A Light-Weight Programming Interface for XML

Tetsuo Kamina[1] Taiichi Yuasa[2] Tetsuo Tamai[3]

[1]Graduate School of University of Tokyo
kamina@graco.c.u-tokyo.ac.jp
[2]Graduate School of Kyoto University
yuasa@kuis.kyoto-u.ac.jp
[3]Graduate School of University of Tokyo
tamai@graco.c.u-tokyo.ac.jp

Abstract

Programming interface in general and particularly for XML data manipulation should be simple and flexible. For this purpose, we introduce light-weight and flexible programming interface for XML that provides only some basic operations such as controlling XML parser and XML document generator. Our programming toolkit represents XML documents as S expressions internally; therefore, XML application programs can be simply coded as list processing, and we can make use of advantage of using Lisp such as treating data and programs uniformly.

1 Introduction

XML is a markup language that can be statically typed by DTDs [11]. Static 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.

Current XML processors tend to be based on Document Object Model (DOM) specification [12] and/or Simple API for XML (SAX) [9]. However, DOM and SAX objects require complicated interface to process them. An alternative approach to develop XML processors is to provide only some basic programming interface such as controlling XML parser and XML document generator, and leave processing of internal representation of XML documents to a general purpose programming language. This approach makes it possible to construct light-weight and flexible programming interface for XML. For this purpose, we chose S expressions of Lisp to represent internal structures of XML documents. As tree structures of XML documents can be naturally expressed by S expressions, it is easy to write XML application programs in Lisp, by treating data and programs uniformly.

In this paper, we show our light-weight and flexible programming interface for XML that represents XML documents as S expressions internally, and implementations based on the interface including a validating XML parser, a validating XML generator, and some database programming tools[1]. First of all, after presenting our design goal, we show the architecture of this programming toolkit. Next, we describe how our XML processor validates XML documents using the *XML element server* which acts like DTDs in main memory. Then, examples of using our toolkit are presented, followed by some discussion on why we take this approach as an alternative to DOM and SAX. Appendix A shows the syntax of S expressions used in our programming toolkit. The programming interface of XML parser, XML document generator, and some database programming tools are shown in Appendix B. Programming toolkit presented here is implemented in Allegro Common Lisp 6.0 [5].

[1]The XML programming toolkit from this paper is available on the WWW at `http://www.graco.c.u-tokyo.ac.jp/~kamina/xmltools/`.

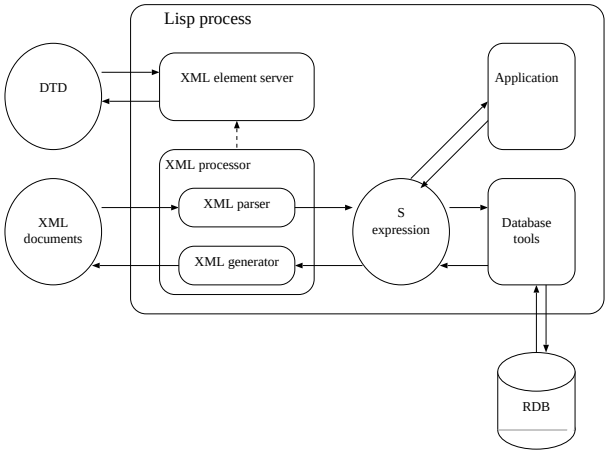

Figure 1: Architecture of the XML programming toolkit

2 Design Goal and Architecture

Programming interface in general and particularly for XML data manipulation should be simple and flexible. For this purpose, our design goal of XML programming toolkit is to provide only some basic programming interface such as controlling XML parser and XML document generator, and leave processing of internal representation of XML documents to a general purpose programming language.

For this design goal, we chose S expressions of Lisp to represent XML documents internally. This makes it possible to manipulate internal representations of XML documents using basic list processing operations such as `car`, `cdr`, and `cons`. The architecture based on our design goal is shown in Figure 1. The XML processor consists of the XML parser and the XML generator for parsing XML documents and generating XML documents, respectively. The database tools transform RDB schema into document types automatically, and output the result of queries for RDB as S expressions. XML applications and the XML processor may be executed in the same Lisp process.

One problem of representing XML documents as S expressions is that it is not possible to validate them without further information. We introduce the *XML element server* to solve it. The XML processor may ask XML element server to validate XML documents which are parsed or generated. We show the detail of XML element server in the next subsection.

