On the Descriptions of Data

The Usability of Notations

Steven Pemberton
CWI, Amsterdam
<steven.pemberton@cwi.nl>

Abstract

Usability describes the ease with which you can use something: how long it takes to achieve your aims, how correctly, and whether it is enjoyable in the process.

While this is normally applied to interactions with processes, such as computer programs, or machines, it is also applicable to notations: how easily can you achieve what you are trying to do, does the notation aid you in avoiding errors, and, indeed, is it enjoyable to do? However, surprisingly little attention is paid to designing notations for usability.

Invisible XML (ixml) 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. It uses a notation for describing data formats that are to be parsed.

Earlier papers on ixml discuss the design of the notation based on functional requirements of the language. This paper discusses changes to the design following experience with using it, giving examples of its use to develop data descriptions, and in passing, suggests other output formats.

Keywords: XML, parsing, notation design, data representation, usability

1. Usability

The study of usability is typically carried out in the field of human-computer interaction [10]. Although there are a small number of definitions of what usability is, all agree on three basic points [7], the degree to which you have:

- Efficiency: the time needed to achieve a purpose
- Effectiveness: being able to achieve the purpose without error
- Enjoyability: being able to enjoy achieving the putrpose.

Some definitions add *learnability* to this, and some add a related concept, *memorability*, which is the ability to return after some time and start using it with less effort than initially; however, others argue that these are orthogonal concepts, and that something that is inherently hard to learn can still be usable despite the steep learning curve.

2. Notations

That notations can affect what is achievable with them is long known. An obvious example is the notation for numbers: the Roman notation CXXVIII is reasonably good for representing the number 128, and is just about acceptable for addition, since with a few simple rules it is possible to add two numbers represented with Roman numerals together, but it is close to impossible to do multiplication in any general way using the numerals alone.

On the other hand, the indo-arabic system that we now use is good for representing numbers, and makes it easy to both add and multiply.

3. Invisible XML

Invisible XML [4], [5], [6] is a methodology for treating heterogenous data representations so that they can be processed as XML: as long as the external data representation is parsable, it is possible to create an internal representation of the data that can further be treated as if it originated from an XML representation, or that can be serialised so that it really is XML, for input to other tools.

4. Data Descriptions

Ixml works by using context-free grammars [1] to describe the format of documents to be converted. A general parsing algorithm, such as Earley [3], is then used to create a parse-tree for the parsed document. This parse tree is then pruned, and the result can be treated as an internal representation of an XML instance, for processing or serialisation.

Using a general-purpose parsing algorithm means that users don't have to be aware of the special rules for languages that are dictated by some types of parsing algorithm, and additionally widens the class of languages that can be described.

In the initial design of ixml [4], focus was put on the functional requirements of the data description language: it was necessary to be able to specify the input format, and describe how to prune the resulting parse-tree to only deliver the useful parts for extracting the enclosed data.

There are two parts to a grammar: the *non-terminal* symbols, roughly speaking describing the semantic concepts represented in a language, and the *terminal symbols* used to express the language.

Terminal symbols can have *intrinsic* role, an *extrinsic* role, or a *structural* role. For instance, to represent "the sum of a and b", you might write `(a+b)`. The characters "a" and "b" are intrinsic, change them and you change the meaning; "+" is extrinsic, since it only identifies that addition is involved; indeed if you change it to "×" the meaning changes, but in a different way; finally the brackets add nothing essential to the meaning, and are only used as punctuation and to disambiguate similar forms.

Nonterminals also have three roles. *Intrinsic*, where they represent a semantic concept in a language, such as "statement" in a programming language; *structural*, where they are used for disambiguation and structuring, such as "factor" in an expression, and finally *refining*, where a name is given largely for convenience to a syntactic structure than can occur in different places in a language.

In the original design, since terminal symbols were almost never intrinsic, it was decided to exclude them by default from the parse-tree, and required specific marking to include a symbol. On the other hand since nonterminals were the *sine qua non* of a parse tree, they were included by default, and required specific marking to exclude them.

Exclusions were always marked at the *use* of a symbol rather than its definition, since that meant that symbols generally were more useful in their use in the descriptions, and since inclusions were by definition only possible at the point of use (since terminal symbols are only used, and have no further definition).

However, while this produced a technically sufficient notation, it turned out in use that nonterminals need to be excluded very often, making for descriptions that were not easy to read. So in the original paper it was described how *all* symbols were excluded by default from the parse tree, and any symbol had to be explicitely marked for inclusion.

5. User Testing

Part of usability-based design is user testing. As Nielsen points out [9], user testing with only a small number of users can reveal large amounts of data about a design.

