國立東華大學資訊工程碩士在職專班

碩士論文

應用編譯技術自動產生 XML 驗證器

Automatic Generation of XML Validators By Applying Compiler Techniques

研究生：葉日進

指導教授：雍　忠 博士

中華民國九十二年七月

Automatic Generation of XML Validators by Applying Compilation Techniques

by

Jih-Chin Yeh

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science and Information Engineering

National Dong Hwa University

July 2003

Approved: _____

Chung Yung

To my parents and my wife.

Acknowledgements

It is with grateful appreciation that I acknowledgement the following who support me to complete this thesis. A special appreciation to Professor Chung Yung, my advisor. He shared his erudite knowledge and warming friendship to me. I appreciate to thesis commitee members, Professor Nai-Wei Lin, Professor Shih-Chieh Chou, for the valuable suggestions to the thesis.

I give thanks to CTA laboratory members, Shin-Lin Chen, Ming-Hao Chiang, Shiang-Wei Kung, Li-Pang Chen, Per-Jen Chuang, Chih-Chiang Lee, Ming-Sian Lin, Fung-Wei Yang, You-Liang Sun, and Yun-Lung Yue. A special thanks to Yu-Ching Lan and Hsun-Her Wang.

I give thanks to my parents Shui-Yuan and Kuei-Mei, my sons Ron and Barry, especially my wife Hsiu-Ping Lin, they give me most powerful supports.

Last but not least, I give thanks to my colleagues at National Hua-Lien Agricultural High School for helping me during this three years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Extensible Markup Language (XML) is a subset of Standard Generalized Markup Language (SGML). Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML [DTD98]. Figure 1.1 shown the contentment relationship diagram among Markup Languages.

Since its first introduction on February 1998 by W3C, XML had five years old today. The XML has become everywhere that information is managed. There are some informal descriptions about the usage of XML:

1. A markup language is a mechanism to identify structures in a document. The XML specification defines a standard way to add markup to documents [MER01].

2. XML is a markup language for documents that contain structured information. Structured information contains both content and the indication of what role that content plays [WAL98].

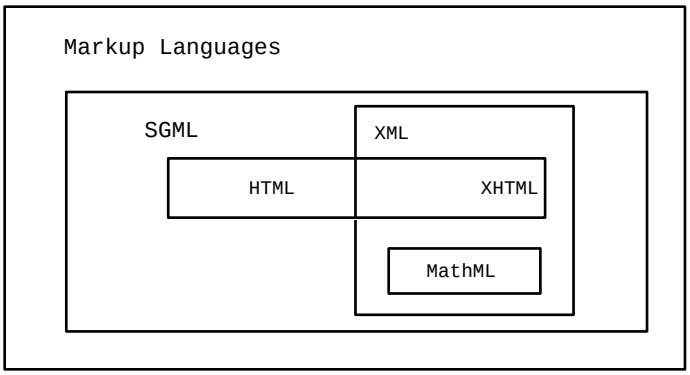

Figure 1.1: Contentment Relationship Diagram among Markup Languages.

3. XML is the universal format for data on the Web. XML allows developers to easily describe and deliver rich, structured data from any application in a standard, consistent way.[MSD03]

1.1 Motivating Examples

An XML document is tagged into a tree of nested elements [Chu01]. Figure 1.2 shows a brief example of XML document. The root element of these document is $< addrbook >$. It has two child elements, named $< person >$. Elements $< name >$ and $< tel >$ are direct descendants of $< person >$. All of these elements are nested properly, we say that it is a *well-formed* document.

XML does not replace HTML; rather, it is a complementary format. XML is not Just Like HTML. In HTML, both the tag semantics and the tag set are fixed. XML provides a facility to define tags and the structural relationships between them. Since there is no predefined tag set, there can not be any preconceived semantics. All of the semantics of an XML document will either

```
<?xml version="1.0"?>
<addrbook>
 <person>
   <name>Jerry</name>
   <tel>0987654321</tel>
 </person>
 <person>
   <name>adams</name>
 </person>
</addrbook>
```

Figure 1.2: An XML Document Example.

```
<?xml version="1.0"?>
<!DOCTYPE addrbook [
<!ELEMENT addrbook (person*) >
<!ELEMENT person (name , tel?) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT tel (#PCDATA) >
]>
```

Figure 1.3: A Document Type Definition.

be defined by the applications that process them or by stylesheets.

The whole concept of XML is to separate data from presentation and to provide a standard format that describes itself in a language both machines and people are comfortable.

XML is extensible because each XML document can include a document type definition (DTD) [DTD98] which lists the tags of the elements and specifies the tagging constraints of the document. Figure 1.3 shows an example of DTD.

One of the keys to XML document processing is *validation* [CP01]. An XML document is *well-formed* if each of its element is enclosed with matching start-tag and end-tag. An XML document is valid if its content is tagged with

the constraints specified in the DTD.

Being well-formed means their tags nested properly, never overlap. But being well-formed does not enough to ensure the meanings of elements. Any documents must have some kind of structure to organize the meanings. An XML document may not covered every elements which the structure possibly appear. In contract, a well-defined DTD includes all elements which may appear within documents. A DTD also constrains the element's attributes and relations between elements.

An XML processor is used to read XML documents, processing their content and structure, and providing the information to the application. A validating XML processor reports the violations of contents in the document according to DTD's constrain.

A valid XML document that the static typing provided by DTDs will improve the safety and correctness of data exchange and processing.

The XML document type definition contains the markup declarations that provide a grammar for a class of documents. Logically, the document is composed of declarations, processing instructions, elements, comments, and character references, all of which are indicated in this document by explicit markups. Validating processors report the violations of the constraints expressed by the DTD, and the failures to fulfill the validity constraints given in the DTD. Only a few XML processors include the validation checks while most XML processors only perform the well-formedness checks. Figure 1.4 shows the Validation Process. The validator must initially have its own def-

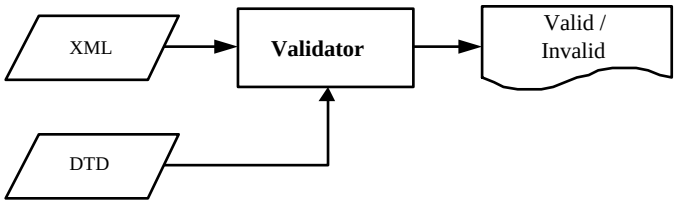

Figure 1.4: The Validation Process.

initions according to its DTD. XML document takes as input, and produces the message either *Valid* or *Invalid* as output.

1.2 Methodology

Present validators play as role of interpreter. Figure 1.4 shows the diagram of a general validator's position.

When an XML document is processed, validating processor must read the DTD that specified the type of the document, constructing the internal type, performing Subtyping algorithms, to decide whether the document is valid or not. Interpreter model of validator has the problems of slowly and inefficiency. Because of the DTD must be read and the type must be checked every times when we process XML documents. The general validator waste a lot of time and resources.

We propose a new approach to solve the XML validation problem. We transform a DTD into a context-free grammar and use the compilation tools to generate the XML validator automatically. With this model, we developed a context-free grammar transformer, called *VG* to generate the lex-like and yacc-like specific codes for the validators of DTDs.

1.3 Organization of Thesis

The remains of thesis is organized as follows:

Chapter 2 provides some background information and examples to explain validation problem with XML and DTD. Section 2.2 describes Regular Expression, Context-Free Grammar, and tools for compilation. Section 2.3 states the property of functional programming, we provided information of compilation tools which are active functional programming languages.

Chapter 3 developed a framework for XML Validator Generator. We described the framework in both syntactic and semantic ways.

Chapter 4 describes the implementation details of our framework. There are four modules in original codes of validator generator, scanner module, parser module, parse tree constructive module, and the translator for validator generator.

Chapter 5 provides experiments of our validator generator both in time and space.

