Modeling Languages

Shriram Krishnamurthi

2003-09-05

1 Separating Wheat from Chaff

A student of programming languages who tries to study a new language can get overwhelmed by the many details he encounters. Virtually every programming language consists of

- a peculiar syntax
- some behavior associated with each syntax
- numerous useful libraries
- a collection of idioms that programmers of that language use

All four of these attributes are important to a programmer who wants to adopt a language. To a scholar, however, one of these is profoundly significant, while the other three are of lesser importance.

The first insignificant attribute is the syntax. Syntaxes are highly sensitive topics,[1] but in the end, they don't tell us very much about a programs *behavior*. For instance, the following code fragment

```
a [25] + 5
```

has more in common with

```
(+ (vector-ref a 25) 5)
```

than with

```
a [25] + 5
```

How come? Because the first is in Java and the second is in Scheme, both of which signal an error if the vector associated with `a` has fewer than 25 entries; the third, in C, blithely ignores the vector's size, leading to unspecified behavior, even though its syntax is exactly the same as that of the Java code.

That said, syntax does matter, at least inasmuch as its brevity can help programmers express and understand more by saying less.[2] For the purpose of this course, however, syntax will typically be a distraction, and will often get in the way of our understanding deeper similarities (as in the Java-Scheme-C example above). We will therefore use a uniform syntax for all the languages we implement.

The size of a language's library, while perhaps the most important characteristic to a programmer who wants to get a job done, is usually a distraction when studying a language. This is a slightly tricky contention, because the line between the core of a language and its library is fairly porous. Indeed, what one language considers an intrinsic primitive, another may regard as a potentially superfluous library operation. With experience, we can learn to distinguish between what must belong in the core and what need not. It is even possible to make this distinction quite rigorously using mathematics. Students who opt for the graduate reading portion of this course will learn about this.

[1] Some computer scientist—I don't know who, but almost certainly an American—once said, "Syntax is the Viet Nam of programming languages". Think that over for a moment.

[2] Alfred North Whitehead: "Civilization advances by extending the number of important operations which we can perform without thinking of them." This is one of the key axioms behind programming language design: as we learn more about topics, we codify that knowledge in the form of language.

Finally, the idioms of a language are useful as a sociological exercise ("How do the natives of this linguistic terrain cook up a Web script?"), but it's dangerous to glean too much from them. Idioms are fundamentally human, therefore bearing all the perils of faulty, incomplete and sometimes even outlandish human understanding. If a community of Java programmers has never seen a particular programming technique—for instance, the principled use of objects as callbacks—they are likely to invent an idiom to take its place, but it will almost certainly be weaker, less robust, and less informative to use the idiom than to just use callbacks. In this case, and indeed in general, the idiom tells us more about the programmers than it does about the language. Therefore, we should be careful to not read too much into it.

In this course, therefore, we will focus on the behavior associated with syntax, namely the *semantics* of programming languages. In popular culture, people like to say "It's just semantics!", which is a kind of put-down: it implies that their correspondent is quibbling over minor details of meaning, perhaps even in a jesuitical way. But communication is all about meaning: even if you and I use different words to mean the same thing, we understand one another; but if we use the same word to mean different things, great confusion results. Therefore, in this course, we will wear the phrase "It's just semantics!" as a badge of honor, because semantics leads to discourse which (we hope) leads to civilization.

Just semantics. That's all there is.

2 Modeling Meaning

So we want to study semantics. Great. But how shall we talk about meaning? Because human language is notoriously slippery, we rapidly run into dark corners and alleys without exit. When all else fails, we can always ruin a conversation by saying, "It depends on what the meaning of 'is' is." So what do we do?

Computer scientists use a variety of techniques for capturing the meaning of a program, all of which rely on the following premise: the most precise language we have is that of mathematics (and logic). Traditionally, three mathematical techniques have been especially popular: *denotational*, *operational* and *axiomatic* semantics. I list these names here primarily to tell you we won't be using any of them. Each of these is a rich and fascinating field of study in its own right, but these techniques are either too cumbersome or too advanced for our use. We will instead use a method that is a first cousin of operational semantics, which we'll call *interpreter* semantics.

The idea behind an interpreter semantics is simple: to explain a language, write an interpreter for it. The act of writing an interpreter forces us to understand the language, just as the act of writing a mathematical description of it does. But when we're done writing, the mathematics only sits on paper, whereas we can run the interpreter to study its effect on sample programs. We might incrementally modify the interpreter if it makes a mistake. When we finally have what we think is the correct representation of a language's meaning, we can then use the interpreter to explore what the language does on interesting programs. As we will briefly see in this course, we can even convert an interpreter into a compiler, thus leading to an efficient implementation that arises directly from the language's definition.

A careful reader should, however, be either confused or enraged (or both). We're going to describe the meaning of a language through an interpreter, which is a program. That program is written in some language. How do we know what *that* language means? Without establishing that first, our interpreters would appear to be mere scrawls in an undefined notation. What have we gained?

This is an important philosophical point, but it's not one we're going to worry about much in practice. We won't for the practical reason that the language in which we write the interpreter is one that we understand quite well: it's pretty simple and succint, so it won't be too hard to hold it all in our heads. The superior, theoretical, reason is this: others have already worked out the mathematical semantics of this simple language. Therefore, we really are building on rock. With that, enough of these philosophical questions for now. We'll see a few other ones later in the course.

3 Modeling Syntax

I've argued briefly that it is both futile and dangerous to vest too much emotion in syntax. In a platonic world, we might say

> Irrespective of whether we write $1 + 2$ or `1 2 +` or $(+\ 1\ 2)$, we always mean the idealized operation of adding the idealized numbers (represented by) "1" and "2".