What we learned after experience of ixml in use was:

1. It is easier to design the data description by starting from the full parse tree, and incrementally pruning the parts that are not needed.
2. Very many non-terminals are refinements or extrinsic, and it is more sensible to prune these at the definition rather than the use-point.
3. Occasionally you want to prune all uses of a nonterminal but one, so it is useful to be able to mark a definition as deleted, but mark it as inserted at a use-point.
4. There are occasions where you need to say "any character *except* this small list is acceptable at this position" (this had as consequence that a notation for character sets was necessary, something that was rejected in the initial design).
5. It is useful to have an explicit notation for something that is optional.

6. It is useful to be able to use Unicode character classes.

6. An Example

To show the process of interatively building an ixml grammar, let's take an example. In order to keep the example manageable we will look at the grammar for small expressions, such as `pi×(10+b)`.

The raw grammar for expressions could look like this:

```
expr: term; sum.
sum: expr, "+", term.
term: factor; prod.
prod: term, "×", factor.
factor: id; number; "(", expr, ")".
id: letter+.
number: digit+.
letter: ["a"-"z"].
digit: ["0"-"9"].
```

The parse of the string "`pi×(10+b)`" using this grammar and then serialised as XML looks like this:

```
<expr>
   <term>
      <prod>
         <term>
            <factor>
               <id>
                  <letter>p</letter>
                  <letter>i</letter>
               </id>
            </factor>
         </term>×
         <factor>(
            <expr>
               <sum>
                  <expr>
                     <term>
                        <factor>
                           <number>
                              <digit>1</digit>
                              <digit>0</digit>
                           </number>
                        </factor>
                     </term>
                  </expr>+
                  <term>
```

```
                     <factor>
                        <id>
                           <letter>b</letter>
                        </id>
                     </factor>
                  </term>
               </sum>
            </expr>)
         </factor>
      </prod>
   </term>
</expr>
<expr>
   <term>
      <prod>
         <term>
            <factor>
               <id>
                  <letter>p</letter>
                  <letter>i</letter>
               </id>
            </factor>
         </term>×
         <factor>(
            <expr>
               <sum>
                  <expr>
                     <term>
                        <factor>
                           <number>
                              <digit>1</digit>
                              <digit>0</digit>
                           </number>
                        </factor>
                     </term>
                  </expr>+
                  <term>
                     <factor>
                        <id>
                           <letter>b</letter>
                        </id>
                     </factor>
                  </term>
               </sum>
            </expr>)
         </factor>
      </prod>
   </term>
</expr>
```

As you can see, for such a small input string, this is a surprisingly long XML document, since it records every little part of the parse.

The first thing that is noticeable is the large numbers of `term`s and `factor`s in the tree. Since we don't need these at all in the parse tree, we mark them for exclusion at their definitions, by prepending a "-" sign:

```
-term: factor; prod.
-factor: id; number; "(", expr, ")".
```

(This only excludes the elements for `term` and `factor` from the serialisation, but not their children.)

The parse tree has already become *considerably* smaller:

```
<expr>
   <prod>
      <id>
         <letter>p</letter>
         <letter>i</letter>
      </id>×(
      <expr>
         <sum>
            <expr>
               <number>
                  <digit>1</digit>
                  <digit>0</digit>
               </number>
            </expr>+
            <id>
               <letter>b</letter>
            </id>
         </sum>
      </expr>)
   </prod>
</expr>
```

Now we exclude the letters and digits (the nonterminals, not the terminals):

```
-letter: ["a"-"z"].
-digit: ["0"-"9"].
```

which gives as parse:

```
<expr>
   <prod>
      <id>pi</id>×(
      <expr>
```

```
            <sum>
               <expr>
                  <number>10</number>
               </expr>+
               <id>b</id>
            </sum>
         </expr>)
      </prod>
   </expr>
```

and finally we get rid of the `exprs`:

```
<prod>
   <id>pi</id>×(
   <sum>
      <number>10</number>+
      <id>b</id>
   </sum>
</prod>
```

to give us a 'minimal' parse: structured, unambiguous, and still with all semantic information.

