote</footnote>.</para>
</section>
```

```
<!-- This example is illegal. -->
<table>
  ...
  <para>This paragraph cannot contain a footnote <footnote>This
        is an illegal footnote</footnote>.</para>
  ...
</table>
```

したがって，段落が章の中にあるときと，表の中にあるときとでは，下記の二つの内容モデルを使いわけたくなります.

```
<!-- Case 1: subordinate to <section> elements. -->
<!ELEMENT para (#PCDATA|footnote)*>

<!-- Case 2: subordinate to <table> elements. -->
<!ELEMENT para (#PCDATA)>
```

出現位置によって内容モデルを変えたいという例は HTML にもあります. HTML では，a 要素が，他のa 要素の中に直接的または間接的に現れることを禁止しています．同様のことは，form 要素にもあてはまります．form の中にform が現れることは禁止されています.

```
<!-- This example is illegal. -->
<a href="foo"><span><a href="bar">dmy</a></span></a>
```

```
<!-- This example is also illegal. -->
<form>
  ...
  <div>
    <form>
      ...
    </form>
  </div>
  ...
</form>
```

HTML では，a 要素はspan 要素を含むことができます．入れ子は間接的であっても禁止したいので，a の外にあるspan ではa を許し，中にあるspan では許さないようにする必要があります．form についても同様で，form の外にあるdiv ではform を許し，中にあるdiv では許さないようにする必要があります.

```
<!-- Case 1: subordinate to <a> elements. -->
<!ELEMENT span (#PCDATA|a)*>

<!-- Case 2: not subordinate to <a> elements. -->
<!ELEMENT span (#PCDATA)>
```

しかし，DTD では，タグ名が同じである限り，内容モデルは常に一つです．したがって，段落の出現箇所によって，内容モデルを変えることはできません．span の内容モデルも常に一つであり，span がa の中にあるかどうかで内容モデルを変えることはできません．div の内容モデルも同様です．

この問題を回避するために，二つの方法が用いられてきました．一つの方法は，出現箇所ごとに別のタグ名を導入することです．この方法を用いた例を次に示します．章の中の段落にはparaInSection というタグ名，表の中の段落はparaInTable というタグ名を導入しています．

```
<!ELEMENT paraInSection (#PCDATA|footnote)*>

<!ELEMENT paraInTable (#PCDATA)>
```

```
<!-- This example is legal. -->
<section>
  <paraInSection>This paragraph can contain footnotes <footnote>This
        is a footnote</footnote>.</paraInSection>
</section>
```

```
<table>
  ...
  <paraInTable>This paragraph cannot contain a footnote.</paraInTable>
  ...
</table>
```

この方法には，DTD が大きくなると，ほとんど同じタグ名が急速に増えるという問題があります．段落，脚注，箇条書き等のような通常のタグ名が何倍にも増えるからです．

もう一つの方法は，タグ名は一つで済ませる代わりに，必要ないくつかの内容モデルをすべて足した内容モデルを作るというものです．この方法を用いた例を次に示します．章の中の段落だけではなく，表の中の段落にも，注釈が許されています．

```
<!ELEMENT para (#PCDATA|footnote)*>
```

この方法には，検証が不十分になるという問題があります．いまの例だと，以下の文書が検証を通ってしまいます．

```
<!-- This example is illegal. -->
<table>
  ...
  <para>This paragraph cannot contain a footnote <footnote>This
        is an illegal footnote</footnote>.</para>
  ...
</table>
```

8.1.2 elementRuleのlabel属性

同じタグ名が出現位置によって異なる生け垣モデルを持てるようにするため，RELAXは**ラベル**を導入しています．タグ名が同じであってもラベルが違えば，異なる生け垣モデルが適用されます．

elementRuleはlabel**属性**を持つことができます．次に，elementRuleの基本的な形を示します．

```
<elementRule role="name" label="label">
  ...content model...
</elementRule>
```

label属性が省略された場合は，role属性と同じ値を指定したものと解釈されます．したがって，次の二つのelementRuleは等価です．