Chapter 6 gives some of the significant related work in the problem of validation for XML document.

Conclusions and future directions are given in Chapter 7.

Chapter 2

Background

2.1 XML and DTD

An XML document is well-formed if it follows the definition in the specification of [XML00]. There is exactly one element, called the root, or document element, no part of which appears in the content of any other element. For all other elements, if the start-tag is in the content of another element, the end-tag is in the content of the same element. More simply stated, the elements, delimited by start-tags and end-tags, nested properly within each other. The document listed in Figure 1.2 is a well-formed XML document because it begins with <?xml version="1.0"?>, which declared it is an XML document, and follow by the start-tag <addrbook> and end-tag </addrbook> as root element which enclosed all other elements.

Being well-formed just means the document followed the basic XML syntax rules. A well-formed XML document is *valid* if it has an associated document type definition, DTD, and if the document complies with the constraints ex-

pressed in the DTD. Figure 1.3 is a DTD that constraints XML documents. The document shown in Figure 1.2 is just one of that conformed to this DTD. The root element *addrbook* may contents zero or more *person* element. The *person* element must contents just one *name* element, follows zero or one *tel* element. The *name* element and the *tel* element may only content Parsed Character Data within it.

Documents may make a reference to an external DTD or an internal DTD. Listed below are the syntax with parameters *SYSTEM* and *PUBLIC* respectively.

<!DOCTYPE element_name SYSTEM DTD_URL>

<!DOCTYPE element_name PUBLIC DTD_name DTD_URL>

DTD may appear directly between XML declaration and element declarations of the XML document. Internal DTD declaration follows informal grammar listed below:

<!DOCTYPE element_name [

<!ELEMENT element_name (contents model)>

<!ATTRIBUTE element_name Attribute=value>

]>

2.1.1 Applications Embedded Modules as API for XML

Abstract data types and the accompanied library routines are designed for XML documents parsing and document creation, navigation, and manipulation. The XML data is assumed to be valid in a separate phase. Examples in this category include standard XML API in C, C++, Java, or other languages

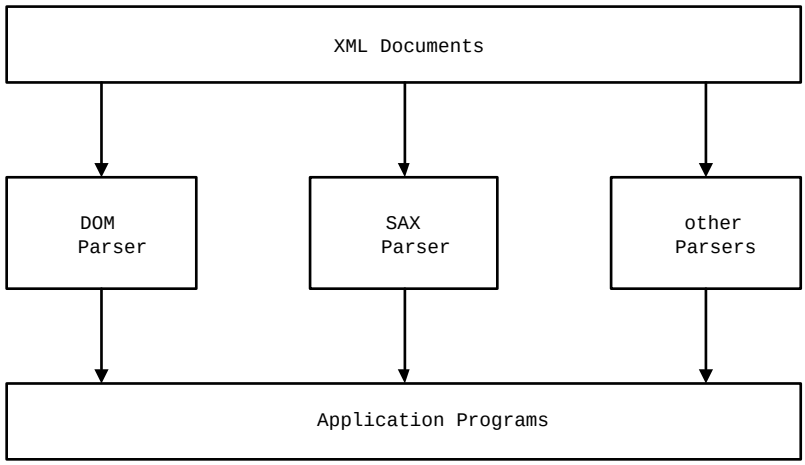

Figure 2.1: Parsers as Interface Between XML Documents and Applications.

(e.g., Document Object Model, DOM [DOM01].) Although many of them are non-validating processor, some of them are validating one.

There are some XML processor designing with strongly typed language, the type system of the language is used to embed DTDs. If the strongly typed language is statically typed, then the soundness proof is done by the type checker at compile-time. Hence no type-correct program will produce invalid XML elements [Chu01]. Examples in this kind included HaXml [WR03] and PXP. PXP means Polymorphic XML Parser. It emphasizes its most useful property that the API is polymorphic and can be configured such that different objects are used to store different types of elements.

2.1.2 Popularly Used DTDs

In recent years, XML is popularly used for data exchange over Internet. The associated DTD is developed as their standard. The Examples are listed below:

- Health Level Seven (HL7) construct health care data , including medical Publishing, messaging and clinical documents.[HL03].

- Interactive Financial Exchange (IFX) [IFX03] is a broad-based framework for exchanging financial data and instructions among customers, their Customer Service Providers (CSP), and financial service providers. They expect IFX to serve as the primary means of communication between Financial Institutions functioning as CSP's and their back-end service providers [IFX02].

- The XML Bookmark Exchange Language (XBEL) is an Internet bookmarks interchange format. It was designed by the Python XML Special Interest Group on the group's mailing list [DRA98].

- Chemical Markup Language (CML) is a new approach to managing molecular information using Internet tools such as XML and Java. It is based strictly on SGML [CML03].

- MathML 2.0, a W3C Recommendation was released on 21 Feb 2001. A product of the W3C Math working group, MathML is a low-level specification for describing mathematics as a basis for machine to machine communication. It provides a much needed foundation for the inclusion of mathematical expressions in Web pages [MAT03]

- RosettaNet, an internet based common messaging standard for global supply chain management. It enables companies in the supply chain to communicate and conduct business electronically through common codes

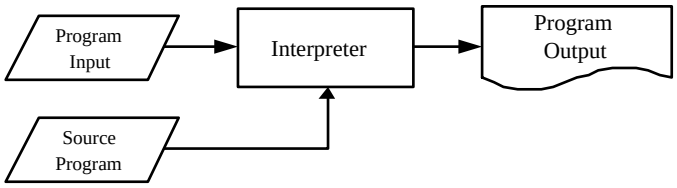

Figure 2.2: An Interpreter.

for sourcing of parts and components [ROS03].

2.2 Techniques in Compiling

An interpreter runs the source program and do something the program specified which always take something for input, and produce another things for output. For example, a *BASIC* interpreter runs source program named *sort.bas*, taking a list of unsorted number *3,1,2,*$\cdots$ as input, and producing a list of sorted number *1,2,3,*$\cdots$ as output. Figure 2.2 shows an interpreter.

A compiler reads a source program and translates it into an equivalent target program [ASU86]. Source programs are written by source languages, target programs are written by target languages. After linked and loaded, the target programs may then executed for specified goals (e.g., sort). Figure 2.3 shows an compiler.

2.2.1 Regular Expression

The symbol that appear in regular expressions are the letters of the alphabet Σ, null string ϵ, parentheses (), kleene star *, and the plus sign +. The set of regular expressions is defined by the following rules [Coh97]:

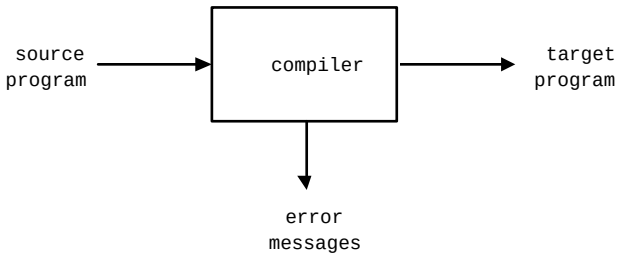

Figure 2.3: A Compiler. (adopted[ASU86])

Rule 1: Every letter of Σ can be made into a regular expression; including ϵ.

Rule 2: If r_1 and r_2 are regular expressions, then so are:

(r_1)	Parentheses
$r_1 r_2$	Concatenation
$r_1 + r_2$	Alternation
r_1*	Kleene-star

Rule 3: Nothing else is a regular expression.

Note that in DTD, plus sign have deferent meaning from rule 2 that shown above, that is: $r_1+ = r_1 r_1*$, and $r_1|r_2$ means r_1 or r_2.