A consequence of including everything by default in the parse is the presence of 'extraneous' terminals in the tree, such as "+" and "(" above. These are harmless in themselves, and can be ignored in processing (since they are identifiable as characters not immediately surrounded by an element); they also provide an advantage that all input characters are present in the output, making the original document easier to recreate. However, if they are not wanted, they can still be explicitly removed by marking them in the same way:

```
sum: expr, -"+", term.
-factor: id; number; -"(", expr, -")".
```

giving as parse:

```
<prod>
   <id>pi</id>
   <sum>
      <number>10</number>
      <id>b</id>
   </sum>
</prod>
```

7. Attributes

Ixml grammars allow you to express that some nonterminals should be serialised as attributes in the XML, so for instance, changing the rules for `id` and `number` in the above grammar to the following:

```
id: name.
@name: letter+.
number: value.
@value: digit+.
```

gives the serialisation:

```
<prod>
   <id name='pi'/>
   <sum>
      <number value='10'/>
      <id name='b'/>
   </sum>
</prod>
```

Again, non-terminals can be marked like this at the point of definition, or the point of use.

Note that you have to be careful when defining non-terminals as attributes, since while child elements are ordered in XML, and several child elements may have the same name, this is not true of attributes.

8. Adding Nodes

If we change the input parse string from "`pi×(10+b)`" to "`pi+(10×b)`" and process it, we get:

```
<sum>
   <id name='pi'/>
   <prod>
      <number value='10'/>
      <id name='b'/>
   </prod>
</sum>
```

A possible problem here, is that this is the identical parse to what you would get for the string "`pi+10×b`": that is, the brackets in the input do not affect the parse tree. This is understandable, since the brackets add no extra information: the two strings are semantically identical. We can fix this, if required, by adding back the node for `expr` in the case that it is a bracketed expressions:

```
-factor: id; number; -"(", ^expr, -")".
```

giving for the bracketed case:

```
<sum>
   <id name='pi'/>
   <expr>
      <prod>
         <number value='10'/>
         <id name='b'/>
      </prod>
   </expr>
</sum>
```

Another solution would be to add a rule:

```
-factor: id; number; bracketed.
bracketed: -"(", expr, -")".
```

to give:

```
<sum>
   <id name='pi'/>
   <bracketed>
      <prod>
         <number value='10'/>
         <id name='b'/>
      </prod>
   </bracketed>
</sum>
```

9. Other Examples

9.1. URLs

As another small example, consider this restricted grammar for URLs:

```
url: scheme, ":", authority, path.
scheme: name.
@name: letter+.
authority: "//", host.
host: sub+".".
sub: name.
path: ("/", seg)+.
seg: sname.
@sname: fletter*.
-letter: ["a"-"z"]; ["A"-"Z"]; ["0"-"9"].
-fletter: letter; ".".
```

Here you can see the use of repetitions with separators (`sub+"."` means one or more `sub`s separated by points), as well as a grouped repetition: `("/", seg)+`

Given the input string "`http://www.w3.org/TR/1999/xhtml.html`" you get the parse:

```
<url>
   <scheme name='http'/>:
   <authority>//
      <host>
```

```
            <sub name='www'/>.
            <sub name='w3'/>.
            <sub name='org'/>
         </host>
      </authority>
      <path>/
         <seg sname='TR'/>/
         <seg sname='1999'/>/
         <seg sname='xhtml.html'/>
      </path>
   </url>
```

This illustrates a point about attributes and elements: if they have a different syntax in the input, they have to have a different name in the output. Here `sub` has an attribute called `name` and `seg` has an attribute called `sname`. The attribute `sname` cannot be called `name` because it has a different syntax to a `name` (an `sname` can contain points ".", whereas a `name` may not).

9.2. Parsing JSON

We can take the grammar for JSON [8], and convert it to ixml:

```
json: S, object.
object: "{", S, members, "}", S.
-members: pair*(",", S).
pair: @string, S, ":", S, value.
array: "[", S, value*(",", S), "]", S.
-value: string, S; number, S; object; array; "true", S; "false", ▶
S; "null", S.
string: -"""", char*, -"""".
-char: ~['"'; "\"; #0-#1F];
  '\', ('"'; "\"; "/"; "b"; "f"; "n"; "r"; "t"; "u", hexdigits).
number: "-"?, int, frac?, exp?.
-int: "0"; digit19, digit*.
-frac: ".", digit+.
-exp: ("e"; "E"), sign?, digit+.
-sign: "+"; "-".
-S: " "*.
-digit: ["0"-"9"].
-digit19: ["1"-"9"].
-hexdigits: hexdigit, hexdigit, hexdigit, hexdigit.
-hexdigit: digit; ["a"-"f"; "A"-"F"].
```

(There are some potential quibbles here: json.org says that whitespace may appear *between* any two tokens, without saying what a token is, or what whitespace is; this means that leading and trailing spaces are not allowed, which this grammar does allow).

