Embedding XML Processing Toolkit on General Purpose Programming Language

Tetsuo Kamina and Tetsuo Tamai
Graduate School of Arts and Sciences
University of Tokyo
Tokyo, Japan
{kamina,tamai}@graco.c.u-tokyo.ac.jp

Abstract

Many methods for XML processing have been proposed in the last few years. One popular approach is to process XML documents by using existing programming languages. Another popular approach is to create a new programming language specialized to the domain of XML processing. We propose a new approach of constructing XML processors: embedding XML processing language on Lisp. Owing to this approach, we may seamlessly invoke the functions of XML-specific language from Lisp.

The other novel features of our approach are shuffle expression pattern matching and dynamic validation of XML documents. A shuffle expression is an extension of a regular expression; it supports shuffle (interleave) operator that is useful, for example, to represent unordered records such as bibliography data. Dynamic validation makes it possible to validate XML documents with respect to the schema or patterns at run time.

1. Introduction

XML[15] is a markup language that can be typed by schema languages such as DTD, XML Schema[13], and RELAX NG[8]. This ability of typing will improve safety of data exchange and processing; therefore, XML has been considered as a standard format for data exchanged over networks such as the Internet.

Many methods for XML document processing have been proposed in the last few years. One popular approach is to process XML documents by using existing programming languages[14, 18]. The advantage is that we can make use of all the features provided by the host language. The disadvantage is that XML documents must be "injected" to the host language: it must be restricted within the value and type space of the host language. Another popular approach is to create a new programming language specialized to the domain of XML processing [1, 4, 7, 9, 16, 17]. By this approach, we can design comprehensive syntax and semantics suitable for XML processing with the cost of difficulty in complex computation.

We set our goal to take both of these advantages: to use all the features provided by a general purpose programming language and devise a convenient XML-specific language in the same environment. In this paper, we propose a new approach: embedding an XML-specific language on Lisp. Using the strong macro feature of Lisp, we embed new syntax on Lisp that defines XML transformation rules. Without seams, this embedded language can be invoked from Lisp. It also may use functions and macros provided by Lisp.

The other novel features of this embedded language are *shuffle expression pattern matching* and *dynamic validation of XML documents*. Shuffle expression pattern matching is the process of extracting data from the middle of an XML document. It also may be regarded as the process of validating the input (the unprocessed) XML document, that is the process of judging whether the XML document is matched against the shuffle expression pattern (which corresponds to the schema of XML). We say an XML document is valid if it does not violate the constraints expressed by the declarations and definitions in the corresponding schema or pattern. As we described above, this ability of validation (also regarded as type checking) is one of the most important features of XML. Our shuffle expression pattern matching is similar to XDuce's regular expression pattern matching [3, 4], but it supports shuffle (interleave) operator, as RELAX NG schema language does [8], and shuffle closure operator. Shuffle operator is very powerful, for example, to represent unordered records such as bibliography data.

Besides the input documents, we also require to validate the output (the processed) XML documents either. Dynamic validation makes it possible to validate the processed XML documents with respect to the schema at run time. To

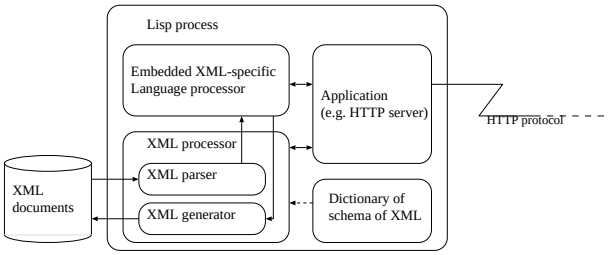

Figure 1. Architecture

implement it, we introduce the *XML element server* which acts like a dictionary of DTDs.

In the next section, we show the architecture of our embedded language and related tools. Then, after presenting the definition of shuffle expressions, we show the features of the language by examples. In appendix, we define the formal syntax and operational semantics of shuffle expression pattern matching.

2. Architecture

Our XML processing toolkit is composed of the following constructs (Figure 1):

- XML processor

  It consists of XML parser and XML generator. The XML parser converts XML documents to S expressions of Lisp, and the XML generator does the inverse operation, conversion of S expressions to XML.

- XML element server

  This is a dictionary of schemas of XML. The XML processor uses it at the dynamic validation stage (see Section 4).