```
<elementRule role="foo">
  ...content model...
</elementRule>
```

```
<elementRule role="foo" label="foo">
  ...content model...
</elementRule>
```

脚注を含む段落と含まない段落とでラベルを使い分け，異なる内容モデルを指定した例を次に示します．

```
<elementRule role="para" label="paraWithFNotes">
  <mixed>
    <ref label="footnote" occurs="*"/>
  </mixed>
</elementRule>

<elementRule role="para" label="paraWithoutFNotes">
  <mixed>
```

```
    <empty/>
  <mixed/>
</elementRule>

<tag name="para"/>
```

一番目のelementRuleは，ラベルがparaWithFNotesである段落の内容は脚注を含んだ文字列であることを示しています．二番目のelementRuleは，ラベルがparaWithoutFNotesである段落の内容は単なる文字列であることを示しています．

多くの場合にラベルとタグ名は一対一に対応します．実際，STEP 6までの例ではそうでした．しかし，前節で示したような問題を扱うには，タグ名と一対一に対応しないラベルが必要になります．

8.1.3 ref要素のlabel属性

次に，ref要素のlabel属性を説明します．この属性の値は，常にラベルです．STEP 1では，値が要素型名だと書きましたが，あの説明は正確ではありません．RELAXには要素型という概念はありません．(実はXML 1.0にもありません．要素型宣言は定義されていますが，要素型の定義はどこを捜してもありません.)

前節の例にあるparaWithFNotesとparaWithoutFNotesはラベルですから，ref要素から参照することができます．章のための内容モデルからは，paraWithFNotesを参照します．表（正確には表のセル）のための内容モデルからは，paraWithoutFNotesを参照します．

```
<elementRule role="section">
  <ref label="paraWithFNotes" occurs="*"/>
</elementRule>

<elementRule role="cell">
  <ref label="paraWithoutFNotes" occurs="*"/>
</elementRule>
```

8.2 ラベルの共有

8.2.1 ラベルを共有する複数のhedgeRule

複数のhedgeRuleがlabel属性に同一のラベルを指定しても構いません．次の例では，ラベルblockElemに対してhedgeRuleが二つ存在しています．

```
<hedgeRule label="blockElem">
  <ref label="para"/>
</hedgeRule>

<hedgeRule label="blockElem">
  <ref label="itemizedList"/>
</hedgeRule>
```

次の**elementRule** は，この**blockElem** を参照しています．

```
<elementRule role="doc">
  <sequence>
    <ref label="title"/>
    <hedgeRef label="blockElem" occurs="*"/>
  </sequence>
</elementRule>
```

RELAX 文法との照合のとき，**hedgeRef** は最初にすべて展開されます．この**hedgeRef** を例に展開の仕方を説明します．

ラベル**blockElem** を記述する二つの**hedgeRule** の生け垣モデルは，どちらも**ref** 要素です．それらをまとめて**choice** 要素で括った結果は次のようになります．

```
<choice>
  <ref label="para"/>
  <ref label="itemizedList"/>
</choice>
```

展開したい**hedgeRef** は，**occurs** 属性として*****を指定しています．この**choice** に，これを転記します．

```
<choice occurs="*">
  <ref label="para"/>
  <ref label="itemizedList"/>
</choice>
```

最後に，この**choice** 要素で**hedgeRef** を置き換えます．

```
<elementRule role="doc">
  <sequence>
    <ref label="title"/>
    <choice occurs="*">
      <ref label="para"/>
      <ref label="itemizedList"/>
    </choice>
  </sequence>
</elementRule>
```

あるラベルを参照するhedgeRefを展開するための手順をまとめておきます.

1. このラベルを記述するhedgeRuleをすべて探す.
2. これらのhedgeRuleの生け垣モデルをchoice要素で一つに括る.
3. hedgeRefのoccurs属性をこのchoice要素に転記する.
4. このchoiceで，hedgeRefを置き換える.

複数のhedgeRuleによるラベルの共有が認められているので，無理に一つのhedgeRuleにまとめる必要がありません. たとえば, numberedItemizedListをblockElemの一種として追加しても，次のhedgeRuleを書き足すだけで済みます. 他のhedgeRuleを変更する必要はありません.