Regular expression types are proposed for describing the structures in XML documents using regular expression operators (i.e., *, ?, |, etc.). Regular expression types are proposed as a foundation for statically typed processing of XML documents [HVP00]. Hosoya et al. designed a new programming language called XDuce that similar to mainstream functional languages but specialized to the domain of XML processing. They took the facilities of static types and pattern matching of ML-like programming language, and get

the novel features in regular expression types and a corresponding mechanism for regular expression pattern matching.

2.2.2 Context-Free Grammar

A Context-Free Grammar, CFG, is a collection of three things [Coh97]:

1. An alphabet Σ of letters called *terminals* .

2. A set of symbols called *nonterminals*, one of which is the symbol S, standing for "start symbol".

3. A finite set of *productions* of the form:

   One nonterminal $\rightarrow$ *finite string of terminals and/or nonterminals.*

   We require that at least one production has the nonterminal S as its left side.

Many programming language constructs have an recursive structure that can be defined by CFG. For example, we might have a conditional statement defined by a rule such as

> If S_1 and S_2 are statements and E is an expression, then
> if E then S_1 else S_2 is a statement.

This form of conditional statement cannot be specified using the notation of regular expressions. On the other hand, using a *nonterminal* symbol *stmt* to denote statements, we can have readily express using the CFG production:

> *stmt* $\rightarrow$ if *expr* then *stmt* else *stmt*

2.2.3 Tools For Compilation

Lex-like tools are scanner generators that automatically generate lexical analyzers, normally from a specification based on regular expressions. The basic organization of the resulting lexical analyzer is in effect a finite automaton [ASU86].

Yacc-like tools are parser generators that produce syntax analyzers, normally from input that is based on a context-free grammar. Many parser generators utilize powerful parsing algorithms that are too complex to be carried out by hand [ASU86].

2.3 Functional Programming

R. Sethi[SET00] classified programming paradigms into the following categories:

1. Imperative Programming: There are *Fortran*, *Pascal* and *C* in the imperative family.

2. Functional Programming: The basic concepts of functional languages originated with *Lisp*, a language designed in 1958 for applications in artificial intelligence. Another functional programming languages are *ML*, *Scheme*, *Haskell*,etc.

3. Object-Oriented Programming: *Simula* change the way people think about programming. C++, Java and Smalltalk are popular languages for object-oriented programming as well.

4. Logic Programming: *Prolog* was developed in 1972 for natural language.

Functional programs can be charmingly simple and surprisingly powerful. The simplicity comes from the emphasis on values, independent of an underlying machine with its assignments and storage allocation. The power comes from recursion and the status of functions as "first-class" values. The absence of explicit code for deallocation makes programs simpler and shorter. The power of functional programming is due in part to first-class functions. Functions have the same status as any other values. A function can be the value of an expression, it can be passed as an argument, and it can be put in a data structure.

Chapter 3

An XML Validation Framework With A Validator Generator

To overcome problems that we stated in section 1.1, we proposed a framework for automatic generation of XML validator . Figure 3.1 illustrated the framework.

We created CFG for a Validator Generator in Phase 0 of Figure 3.1. The CFG is the mapping result from the structure of DTD. With A proper compiler

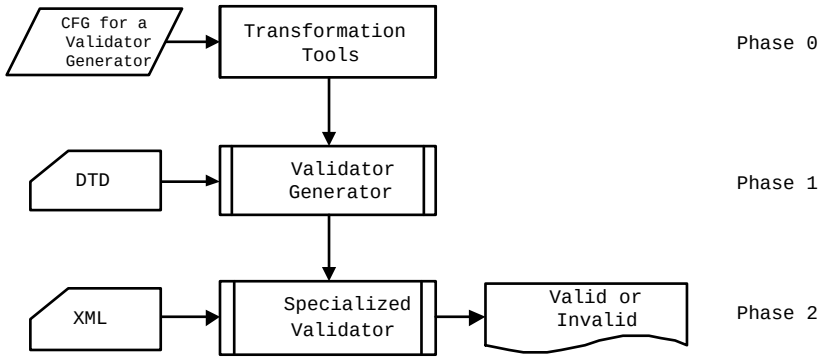

Figure 3.1: A Framework for a Validator Generator.

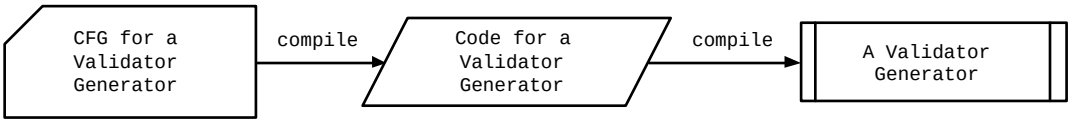

Figure 3.2: Process of Generating A Validator Generator.

tools, the CFG can be transformed to a Validator Generator. This Generator can always read specific DTD as input, and generate a Validator that was specified with this DTD.

There are two phases to perform validating action. In the Phase 1, we generate the specialized Validator according to the specific DTD. In Phase 2, XML documents are validated by this specialized Validator. All of this sort of XML documents will be validated with just phase 2, there is no need phase 1 except the DTD had been changed.

3.1 Phases

First of all, we code the CFG for a Validator Generator by hand. Figure 3.2 shows the process of generating a validator generator.

With compilation tools (Lex- and Yacc-like), we translate CFG into the source code for a Validator Generator in a kind of programming language. With this programming language compiler, we can get the executable Validator Generator.

We listed the separated phase of whole framework as below.

1. In phase 1, the Validator Generator takes a DTD as input, and produces another CFG for specialized Validator as output. The output CFG can

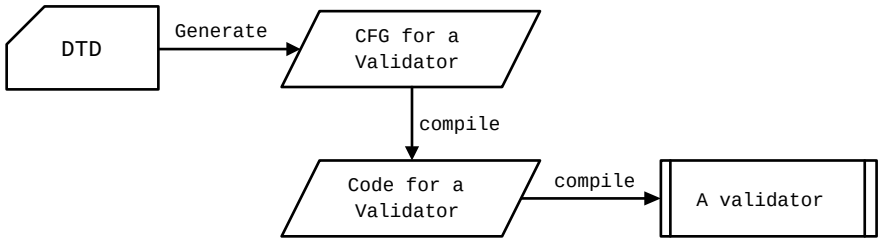

Figure 3.3: Phase of Generating Validator.

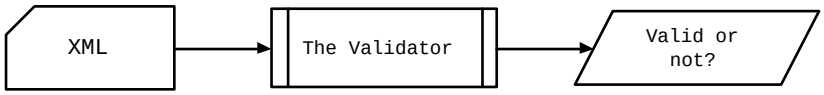

Figure 3.4: Phase of Validation.

then be translated into the source code for a Validator in a kind of programming language. Therefor, the codes can be compiled to a validator. Figure 3.3 shows the process of phase 1.

2. In phase 2 of Figure 3.1, the Validator is ready for the class of XML documents. Any documents which is required for validation will validate directly by the Validator. Figure 3.4 shows the usage of Validator.

There is no need to read DTD whenever we validate this class of XML documents, hence we saved the time cost for DTD processing, this makes our framework valuable.

3.2 Transformation Functions

Constructs in a language is described by a combination of formal syntax and informal semantics [ASU86]. We will try to form this validator generator in manners of both syntactics and semantics.

Abstract Syntax

c	$\in$	Con	Constants, where $Con = \{$"#PCDATA"$\}$
t	$\in$	Tag	User defined tag in DTD
p	$\in$	Po	Regular Expression Operator, where $p = *, ?, +, \epsilon$
e	$\in$	Exp	Expressions, where $e = c \mid t \mid e\ p \mid e' \mid' e \mid (e) \mid e, e$
d	$\in$	Def	Element definition, where $d = t\ e$
dtd	$\in$	Dtd	Data Type Definition, where $dtd = \{d_i\}$

Figure 3.5: Abstract Syntax of DTDs

3.2.1 Formal Syntax

We abstract the syntax of DTDs in Figure 3.5. Without loss of generality, we simplified the ENTITY declarations, the Processing Instructions, and the ATTLIST declarations. The *dtd* consist of element definition set, where every definition d consist of tag t follow by expression e. Expression e may either be constant c or consist of content model with regular expression operator p as shown in Figure 3.5.