- XML transformation language

  This is an embedded XML-specific language that transforms XML documents into other XML or non-XML documents. It has an ability of shuffle expression pattern matching (see Section 3).

These tools and other applications (such as HTTP server, for example) may be executed in the same Lisp process.

S expressions representing XML. In our toolkit, XML documents are represented as S expressions of Lisp internally. For example, the XML document shown in Figure 2 is represented by S expressions shown in Figure 3. In this example, XML element `<last>` is represented as `:last`, and `<profile xml:lang="en">` is represented as `(:profile :|xml:lang| "en")`. Besides

```
<profile xml:lang="en">
 <last>Kamina</last>
 <first>Tetsuo</first>
 <affiliation>Univ. of Tokyo</affiliation>
 <office>15-610</office>
 <email>kamina</email>
</profile>
```

Figure 2. An example XML document

```
((:profile :|xml:lang| "en")
 (:last "Kamina")
 (:first "Tetsuo")
 (:affiliation "Univ. of Tokyo")
 (:office "15-610")
 (:email "kamina"))
```

Figure 3. An example S expression representing XML document

these logical structure of XML, we may also represent the physical structure of XML (that is the reference structure) using S expressins. We built the XML parser compliant to the W3C XML specification [15], that is, if internal entities are referenced in the content, then include them in the content, or if external parsed entities are referenced in the content, then include them for validation, or if external unparsed entities are referenced as attribute values, then just notify them to the application, and so on.

The formal syntax of S expressions representing XML documents is as follows:

```
sexpr   ::= (element content*)
element ::= name
            (name attlist*)
attlist ::= name string
content ::= string
            lispCommand
            sexpr
```

`name` is a string prefixed by '`:`' (Lisp symbols interned in the *keyword* package). `lispCommand` is Lisp expressions which are to be evaluated.

In general, API for XML processing such as DOM[14] is rather complicated. In contrast, adopting S expression based approach, we are free from designing and implementing such API because S expressions can be manipulated directly using Lisp primitives.

3. Embedding XML-specific language in Common Lisp

As mentioned in the previous sections, we require to use general purpose programming language and XML-specific

language in the same environment. We embed new syntax for XML processing that behaves as a part of Common Lisp processing.

For XML document processing, we use shuffle expression pattern matching that extracts data from the middle of an XML document. Shuffle expressions are extension of regular expressions equipped with shuffle operations (shuffle and shuffle closure). This section provides the definition of shuffle operations and shows how shuffle expression pattern matching is used in XML processing by examples.

3.1. Shuffle expressions

Let Σ be a set of alphabet, Σ^* be a set of words over Σ and the empty word λ is included in Σ^*. The shuffle operator $\odot$ ($\Sigma^* \times \Sigma^* \rightarrow 2^{\Sigma^*}$) is defined inductively as follows:

- for all $u \in \Sigma^*$, $u \odot \lambda = \lambda \odot u = \{u\}$,
- for all $u, v \in \Sigma^*$, $a, b \in \Sigma$, $au \odot bv = a(u \odot bv) \cup b(au \odot v)$.

For any languages $L_1, L_2 \subset \Sigma^*$, $L_1 \odot L_2$ is defined as follows:

$$L_1 \odot L_2 = \bigcup_{u \in L_1, v \in L_2} u \odot v$$

For any languages $L \subset \Sigma^*$, the shuffle closure operator $\otimes$ is defined as follows:

$$L^{\otimes} = \bigcup_{i=0}^{\infty} L^{\odot i}, where\, L^{\odot 0} = \{\lambda\}, L^{\odot i} = L \odot L^{\odot i-1}$$

Jedrzejowicz and Szepietowski proposed shuffle automaton that accepts shuffle expressions[5]. Although shuffle expression pattern matching may be implemented upon it, to make algorithm simpler and get higher performance, we restrict positions of shuffle operators to appear only at the root of parse tree deriving the following definition of *shuffle*$_1$ expressions.

Definition 3.1 *If R_1, R_2 are regular expressions, then $R_1 + R_2$, $R_1 \cdot R_2$, R_1^*, $R_1 \odot R_2$ and $R_1^{\otimes}$ are shuffle$_1$ expressions, and nothing else is a shuffle$_1$ expression.*

3.2. A simple example

We illustrate how shuffle expression pattern matching is powerful by taking a simple example: a bibliography database.