```
<hedgeRule label="blockElem">
  <ref label="numberedItemizedList"/>
</hedgeRule>
```

8.2.2 hedgeRuleとelementRuleによるラベル共有の禁止

hedgeRuleとelementRuleが同じラベルを共有することは許されません. たとえば，次の例は構文エラーです.

```
<hedgeRule label="foo">
  <ref label="bar"/>
</hedgeRule>

<elementRule role="foo" label="foo">
  <empty/>
</elementRule>
```

8.2.3 ラベルを共有する複数のelementRule

複数のelementRuleがlabel属性に同一のラベルを指定しても構いません. また，複数のelementRuleのrole属性が同一であっても構いません.

以下の例では，二つのelementRuleがrole属性としてsectionを指定しています. どちらもlabel属性は省略されていますから，sectionを指定しているものとみなされます.

```
<tag name="section"/>

<elementRule role="section">
  <ref label="para" occurs="*"/>
</elementRule>

<elementRule role="section">
  <choice occurs="*">
    <ref label="para"/>
    <ref label="fig"/>
  </choice>
</elementRule>
```

ラベルを共有する複数のelementRuleが存在するときは，どれか一つが成立するだけで十分です．

次のsection要素を考えます．一番目のelementRuleも二番目のelementRuleも，これを認めています．したがって，ラベルsectionを振ることができます．

```
<section><para/></section>
```

次のsection要素はfig要素を含んでいるので，二番目のelementRuleだけが成立しています．どれか一つが成立していれば十分ですから，やはりラベルsectionを振ることができます．

```
<section><para/><fig/><para/></section>
```

複数のelementRuleが同一のラベルを記述することの利点を説明します．すでにモジュールが作成されているとします．より広い範囲の文書が合法になるようにこのモジュールを変更することを考えます．

従来の方法では，既存のelementRuleを変更する必要があります．今まで合法だった文書が，この変更後にも合法であるという保証はありません．

RELAXでは，既存のelementRuleを変更せず，新たにelementRuleを追加することで対処できます．この方法だと，今まで合法だった文書がやはり合法であることは原理的に保証されています．

先の例では，paraだけをsectionの内容として許すのが当初の予定であり，一番目のelementRuleはそのために書かれています．その後に，figも内容として許すように二番目のelementRuleを追加しています．一番目のelementRuleは依然として有効ですから，今まで合法だった文書が合法でなくなることはありません．

8.3 まとめ

いままで大規模な DTD を書くのに苦労してきた人には，STEP 7 は魅力的に映ると思います．必ず問題になる部分が RELAX では綺麗に書けます．RELAX!

第9章　STEP 8: tagとattPool再訪

$Id: step8.sdoc 1.15 2000/08/26 14:03:04 murata Exp murata $

STEP 2 では，tag が属性リスト宣言に相当し，attPool が属性だけを定義するパラメータ実体に相当すると説明しました．実際には，より一般的な考え方が RELAX では存在しています.

9.1　tag要素のrole属性

tag 要素は，name 属性のほかに，role 属性を持つことができます．この節では，まずこの拡張の動機を説明し，つぎにrole 属性を導入します.

9.1.1　属性によって異なる内容モデル

同じ名前のタグであっても属性の値によって内容モデルを変えたいということがしばしばあります．たとえば，val という要素のtype 属性の値によってval の内容モデルを変えたいとします．type="integer"と指定されていれば内容モデルはデータ型integer への参照ですし，type="string"と指定されていればデータ型string への参照です.

