Data Just Wants to Be Format-Neutral

Steven Pemberton, CWI, Amsterdam

Abstract

Invisible XML is a technique for treating any parsable format as if it were XML, and thus allowing any parsable object to be injected into an XML pipeline. The parsing can also be undone, thus allowing roundtripping.

This paper discusses issues with automatic serialisation, and the relationship between Invisible XML grammars and data schemas.

Presented at XML Prague, 2016, Prague, Czech Republic, http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf, pp109-120.

Contents

Abstraction and Representation

All numbers are abstractions. There is no *thing* that is the number 3, you can't point to it, only to a representation of it. The best that we can say is that the number three is what it is that three apples and three chairs have in common.

Given the right context, we understand that "CXXVII", "127", "7F", "1111111" and "one hundred and twenty-seven" are all representations of the same number: the underlying concept is identical. We choose the representations we use either through familiarity, or for convenience. For instance, while it is relatively easy to add numbers expressed in roman numerals together, it is very hard to multiply them; binary representations are used in computers because the electronics needed to manipulate them are much simpler.

And so it is with data representations in general. To take an example, there is no *essential* difference between the JSON

```
{"temperature": {"scale": "C"; "value": 21}}
```

and an equivalent XML

```
<temperature scale="C" value="21"/>
```

or

```
<temperature>
  <scale>C</scale>
  <value>21</value>
</temperature>
```

since the underlying abstractions being represented are the same. We choose which representations of our data to use, JSON, CSV, XML, or whatever, depending on habit, convenience, or the context we want to use that data in.

On the other hand, having an interoperable *generic* toolchain such as that provided by XML to process data is of immense value. How do we resolve the conflicting requirements of convenience, habit, and context, and still enable a generic toolchain?

Invisible XML

Invisible XML[ixml] is a method for treating non-XML documents as if they were XML, enabling authors to write documents and data in a format they prefer while providing XML for processes that are more effective with XML content.

The essence of Invisible XML is based on the observation that, looked at in the right way, an XML document is no more than the parse tree of some external form, so that all that is needed is to parse the external form using some general-purpose parsing algorithm, and then serialise the resulting parse-tree as XML.

To take a very simple example, imagine a grammar for a very simple expression language that allows such expressions as:

```
a×(3+b)
```

The grammar could look like this:

```
expression: ^expr.
expr: term; ^sum; ^diff.
sum: expr, "+", term.
diff: expr, "-", term.
term: factor; ^prod; ^div.
prod: term, "×", factor.
div: term, "÷", factor.
factor: ^letter; ^digit; "(", expr, ")".
letter: ^["a"-"z"].
digit: ^["0"-"9"].
```

The format used is a 1-level van Wingaarden grammar [vwf], a variant of BNF [bnf]. Each rule consists of a non-terminal to be defined, followed by a colon, and a *definition* followed by a full-stop. A definition consists of a number of *alternatives* separated by semicolons. Each alternative consists of a list of *non-terminals* and *terminals* separated by commas. A terminal is enclosed in quotes. An alternative, as a shorthand, may also consist of a range of characters enclosed in square brackets.

The only thing that needs to be explained here is the use of the "^" symbol, which marks non-terminals in the parse tree that are required to show up in the final XML serialisation. To illustrate, if we parse the example expression "`a×(3+b)`" with this grammar we would get the following parse tree:

```
^expr
|   term
|   |   ^prod
|   |   |   term
|   |   |   |   factor
|   |   |   |   |   ^letter
|   |   |   |   |   |   ^"a"
|   |   |  "×"
|   |   |   factor
|   |   |   |   "("
|   |   |   |   expr
|   |   |   |   |   ^sum
|   |   |   |   |   |   expr
|   |   |   |   |   |   |   term
|   |   |   |   |   |   |   |   factor
|   |   |   |   |   |   |   |   |   ^digit
|   |   |   |   |   |   |   |   |   |    ^"3"
|   |   |   |   |   |   "+"
|   |   |   |   |   |   term
|   |   |   |   |   |   |   factor
|   |   |   |   |   |   |   |   ^letter
|   |   |   |   |   |   |   |   |   ^"b"
|   |   |   |   ")"
```