In this database, bibliography data are divided into four types: book, article, inproceedings, and misc. Each datum is stored as an XML document. The applications are:

- Conversion from BibTeX to XML
- Conversion from XML to HTML

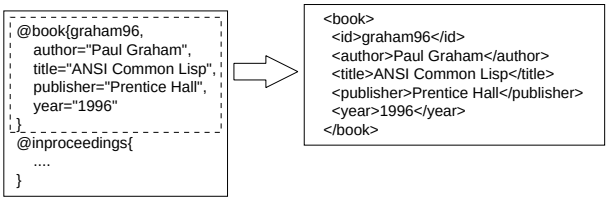

Figure 4. Conversion from BibTeX to XML

```
<element name="article">
 <interleave>
  <element name="id"><text/></element>
  <element name="author"><text/></element>
  <element name="title"><text/></element>
  <element name="journal"><text/></element>
  <element name="year"><text/></element>
  <optional>
   <element name="volume"><text/></element>
  </optional>
  <optional>
   <element name="number"><text/></element>
  </optional>
  <optional>
   <element name="pages"><text/></element>
  </optional>
  <optional>
   <element name="month"><text/></element>
  </optional>
  <optional>
   <element name="note"/><text/></element>
  </optional>
 </interleave>
</element>
```

Figure 5. A fragment of RELAX NG document

3.3. Conversion from BibTeX to XML

To enrich our database, we exploit existing bibliography resources; we convert BibTeX files into XML documents.

How to convert BibTeX files into XML documents is shown in Figure 4. One BibTeX entry corresponds to an XML document. Different types of bibliography correspond to different schemas of XML documents. For example, `@article{ ... }` bibliography type corresponds to a schema written by RELAX NG document shown in Figure 5. RELAX NG is an XML schema language that is equipped with interleave (i.e. shuffle) types. As the order of fields of BibTeX entries is not important, we use the `<interleave>` p_1 p_2 `</interleave>` pattern in the RELAX NG document, which means the elements in pattern p_1 and the elements in pattern p_2 may be interleaved.

```
(:article (% (:id $id)
             (:author $author)
             (:title $title)
             (:journal $journal)
             (:year $year)
             (? (:volume $volume))
             (? (:number $number))
             (? (:pages $pages))
             (? (:month $month))
             (? (:note $note))))
```

Figure 6. An example of patterns

3.4. Conversion from XML to HTML

In the following subsections, we show some example programs of our embedded XML transformation language and discuss its advantages.

Patterns as a schema language. One of the novel features of our XML transformation language is shuffle expression pattern matching. An example of XML pattern is shown in Figure 6. Strings prefixed by '`:`' denote names of XML elements[1], and strings prefixed by '`$`' denote pattern variables. Furthermore, in XML patterns, we may put shuffle expression operators as well as regular expression operators: `seq` (sequence), `or` (choice), `?` (optional), `*` (Kleene's closure), `+` (zero or more), `%` (shuffle operator $\odot$) and `&` (shuffle closure operator $\otimes$).

For example, the XML document

```
<article>
 <id>helzmann97</id>
 <author>G. J. Holzmann</author>
 <title>The Model Checker SPIN</title>
 <journal>IEEE Transactions on
          Software Engineering</journal>
 <volume>23</volume>
 <number>5</number>
 <year>1997</year>
</article>
```

matches the pattern of Figure 6. The bindings of pattern variables are as follows:

```
$id = ("holzmann97")
$author = ("G. J. Holzmann")
$title = ("The Model Checker SPIN")
$journal = ("IEEE Transaction on
             Software Engineering")
$volume = ("23")
$number = ("5")
$year = ("1997")
```

[1]To make discussion simple, we assume all XML elements belong to the default namespace, and no XML elements have attributes

```
(defrule article (:article
                  (% (:id $id)
                     (:author $author)
                     (:title $title)
                     (:journal $journal)
                     (:year $year)
                     (? (:volume $volume))
                     (? (:number $number))
                     (? (:pages $pages))
                     (? (:month $month))
                     (? (:note $note))))
  (:table
   (:caption
    (format nil "Article ID: ~A" $id))
   (:tr (:th "Author") (:th "Title")
        (:th "Journal") (:th "Year")
        (:th "Volume") (:th "Number")
        (:th "Pages") (:th "Month")
        (:th "Note"))
   (:tr (:td $author) (:td $title)
        (:td $journal) (:td $year)
        (:td $volume) (:td $number)
        (:td $pages) (:td $month)
        (:td $note))))
```

Figure 7. An example program

It is important to note that we may regard this pattern as a schema of XML. In fact, the pattern of Figure 6 is equivalent to RELAX NG document of Figure 5. The difference between a pattern and a schema is that a pattern has operational semantics such as binding of variables.