3.2.2 Semantic Transformation

We usually use *function* to denote or represent meanings. The meaning of an expression E, written as $\mathcal{E}[\![E]\!]env$, is a function from environment to values. It should be read as a function $\mathcal{E}$ performed an expression [[E]] in the environment env, the results of function $\mathcal{E}$ will be a value which is what the function means.

Semantic Domains

$Env = Tag \rightarrow D$	Environment.
$Bl = Lab \rightarrow D$	Internal Bound Labels.
$D = \{\bot\} + \{\epsilon\} + \{Tag\} + \{Lab\} +$	
$\quad (DD) + (D\|D) + error$	Denotable values.

Semantic Functions for Computation of Symbol Tables

$$
\begin{array}{rcl}
\mathcal{K}_p & : & Con \rightarrow \{\bot\} \\
\mathcal{P}_p & : & Po \rightarrow D \\
\mathcal{E}_p & : & Exp \rightarrow Env \\
\mathcal{D}_p & : & Def \rightarrow Env \\
\mathcal{D}_{dtd_p} & : & Dtd \rightarrow Env
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{K}_p[\![c]\!] & = & \bot \\
\mathcal{E}_p[\![c]\!] & = & \mathcal{K}_p[\![c]\!] \\
\mathcal{E}_p[\![t]\!] & = & env[\![t]\!] \\
\mathcal{E}_p[\![e\ p]\!] & = & env[\mathcal{P}_p[\![e\ p]\!]/l], Bl[l/ep], \text{ where } l \text{ is a fresh label.} \\
\mathcal{E}_p[\![e_1\ '|'\ e_2]\!] & = & env[\mathcal{E}_p[\![e_1]\!] \mid \mathcal{E}_p[\![e_2]\!] \ /\ l], \\
 & & Bl[\ l\ /\ e_1\ '|'\ e_2], \text{ where } l \text{ is a fresh label.} \\
\mathcal{E}_p[\![(e)]\!] & = & \mathcal{E}_p[\![e]\!] \\
\mathcal{E}_p[\![e_1, e_2]\!] & = & env[\mathcal{E}_p[\![e_1]\!]\ \mathcal{E}_p[\![e_2]\!] \ /\ l], \\
 & & Bl[\ l\ /\ e_1, e_2], \text{ where } l \text{ is a fresh label.}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{P}_p[\![e\ *]\!] & = & e\ l \mid \epsilon, \text{ where } l \text{ is a fresh label.} \\
\mathcal{P}_p[\![e\ +]\!] & = & e\ l \mid e, \text{ where } l \text{ is a fresh label.} \\
\mathcal{P}_p[\![e\ ?]\!] & = & e \mid \epsilon \\
\mathcal{D}_p[\![t\ e]\!] & = & env[\mathcal{E}_p[\![e]\!] \ /\ t] \\
\mathcal{D}_{dtd_p}[\![\{d_i \rightarrow t_i\ e_i\}]\!] & = & \{\mathcal{D}_p[\![t_i\ e_i]\!]\}
\end{array}
$$

Figure 3.6: Semantic Functions for Computation of Symbol Tables

Semantic Functions for CFG Transformation

$$
\begin{array}{rcl}
\mathcal{K}_g & : & Con \rightarrow D \\
\mathcal{P}_g & : & Po \rightarrow D \\
\mathcal{E}_g & : & Exp \rightarrow D \\
\mathcal{D}_g & : & D \rightarrow D \\
\mathcal{D}_{dtd_g} & : & D \rightarrow D
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{K}_g[\![c]\!] & = & \perp \\
\mathcal{K}_g[\![t]\!] & = & t \\
\mathcal{P}_g[\![e\ p]\!] & = & Bl[e\ p] \\
\mathcal{E}_g[\![c]\!] & = & \mathcal{K}_g[\![c]\!] \\
\mathcal{E}_g[\![t]\!] & = & \mathcal{K}_g[\![t]\!] \\
\mathcal{E}_g[\![e\ p]\!] & = & \mathcal{P}_g[\![e\ p]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_g[\![e_1\ '|'\ e_2]\!] & = & Bl[\![\ l\ /\ e_1\ '|'\ e_2]\!] \\
\mathcal{E}_g[\![(e)]\!] & = & \mathcal{E}_g[\![e]\!] \\
\mathcal{E}_g[\![e_1, e_2]\!] & = & Bl[\![\ l\ /\ e_1, e_2]\!] \\
\mathcal{E}_g[\![t\ e]\!] & = & generate(t \rightarrow \mathcal{E}_g[\![e]\!]) \\
\mathcal{D}_g[\![d]\!] & = & \mathcal{E}_g[\![t\ e]\!], \text{ where } d \rightarrow t\ e \\
\mathcal{D}_{dtd_g}[\![dtd]\!] & = & \{\mathcal{E}_g[\![d_i]\!]\} + \{\mathcal{E}_g[\![l_i]\!]\}, \forall l_i \in Bl \\
 & & \text{where } d \rightarrow t\ e \text{ and } l \text{ is an internal bound label.}
\end{array}
$$

Figure 3.7: Semantic Functions for CFG Transformation

Chapter 4

Implementation

The tools that popularly used for scanner and parser generator should be *lex* and *yacc* in UNIX. There are *flex* and *bison* in GNU, or *ocamllex* and *ocamlyacc* in Objective-camel programming language. All of them are suitable to manipulate regular expressions and context-free grammars to generate a scanner and parser for a translator or compiler. In this thesis, we use ocamllex and ocamlyacc in Objective-camel programming language [LER01] to implement the validator generator. We concerned the benefits of Regular Expression Types, Subtyping, and Static Type System as stated in Section 6.1, Objective-Caml is one of the powerful functional programming languages, we choose it to implement this framework so that we can easily got the whole application.

4.1 Validator Generator

Figure 4.1 shows the implementation modules for our validator generator that adapted from Section 3.1 and Figure 3.2.

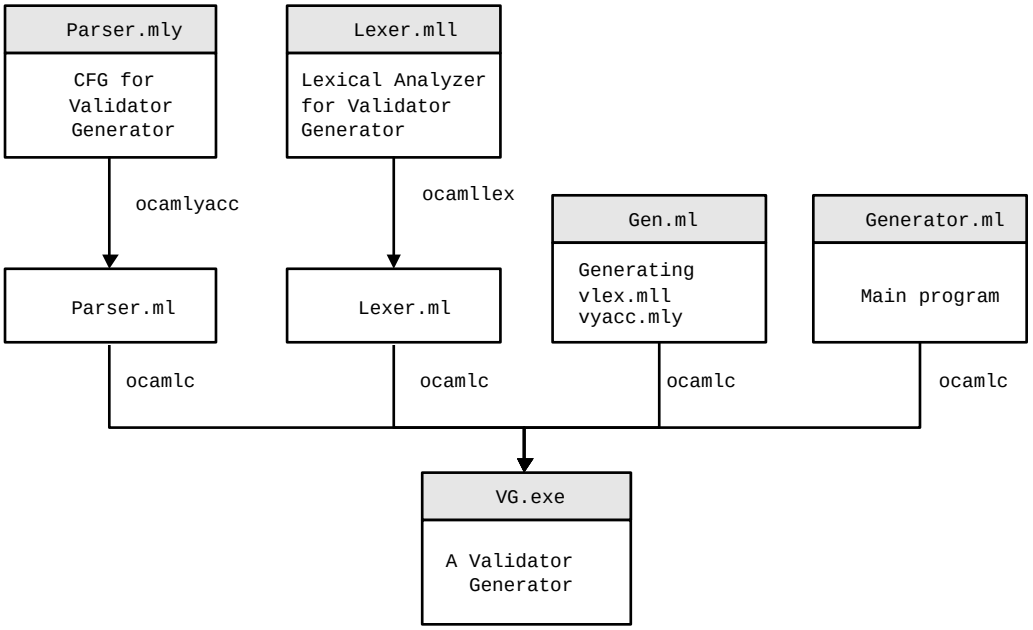

Figure 4.1: The Implementation Modules for Validator Generator.