Serialising this as XML, retaining all nodes, would give the following XML instance:

```
<expr>
  <term>
    <prod>
      <term>
```

```
        <factor>
          <letter>a</letter>
        </factor>
      </term>
      x
      <factor>
        (
        <expr>
          <sum>
            <expr>
              <term>
                <factor>
                  <digit>3</digit>
                </factor>
              </term>
            </expr>
            +
            <term>
               <factor>
                 <letter>b</letter>
               </factor>
            </term>
          </sum>
        </expr>
        )
      </factor>
    </prod>
  </term>
</expr>
```

However, serialising it retaining only the marked nodes gives us the following XML:

```
<expr>
    <prod>
        <letter>a</letter>
        <sum>
            <digit>3</digit>
            <letter>b</letter>
        </sum>
    </prod>
</expr>
```

Serialisation

Since in general the input form and the generated XML are isomorphic, returning the generated XML to its original format is just a process of serialisation, nothing that a suitable bit of XSLT couldn't do, or even CSS in some simple cases.

For instance, to take an example from the original paper, where a piece of CSS

```
body {color: blue; font-weight: bold}
```

is parsed into XML as:

```
<css>
   <rule>
      <selector>body</selector>
      <block>
         <property>
            <name>color</name>
            <value>blue</value>
         </property>
         <property>
            <name>font-weight</name>
            <value>bold</value>
         </property>
      </block>
   </rule>
</css>
```

Rejoicing in the possibility of formatting CSS with CSS, the following simple bit of CSS would return the XML back into regular CSS format:

```
block::before {content: "{"}
```

```
block::after {content: "}"}
name::after {content: ":"}
property::after {content: ";"}
```

The original paper also shows how to produce an alternative XML serialisation of the CSS snippet using attributes:

```
<css>
   <rule>
      <selector>body</selector>
      <block>
         <property name="color" value="blue"/>
         <property name="font-weight" value="bold"/>
      </block>
   </rule>
</css>
```

which could be round-tripped with the following piece of CSS:

```
block::before {content: "{"}
block::after {content:"}"}
property::before {content: attr(name) ":" attr(value) ";"}
```

However, considering the XML above for the expression, it is harder (a combinatorial problem) to round-trip the XML using CSS because of the loss of context caused by eliding intermediate nodes like `term`, `factor` and `expr`. For instance, if `<sum>` is a direct child of `<prod>`, then it must have been enclosed in brackets in the original expression, and therefore the serialisation must include brackets around such `<sum>`s, but you can only infer it, and it is impossible to infer if the original had *two* pairs of brackets around it.

An alternative option to such inference is to regard the grammar of a format as a specification of a presentation language for the parse-tree of that format, and write a suitable program that walks the tree hand-in-hand with the grammar.

Serialisation by Tree Walking

If you have the parse tree that was used to generate the XML serialization, then serialising it back to its original form is trivially easy: the parse tree is traversed depth first, and each time a terminal symbol is reached, it is copied to the output:

```
serialise(t)=
   for node in children(t):
      select:
         terminal(node):
            output(node)
         nonterminal(node):
            serialise(node)
```

However, in the general case you will not have the original parse-tree, and so life is harder. Because of the lack of context referred to earlier, caused by the elision of intermediate nodes in the parse tree, you essentially have to recreate the parse-tree. This can be done by 'parsing' the XML serialisation using the original grammar.

Earley Parsing

In the literature, the Earley parsing algorithm [earley] is often referred to as a "state chart" parsing algorithm [aho]. However, from a modern computing perspective, it is more useful to see it as a serialised parallel parsing processor.

Each rule, such as

```
sum: expr, "+", term.
```

represents, in Unix terms, a process. The right hand side is a series of 'instructions' for matching the input, starting at the current position. These are executed sequentially.

However, if a right hand side has several alternatives (separated by ";" in ixml grammars), such as

```
factor: ^letter; ^digit; "(", ^expr, ")".
```

then the process is 'forked' (again in Unix terminology) to produce a sub-process for each alternative, each processing from the same start position.

The processes are put in a queue, ordered on the position in the input they are parsing from. All processes for position *n* are run before processes for position *n+1* (not essential, but reduces the need for keeping the whole input around during processing).

If a process successfully matches an input symbol, it is paused and added to the queue for position *n+1*.

If a process reaches the end of its 'instructions', it *succeeds* (terminates successfully, and returns to its parent rule).

If a process meets an input symbol it wasn't expecting, it *fails* (terminates unsuccessfully, and returns to its parent rule).

For a rule with more than one alternative, if one or more succeeds, the rule itself succeeds, otherwise it fails.