Embedded XML transformation language. We now explain how to process XML by shuffle expression pattern matching. Basically, we process XML documents by transformation. An example of transformation rules is shown in Figure 7. The syntax of `defrule` is as follows:

```
(defrule <name> <input_pattern>
         <output_pattern>)
```

It returns a Common Lisp function that accepts an instance of XML pattern `<input_pattern>` and transforms it into `<output_pattern>`. If it receives an XML document which is not an instance of `<input_pattern>`, it returns `nil`. By pattern matching described above, some data extracted from the middle of an XML document that matched against `<input_pattern>` are bounded to the variables. These variables are used in `<output_pattern>`. In `<output_pattern>`, we may also put any Lisp expressions that are to be evaluated. In Figure 7, we put a `format` expression inside the `:caption` XML element. This is a very simple example but we may put any complex computation inside `<output_pattern>`.

Most interestingly, `defrule` returns an ordinary Common Lisp function. We can invoke it from other part of Lisp program as follows:

```
(generate-xml "html"
 `(:html
   (:head (:title "Foo"))
   (:body
    ,(article (parse-xml p))))
 :public "-//W3C//DTD XHTML 1.0 Strict//EN"
 :validate t)
```

This example shows the `article` function defined in Figure 7 is called inside the backquoted parameter of other Lisp function named `generate-xml`. `generate-xml` is XML generator introduced in Section 2. It can validate generated XML documents using the XML element server; therefore, validity of transformed XML documents (i.e. validity of transformation programs) may be assured dynamically. `parse-xml` is an XML parser introduced in Section 2. It parses an XML document that is sent to the input stream (named `p`) and returns S expression of XML[6].

3.5. Ambiguous patterns

In general, the semantics of shuffle expression pattern matching can be ambiguous. For example, the following pattern

```
(seq (* (:a $foo)) (* (:a $bar)))
```

is ambiguous because how many contents of element `a` the variable `$foo` should take cannot be decided. There are two approaches to treat this problem: the all match approach and the single match approach. The all match approach returns all possible bindings of variables, while the single match approach yields just a single binding. Even though some query language takes the all match approach[1, 9, 12], we concentrate on the single match approach here. We resolve this ambiguity by taking the longest match policy, that is patterns appearing earlier have higher priority. For example, the value `((:a "a1") (:a "a2") (:a "a3"))` matches the previous pattern and the result of pattern match is as follows:

```
$foo = ("a1" "a2" "a3")
$bar = nil
```

3.6. Implementation issues

Implementing shuffle expression pattern matching is not trivial work. Shuffle expressions can be simulated by regular expressions; however, with this approach, the number of states will easily explode. Shuffle automaton[5] resolves this problem but it requires backtracking, which leads to the performance decline. In our toolkit, performance is important because it validates input values dynamically.

```
<!DOCTYPE xdoc [
...
<!ELEMENT foo (a,(b|c)*)+>
...
]>
```

Figure 8. An example of element declaration

```
name:         :foo
attlist:      nil
children:     (((6 :a) 7)
               ((7 :a) 7)
               ((7 :b) 7)
               ((7 :c) 7))
begin-state:  6
final-states: (7)
doctype:      "xdoc"
```

Figure 9. CLOS instance for element declaration

In the previous section, we defined the restricted version of shuffle expressions. According to this definition, we may simulate *shuffle*$_1$ expression with multiple small size finite state automata. For example, the shuffle automaton for $R_1 \odot R_2$ can be simulated by executing the automata for R_1 and R_2 concurrently. Each automaton may be deterministic; therefore, the pattern matching engine may be run without backtracking.

How about ambiguous shuffle expressions, such as $ab \odot ac$? If it receives the sequence of alphabets like `acab`, which "a" of shuffle expressions each `a` will be associated with? To enumerate all combinations, backtracking is required. However, we are not sure about the usefulness of such ambiguous shuffle expressions and believe there is little possibility for them to be used. Therefore, our implementation of *shuffle*$_1$ expression pattern matching restricts the occurrence of the same elements in the operands of shuffle operators. The same restriction is also found in the RELAX NG specification[8].