```
<!-- This is legal. -->
<val type="integer">10</val>

<!-- This is also legal. -->
<val type="string">foo bar</val>

<!-- This is illegal. -->
<val type="integer">foo bar</val>
```

したがって，type 属性の値がinteger かstring かで，次の二つのelementRule を使い分けたくなります.

```
<!-- Case 1: type="integer" -->
<elementRule role="val" type="integer"/>
```

```
<!-- Case 2: type="string" -->
<elementRule role="val" type="string"/>
```

しかし，STEP 7 までの機能では，タグ名に対して内容モデルを記述しています．属性値は使われません．したがって，type 属性の値に関わらず，同じelementRule が適用されてしまいます．

9.1.2 tag 要素が表す制約条件

tag 要素は，name 属性のほかに，role 属性を持つことができます．tag の基本的な形は次の通りです．name 属性はタグ名を指定しますが，role 属性が指定するのは**役割**です．

```
<tag name="tag-name" role="role-name">
  ...
</tag>
```

tag 要素は，タグ名と属性に関する制約条件の集まりを役割に関連づけます．tag 要素に記述された条件を開始タグ（または空要素タグ）が満たすとき，このタグは指定された役割を持つと言います．

例として，次のtag 要素を考えます．

```
<tag name="val" role="val-integer">
  <attribute name="type" type="NMTOKEN" required="true">
    <enumeration value="integer"/>
  </attribute>
</tag>
```

このtag 要素は,「タグ名がval であり，type 属性の値はinteger という文字列である」という制約条件を表しています．この条件を満たす開始タグ（または空要素タグ）は，役割val-integer を持ちます．したがって，次の開始タグは，役割val-integer を持ちます．

```
<val type="integer">
```

次のtag 要素では，type 属性の値に関する条件が「値がstring という文字列である」に変わり，役割がval-string に変わっています．

```
<tag name="val" role="val-string">
  <attribute name="type" type="NMTOKEN" required="true">
    <enumeration value="string"/>
  </attribute>
</tag>
```

次の開始タグは役割val-integer は持ちませんが，役割val-string は持ちます.

```
<val type="string">
```

tag 要素によって指定されていない属性があっても構いません．たとえば，次の開始タグの属性unknown は，先のtag 要素で言及されていません．しかし，この開始タグは，役割val-string を持ちます．ただし，診断メッセージは出力されます.

```
<val type="string" unknown="">
```

STEP 1 から 7 までにあるようなrole 属性がないtag 要素はどう解釈されるのでしょうか．role 属性がない場合は，name 属性と同じものが指定されたものとみなされます．したがって次の二つのtag 要素は同じ意味を持ちます.

```
<tag name="foo">
  <attribute name="bar" type="int"/>
</tag>

<tag name="foo" role="foo">
  <attribute name="bar" type="int"/>
</tag>
```

9.1.3 elementRule 要素のrole 属性

elementRule 要素のrole 属性は，タグ名を指定するのではなく役割を指定します．したがって，タグ名が同じであっても属性が違えば，別の生け垣モデルを使うことができます.

先の例で示した役割val-string とval-integer を使えば，タグ名がval である開始タグに対して二つのelementRule を使い分けることが出来ます．述語名val-string を指定するelementRule ば，type 属性の値がstring である開始タグを扱います．役割val-integer を指定すれば，type 属性の値がinteger であるものを扱います.

```
<!-- Case 1: type="integer" -->

<tag name="val" role="val-integer">
  <attribute name="type" type="NMTOKEN" required="true">
    <enumeration value="integer"/>
  </attribute>
</tag>

<elementRule role="val-integer" label="val" type="integer"/>

<!-- Case 2: type="string" -->

<tag name="val" role="val-string">
  <attribute name="type" type="NMTOKEN" required="true">
    <enumeration value="string"/>
  </attribute>
</tag>

<elementRule role="val-string" label="val" type="string"/>
```

二つのtag 要素がタグ名val"と属性名"type を指定していることに注意して下さい．RELAX におけるtag 要素は，一度だけしか行われない宣言ではなく，何度でも書ける制約条件なのです.

9.1.4 ref 要素による参照の禁止

ref 要素が参照する役割を，tag 要素で記述してはいけません．記述するのなら，attPool 要素で記述する必要があります.

次の例にあるref 要素は，tag 要素によって記述された役割foo を参照しています．したがって，構文エラーです.