The generator needs the follow four modules, a lexical scanner for all kinds of DTDs (lexer.mll), a parser for all kinds of DTDs (parser.mly), functions module that makes up the parse tree(gen.ml), and a main program (generator.ml) that put all things together to build a Generator.

lexer.mll

A Lexical Scanner for DTD should be able to recognize as tokens from DTD that consist of general lexeme and DTD specific lexeme, It also should be able to remove white space and comments from documents.

Figure 4.2 shows partial definitions of the pattern of lexical token that will appear in DTDs. For example, "<!ELEMENT" is a token named TAG_H, "♯PCDATA" is also a token named PCDATA.

There are two parts every single row. The left hand side which enclosed with double-quote " and " is the pattern of input, this pattern may be defined

```
1.    "<!DOCTYPE"          { DOCTYPE }
2.    "["                  { DOCTYPESTART }
3.    "]>"                 { DOCTYPEEND }
4.    "<!ELEMENT "         { TAG_H }
5.    IDENT                { ID (Lexing.lexeme lexbuf) }
6.    "#PCDATA"            { PCDATA }
7.    ">"                  { RANGLE }
8.    "*"                  { STAR }
9.    "|"                  { OR }
```

Figure 4.2: Samples of Lexical Tokens Definitions In Lexer.mll

as regular expression. The right hand side which enclose with brace { and } is the token, in certain case, the token associated some value which described attribute about this token.

Note that the token IDENT listed in line 5 of Figure 4.2 is a defined regular expression as below:

IDENT=['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9" _']*

This token should associate what the IDENT regular expression pattern real hold with in *lexbuf*.

parser.mly

The module *parser.mly* makes us to have the ability to parse DTD. The productions which adapted from Context-Free Grammar for parsing DTDs are implemented as below.

```
dtd :   XMLDEC DOCTYPE ID DOCTYPESTART elementlist DOCTYPEEND
        {Gen.Tree.Root ( Gen.Tree.Label($3,Gen.Tree.Single),$5)}
    ;
```

Nonterminals are presented in lowercase, and must be reduced to a sequence of terminals or nonterminals. The production *dtd* reduces to XML declarations, pattern "⟨!DOCTYPE, the root label identifier, pattern "[", a serials element definitions as nonterminal *elementlist*, and pattern "]⟩". there is only one nonterminal which must be reduced in progress. Follows after sequent terminals or nonterminals which enclosed by brace { and } are called *action*. The action will be done right after the production had been deduced. The action of *dtd* constructed the root node of the parse tree.

The production:

```
elementlist :   element elementlist
                { ($1::[]) @ $2  }
            |   element
                { $1::[] }
            ;
```

says that the nonterminal *elementlist* is reduced to either one element or more elements.

The production:

```
element :   TAG_H   ID  LPAREN  seqlist  RPAREN  repeat RANGLE
        {Gen.Tree.Element (Gen.Tree.Label ($2,$6) , $4 )}
    |   TAG_H   ID  LPAREN  PCDATA  RPAREN  RANGLE
        {Gen.Tree.Pcdata (Gen.Tree.Label ($2,Gen.Tree.Single))}
    |   TAG_H   ID  LPAREN  PCDATA OR orseqlist  RPAREN  repeat RANGLE
        {Gen.Tree.Mixtype (Gen.Tree.Label ($2,$8) , $6 )}
```

```
    ;
```

have three branches, the first one consists of element label follow by content model, the second one reduces to a definition of PCDATA. The tired branch of the production represents mixtype of content model, because it can be reduced to PCDATA or other content model. The productions have their associated actions which construct a node of the parse tree.

The next production denotes sequent list of content items:

```
seqlist  :    seq COMMA seqlist         {$1 :: $3 }
         |    seq                       {$1 :: []}
         ;
```

For example, sequence (a,b,c) can be linked into seqlist.

```
seq :   LPAREN  orseqlist RPAREN  repeat
            { incr (i);
            Gen.Tree.Seq (Gen.Tree.Label (("seq_" ^
                          string_of_int !i),$4),$2 );
            }
        |   label        {Gen.Tree.Tag ($1);}
        ;
```

The seq production might have two branch of reduction, either a sequence of contents separated by vertical bar, or the single label. Note that, if orseqlist is performed, we must create an increment label to construct a note in the tree.

```
orseqlist:  orseq OR orseqlist        {$1 :: $3 }
```

```
        |  orseq          {$1 :: []}
        ;
```

For example, sequence (a—b—c) can be linked into orseqlist.

```
orseq   :   LPAREN  seqlist  RPAREN  repeat
        { incr (i);
        Gen.Tree.Orseq (Gen.Tree.Label (("orseq_" ^
                        string_of_int !i),$4),$2 );
        }
    |   label
        {Gen.Tree.Ortag ($1);
        }
    ;
```

The orseq production might have two branch of reduction, which are just the same as seq reductions described above, but only be reduced by orseqlist productions. Note that, if orseq production is performed, we must create an increment label to construct a note in the tree.

```
label   :   ID repeat        {Gen.Tree.Label ($1,$2)}
        ;
```

If a single content is reduced, it must be a label of element. The label represent another element in the DTD. The label might associated a regular expressive operator which expressed as Single, Star, Option and Plus, verbatim listed as below:

```
repeat  :               {Gen.Tree.Single}
        |   STAR        {Gen.Tree.Star }
        |   OPT         {Gen.Tree.Option}
        |   PLUS        {Gen.Tree.Plus}
        ;
```

The CFG rules of Validator Generator should generate associate action to construct the tree node. These generating functions are all construct in module *Gen*.

The format of ocamllex and ocamlyacc are what we have to define for the scanner and parser of the Validator Generator. Also it is the format that the Validator Generator have to produce automatically for the scanner and parser of the validator. We have got *VG* as our Validator Generator and *Isvalid* as a Validator output via *VG*.

Without loss of generality, we simplified the syntax grammar of XML and DTD, we did not implement the properties of Entity and Attribute and others in DTD.

4.2 Modules

All functions which the Validator Generator needed are provided in file *Gen.ml*. Figure 4.3 shows the detail.

- The module *Gen.Tree* have functions for constructing parse-tree of DTDs.

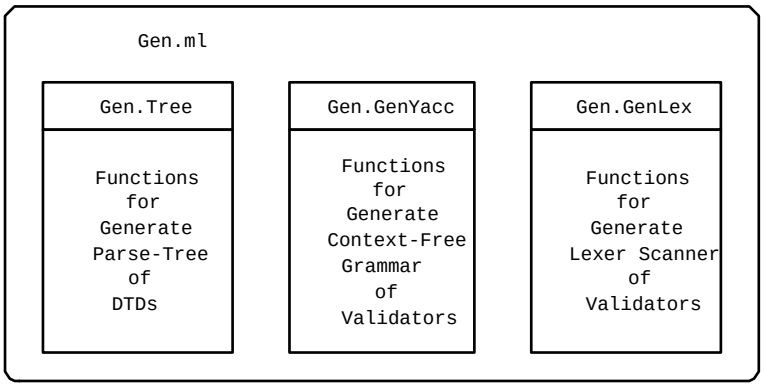

Figure 4.3: Modules for Validator Generator in file Gen.ml.