4. Dynamic validation

Owing to the ability of pattern matching, the validity of input (unprocessed) XML documents of `defrule` is checked at run time. But how about the output values produced by `defrule`? To check the validity of output (processed) XML documents automatically, out toolkit should be able to validate output XML documents as well as input. To tackle this problem, we introduce the *XML element server*.

XML element server acts like a dictionary of DTDs, which remains permanently in the run-time memory of the

Lisp process. It is implemented as an association list of document type names (the key) and CLOS (Common Lisp Object System) instances of *DOCTYPE* class (the value). Instances of DOCTYPE class contain hashtables of XML elements, entities, and other information specific in that document type. XML elements are also represented as CLOS instances. For example, the instance representing XML element `foo` in Figure 8 is shown in Figure 9.

The `children` slot (member) plays the main role of validating XML documents. It represents an automaton converted from the regular expression in the declaration of Figure 8[2]. The `begin-state` and `final-state` slots are the initial state and the final state(s) of this automaton, respectively. The XML processor can validate XML documents at run time by executing this automaton.

Suppose the XML processor tries to validate the given XML documents. Using the name from DOCTYPE declaration of the XML documents, the XML processor searches the corresponding DOCTYPE instance in the XML element server. If it is found, the XML processor uses it. If it is not found, the XML processor tries to parse the DTD file that exists at the URI path in the DOCTYPE declaration, and registers it to the XML element server so that no more parsing of DTD is necessary.

Multiple schema validation and namespace aware validation[11] should be explored in the future work.

5. Related work

DOM (Document Object Model)[14] is a popular API for XML processing. DOM based parser converts XML documents into object trees. Each object may be manipulated via DOM API, so we can write XML processing programs using this API. The problem is although DOM based parsers validate XML documents with respect to schemas of XML, there is no systematic way of validating processed DOM trees.

Representing XML documents as S expressions itself is not new. Franz Inc. released a Lisp based XML parser[2]. It represents XML documents as S expressions internally, but there is no systematic way of validating XML documents. SXML[10] is also a representation of XML documents in the form of S expressions whose processor is developed by Scheme.

Wallace and Runciman proposed to use Haskell for XML processing [18]. It defines mapping from DTDs into Haskell data types. XML documents are represented as instances of Haskell data types. Using Haskell's type system, validity of XML documents can be (partially) assured statically.

[2]The automaton in Figure 9 looks like a DFA of `(a,(a|b|c)*)` but it is equal to a DFA of `(a,(b|c)*)+`.

These approaches may be categorized as "injected" approaches, that is to make XML documents accessible from existing programming languages. Even though these approaches make it possible to use all the features provided by the programming languages, XML documents are restricted by the value and type space of the host language. For example, representing XML documents as trees of objects requires that internals of each object should be accessed only via interfaces. The problem is that recent extension of requirements of XML processing makes DOM API very complex. Haskell approach has another kind of problem. The types of XML documents must be restricted to type space which can be represented by Haskell type system.

On the contrary, there are many XML-specific programming languages such as XSLT[16], XQuery[17], XDuce[4], YAT[9], and XMlambda[7]. XSLT is an XML based XML transformation language called a stylesheet language. XQuery is a query language of XML databases. There is no systematic way of validating processed XML documents for them.

XDuce is a statically typed functional programming language for XML processing. It has novel features of regular expression pattern matching and regular expression types. Using its own type system, the validity of XML documents (and validity of XDuce programs) is assured statically. XMlambda also takes a similar approach. There are no static type systems for shuffle expressions; by taking a dynamically typed approach, we make it possible to perform shuffle expression pattern matching.

By taking approaches of constructing XML-specific languages, we may design programming languages suitable for XML processing; however, it is difficult to perform complex and flexible computation. We take the approach of designing an XML-specific language and implementing it "on" the existing programming language. The XML-specific language still makes use of all the features of the host language (Lisp), and we may also invoke the functions of XML-specific language seamlessly from Lisp.

The relationship between these related approaches and our's is shown in Table 1.