More than one alternative can succeed if the grammar is ambiguous. For instance, with the simple grammar:

```
div: "i"; div, "÷", div.
```

the string

```
i÷i÷i
```

can be parsed in two ways, essentially either as

```
div(i, div(i, i))
```

or

```
div(div(i, i), i)
```

or in other words, either as

```
i÷(i÷i)
```

or as

```
(i÷i)÷i.
```

The whole process ends when all the sub-processes have terminated; if the top-level process succeeds, than you have successfully parsed the input, and otherwise not.

There is one other issue: if a rule has already been queued for a particular position, it is not added a second time, instead being linked to the already-queued version. THis implies that a process can have more than one parent, and if a process succeeds, all its parents must be restarted.

Parsing a Parse Tree

Parsing a parse tree is a similar procedure. The top level rule must be matched against the XML tree. A 'marked' terminal in the grammar must be present in the XML, as must a marked nonterminal, which is then further treated as a nonterminal in the original algorithm. A non-marked terminal is assumed to be present. Finally an unmarked nonterminal is treated the same as any nonterminal in the original algorithm.

This parsing will produce a parsetree that can then be used for serialisation as described above.

The only thing to note is that parsing the parsetree can also produce an ambiguous result. For example, suppose an expression grammar allowed the use of several sorts of brackets, where the brackets had no separate semantic meaning, so that as well as

```
a×((b+1)×(c+1))
```

you could also write

```
a×({b+1}×{c+1})
```

with the following grammar fragment:

```
factor: ^letter; ^digit; "(", expr, ")"; "{", expr, "}".
```

Since the brackets do not appear in the final serialised parsetree, there is no way to tell from it if an original bracketed expression had been

```
(b+1)
```

or

```
{b+1}
```

since they both produce the identical serialisation. This implies that while roundtripping will be semantically identical, it won't necessarily be character-by-character identical. If this is not wanted, then to overcome it, effective information about the elided characters *has* to appear in the serialisation. For instance by using rules like:

```
factor: ^letter; ^digit; ^pexpr; ^bexpr.
pexpr: "(", expr, ")".
bexpr: "{", expr, "}".
```

Representation Neutrality

A major consequence of Invisible XML is that the external representation of any format is relatively unimportant: it is the data represented that matters, and in particular the resulting parse-tree. This means from the point of view of *ixml* that any external representation of a format is equivalent, as long as it has the same parse tree.

Take for instance the syntax of an ixml grammar, a part of which looks like this:

```
ixml: (^rule)+.
rule: @name, colon, definition, stop.
definition: (^alternative)+semicolon.
alternative: (term)*comma.
term: symbol; repetition.
 ...
name: (letter)+.
colon: ":".
```

As long as the resulting serialised parsetree is the same, we could easily choose another format for the grammars. For instance:

```
<ixml> ::= (^<rule>)+
<rule> ::= @<name> <define-symbol> <definition>
<definition> ::= (^<alternative>)+<bar>
<alternative> ::= (<term>)*
<term> ::= <symbol> | <repetition>
 ...
<name> ::= "<" (<letter>)+ ">"

<define-symbol> ::= "::="
<bar> ::= "|"
```

(Note that these two grammar fragments, both describe *and* use the format described).

The only repercussion this has on Invisible XML is during the delivery, we not only have to say what the syntax is of the document that we are parsing, but also what syntax of that syntax is, if it is not the standard one. So for instance the mediatype could look like:

```
application/xml-invisible; syntax=http://example.com/syntax/css; in=http://example.com/syntax
/invisible-xml-alt
```

Normalising Grammars

So what is the resulting parse tree of a particular ixml grammar? The way to find out is to process the grammar in the following way. For each symbol in every rule:

1. if it is an implicit terminal delete it
2. if it is a refinement, replace it with the definition of that refinement enclosed with brackets, unless this refinement is already a part of it (i.e. the refinement is recursive).

and then delete all rules that are no longer used.

So for example, for the expressions grammar, we would end with:

```
expr: (^letter; ^digit; ^prod; ^div; ^sum; ^diff).
sum:  (^letter; ^digit; ^prod; ^div; ^sum; ^diff),
      (^letter; ^digit; ^prod; ^div; ^sum; ^diff).
diff: (^letter; ^digit; ^prod; ^div; ^sum; ^diff),
      (^letter; ^digit; ^prod; ^div; ^sum; ^diff).
prod: (^letter; ^digit; ^prod; ^div; ^sum; ^diff),
      (^letter; ^digit; ^prod; ^div; ^sum; ^diff).
div:  (^letter; ^digit; ^prod; ^div; ^sum; ^diff),
      (^letter; ^digit; ^prod; ^div; ^sum; ^diff).
letter: ^["a"-"z"].
digit: ^["0"-"9"].
```

which has eliminated all refinements, and all non-productive terminal symbols.