```
<tag name="foo"/>

<attPool role="bar">
  <ref role="foo"/>
</attPool>
```

9.1.5 データ型none 再訪

STEP 3 ではデータ型none を導入しました．属性が指定された場合と指定されない場合とで異なる内容モデルを用いるときに，none は有益です.

たとえば と で内容モデルを変えることを考えます．前者を表現する役割divSec は次のように記述されます.

```
<tag name="div" role="divSec">
  <attribute name="class" type="string">
    <enumeration value="sec"/>
  </attribute>
</tag>
```

問題は を表現する役割divWithoutClassをどう記述するかです．次のように記述したとします．

```
<tag name="div" role="divWithoutClass"/>
```

この記述では， に対しても，divWithoutClassが成立してしまいます．宣言されていない属性があるというメッセージは出ますが，両方の役割を持つとみなされてしまいます．[1]

属性classを持たないということを陽に指定するには，データ型noneを用いて次のように書く必要があります．

```
<tag name="div" role="divWithoutClass">
  <attribute name="class" type="none"/>
</tag>
```

データ型noneに属する文字列は存在しないので，どんな値を属性classに指定しても役割divWithoutClassを持つことはありません．

9.2 attPool要素

attPool要素は，パラメタ実体のように単に展開されるものではありません．attPool要素は，tag要素とほとんど同等の概念です．

9.2.1 attPoolが表す制約条件

tag要素は，タグ名に関する制約条件と属性に関する制約条件の集まりに役割を関連付けるものでした．attPool要素もほとんど同じで，違いはタグ名に関する制約条件を指定できないことだけです．すなわち，属性に関する制限の集まりに役割を関連付けるものがattPool要素です．

つぎのattPoolを考えます．

[1] RELAXでは，宣言されていない属性があっても，役割が成立することにしています．こうしている理由は二つあります．一つは，宣言されていない属性があっても処理を続行するというのが従来のXMLパーサの挙動であることです．もう一つは，HTMLでは宣言されていない属性を許していることです．

```
<attPool role="info">
  <attribute name="class" required="true">
    <enumeration value="informative"/>
  </attribute>
</attPool>
```

このattPool要素は,「class という属性が指定されており，その値はinformative という文字列である」という条件にinfo という役割を関連付けています．タグ名が何であっても構いません．このattPool によって，次の空要素タグは役割info を持ちます.

```
<some class="informative"/>
```

tag 要素のときと同様に，宣言されていない属性があっても構いません．たとえば，次の開始タグに対して役割info は成立します.

```
<some class="informative" unknown=""/>
```

9.2.2 elementRule要素による参照の禁止

elementRule 要素のrole 属性で指定する役割を，attPool 要素で記述してはいけません．記述するのなら，tag 要素で記述する必要があります.

次のelementRule は役割info を参照しています．役割info はattPool 要素によって規定されています．したがって，この例は構文エラーです.

```
<attPool role="info"/>
<elementRule role="info" label="informative" type="emptyString"/>
```

9.3 複数のtag またはattPool 要素による役割の共有の禁止

複数のtag またはattPool 要素が，一つの役割を共有することはできません.

以下の例では，二つのtag 要素が役割bar を共有しています．したがって，構文エラーです.

