PARSING AND AMBIGUITY TESTING OF XML DOCUMENTS

Dr. Eng. Ioan Cauțil

Redline Communications, Craiova

Abstract: The paper analyses the grammar of XML elements an presents algorithms for testing ambiguity of elements and to check the structure of XML documents versus the definition of the elements.

Keywords: XML

1. INTRODUCTION

The paper presents the implementation of an XML parser in [4]. XML is widely used to describe the data structures. An XML document consists of elements defined by the user with attributes and content (other elements as children). The first task of an XML processor is to parse the DTD statements (in many cases in a DTD file). The second task is to check the structure of the XML document versus the definition of the element. The content of an XML element is defined by element type declaration with the form

```
elementdecl ::= ‘<!ELEMENT’ S Name S
                contentspec S? ‘>’
contentspec ::= ‘EMPTY’ | ‘ANY’ | Mixed | Children
```

In the above productions S is a sequence of one or more white spaces and Name is the element name. In the sequel ? + and * are operators used to describe regular expressions. If A is a regular expression A? matches A or nothing, A+ matches one or more occurrences of A and A* matches zero or more occurrences of A. The parentheses, (and), can be used to group sub expressions. The | (pipe) means or (if A and B are regular expressions, A | B means A or B). The following grammar of element content is taken from the XML specification (W3C, 2004).

```
[51] Mixed ::= ‘(‘ S? ‘#PCDATA’ (S? ‘|’ S? Name)* ‘)*’
             | ‘(‘ S? ‘#PCDATA’ S? ‘)’
```

The keyword #PCDATA denotes character data. The mixed content means the element may contain character data interspersed with child elements. The element content is defined as

```
[47] children ::= (choice | seq) (‘?’ | ‘*’ | ‘+’)?
[48] cp ::= (Name | choice | seq ) (‘?’ | ‘*’ | ‘+’)?
[49] choice ::= ‘(‘ S? cp (S? ‘|’ S? cp)+ S?’)’
[50]  seq ::= ‘(‘ S? cp (S? ‘,’ S? cp)* S? ‘)’
```

In this case the element must contain only child elements. An EMPTY content means the element may not have any content and an element defined with ANY content may contain any other elements. The number of productions is that of XML specification (W3C, 2004).

2. PARSING THE GRAMMAR

We will analyze only the grammar for children. The production [49] is transformed as

```
[49a] choice ::= ‘(‘ S? cp S? ‘|’ S? cp (S? ‘|’ S? cp) * S? ‘)’
```

The grammar for children is rewritten as

Table 1 XML Grammar

		FIRST	FOLLOW	SELECT
1	<S> → (<P><Z>)	(		(
2	<Z> → <P> <Clist>	|		|
3	<Z> → <Elist>	,	)	,)
4	<Clist> → |<P><Clist>	|		|
5	<Clist> → epsilon		)	)
6	<Elist> → , <P> <Elist>	,		,
7	<Elist> → epsilon		)	)
8	<P> → a	a		a
9	<P> → <S>	(		(

<S> is the starting symbol of the grammar and epsilon is the null sequence. In the grammar we denoted the terminal symbol) and the strings of terminal symbols)?,)* and)+ as). We will show the above grammar is an LL(1) grammar. For presentation of such grammar see (Aho et al., 1997; Lewis et al., 1976). We need the following definitions from (Lewis et al., 1976).

The FIRST sets. Given a context-free grammar and an intermediate string alpha of symbols from the grammar, define FIRST(alpha) to be the set of terminal symbols that occur at the beginning of intermediate string derived from alpha.

The FOLLOW sets. Given a context-free grammar with starting symbol <X> define FOLLOW(<X>) to be the input symbols that can immediately follow an <X> in an intermediate string derived <S>@ where @ is the end marker of the input sequence.

The SELECT sets. A string alpha of symbols from grammar is nullable if alpha can be transformed in epsilon by zero or more substitutions. A production of grammar is nullable if its right hand size is nullable. Given a production

<A> → alpha

where alpha is a string of terminals and nonterminals, define

SELECT(<A>→ alpha) = FIRST(alpha)

if alpha is not nullable, and

SELECT(<A> → alpha) = FIRST(alpha) U FOLLOW(<A>)

if alpha is nullable.

Finally, a context-free grammar is called an LL(1) grammar if productions with the same left hand size have disjoint SELECT sets.

In the above grammar we need to compute the FOLLOW sets for productions 3, 5, and7. Examining the SELECT sets for productions with the same left hand size, they are disjoint so the grammar is LL(1).

3. THE ELEMENT TREE

The element tree is a modification of the derivation tree, see (Aho et al., 1997; Lewis et al., 1976). The element tree has as leaves terminal symbols (Name in production [48] or a in the grammar) and operators |, *+? in nodes (| for or and , for and operators). For example, the sequence

(a, b, d)

has the tree

```
(,)
--- (a)
--- (b)
--- (d)
```

The element

(a | b | d)*

has the tree

```
(*)
--- (|)
       --- (a)
       --- (b)
       --- (d)
```

The composed element

((a, b), (x | y))

has the tree

```
(,)
--- (,)
       --- (a)
       --- (b)
--- (|)
       --- (x)
       --- (y)
```

In the sequel we will use the element tree to calculate the FIRST and FOLLOW sets for element.