- The module *Gen.GenYacc* have functions for generating Context-Free Grammar for parsing XML documents.
- The module *Gen.GenLex* have functions for generating lexer scanners for the Validator.

4.3 Generated Validators

To implement a validator for a DTD, we need the follow three modules, a lexical scanner, a parser, and a main program.

According to the DTD, the generator should automatically create context-free grammar rules for *yacc*. Note that the scanner ignored all nonstructural data, so that the parser of validator may concern at tree structure of DTD. In practice, the errors reported by ocamlyacc imply the incompleteness of this DTD.

A Lexical Scanner for XML Document should be able to recognize as token

from XML douments that consist of general lexeme and DTD specific lexeme, It also should be able to remove white space and comments from documents. For examples, the definition of regular expressions:

```
IDENT =['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9''\_']*
```

defined *identifiers* to have the regular expression of one letter follows by zero or more letter, digit, or underline. The square bracket [ ] expressed just one character in there. Star sign * expressed zero or more the characters in the bracket.

The definition "<name>" is a token named NAME_TAG_H, "</name>" is also a token but named NAME_TAG_T. they are all *ELEMENT* specified in the DTD. One may take "<", ">", "name" as tokens, it depends on the definitions of our Context-Free Grammar.

The example DTD shown in Figure 1.3 are transformed as listed in Figure 4.5 and Figure 4.4.

```
{
    open Vyacc
    exception Eof
}
let   COM_H="<!- -"
let   COM_T="- ->"
let   COM_M=_*
let   RANGLE='>'
let   SLASH='/'
let   COMM=COM_H COM_M COM_T
let   IDENT=['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9"_']*
let   CONTENT=['a'-'z' 'A'-'Z' '0'-'9"_"!'-')']*
let   EMPTYTAG=SLASH RANGLE
let   ATTR=[^ '/' '>' ]*
rule token   =   parse
                 [' ' '\t' '\r' '\n']              { token lexbuf }
             |   COMM                              { token lexbuf }
             |   "<?xml" COM_M "?>"                {XMLDOC}
             |   _                                 {token lexbuf}
             |   eof                               {raise Eof}
             |   "<addrbook" ATTR RANGLE           {ADDRBOOK_TAG_H}
             |   "</addrbook>"                     {ADDRBOOK_TAG_T}
             |   "<person" ATTR RANGLE             {PERSON_TAG_H}
             |   "</person>"                       {PERSON_TAG_T}
             |   "<name" ATTR RANGLE               {NAME_TAG_H}
             |   "</name>"                         {NAME_TAG_T}
             |   "<name" ATTR EMPTYTAG             {NAME_TAG_E}
             |   "<tel" ATTR RANGLE                {TEL_TAG_H}
             |   "</tel>"                          {TEL_TAG_T}
             |   "<tel" ATTR EMPTYTAG              {TEL_TAG_E}
```

Figure 4.4: The regular expressions transformation result of the example DTD shown in fig.1.3

```
%token ADDRBOOK_TAG_H ADDRBOOK_TAG_T ADDRBOOK_TAG_E
%token PERSON_TAG_H PERSON_TAG_T PERSON_TAG_E
%token NAME_TAG_H NAME_TAG_T NAME_TAG_E
%token TEL_TAG_H TEL_TAG_T TEL_TAG_E
%token XMLDOC BADTAG
%type <string> main
%start main

%%
main        :  XMLDOC addrbook {"Valid."};

addrbook    :  ADDRBOOK_TAG_H person_star ADDRBOOK_TAG_T {};

person_star :  person person_star {} | {()};

person      :  PERSON_TAG_H name tel_option PERSON_TAG_T {};

tel_option  :  tel {} | {()};

name        :  NAME_TAG_H NAME_TAG_T {}
            |  NAME_TAG_E {};

tel         :  TEL_TAG_H TEL_TAG_T {}
            |  TEL_TAG_E {};
```

Figure 4.5: The Context-Free Grammar transformation result of the example DTD shown in fig.1.3

Chapter 5

Experiments

Our implementation is aimed at an application environment that processing XML documents with static DTD. We measured the run time of validation and the size of validator in deference kind of DTD.

To prove the efficiency and effectiveness of our implementation, we searched XML Validator in the CoverPage [CP01] and search engines such as Google. Both of them are recommended by the official cite of w3 consortium. We would like to search some candidate validator which are conformed with:

1. local executable so that we could eliminate the connection and communication time.

2. command line executable or scripting so that we could run with a batch file.

3. running silently, otherwise the output of running information will implicate performance measurement.

4. multi-platformed executable. Though many solutions are freeware that

# of element in DTD	Average validating time		
	VG.isvalid	xmlvalid	rxp -sV
31	0.0200	0.0153	0.0172
281	0.0246	0.0250	0.0264
531	0.0252	0.0335	0.0418
781	0.0279	0.0443	0.0653
1031	0.0306	0.0536	0.0798
1281	0.0318	0.0627	0.1041
1531	0.0348	0.0729	0.1353
1781	0.0394	0.0867	0.1766

Table 5.1: The result of running over deferent number of elements in DTDs.

suited run at Unix-like platform, we hope that it could run at DOS command line.

With these requirements, *xmlvalid* and *RXP* are selected.

5.1 xmlvalid and RXP

xmlvalid is a command-line utility for validating XML files. It is produced and maintained by ElCel Technology [XVD03]. The XMLValid is built using their C++ XML Toolkit, which is designed to be fast, flexible and whole conforming to the [XML00]. Use of ISO-standard C++ makes the toolkit highly portable, enabling to provide binary versions of the XML Validator for both Windows and Linux.

RXP is a validating XML parser written in C by Richard Tobin. It is used by the LT XML toolkit, and the Festival speech synthesis system [TO03].

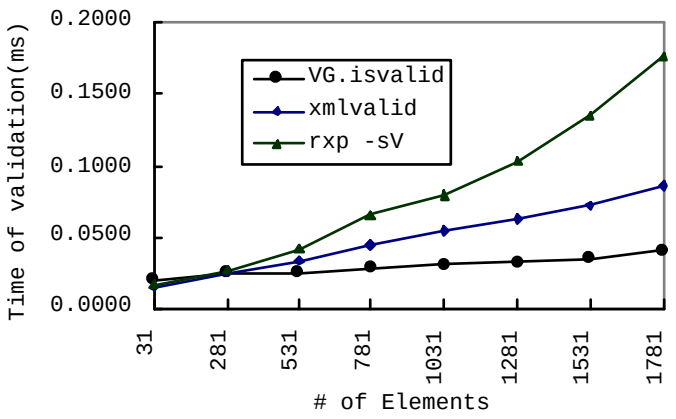

Figure 5.1: The Result of Running Time.

5.2 Benchmark

To experiment the performance of implementation of VG, We run VG and Isvalid with 1.06GHz PIII CPU, 256MB RAM in hardware, and Microsoft Windows XP version 5.1, Objective-Camel version 3.04 in software. Table 5.1 shows the average validating time to the different umber of elements in DTD files which running by VG.isvalid, xmlvalid, and rxp -sV, respectively.

The result shows that the more number of elements in DTD the more efficient validating by VG.isvalid. It will faster then competitors when the number of elements is greater then 281 or more. VG-isvalid will faster two times than the competitors while the number of element in DTD is more than 1281.