```
<tag name="foo1" role="bar">
  <attribute name="a" type="string"/>
  ...
</tag>

<tag name="foo2" role="bar">
  <attribute name="b" type="string"/>
  ...
</tag>
```

以下の例では，役割とタグ名の両方を共有しています．これも構文エラーです．

```
<tag name="foo" role="foo">
  <attribute name="a" type="string"/>
  ...
</tag>

<tag name="foo" role="foo">
  <attribute name="b" type="string"/>
  ...
</tag>
```

role 属性が省略され，name 属性の値が使われたときも，共有は許されません．次の二つのtag 要素は，上の二つのtag 要素と同じ意味を持ちます．したがって，この例も構文エラーです．

```
<tag name="foo">
  <attribute name="a" type="string"/>
  ...
</tag>

<tag name="foo">
  <attribute name="b" type="string"/>
  ...
</tag>
```

つぎの例では，二つのattPool 要素が役割bar を共有しています．したがって，構文エラーです．

```
<attPool role="bar">
  <attribute name="a" type="string"/>
  ...
</attPool>

<attPool role="bar">
  <attribute name="b" type="string"/>
  ...
</attPool>
```

最後に，tag 要素とattPool 要素が役割bar を共有した例を示します．これも，構文エラーです．

```
<attPool role="bar">
  <attribute name="a" type="string"/>
  ...
</attPool>

<tag role="bar" name="foo">
  <attribute name="b" type="string"/>
  ...
</tag>
```

9.4 まとめ

STEP 7 までは，tag 要素はタグ名と属性を宣言するものだと見なしてきました．本当は，タグ名に関する条件と属性に関する条件に役割を関連づけるのがtag 要素です．これまでは，役割とタグ名は常に同じでしたが，必ずしもそうではありません．多くの場合にはラベルと役割とタグ名には一対一に対応しますが，一般的にはそうではありません．

タグ名，ラベル，役割を記述・参照する構文が何であるかを表にまとめておきます．

構文要素	タグ名／ラベル／役割の区別
elementRule のrole 属性	tag で記述された役割への参照
elementRule のlabel 属性	ラベルの記述
hedgeRule のlabel 属性	ラベルの記述
ref のlabel 属性	elementRule によって記述されたラベルへの参照
hedgeRef のlabel 属性	hedgeRule によって記述されたラベルへの参照
tag のname 属性	タグ名の記述
tag のrole 属性	役割の記述
attPool のrole 属性	役割の記述
ref のrole 属性	attPool によって記述された役割への参照

次に，タグ名，ラベル，役割が XML 文書で出現するかどうかを表にまとめておきます．

属性の値によって内容モデルを変えることは，従来の DTD では不可能でしたが，RELAX では可能になりました．必要な拡張はrole 属性だけです．RELAX の簡潔さ，強力さの現れといっていいでしょう．RELAX!

名前の種類	インスタンス中で	RELAX モジュール中で
タグ名	出現する	節の一部として出現する
役割	出現しない	節の一部として出現する
ラベル	出現しない	生成規則の一部として出現する

第10章　STEP 9: 生け垣内容モデルelement

RELAX では，element という要素を生け垣モデルとして許しています．これは単なる略記法であり，ref とelementRule とtag に展開されます．この節では，まずelement 要素の目的を説明し，次に機構を説明します．

10.1　プログラミング言語やデータベース言語の模倣

RELAX は DTD の拡張であり，文法的なデータモデルを持っています．これは，プログラミング言語やデータベースのデータモデルとは異質なものです．しかし，RELAX でプログラミング言語の宣言やデータベース言語のスキーマを真似ることは考えておく必要があります．

プログラミング言語では，変数を宣言するときに同時にその型を宣言します．次の例では，変数x とy を宣言し，型としてint を指定しています．

```
public class Point {
    int x;
    int y;
}
```

変数x が別のクラスで宣言されるときは，異なる型を指定されているかも知れません．次の例では，float 型がクラスFoo のx に対して指定されています．

```
public class Foo {
    float x;
}
```

10.2　element 要素

element 要素は，変数名と型の指定を同時に行うための要素生け垣モデルです．element 要素は，必ずname 属性とtype を持ちます．さらに，occurs 属性も持つことができます．

```
<element name="tag-name" type="datatype-name"/>
```

```
<element name="tag-name" type="datatype-name" occurs="*"/>
```

element 要素を用いることによって次のようなtag 要素とelementRule 要素が書けます.