2.1 Validating XML Documents

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 alist 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 which is specific in that document type. XML elements are also represented as CLOS instances. For example, the instance representing XML element `foo` in Figure 3 is shown in Figure 2:

The `children` slot (member) plays the main role of validating XML documents. It rep-

```
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 2: CLOS instance for element declaration

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

Figure 3: An example of element declaration

resents an automaton converted from the regular expression in the declaration of Figure 3[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.

Next section shows how XML documents are expressed in S expressions in our toolkit.

3 Examples

In our toolkit, the XML document shown in Figure 4 can be represented by S expressions shown in Figure 5. In this example, XML element `<head>` is represented as `:head`, and

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

is represented as

```
(:html :xmlns "http://www.w3.org/1999/xhtml")
```

and so on. The precise syntax of S expressions used in our programming toolkit is shown in Appendix A. This internal representation of XML documents can be controlled by Lisp (using Lisp operations such as `car`, `cdr`, and `cons`).

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

```
<html xmlns="http://www.w3.org/1999/xhtml">
 <head>
  <title>Hello</title>
 </head>
 <body>
  <h1>Hello, World!</h1>
  I'm in Tokyo.
 </body>
</html>
```

Figure 4: An example XML document

```
((:html :xmlns "http://www.w3.org/1999/xhtml")
 (:head (:title "Hello"))
 (:body (:h1 "Hello, World!")
        "I'm in Tokyo."))
```

Figure 5: An example S expression representing XML document

These S expressions may be constructed by programs mechanically, or we may write them by hand. Furthermore, with backquote, we can embed any Lisp expressions inside these S expressions [2]. For example, in Figure 6, `(increment-access-counter)` will be evaluated because it is associated with comma (`,`); however, the whole S expression will not be evaluated because it is associated with backquote (`` ` ``).

4 Discussion

Although this S expression based programming interface looks straightforward, we found that there are several issues to be discussed.

Some may say our interface may be simple to Lisp programmers but not to everyone. How can it be concluded that our interface is simpler than DOM or SAX? In fact, DOM and SAX objects require complicated programming interface (such as `element.getTagName()`,

```
`((:html :xmlns "http://www.w3.org/1999/xhtml")
  (:head (:title "Hello"))
  (:body (:h1 "Hello, World!")
         "You are "
         ,(increment-access-counter)
         " visitor."))
```

Figure 6: An example of using backquote and comma

```
(:html
 (:head (:title $title))
 (:body (:h1 $title)
        $content))
```

Figure 7: An example pattern of XML documents

`element.getElementsByTagName()`, and `attribute.getValue()`) to process them. Although these programming interface is well-organized, these methods are not general in Java or C++ API. In contrast with them, our approach makes it possible to code XML application programs using basic list processing operations such as `car`, `cdr` and `cons`, or users may define their own functions to process some pieces of S expressions. This is why we call our programming interface light-weight and flexible.

Furthermore, our approach may take an advantage of using Lisp such as treating data and programs uniformly. S expressions are writable by hand; therefore, we can code programs as shown in Figure 6. In contrast with our approach, constructing DOM tree using Java or C++ requires much harder work.

5 Related Work

Wallance and Runciman proposed to use Haskell for XML processing [13]. Like our programming toolkit, this XML toolkit embeds XML processing in Haskell language. The difference from our approach is that they use Haskell type system instead of a dictionary; they add mapping from DTDs into Haskell datatypes to Haskell constructs.

XDuce [3] and XMLambda [8] are statically typed functional languages. These are similar to Haskell's approach except that these are XML-specific languages.

Haskell or other functional languages use type systems to map DTDs into language's data types. This approach looks straightforward to represent static typing in XML; however, although XML may be statically typed, such typing is not always necessary. XML documents without DTDs are also useful even if these are well-formed but not valid. XML1.0 recommendation says that validation of XML documents should be enabled via user's option. Our approach using a dictionary looks more suitable to process XML documents which are well-formed but not valid; it can simply be done by not using a dictionary.

Franz Inc. has released non-validating XML parser on lesser GNU public license (LGPL). It checks well-formedness of XML documents and converts them into S expressions called Lisp XML (LXML) [4]; however, it does not validate XML documents. Instead, when it parses DTD files, it generates corresponding S expressions (of course, they are different from S expressions of XML documents), and leaves responsibility of validation to the developers. Although well-formed but invalid XML documents are also useful, one of the novel features of XML is the ability of typing using DTDs. This ability should not be discarded. Another difference of Franz's tool from our toolkit is that Franz's tool is just a parser but not a processor. XML processors should also be able to generate XML documents via application programming interface.

6 Conclusion and Future Work

Our approach of developing XML processors is an alternative to DOM and/or SAX based approaches. Our approach makes it possible to construct light-weight and flexible XML programming interface, and take advantage of using Lisp such as treating data and programs uniformly.

This programming toolkit is intended to be used for constructing the base technology for XML application programming on Lisp. For the next step, using this toolkit, we plan to embed XML transformation language in Common Lisp. S expressions can be used for pattern matching. An example of pattern of XML documents is shown in Figure 7 (In this figure, notations prefixed by `$` mean variables). Using pattern matching of trees, its notation will become very simple and easy to understand for developers who have some experience of functional languages. We also consider a support for customizing conversion rules from database schema to DTD, and other document type declaration schemes like XML Schema [10] and RELAX [1].

7 Acknowledgements

Information-technology Promotion Agency, Japan funded this line of work. Hidehiko Masuhara and Atsushi Igarashi gave very helpful comments on an earlier version of this software.

References

[1] ISO/IEC DTR 22250-1. Document Description and Processing Language – Regular Language Description for XML (RELAX) – Part1: RELAX Core, 2000.

[2] Paul Graham. *ANSI Common Lisp*. Prentice Hall, 1996.

[3] Haruo Hosoya and Benjamin Pierce. XDuce: A Typed XML Processing Language. In *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, 2000.

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

[5] Franz Inc. Allegro Common Lisp. http://www.franz.com.

[6] Franz Inc. Allegro ODBC. http://www.franz.com.

[7] Franz Inc. AllegroServe. http://allegroserve.sourceforge.net.

[8] Erik Meijer and Mark Shields. XMLambda: A Functional Language for Constructing and Manipulating XML Documents. In *USENIX Annual Technical Conference*, 2000.

[9] SAX. http://www.megginson.com/SAX/index.html.

[10] W3C Web Site. XML Schema. http://www.w3.org/XML/Schema.

[11] W3C Web Site. Extensible Markup Language (XML) 1.0. http://www.w3.org, 1998.

[12] W3C Web Site. Document Object Model (DOM) Specification. http://www.w3.org, 2000.

[13] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *Proceedings of the International Conference on Functional Programming*, 1999.

A S Expression Representing XML Documents

The syntax of S expressions used in our programming tool is as follows:

```
S_Expression ::= '(' Element Space Content ')'
Element ::= ElementName | '(' ElementName Space AttributeList ')'
AttributeList ::= ( AttributeName Space Value )*
Content ::= ( String | LispCommand | S_Expression )*
ElementName ::= Keyword
AttributeName ::= Keyword
```

`Space` is the same as the white space found on the XML specification [11]. `ListCommand` is Lisp expressions which are to be evaluated when XML documents are generated. `Keyword` is the same as Lisp symbols which are interned in the *keyword* package.

B The Light-Weight API for XML Programming

The light-weight XML programming interface is described in this section.

B.1 API for Validating XML Parser

The programming interface of XML parser is as follows:

```
(parse-xml stream &key validate)
```

The `stream` parameter is the input stream in which the XML documents are sent. If `validate` is set to `t`, the XML parser validates the XML documents after converting them into S expressions.

B.2 API for Validating XML Generator

The programming interface of XML generator is as follows:

```
(generate-xml-with-stream dest doctype s-expr &key validate public system)

(generate-xml doctype s-expr &key validate public system)
```

These two functions generate XML documents from S expressions passed to `s-expr`. The first function, `generate-xml-with-stream`, allows users to specify the output stream by `dest`. If `validate` is set to `t`, the XML generator validates XML documents before it outputs XML documents to the stream. The `public` and `system` keyword parameters are used to specify the public identifier and the system identifier, respectively. The second function, `generate-xml`, outputs the resulting XML documents to the Web server [7].

B.3 API for Manipulating DTDs

The programming interface for manipulating DTDs is as follows:

```
(parse-dtd-only stream doctype)

(remove-doctype doctype)
```

The function `parse-dtd-only` parses DTD files which are sent to input stream `stream`, then registers them into the XML element server. Document type name is specified by the `doctype` parameter. The function `remove-doctype` removes a document type which is specified by the `doctype` parameter from the XML element server.

```
<booklist>
<book id="1"
      author="Paul Graham"
      title="ANSI Common Lisp"
      publisher="Prentice Hall"/>
<book id="2"
      author="Bob DuCharme"
      title="XML: The Annotated Specification"
      publisher="Prentice Hall"/>
</booklist>
```

Figure 8: Resulting XML for Query

B.4 API for Database Programming Tools

In this section, the API for integration of databases and XML applications is shown. It consists of the following three functions:

```
(register-dtd-from-db data-source &key db)

(sqltoxml query data-source &key db)

(publish-dtd data-source)
```

The first function is intended to define a document type from RDB schema automatically. `db` is a database connection object which is connected to the database whose data source name is `data-source` via Allegro ODBC interface [6]. The definition rules are as follows:

- The document type name is the data source name.
- The root element is declared as follows:
  `<!ELEMENT data-source (table_1| ... |table_n)+>`
- Each `table_i` is declared as follows:
  `<!ELEMENT table_i EMPTY>`
- Every column of the relation `table_i` is declared as XML attribute of XML element `table_i`:
  `<!ATTLIST table_i column_j CDATA #IMPLIED>`

The second function, `sqltoxml`, outputs the result of a `query` as S expressions. Inputting such S expressions to XML generator, we can get an XML document like Figure 8 as a result.

The third function, `publish-dtd` automatically generates DTD from the XML element server.