Parsing this piece of JSON:

```
{"name": "pi", "value": 3.145926}
```

then gives us this XML:

```
<json>
   <object>{
      <pair string='name'>:
         <string>pi</string>
      </pair>,
      <pair string='value'>:
         <number>3.145926</number>
      </pair>}
   </object>
</json>
```

9.3. Parsing XML

To move to another example, let us consider a simplified grammar for XML itself:

```
xml: element.
element: -"<", name, (-" "+, attribute)*, (-">", content, -"</", ▶
close, -">"; -"/>").
@name: ["a"-"z"; "A"-"Z"]+.
@close: name.
attribute: name, -"=", value.
@value: -'"', dchar*, -'"'; -"'", schar*, -"'".
content: (cchar; element)*.
-dchar: ~['"'; "<"].
-schar: ~["'"; "<"].
-cchar: ~["<"].
```

Note the notation for character exclusions: `~['"'; "<"]` means "any character except a double quote or a less-than".

Using input:

```
<test lang="en" class="test">
  This <em>is</em> a test.
</test>
```

gives as parse:

```
<xml>
   <element name='test' close='test'>
      <attribute name='lang' value='en'/>
      <attribute name='class' value='test'/>
      <content>  This
         <element name='em' close='em'>
            <content>is</content>
         </element> a test.</content>
   </element>
</xml>
```

Note XML is not context-free, since you can't check on parsing that open and closing tags match. It is for this reason that the serialisation contains both the opening and the closing tag names, so that they can be checked on later processing.

10. Parsing ixml

Of course, an ixml grammar is itself expressible in ixml. That is to say, we can write a grammar that expresses ixml, and then process it with ixml to give an XML serialisation of the grammar.

So here is ixml expressed in itself. "S" represents a string of spaces or comments. Comments are enclosed in curly braces { and }. The notation `mark?` means that `mark` is optional at that point.

```
ixml: S, rule+.
rule: mark?, name, S, ":", S, def, ".", S.
def: alt+(";", S).
alt: term*(",", S).
-term: factor; repeat0; repeat1; option.
repeat0: factor, "*", S, sep?.
sep: factor.
repeat1: factor, "+", S, sep?.
option: factor, "?", S.
-factor: nonterminal; terminal; "(", S, def, ")", S.
nonterminal: mark?, name, S.
terminal: mark?, (quoted; hex; charset; exclude).

charset: "[", S,  element+(";", S), "]", S.
exclude: "~", S, -charset.

-element: range; character, S; class, S.
range: from, S, "-", S, to, S.
@from: character.
@to: character.
class: letter, letter. {One of the Unicode character classes}

@name: letgit, xletter*.
-letgit: letter; digit.
-letter: ["a"-"z"; "A"-"Z"].
-digit: ["0"-"9"].
-xletter: letgit; "-".

@mark: "@", S; "^", S; "-", S.
```

```
quoted: -'"', dstring, -'"', S; -"'", sstring, -"'", S.
@dstring: dchar+.
@sstring: schar+.
dchar: ~['"']; '""'. {all characters, dquotes must be doubled}
schar: ~["'"]; "''". {all characters, squotes must be doubled}
-character: '"', dchar, '"'; "'", schar, "'"; hex.

hex: "#", number.
number: hexit+.
-hexit: digit; ["a"-"f"; "A"-"F"].

-S: (" "; comment)*.
comment: "{", cchar*, "}".
-cchar: ~["}"].
{the end}
```

Parsing this *with itself* gives 479 lines of output, so only the first couple of rules will be shown:

```
<ixml>
   <rule name='ixml'>:
      <def>
         <alt>
            <nonterminal name='S'/>,
            <repeat1>
               <nonterminal name='rule'/>+
            </repeat1>
         </alt>
      </def>.</rule>
   <rule name='rule'>:
      <def>
         <alt>
            <option>
               <nonterminal name='mark'/>?
            </option>,
            <nonterminal name='name'/>,
            <nonterminal name='S'/>,
            <terminal>
               <quoted dstring=':'/>
            </terminal>,
            <nonterminal name='S'/>,
            <nonterminal name='def'/>,
            <terminal>
               <quoted dstring='.'/>
            </terminal>,
            <nonterminal name='S'/>
         </alt>
      </def>.</rule>
```