```
<tag name="Point"/>

<elementRule role="Point">
  <sequence>
    <element name="x" type="integer"/>
    <element name="y" type="integer"/>
  </sequence>
</elementRule>
```

x=100 かつy=200 であるようなPoint は次の XML 文書によって表現できます.

```
<Point>
  <x>100</x>
  <y>200</y>
</Point>
```

10.3 ref, elementRule, tag への展開

element 要素は単なる構文糖衣 (syntax sugar) です. 生け垣モデル中のelement 要素はref 要素に置き換えられ, elementRule 要素とtag 要素が生成されます.

前節のelementRule をもう一度示します. このelement 要素がどう展開されるかを考えてみます.

```
<elementRule label="Point">
  <sequence>
    <element name="x" type="integer"/>
    <element name="y" type="integer"/>
  </sequence>
</elementRule>
```

どちらのelement 要素もref 要素によって置き換えられます. また, elementRule とtag がそれぞれに対して生成されます. 生成されるelementRule の生け垣モデルは, element のtype 属性で指定されたデータ型への参照です.

```
<elementRule label="Point">
  <sequence>
    <ref label="Point$1"/>
    <ref label="Point$2"/>
  </sequence>
</elementRule>

<elementRule role="Point$1" label="Point$1" type="integer"/>
<tag role="Point$1" name="x"/>

<elementRule role="Point$2" label="Point$2" type="integer"/>
<tag role="Point$2" name="y"/>
```

element 要素がoccurs 属性を持っているときは，生成されるref 要素にoccurs 属性が付与されます．たとえば，最初のelementRule でelement にoccurs="?"が指定されたとします（下記参照）．

```
<elementRule label="Point">
  <sequence>
    <element name="x" type="integer" occurs="?"/>
    <element name="y" type="integer" occurs="?"/>
  </sequence>
</elementRule>
```

展開の結果は次のようになります．

```
<elementRule label="Point">
  <sequence>
    <ref label="Point$1" occurs="?"/>
    <ref label="Point$2" occurs="?"/>
  </sequence>
</elementRule>

<elementRule role="Point$1" label="Point$1" type="integer"/>
<tag role="Point$1" name="x"/>

<elementRule role="Point$2" label="Point$2" type="integer"/>
<tag role="Point$2" name="y"/>
```

10.4 展開手続き

ここではelement 要素がどう展開されるかをまとめておきます．

10.4.1 ref 要素の生成

ref 要素を生成します．そのlabel 属性の値として，他のラベルと衝突しないラベルが自動生成されます．もし，element 要素にoccurs 属性があれば，生成されたref 要素に引き継がれます．

10.4.2 elementRule 要素の生成

elementRule を生成します．その role 属性の値として，他の役割と衝突しない役割が生成されます．label 属性の値は，ref 要素を生成したときに自動生成したラベルです．この elementRule の生け垣モデルとして，element 要素の type 属性を転記します．

10.4.3 tag 要素の生成

tag 要素を生成します．elementRule を生成するときに自動生成した役割が role 属性として指定されます．生成された tag の name 属性としては，元々の element 要素の name 属性の値が指定されます．

10.5 まとめ

プログラミング言語やデータベースの人たちは，element 要素による記述は，とても自然で分かりやすいと思います．RELAX!

第11章　STEP 10: elementRule へのtagの埋め込み

$Id: step10.sdoc 1.8 2000/11/01 13:46:38 murata Exp $

この節では，tag 要素のelementRule 要素への埋め込みについて説明します.

11.1　属性と生け垣モデルの同時記述

STEP 0 から 9 まででは，属性やタグ名と，生け垣モデルは分離して記述されました．属性やタグ名はtag とattPool に，生け垣モデルはelementRule とhedgeRule に記述されます．elementRule は，役割を通じてtag を参照し，このtag がさらにattPool を参照します.

あるelementRule とあるtag が密接に関連している場合には，両者を分離するのではなく，一つの要素にまとめてしまうほうが便利なこともあります.

elementRule とtag とに分離した記述の例として，STEP 8 にある例をもう一度示します.