This resulting parse-tree definition is essentially a definition of the data-structure for the internal representation of the document, in other words a form of schema.

As another example, take this fragment of the ixml grammar for itself:

```
ixml: (^rule)+.
rule: @name, colon, definition, stop.
definition: (^alternative)+semicolon.
alternative: (term)*comma.
term: symbol; repetition.
symbol: terminal; ^nonterminal; ^refinement.
terminal: ^explicit-terminal; ^implicit-terminal.
repetition: ^one-or-more; ^zero-or-more.
```

The first rule:

```
ixml: (^rule)+.
```

doesn't change.

```
rule: @name, colon, definition, stop.
```

becomes:

```
rule: @name, ":", (^alternative)+semicolon , ".".
```

by substituting all the definitions for the refinements. This is then processed again to give:

```
rule: @name, ":", (^alternative)+";" , ".".
```

and finally the (unmarked) terminals are deleted:

```
rule: @name, (^alternative)+.
```

The rule

```
definition: (^alternative)+semicolon.
```

becomes by a similar process:

```
definition: (^alternative)+.
```

(but will be later deleted, as it is no longer used).

The rule

```
alternative: (term)*comma.
```

becomes by a similar process

```
alternative: (symbol; repetition)*.
```

which then can be processed again to give

```
alternative: (terminal; ^nonterminal; ^refinement; ^one-or-more; ^zero-or-more)*.
```

and then one final time to give

```
alternative: (^explicit-terminal; ^implicit-terminal; ^nonterminal; ^refinement; ^one-or-more;
^zero-or-more)*.
```

and so on.

So our final parse-tree description for this grammar fragment is:

```
ixml: (^rule)+.
rule: @name, (^alternative)+.
alternative: (^explicit-terminal; ^implicit-terminal; ^nonterminal; ^refinement; ^one-or-more;
^zero-or-more)*.

one-or-more: (^alternative)+; (^alternative)+, ^separator.
zero-or-more: (^alternative)+; (^alternative)+, ^separator.
separator: ^explicit-terminal; ^implicit-terminal; @nonterminal; @refinement.
symbol: ^explicit-terminal; ^implicit-terminal; ^nonterminal; ^refinement.
terminal: ^explicit-terminal; ^implicit-terminal.
explicit-terminal: @string.
implicit-terminal: @string.
nonterminal: @name.
refinement: @name.
attribute: @name.
```

From a data-structuring point of view, these are type definitions, semicolons representing unions, commas representing structs, and repetitions representing lists.

Subsets

A corollary of the observation above that any external representation of a format is equivalent, as long as it has the same parse tree, is that if a format has a normalised grammar that is a subset of another normalised grammar, and the same root node, then the first language is compatible with the second (but not the other way round).

Data Conversion

Since external representation is no longer important, it would be easy to transform one format to another, as long as their normalised grammars are compatible. So transforming an ixml grammar to one in a different representation is as simple as parsing it with one grammar and serialising it with another.

Conclusion

XML has provided us with a standard data-representation layer, and a standard processing pipeline. With a relatively small addition we can open up the pipeline to all structured documents, making XML truly ubiquitous.

References

[ixml] Pemberton, Steven. “Invisible XML” Presented at Balisage: The Markup Conference 2013, Montréal, Canada, August 6 - 9, 2013. In Proceedings of Balisage: The Markup Conference 2013. Balisage Series on Markup Technologies, vol. 10 (2013). doi:10.4242/BalisageVol10.Pemberton01.

[bnf] Backus-Naur Form, http://en.wikipedia.org/wiki/Backus-Naur_Form

[vwf] S. Pemberton, 1982, Executable Semantic Definition of Programming Languages Using Two-level Grammars, http://www.cwi.nl/~steven/vw.html

[earley] https://en.wikipedia.org/wiki/Earley_parser

[aho] Aho, AV, and Ullman, JD, "The Theory of Parsing, Translation, and Compiling", Prentice-Hall, 1972, ISBN

0139145567