Where Semantics Lies

Stephen Ramsay

Should the syntax of XML have been scrapped in favor of s-expressions? This debate, which raged for years, and which occasionally reappears, has all the ring of a religious war (Windows vs. Mac, Emacs vs. Vi, big-endian vs. little endian, and so forth).

The risks we take in even broaching the subject are manifold. A talk based on a question like this is destined to be both technical and philosophical, which is to say, bad. And try as I might, I will undoubtedly seem guilty of favoring one side or another, whatever protestations I might make to the contrary. It is in the nature of religious warfare to be on one side or another, and to be wrong either way.

But my purpose here really isn't to settle this question, or even to re-introduce the debate. What I want to do is use this mostly wrong-headed back-and-forth to shake out something that I think is actually highly relevant to the topic of data modeling—in the humanities, or anywhere else. That highly relevant point can be stated pithily by asking "where the semantics lies" in our computational systems. In fact, what I'd like to say, is that this issue subtly affects the way we think about data modeling, even when we try to think about data modeling in complete isolation from any concerns about the use of data models, or even, for that matter, computational tractability.

But before I launch on this hopefully meaningful quest for theological insight, perhaps I should explain the terms of the debate that give rise to these meditations. What, to start with, is an s-expression?

An s-expression is a notation for representing tree structures, and it looks like this:

[SLIDE]

```
(dictionary
  (e-mail "electronic mail")
  (html "hypertext transport language")
```

```
  (xml "extensible markup language"))
```

We could define s-expressions much more formally, using an elegant recursive definition, but this is perhaps beside the main point. Because anyone looking at this will say, “You mean like Lisp?”

Yes, like Lisp. But let’s lay that aside for a moment and just consider the fact that anything we could possibly want to express in this notation can be expressed using the tree-structure notation we call XML. To wit:

[SLIDE]

```
<dictionary>
  <email>electronic mail</email>
  <html>hypertext transport language</html>
  <xml>extensible markup language</xml>
</dictionary>
```

Now, I’m leaving off attributes here, but it’s easy to imagine how we might add them in. If we have:

[SLIDE]

```
<dictionary>
  <e-mail acronym="false">electronic mail</email>
  <html acronym="true">hypertext transport language</html>
  <xml acronym="true">extensible markup language</xml>
</dictionary>
```

We could do something like:

[SLIDE]

```
(dictionary
  (e-mail :acronym false "electronic mail")
  (html :acronym true "hypertext transport language")
  (xml :acronym true "extensible markup language"))
```

I don't know if that's the best way; several methods have been proposed, whereby a tree node can be annotated with a key-value pair. But the point is this: These two representations are 100% isomorphic. Anything I can do in one, I can do in the other.

So you might suppose that one element of the debate involves syntax, and that is certainly true. Some people have argued quite vociferously (Notice! I did not use the word "correctly" just there) that XML is simply a needlessly verbose form of s-expression syntax. The standard reply is that "syntax matters" (c.f. Paul Prescod, one of the most vigorous defenders of XML over s-expressions). The s-expression syntax is certainly less busy. On the other hand, do you really want your TEI document to end with seventy-five closing parentheses?

But that is not actually the center of this debate at all. The center of the debate is the charge that the XML version has "no semantics."

[SLIDE]

```
<dictionary>
  <email>electronic mail</email>
  <html>hypertext transport language</html>
  <xml>extensible markup language</xml>
</dictionary>
```

Before I delve into what such a thing could possibly mean in this context, let's dive down one additional rabbit hole (we'll claw our way out; I promise). What does it mean for something to "have a semantics?"

The most frequently-offered answer to that question is that "semantics" refers to what a particular representation "means." Terrence Parr, who is the author of the ANTLR parser generator (and therefore someone who should surely know what semantics is) says:

[SLIDE]

> Loosely speaking, semantic analysis figures out what the input means (anything beyond syntax is called *the semantics.)* (4)

Now, this appears in a book called *Language Implementation Patterns.* Hardly light reading, but not a textbook on formal languages. He can surely be forgiven for speaking loosely. But when we turn to actual textbooks on formal languages, we get statements like this:

[SLIDE]

> This book is an analytic study of programming languages. Our goal is to provide a deep, working understanding of the essential concepts of programming languages [. . .] Most of the essentials relate to the semantics, or meaning, of program elements. (Friedman xv)

Now, both of these statements (and I could cite dozen more) seem to me to beg the question: What is meaning? Or to put it more awkwardly: What does it mean for something to mean . . . something? Obviously, this paper can only get worse.