```
<!-- Case 1: type="integer" -->

<tag name="val" role="val-integer">
  <attribute name="type" type="NMTOKEN" required="true">
    <enumeration value="integer"/>
  </attribute>
</tag>

<elementRule role="val-integer" label="val" type="integer"/>

<!-- Case 2: type="string" -->

<tag name="val" role="val-string">
  <attribute name="type" type="NMTOKEN" required="true">
    <enumeration value="string"/>
  </attribute>
</tag>

<elementRule role="val-string" label="val" type="string"/>
```

ここで，役割val-integer, val-stringがこの二つのelementRule要素からだけしか参照されていないとします．わざわざval-integer, val-stringという二つの名前をつけて参照するより，以下のように直接埋め込みたくなります．

```
<!-- Case 1: type="integer" -->

<elementRule label="val" type="integer">
  <tag>
    <attribute name="type" type="NMTOKEN" required="true">
      <enumeration value="integer"/>
    </attribute>
  </tag>
</elementRule>

<!-- Case 2: type="string" -->

<elementRule label="val" type="string">
  <tag>
    <attribute name="type" type="NMTOKEN" required="true">
      <enumeration value="string"/>
    </attribute>
  </tag>
</elementRule>
```

この書き方の利点は，役割に名をつけて参照する必要がないことです．書き直す前は，タグ名ともラベルとも違う名前をつけていました．この名前がなくなった分だけ，読むときの負担が減っています．

属性と生け垣モデルとを同時に記述できることを利点と感じる人もいます．たとえば，x 座標と y 座標を持つ点を，XML で二通りに表現した例を以下に示します．前者では，属性によって表現し，後者では要素によって表現しています．両者の差異は僅かであり，書き換えが容易に可能です．

```
<elementRule label="point" type="emptyString">
  <tag>
    <attribute name="x" type="integer"/>
    <attribute name="y" type="integer"/>
  </tag>
</elementRule>
```

```
<elementRule label="point">
  <tag/>
  <sequence>
    <element name="x" type="integer"/>
    <element name="y" type="integer"/>
  </sequence>
</elementRule>
```

tag を含むelementRule は，role 属性を持つことは許されません．その代わりに，label 属性は必須です．

埋め込まれたtag がrole 属性をもつことも許されません．この例では省略されていますが，name 属性を持つことは許されています．

11.2 埋め込まれたtag の処理

埋め込まれたtag は，elementRule から取り出され，その兄弟要素になります．本 STEP の最初の例を用いて説明します．

```
<elementRule label="val" type="integer">
  <tag>
    <attribute name="type" type="NMTOKEN" required="true">
      <enumeration value="integer"/>
    </attribute>
  </tag>
</elementRule>
```

まず，適当な役割を生成します．この例では，val$1 という役割を生成しています．

次に，埋め込まれたtag 要素をelementRule から取り出し，その兄弟要素として配置します．role 属性を追加し，先に生成した役割のval$1 を属性値として指定します．

tag 要素がname 属性を持っていないときに限って，name 属性を追加します．値としては，elementRule 要素のlabel 属性の値を用います．この例では，tag 要素のname 属性の値として"val"を指定します．

最後に，elementRule にrole 属性を追加し，先に生成した役割を指定します．

```
<elementRule label="val" type="integer" role="val$1">
</elementRule>

<tag name="val" role="val$1">
  <attribute name="type" type="NMTOKEN" required="true">
    <enumeration value="integer"/>
  </attribute>
</tag>
```

11.3 まとめ

要素と属性をまとめて記述したいときには，tag の埋め込みによって，簡潔で分かりやすい記述が可能になります．RELAX!