6. Concluding remarks

We show a new approach for developing XML processing toolkit. By representing XML documents as S expressions, we are free from designing and implementing API for XML processing because it can be directly accessed using Lisp primitives. Shuffle expression pattern matching is a powerful language construct. Example programs shown in this paper are simple but adequate to show how our embedded XML transformation language accepts XML documents with schema equipped with interleave types such as RELAX NG. They also show how we can write flexible pro-

Table 1. Related work

	General purpose languages	XML-specific languages
Dynamic validation (with shuffle)	**Our approach**	
Static validation (without shuffle)	Haskell	XDuce, XMLambda, YAT
No systematic validation	DOM, Franz Inc. , SXML	XSLT

grams with features of Lisp, how XML transformation rules can be invoked from Lisp programs seamlessly, and how we can write robust programs exploiting dynamic validation.

We consider our embedded XML transformation language is suitable for applications involving complicated processing. We plan to implement some large scale applications using it.

Acknowledgements. Information-technology Promotion Agency, Japan partially funded this research. Hidehiko Masuhara and Atsushi Igarashi gave very helpful comments on the earlier version of our software.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[2] Franz Inc. A Lisp Based XML Parser. http://www.franz.com.

[3] H. Hosoya and B. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–80, January 2001.

[4] H. Hosoya and B. Pierce. XDuce: A Typed XML Processing Language. In *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, pages 226–244, May 2000.

[5] J. Jedrzejowicz and A. Szepietowski. Shuffle languages are in P. *Theoretical Computer Science*, 250:31–53, 2001.

[6] T. Kamina, T. Yuasa, and T. Tamai. A Light-Weight Programming Interface for XML. In *Proceedings of the Fourth Workshop on Internet Technology (WIT2001)*. Japan Society for Software Science and Tecknology (JSSST), September 2001.

[7] E. Meijer and M. Shields. XMlambda: A Functional Language for Constructing and Manipulating XML Documents. Submitted to USENIX 2000 Technical Conference, 1999.

[8] OASIS. RELAX NG Specification. http://www.oasis-open.org/committees/relax-ng/, November 2001.

[9] J. Sim'eon and S. Cluet. Using YAT to Build a Web Server. In *Proceedings of International Workshop on the Web and Databases (WebDB'1998)*, 1998.

[10] SXML. http://okmij.org/ftp/Scheme/SXML.html.

[11] W3C. Namespaces in XML. http://www.w3.org/TR/1999/REC-xml-names-19990114/.

[12] W3C. XML-QL: A Query Language for XML. http://www.w3.org/TR/NOTE-xml-ql/.

[13] W3C. XML Schema. http://www.w3.org/XML/Schema.

[14] W3C. Document Object Model (DOM) Level 1 Specification. http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/, 1998.

[15] W3C. Extensible Markup Language (XML) 1.0 2nd edition. http://www.w3.org/TR/2000/REC-xml-20001006, 1998.

[16] W3C. XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, 1999.

[17] W3C. XQuery 1.0 and XPath 2.0 Data Model. http://www.w3.org/TR/query-datamodel/, 2001.

[18] M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*, pages 148–159, September 1999.

A. Formalization

We formally define the syntax and operational semantics of shuffle expression pattern matching here.

A.1. Syntax of patterns

A shuffle expression pattern is an S expression whose head is an XML element, a *shuffle*$_1$ expression pattern, or a variable:

```
pattern ::= (element pattern*)
            shuffle1
            variable
```

A *shuffle*$_1$ expression pattern is a regular expression pattern, or S expression whose head is an shuffle operator or shuffle closure operator and whose tail is regular expression patterns. A regular expression pattern is an S expression whose head is `seq`, `or`, `*`, `+` or `?`, or a fragment of XML document.

```
shuffle1 ::= ('%' regular+)
             ('&' regular)
             regular

regular ::= ('seq' regular+)
            ('or' regular+)
            ('*' regular)
            ('+' regular)
            ('?' regular)
            sexpr
```

```
sexpr   ::= (element sexpr*)
            string
            variable
```

`element` is a string prefixed by '`:`'. `variable` is a string prefixed by '`$`'.

A.2. Operational semantics of shuffle expression pattern matching

We provide an internal form of operational semantics of shuffle expression pattern matching. Internal tree values t are defined as follows:

t	::=	ϵ	leaf
		$(l\ t^*)$	label

A label is an XML element (and its attributes). t^* means a sequence of trees. Sometimes t^* denotes $(t_1 \cdots t_n)$ but sometimes it also denotes $t_1, \cdots, t_n$. We do not distinguish them in the internal form.