Space Required for running validation Modules (bytes)				
♯ of	VG.isvalid			
element	exe	dtd	xml	total
31	38,448	-	79	38,527
281	148,804	-	104	148,908
531	257,484	-	104	257,588
781	365,048	-	125	365,173
1,031	474,134	-	127	474,261
1,281	582,970	-	127	583,097
1,531	691,698	-	106	691,804
1,781	799,186	-	106	799,292
♯ of	xmlvalid			
element	exe	dtd	xml	total
31	733,184	945	79	734,208
281	733,184	9,980	104	743,268
531	733,184	17,395	104	750,683
781	733,184	43,130	125	776,439
1,031	733,184	34,605	127	767,916
1,281	733,184	43,455	127	776,766
1,531	733,184	52,284	106	785,574
1,781	733,184	61,157	106	794,447
♯ of	rxp -sV			
element	exe	dtd	xml	total
31	151,552	945	79	152,576
281	151,552	9,980	104	161,636
531	151,552	17,395	104	169,051
781	151,552	43,130	125	194,807
1,031	151,552	34,605	127	186,284
1,281	151,552	43,455	127	195,134
1,531	151,552	52,284	106	203,942
1,781	151,552	61,157	106	212,815

Table 5.2: Space required for running validation Modules.

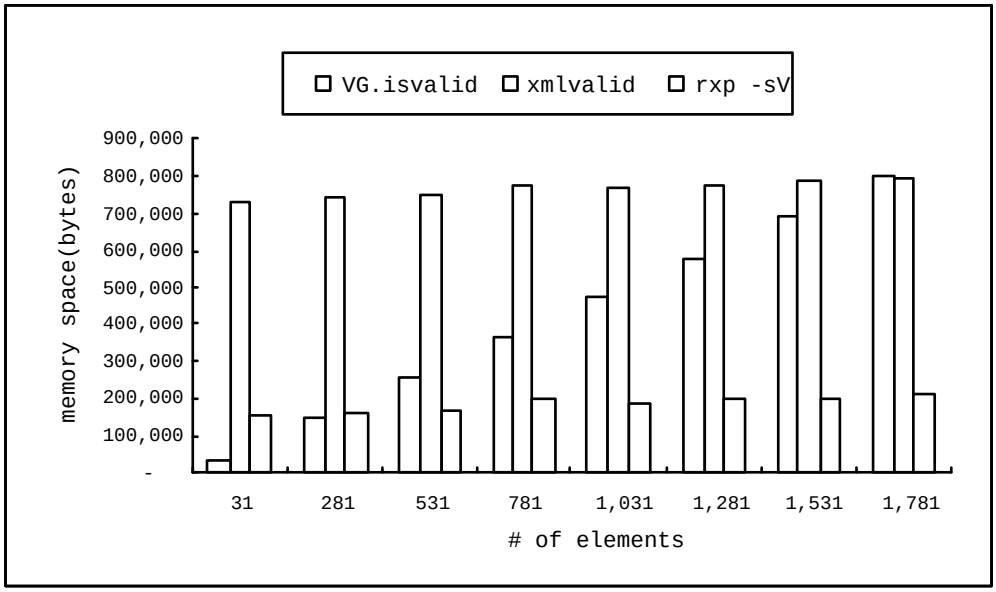

Figure 5.2: Space Required for running validation Modules.

Chapter 6

Related Work

6.1 Regular Expression Types

Regular expression types are a natural generalization of DTDs, describing the structure in XML documents using regular expression operators (i.e., *, ?, |, etc.) [HOS01]. The following are examples of regular expression type definitions in an ML-like programming language style according to the DTD shown in Figure 1.3.

```
type addrbook = Addrbook[person*]

type person = Person[name,tel?]

type name = Name[String]

type tel = Tel[String]
```

Regular expression types support a simple but powerful notion of subtyping. An XML document has a tree structure with a root node. One can inference the types from the root down to the leaves, or subtyping from the

leaves up to the root. The following are examples of subtyping.

- () <: Tel?
- Tel <: Tel?
- () | Tel <: Tel?

  ⋮

- person[Name] <: person[Name,Tel?]
- person[Name,Tel] <: person[Name,Tel?]
- person[Name,Tel], person[Name] <: (person[Name,Tel?])*

Subtyping mechanism can be used as the core of an XML validating algorithm. Figure 6.1 shows the structure of a subtyping mechanism. Both XML documents and its corresponding DTD have regular expression types. They must be represented in an internal form for the representations of the types. With the subtyping mechanism, if the internal type for the XML document is a subtype of the internal type for the DTD, the XML document is valid.

6.2 Native Language Support for XML

New languages are in designing with built-in XML support to help build XML-related applications. *XDuce* (pronounced "transduce") [HOS01] is a statically typed, tree transformation language. Similar to functional languages but specialized to the domain of XML processing. Its novel features are *regular expression types* and a corresponding mechanism for *regular expression pattern*

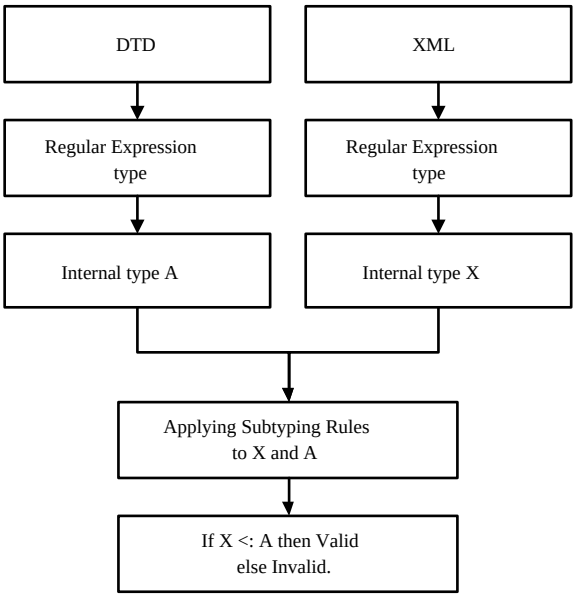

Figure 6.1: The structure of a subtyping mechanism for XML Validation.

matching.

In addition, there are several developers working on developing new languages for XML application with validity check; e.g., CDuce, which pronounced as "seduce", is a typed functional programming language [CDU02], which is an extension of XDuce with higher-order and precise typing of basic values. For instance, string types are described by regular expressions. CDuce has the ability to inference the types and the subtypes of the elements in the document.

6.3 Generic Validation for XML with Parametric Modules

A generic validation of XML documents with parametric modules is developed by Chuang [Chu01]. They demonstrate a natural mapping from the element types of XML to the module expressions of ML-like programming languages. ML-like languages are suitable for type-safe prototyping of DTD-aware XML applications. One can view the validation function as giving types to XML elements. The validation functions mapped from element types of DTD to module expressions in a generic way, such that the function checks an XML document for conformance to the content model specified by its DTD [Chu01]. The mapping is inductive, and the definitions of common XML operations can be derived as the module expressions are constructed. They show how to derive, in a generic way, the validation function, which checks an XML document for conformance to the content model specified by its DTD.

From the viewpoint of generic programming, software development often consists of designing a datatype, to which functionality is added. A program that works on many datatypes is called a generic program [JM99]. In general, an XML validator works on all the XML documents, and recognize whether the document is valid or invalid against to its DTD.

Generic programming should be the appropriate method [Chu01]. Every element can be defined as module expression using a predefined set of constant modules and parametric modules, such as hd, tl, empty, star, plus, alt, seq, etc. Generic validation procedures give types to valid elements, such that we

can construct XML processors in a typeful way. There is no need to recode generic operations, and no need to design new languages either. For example, The structure of A Wireless Markup Language document [Chu00].

```
<?xml version="1.0"?>
<wml>
 <card id="card1" title="School">
 <do type="accept" label="Answer">
   <go href="#card2"/>
 </do>
</wml>
```

can be translated into the following well-formed element u,

```
let u = U.wml [U.card [U.do [U.go []]]]
```

Element u is then validated,

```
let v = Dtd.validate u
let w = Dtd.forget   v
let x = Dtd.validate w
```

and

```
let ok0 = u = w
let ok1 = v = x
```

The result messages depend on whether u=w and v=x or not. If ok0 and ok1 then its valid else invalid. The process is shown in figure 6.2 which is adapted from [Chu01].