4. AMBIGUITY TESTING

When we define an XML element having children, is important for an XML processor to be able to test if the element content conforms its definition. The standard includes an optional requirement that the content model of an element be deterministic. The content model of the element

((b, c) | (b, d))

is called nondeterministic because giving an initial b terminal symbol, the processor cannot know which branch in the element tree to follow without looking ahead which element follows the terminal symbol b. Considering the grammar of the above content model

1. <X> → bc
2. <X> → bd

in not a LL(1) grammar. To test if an element is ambiguously defined we will test if its grammar is LL(1). We use the element the tree of element to calculate its FIRST and FOLLOW sets to test if its content model is deterministic. To get an algorithm we will derive the grammar of element from the regular expression. In regular expression we have the following rules

1) The unary operators * + ? have the highest precedence and are left associative
2) The concatenation operator , has the second highest precedence and is left associative
3) The union operator | has the lowest precedence and is left associative

4) The parenthesis (and) are used to change the precedence of operators

The rules used to derive the grammar are following

1) Each expression in parenthesis is denoted by a nonterminal symbol
2) Each terminal and nonterminal symbol followed by an unary operator * + ? is denoted by a nonterminal symbol

For the nonterminal symbol

Z = R*   (1)

where R is a nonterminal symbol, we define the following productions of the grammar

```
<Z> → <R><Z>       (2a)
<X> → epsilon      (2b)
```

For the nonterminal symbol

R = (A1 | A2 | … |An)   (3)

we define the following productions

```
<R> → <A1>         (4a)
<R> → <A2>         (4b)
……………
<R> → <An>         (4n)
```

For the nonterminal symbol

R = (A1, A2, …, An)   (5)

we define the following productions

```
<R> → <A1><X2>     (6a)
<X2> → <A2><X3>    (6b)
……………
<Xn> → <An>        (6n)
```

For example, the element

(a, (b | c)*, d)

is described by the following grammar

```
<S> → a<X1>
<X1> → <Z><X2>
<X2> → d
<Z> → <R><Z>
<Z> → epsilon
<R> → b
<R> → c
```

The element tree from the previous section corresponds to this grammar. We will use the element tree to calculate the FIRST and FOLLOW sets of the grammar.

4.1 The FIRST sets

1)For the nonterminal symbol (3) the FIRST set is the union of FIRST sets of nonterminals <A1>, <A2>, …, <An>
2) For the nonterminal symbol (5) the FIRST set is the FIRST set of <A1> if <A1> cannot be transformed in epsilon, else is the union of the FIRST sets of <A1> and <A2> if <A2> cannot be transformed in epsilon, and so on.
3) For a node labeled as * + ? the FIRST set is the FIRST set of its child.

4.2 The FOLLOW sets

The FOLLOW sets are computed only for nodes labeled *+?. Consider the FIRST sets for all nodes already computed. Let n be such node and p its parent. Let followset(n) be the FOLLOW set of node n and firstset(n) its FIRST set. Let

n.getParent()

be a function that gives the parent of node n and

n.getNextSibling()

be a function that gives the next sibling of node n. The algorithm to compute the FOLLOW set of node n is the following

```
p = n.getParent()
followset(n) = empty set
while(n = = '*' || n = = '?' || n = = '+')
{
    s = n.getNextSibling()
    if(s != null)
    {
        followset(n) =
        followset(n) U firstset(s)
        n = s
    }
    else
    {
        n = p
        p = n.getParent()
    }
}
```

5. TESTING THE ELEMENT TREE

The last task of an XML parser is to compare the content each element of XML document (element tree) with the element definition. This is done by comparing the element tree from XML document with the derivation tree. Consider the function

int procChild(List elemChildren, Node node)

where elemChildren is the list of children of a given element from the XML document and node is the root node of the derivation tree. In the sequel the name of the node is * or + or ? for unary operators and | and respectively , for binary operators. For leaves the node name is the element name. The function returns 1 if there is a match for the first element in elemChildren list and a leaf in the derivation tree, else 0. In the case of a , node (operator ,) the algorithm is

```
if(node = = ',')
{
    // in this case we have to match all
    elements
    // of the derivation tree with the
    elements of the list
    viter = node.children()
    while(viter.hasMoreElements())
    {
        a = viter.nextElement()
    x = procChildren(elemChildren, a)
        if(x = = 0)
            return 0
    }
```

```
        return 1
    }
```

The case of a | (operator |) is treated in the same manner. In the case of an * node (operator *) we have the following algorithm

```
    if(node = = '*')
    {
        // try to match as many as possible
        elements of the list
        a = node.getFirstChild()
        x = procChildren(elemChildren, a)
        if(x = = 0)
            return 1
        while(x > 0)
        x = procChildren(elemChildren, a)
    }
```

The case of + and ? nodes is simple and is not presented. In the case of a leaf we try to match the first element from the elemChildren list.

```
    if(elemChildren.size() = = 0)
        return 0
    if(node  = = elemChildren.getFirst())
    {
        elemChildren.removeFirst()
        return 1
    }
    else
        return 0
```

For the source code and an application implementing the above algorithms see (Cautil, 2004).

6. CONCLUSIONS

The paper proposed an algorithm for ambiguity testing of XML elements. The FIRST and FOLLOW sets of the grammar are computed using the element tree. An algorithm for testing the content of elements using the derivation tree is proposed.

REFERENCES

Aho A, Sethi R., Ullman J. D., (1997). Principles of Compilers Design, Addison Wesley

Cautil, I.,(2004). www.cautil.org

Lewis P. M., Rozenkrantz D., Stearns R. E., (1976). Compiler Design Theory, Addison Wesley

W3C, (2004). Extensible Markup Language (XML) 1.0 (Second Edition), http://www.w3c.org/TR/RECXML