My favorite denizen of this particular rabbit hole is Ludwig Wittgenstein, who offered what I continue to think is the most provocative answer to that question ever given. Some of you may be familiar with the canonical quotation in which his basic idea appears:

[SLIDE]

> For a large class of cases—though not for all—in which we employ the word "meaning" it can be defined thus: the meaning of a word is its use in the language.

It can be difficult to see at first what is so radical about this conception (Wittgenstein spends a few hundred pages drawing it out). Taken superficially, it might seem to be a statement about context—that "how a word is used" is important. But Wittgenstein goes considerably further than that, by rejecting the entire notion that propositions are true or false—or otherwise "meaningful"—based on some condition exterior to those propositions. In fact, what he really says is that there isn't anything other than use in context. There isn't anything to speak of beyond this complex web of relations. "What is justice?" "Justice" is the set of moments in which the term is deployed. That doesn't make the question itself nonsensical or unanswerable ("What is justice?" is, after all, an instance in which the term is employed), but it does make it unlikely that we'll get very far in forming a useful, all-purpose definition. And since forming useful, all-purpose definitions is presumably one of the goals of philosophy, we may find that posing questions like this gets us exactly nowhere.

What is useful about all of this for my purposes, though, is the fact that this idea of "meaning as use" give us not only as way to talk about computational representations, but as a way to describe computation itself. Computation, stated in the most minimalistic way possible, is about taking information from one state to another. In the normative case, it is about taking some linguistic construct and

producing another linguistic construct, though that is not essential. If you have a process that can take information and produce more information, we call that process "a computation." It's what happens when you press the equals sign on a calculator, and it's what happens when you Friend someone on Facebook.

The fact that we have some process by which to affect that transformation indicates something in particular about the information with which we began: We say that it "has a semantics." And this restates Wittgenstein's point quite succinctly. The information has meaning—has a semantics—because we can produce other states from it ("states" being anything from reorganizations to physical actions). In the absence of such productions (whether actual or potential), the information is literally "meaning-less." And while that condition might be rare, it sets a boundary condition on semantics. Most computational representations "have a semantics," because it is at least possible to imagine computations being performed on them. This is perhaps why Friedman and Wand (from whom I drew that previous quotation about the essentials of programming languages having to do with semantics), go on a few sentences later to say:
[SLIDE]

> The most interesting question about a program as object is, "What does it do?" (4)

If meaning is use, then who can argue?

So when the Lispers say that XML "has no semantics," they are presumably referring to the fact that *by itself,* XML has no inherent ability to produce anything at all. You need to describe that semantic meaning somewhere else; which is exactly the same as saying that you need some process by which that representation is either transformed into some other kind of representation, or otherwise results in another representation being produced.

But is that any less true of s-expressions? Isn't an s-expression *also* a representation in search of a means by which it can be translated into some other representation? What could possibly cause someone to say that s-expressions "have a semantics," while XML does not?

And the answer to that question does have to do with Lisp. Because in Lisp, there is no inherent difference between the representation you use for data and the representation you use for the process (i.e. the code). This, by the way, is called "homoiconicity," and it's an inherent property of all languages within the Lisp family. The most striking example of this *outside* the Lisp family is . . . wait for it . . . XSLT. In either case, it means that any code you write is also a data structure in

the language, and conversely, any data structure you create is at least potentially an executable process.

I say "potentially," because the Lispers are completely and totally wrong when they say that s-expressions have a semantics. They have a semantics if and only if you also have a way of taking that representation and using it to produce something else. That is to say, s-expressions have a semantics if you also have a Lisp to process them.

The consequent notion for XML, is that XML has a semantics if and only if you also have a way of taking that representation and using it produce something else. That is to say, XML has a semantics if you also have a schema combined with some way to process it.

But notice the difference there. If you have s-expressions, you need a Lisp runtime. If you have XML, you need a schema (which is to say, a grammar description combined with a type and structure ontology) combined with a (presumably Turing-complete) language. The difference, in other words, has less to do with angle brackets and parentheses, and much more to do with *where* the semantics lies within the overall system.

It is possible, of course, to process s-expressions without Lisp. It would also be possible to separate the grammatical description of type and structure constraints from the entity responsible for affecting the transformation and still be "doing Lisp." We are not talking about some kind of new affordance offered by Lisp, some deficiency in the XML ecosystem, or the other way around. When it comes to taking things from one information state to another, either system could be designed either way.