Even though we presented a syntax of pattern matching above, to make discussion simple, we provide following definitions of internal form of patterns:

P	::=	x	variable
		ϵ	leaf
		$(\%\ P\ P)$	shuffle
		$(\&\ P)$	shuffle closure
		$(seq\ P^*)$	sequence
		$(or\ P\ P)$	selection
		$(*\ P)$	Kleene's closure
		$(l\ P^*)$	label

$\%$, $\&$, seq, or, and $*$ are same as `%`, `&`, `seq`, `or`, and `*`. We define $\%$ and or only in the case of binary trees, but it can be generalized. l is an XML element (and its attributes).

The semantics of pattern matching is similar to the regular expression pattern matching of XDuce[3]. The difference is we provide the rules for shuffle (MAT-SFL) and shuffle closure (MAT-SFC). The semantics of pattern matching is given by the acceptance relation and matching relation. The acceptance relation $t^* \in P$ is read "sequence of value tree t^* is an instance of pattern P". The matching relation of pattern P is defined as $t^* \in P \Rightarrow V$ that is read "substitution of variables within P is defined as V". V is a map from variables to value trees (or sequences of value trees). $t \in P \Rightarrow V$ is a special case of $t^* \in P \Rightarrow V$. Matching relation of sequence of patterns P^* is defined as $t^* \in P^* \Rightarrow V$. It is important to distinguish P^* (sequence of patterns) and $(*\ P)$ (Kleene's closure of pattern). The rules of matching relations are defined as follows:

$$t^* \in x \Rightarrow \{x \mapsto t^*\} \qquad \text{(MAT-BND)}$$

$$\epsilon \in \epsilon \quad \Rightarrow \quad \emptyset \qquad \text{(MAT-EPS)}$$

$$\frac{t_1^* \in P_1 \Rightarrow V_1, \cdots, t_n^* \in P_n \Rightarrow V_n}{(t_1^* \cdots t_n^*) \in P_1, \cdots P_n \Rightarrow V_1 \cup \cdots \cup V_n} \qquad \text{(MAT-CDR)}$$

$$\frac{t_1^* \in P \Rightarrow V_1, \cdots, t_n^* \in P \Rightarrow V_n}{(t_1^* \cdots t_n^*) \in (*\ P) \Rightarrow V_1 \cup \cdots \cup V_n} \qquad \text{(MAT-CLS)}$$

$$\frac{t_1^* \in P \Rightarrow V_1, \cdots, t_n^* \in P \Rightarrow V_n}{t_1^* \odot \cdots \odot t_n^* \in (\&\ P) \Rightarrow V_1 \cup \cdots \cup V_n} \qquad \text{(MAT-SFC)}$$

$$\frac{t^* \in P^* \Rightarrow V}{t^* \in (seq\ P^*) \Rightarrow V} \qquad \text{(MAT-SEQ)}$$

$$\frac{t^* \in P_1 \Rightarrow V}{t^* \in (or\ P_1\ P_2) \Rightarrow V} \qquad \text{(MAT-OR1)}$$

$$\frac{t^* \in P_2 \Rightarrow V}{t^* \in (or\ P_1\ P_2) \Rightarrow V} \qquad \text{(MAT-OR2)}$$

$$\frac{t_1^* \in P_1 \Rightarrow V_1 \qquad t_2^* \in P_2 \Rightarrow V_2}{t_1^* \odot t_2^* \in (\%\ P_1\ P_2) \Rightarrow V_1 \cup V_2} \qquad \text{(MAT-SFL)}$$

$$\frac{t^* \in P^* \Rightarrow V}{(l\ t^*) \in (l\ P^*) \Rightarrow V} \qquad \text{(MAT-LAB)}$$

We write $t^* \in P \Rightarrow (x \mapsto u)$ when $t^* \in P \Rightarrow V$ and $V(x) = u$. $\{x \mapsto u\} \cup \{x \mapsto v\}$ is interpreted as $\{x \mapsto (u\ v)\}$.

These matching rules are ambiguous: a single pattern can be matched in different ways. We resolve this ambiguity by adopting a "longest match" policy where patterns appearing earlier have higher priority.