The upshot of this is that *any* grammar that produces a similar output (ignoring extraneous terminals) can be used to parse a document. That means that the grammar format for ixml is not set in stone, but alternatives can be offered. For instance, here is ixml expressed in a different grammar format, BNF [2], representing itself:

```
<ixml>::= <S> <rules>
-<rules>::= <rule> | <rule> <rules>
<rule>::= <mark> <name> "::=" <S> <def> |
      <name> "::=" <S> <def>
<def>::= <alts>
-<alts>::= <alt> | <alt> "|" <S> <alts>
<alt>::= <terms> | <empty>
-<terms>::= <term> | <term> <S> <terms>
<empty>::=
<term>::= <mark> <name> | <name> | <string> | <range>
@<name>::= "<" <letters> ">" <S>
@<mark>::= "@" <S> | "^" <S> | "-" <S>
<letters>::= <letter> <more-letters>
<letter>::= ["a"-"z"] | ["A"-"Z"] | ["0"-"9"]
<more-letters>::= <letter> <more-letters> | "-" <more-letters> | <empty>
@<string>::= """" <chars> """" <S>
<chars>::= <char> <chars> | <char>
<char>::= [" "-"!"] | ["#"-"~"] | """""" {all characters, quotes must be ▶
doubled}
<range>::= "[" <S>  <character> <S> "-" <S> <character> <S> "]" <S>
-<character>::= """" <char> """" | """" """" """" """"
-<S>::= " " <S> | <comment> <S> |
<comment>::= "{" <schars> "}"
-<schars>::= <schar> <schars> |
-<schar>::= [" "-"|"] | "~" {Everything except: }
```

which parsed with itself gives (again, only showing the first few lines):

```
<ixml>
   <rule name='ixml'>::=
      <def>
         <alt>
            <term name='S'/>
            <term name='rules'/>
         </alt>
      </def>
   </rule>
   <rule mark='-' name='rules'>::=
      <def>
         <alt>
            <term name='rule'/>
         </alt>|
         <alt>
            <term name='rule'/>
            <term name='rules'/>
         </alt>
      </def>
   </rule>
```

11. Other Output Formats

In the true spirit of "data wants to be format neutral", there is strictly speaking no reason why the parse tree need be in XML, but could be equally well serialised in some other form, such as JSON. Taking an example like this:

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

you might be tempted to try to express this as:

```
{"expr":
    {"prod":
        {"letter": "a";
         "sum": {"digit":"3"; "letter":"b"}
        }
    }
}
```

However, JSON object members are more like XML attributes than child elements, because they are not ordered, and (probably) member names may not be duplicated.

Therefore you have to use arrays, which *are* ordered, where each array element is a single-member group:

```
{"expr":
    [{"prod":
        [{"letter": "a"}],
        [{"sum":
            [{"digit":"3"}],
            [{"letter":"b"}]
        }]
    }]
}
```

There still remains the question of what to do with extraneous terminals, such as here:

```
<expr>
    <prod>
        <letter>a</letter>×(
        <sum>
            <digit>3</digit>+
            <letter>b</letter>
        </sum>)
    </prod>
</expr>
```

The best approach is to treat them as members without a name, like this:

```
{"expr":
    [{"prod":
        [{"letter": "a"}],
        [{"": "×("}],
        [{"sum":
            [{"digit":"3"}],
            [{"":"+"}]
            [{"letter":"b"}]
        }],
        [{"":")"}]
    }]
}
```

12. Conclusion

Notations can have a profound effect on what we are able to express. In the end notations are a human artifact, for use by people, and we should not lose sight of the fact that what is suitable for an automaton is not necessarily ideal for a person. We can learn from the techniques of usability, such as user testing and iterative design, to make notations that are more suitable for human use.

Bibliography

[1] AV Aho, and JD Ullman, The Theory of Parsing, Translation, and Compiling, Prentice-Hall, 1972, ISBN 0139145567

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

[3] Earley Parser, https://en.wikipedia.org/wiki/Earley_parser

[4] Steven Pemberton. 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. http://www.cwi.nl/~steven/Talks/2013/08-07-invisible-xml/invisible-xml-3.html

[5] Steven Pemberton. Data Just Wants to Be Format-Neutral, Proc. XML Prague, 2016, Prague, Czech Republic, pp109-120. http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf

[6] Steven Pemberton. Parse Earley, Parse Often: How to Parse Anything to XML, Proc. XML London 2016, London, UK, pp120-126. http://xmllondon.com/2016/xmllondon-2016-proceedings.pdf#page=120

[7] International Standards Organisation, ISO 9241 - Ergonomic requirements for office work, 1997.

[8] Introducing JSON, http://json.org/

[9] Jakob Nielsen, Why You Only Need to Test with 5 Users, 2000. https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/

[10] Ben Shneiderman *et al.*, Designing the User Interface, Pearson, 2013, ISBN 1292023902.