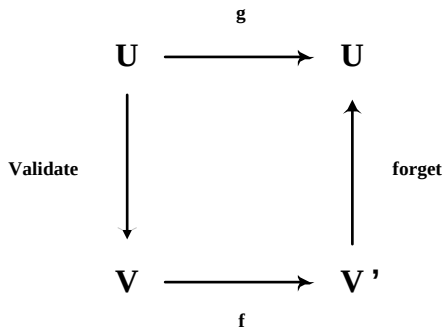

Figure 6.2: Typeful XML Programming Validation.

U is the ML type for well-formed elements and V and V' are the ML types for specific XML element types. Functions in $g{:}U \rightarrow U$ are untyped as they may produce invalid elements. On the other hand, functions in $f{:}V \rightarrow V'$ are typed as they always output valid elements. So that we compose functions as:

$$g = forget \circ f \circ validate$$

to produce valid output.

Chapter 7

Conclusion and Future Directions

7.1 Conclusion

To solve XML validation problem, A Context-Free Grammar Transformation Framework has proved efficiency and effectiveness. Because of documents' type structure had been builded before documents are validated.

Actually, a DTD defined by stably activated association will rarely change, DTD must to keep stability for the develop of application (e.g. IFX-DTD [IFX03]).

Our framework applied the front end of compiler techniques to transform the DTD types into automatic programming generator. We control the rules of transformation in order to generate validator automatically. We proved the efficiency and effectiveness of our implementation.

7.2 Future Directions

In the future, we would like to state follows to whom interested in:

- Optimizing the efficiency of our implementation. There are many optimization techniques of compilation which can always improve efficiency of compilers in deference way.

- Applying our implementation to mobile or other device. Mobile devices have limited memory and restricted computing ability, the space require of applications in mobile devices is better lower.

Bibliography

[APP99] Andrew W. Appel "Modern Compiler Implementation in ML", CAMBRIDGE UNIVERSITY PRESS. 1999.

[ASU86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman "Compilers principles, Techniques, and Tools", ADDISON-WESLEY PUBLISHING COMPANY. 1986.

[CDU02] V. Benzaken, G. Castagna, and A. Frisch. "CDuce: a white paper.(working document)", http://www.cduce.org/ . 2002.

[Chu00] Tyng-Ruey Chuang. "Project x.ml Modeling XML in ML", *http://www.iis.sinica.edu.tw/ trc/x_dot_ml.html*, Visited July 2001.

[Chu01] Tyng-Ruey Chuang. "Generic Validation of Structural Content with Parametric Modules." *2001 International Conference on Functional Programming*, pp.98-101, September, 2001.

[CML03] "The Site for Chemical Markup Language", *http://www.xml-cml.org/* , Visited march 2003

[CMP00] Emmanuel Chailloux, Pascal Manoury and Bruno Pagano, "Devel-

oping Applications With Objective Caml", Editions O'Reilly Frence, 2000. Avilable at *http://caml.inria.fr/oreilly-book/*

[CML01] INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE. "The Caml language", *http://caml.inria.fr/*, Visited July 2001.

[Coh97] Daniel I.A. Cohen, "Introduction To Computer Theory" Second Edition, JOHN WILLY & SONS, INC. 1997.

[CP01] The XML Cover Pages. "Check or Validate XML", *http://www.oasis-open.org/cover/check-xml.html*, Visited July 2001.

[DOM01] The World Wide Web Consortium. "XML DOM", *http://www.w3.org/XML/Schema*, W3C Recommendation, May 2001.

[DRA98] Fred L. Drake, Jr. , "The XML Bookmark Exchange Language", *http://pyxml.sourceforge.net/topics/xbel/docs/html/xbel.html*, 1998

[DTD98] The World Wide Web Consortium. "Guide to the W3C XML Specification ("XMLspec") DTD, Version 2.1", *http://www.w3.org/XML/1998/06/xmlspec-report*, June 1998.

[IFX03] IFX forum.*http://www.ifxforum.org/*, visited march, 2003.

[IFX02] IFX 1.0.1 Public Review Draft. "Interactive Financial Exchange XML Reference". February 14th, 2000

[JM99] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. "Generic programming, An introduction." Advanced Functional Programming, volume 1608 of LNCS, pages 28 - 115. Springer-Verlag, 1999.

[JP02] Johan Jeuring and Paul Hagg. "Generic Programming for XML Tools". Technical report ICS Utrecht University, UU-CS-2002-023.

[HL03] Health Level Seven.*http://www.hl7.org/*, visited march, 2003.

[HP00] Haruo Hosoya and Benjamin C. Pierce. "XDuce: A Typed XML Processing Language", *In Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, volume 1997 of Lecture Notes in Computer Science, pages 226-244, May 2000.

[HP01] Haruo Hosoya and Benjamin C. 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.

[HVP00] Haruo Hosoya, Jerome Vouillon, Benjamin C. Pierce. "Regular Expression Types for XML", *In Proceeding of the International Conference on Functional Programming (ICFP)*, page 11-22, September 2000.

[HOS01] Haruo Hosoya. "XDuce Homepage." *http://xduce.sourceforge.net/*, Visited July 2001.

[LER01] Xavier Leroy. "The Objective Caml system, Documentation and user's manual", December 10, 2001.

[MER01] Dave Mercer, "XML: A Beginner's Guide", Osborne/McGraw-Hill Press, 2001

[MAT03] MathML 2.0, *http://www.w3.org/Math/* , Visited march 2003

[MSD03] Microsoft Co. "MSDN library-XML Web Services", *http://msdn.microsoft.com/library/* , Visited march 2003

[ROS03] RosettaNet Home, *http://www.rosettanet.org/* , Visited march 2003

[SET00] Ravi Sethi "Programming Languages CONCEPTS & CONSTRUCTS", ADDISON-WESLEY PUBLISHING COMPANY, 2nd Edition, 2000.

[SCH01] The World Wide Web Consortium. "XML Schema", *http://www.w3.org/XML/Schema*, W3C Recommendation, May 2001.

[STO03] Gerd Stolpmann, "The polymorphic XML parser", *http://www.ocaml-programming.de/programming/pxp.html*

[XVD03] XML validator. *http://www.elcel.com/products/xmlvalid.html*, visited march, 2003.

[TO03] Richard Tobin, RXP - an XML parser. *http://www.cogsci.ed.ac.uk/ richard/rxp.html*, visited march, 2003.

[WR03] Malcolm Wallace and Colin Runciman, "HaXml: Haskell and XML" , *http://www.cs.york.ac.uk/fp/HaXml/*.

[WRB03] Malcolm Wallace, Colin Runciman and John Boyer "Haskell and XML: Generic Combinators or Type-Based Translation?" *http://www.cs.york.ac.uk/fp/HaXml/icfp99.html*

[WAL98] Norman Walsh, "What is XML?", *http://www.xml.com/pub/a/98/10/guide1.html*, 1998

[XML00] The World Wide Web Consortium. "Extensible Markup Language (XML) 1.0 (Second Edition)", *http://www.w3.org/TR/2000/REC-xml-20001006*, October 2000.

Automatic Generation of XML Validators by Applying Compilation Techniques

Jih-Chin Yeh

Advisor: Chung Yung

Abstract

The validating process of XML documents is significant for improving the safety and correctness of data exchange and processing over Internet.

In this thesis, we develop an approach for automatically generating XML validators. We transform the document type definition, DTD, into a context-free grammar, CFG, so that we can freely generate a static typed program which can validates XML documents according to the DTD.

We implement the CFG transformation with functional programming language-Objective Caml that associated compiler tools named ocamllex and ocamlyacc. Compared with the popular approaches, our experiments show the high efficiency and effectiveness in solving the XML validation problem.