So my question is this: Does it matter at all where you put the semantics? And the answer to that is, I think, "yes"—and for more-or-less the same reason that "syntax matters."

The XML ecosystem implicitly imagines a radical decoupling between the act of data modeling and the act of processing data. In fact, it breaks the act of data modeling itself into several discrete stages, which, in practical terms, translates into a decoupling of the social act of marking up texts from the social act of modeling data, and both from the social act of processing data. I use the term "social act" as a way of designating different potential functions—"job descriptions," if you like—in the overall job of computation. You can be the person who decides how a grammar is applied in a particular document instance. Or, you can be the person who defines the grammar. Or, you can be the person who uses the grammar and the document to translate the information to another state.

What the Lispers argue for, is really a world in which the three things are

combined. Some partition of roles is, of course, still possible, but in practice, the Lisp ecosystem more-or-less demands that data modeling and data processing are never far from one another. While it's possible to imagine an s-expression tagger (maybe that would be a "parenner"), it is less easy to imagine that person not also being, at some level, a programmer.

But forget about Lisp (again). Because the real issue is not whether Lisp is good or bad. The issue is whether the distributed, decoupled model embodied in the XML ecosystem limits or expands our ideas about data modeling, as compared to a more centralized workflow in which data modeling is never far from data processing. And here, I will risk starting my own flame war by saying that, practically speaking, it does.

It does, because it is not possible to fully describe the semantics of anything apart from the processing that is enabled by the semantic relationships so described. An XML schema (and here, I'm talking about any kind of schema whatsoever) describes a grammar. It is, in fact, explicitly based on BNF grammars, which, of course, are used to describe programming languages. *This*, and not any particular instantiation, is the "data model" (a statement with which the designers of XML, by the way, are in full agreement). Typically, a schema defines a set of data types and a set of ordering constraints (which, again, are semantically meaningful *only at the point at which the document is processed).* But why stop there? Why not use that schema to define a set of control structures for processing data? Why not state whether variables are bound late or early, lazily or not? Why not define a set of additional data structures into which the data may be trivially transformed?

Well, you say, they did! It's called XSLT. And it's separate. And optional. And that's good. And you might be right. In fact, I think you are. But the fact still remains: Every data model is asymptotically approaching a processing model. I would even suggest that the question, "Are the data models we have proposed for the humanities sufficient to the task?" is equivalent to the question, "Does the semantics reside in the right place in our model?"

Not because shifting the semantics around give you new processing powers, but because to the degree that any data model attempts to stay neutral with respect to future processing regimes, it must limit the practical affordances offered by that model to the data modeler. To do so might be to commit an act of magnanimity; to construct a data model in the absence of any particular judgment about future processing is presumably to allow a thousand processes to flourish. But it is also to limit what can be modeled—because that is *exactly* where a good number of the decisions about semantics is being made. We may comfort ourselves with the

thought that every step up the chain of abstraction allows more flexibility at the processing level. But a dark voice remains—and should remain: Every step up the chain of abstraction also means separating further and further from what is presumably the point of all of this—namely, the attempt to exploit the computational tractability of the data. To give the processor more power, is necessarily to give the data modeler less control. Not just less control over the processing, but less control over the data model itself. So we really must ask ourselves this: Does having *less* control over the data model—which is not the same thing as saying "more flexibility"—make sense for our data?

Should we have gone this way? Should we have attempted to create a more tightly coupled ecosystem in which the line between data modeling and data processing vanishes as a practical matter (as it does in, for example, SQL)? Should we *now* think about doing that? I don't know. And I'm sorry to end with something so obviously decoupled from a practical recommendation of any kind. But I take the point of this symposium to have been, "Can we do what we want to do?" And I think it's at least apposite to point out that as long as we're talking about what we want to *do,* we are talking at least partially about what our data models cannot do by themselves.

Works Cited

Friedman, Daniel P and Mitchell Wand. *Essentials of Programming Languages*. 3rd ed. Boston: MIT P, 2006.

Parr, Terence. *Language Implementation Patterns.* Raleigh, NC: Pragmatic, 2010.

Prescod, Paul. "XML is not S-Expressions." Web. 11 March 2012. http://www.prescod.net/xml/sexprs.html.

Wittgenstein, Ludwig. *Philosophical Investigations.* Ed. P. M. S. Hacker. 4th ed. Malden, MA: Wiley-Blackwell, 2009.