What we want for our interpreter is essentially a procedure that carries out this conviction. That is, no matter which of many input syntaxes the user chooses to type in a program, this procedure converts it into a standard form that represents the user's *intent* over their *syntax*. We call such a program a *parser*.

In more formal terms, a parser is a program that converts *concrete syntax* (what a user might type) into *abstract syntax*. The word *abstract* signifies that the output of the parser is divorced from physical representations: it's the idealized representation of a physical object, the kind of representations that computers are so good at manipulating.

We'll now temporarily set parsing aside and get to the heart of the matter, which is writing an interpreter semantics. We will assume only numbers, addition and subtraction, and further assume both these operations are binary. This may seem like a rather silly exercise—you've known how arithmetic works since, oh, first grade?—but that's the whole point. By picking something you know really well, we can focus on the mechanics. Once you have a feel for the mechanics, we can use them to explore languages you have never seen before.

4 Interpreting Arithmetic

Let's assume, momentarily, that someone has written us a parser. For its abstract syntax, the parser chooses the following Scheme representation (the data definition):

```
(define-datatype AE AE?
  [num (n number?)]
  [add (lhs AE?)
       (rhs AE?)]
  [sub (lhs AE?)
       (rhs AE?)])
```

Here are some examples of parsed arithmetic expressions:

```
(num 3)
(add (num 3) (num 4))
(add (sub (num 3) (num 4))
     (num 7))
```

Notice that we don't care what concrete input led to these representations. The second expression, for instance, could have been written in at least three different ways:

1. `3 + 4` (conventional infix)
2. `3 4 +` (postfix)
3. `(+ 3 4)` (parenthesized prefix)

and other variations on each. Indeed, for each of these syntaxes, there exists at least one programming language that uses it. The parser lets us ignore this distinction.

Now we're ready to start writing the interpreter. The interpreter has the following contract and purpose:

```
;; calc : AE ⟶ number
;; reduces an AE to the number corresponding to it
```

which leads to these test cases:

```
(calc (num 3)) "should equal" 3
(calc (add (num 3) (num 4))) "should equal" 7
(calc (add (sub (num 3) (num 4)) (num 7))) "should equal" 6
```

(notice that the tests must be consistent with the contract and purpose statement!) and this template:

```
(define (calc an-ae)
  (cases AE an-ae
         [num (n) ...]
         [add (l r) ... (calc l) ... (calc r) ...)]
         [sub (l r) ... (calc l) ... (calc r) ...)]))
```

In this instance, we can convert the template into a function easily enough:

```
(define (calc an-ae)
  (cases AE an-ae
         [num (n) n]
         [add (l r) (+ (calc l) (calc r))]
         [sub (l r) (- (calc l) (calc r))]))
```

Running the test suite helps validate our interpreter.

What we have seen is actually quite remarkable. We have shown that a programming language with sufficient data structure capabilities can represent very significant data, namely programs themselves. That is, we can write programs that consume, and perhaps even return, programs. This is the foundation for both an interpreter semantics and for all compilers. This same idea—but with a much more primitive language, namely arithmetic, and a much poorer collection of data, namely just numbers—is at the heart of the proof of Gödel's Theorem.

5 A Primer on Parsers

We could duck the problem of parsing entirely this semester by asking you to always write programs in their *abstract*, rather than concrete syntax. However, that can grow tiring pretty quickly. Eventually, you're going to want a tool that lets you write programs more concisely, then converts that concise notation into abstract syntax. Voilà, that would be a parser! So here's a primer on how to roll one quickly.

As we've seen, there are many concrete syntaxes that we could use for AE programs. We're going to pick one particular, slightly peculiar notation. We will use a prefix parenthetical syntax that, for arithmetic, will look just like that of Scheme. With one twist: we'll use {braces} instead of (parentheses), just so we can keep AE code separate from Scheme code just by looking at the delimiters. That is, the three sample abstract syntaxes would correspond to the following source programs:

1. `3`
2. `{+ 3 4}`
3. `{+ {- 3 4} 7}`

Our choice is, admittedly, fueled by the presence of a convenient primitive in Scheme—the primitive that explains why so many languages built atop Lisp and Scheme *look* so much like Lisp and Scheme (i.e., they're parenthetical), even if they have entirely different meanings. That primitive is called *read*.

Here's how *read* works. It consumes an input port (or, given none, examines the standard input port). If it sees a sequence of characters that obey the syntax of a number, it converts them into the corresponding number in Scheme and returns that number. That is, the input stream

```
1 7 2 9 <eof>
```

(the spaces are merely for effect, not part of the stream) would result in the Scheme number 1729. If the sequence of characters obeys the syntax of a symbol (sans the leading quote), *read* returns that symbol: so

```
c s 1 7 3 <eof>
```

(again, the spaces are only for effect) evaluates to the Scheme symbol 'cs173. Likewise for other primitive types. Finally, if the input is wrapped in a matched pair of parenthetical delimiters—either (parentheses), [brackets] or {braces}—*read* returns a list of Scheme values, each the result of invoking *read* recursively. Thus, for instance, *read* applied to the stream

```
(1 a)
```

returns (*list* 1 'a), to

```
{+ 3 4}
```

returns (*list* '+ 3 4), and to

```
{+ {- 3 4} 7}
```

returns (*list* '+ (*list* '− 3 4) 7).

The *read* primitive is a crown jewel of Lisp and Scheme. It reduces what are conventionally two quite elaborate phases, called *scanning* (or *tokenizing*) and *parsing*, into two different phases: *reading* and *parsing*. Furthermore, it provides a single primitive for the first, so all that's left to do is the second.

The parser needs to identify what kind of AE program it's examining, and convert it to the appropriate abstract syntax. To do this, it needs a clear specification of the syntax of our language. We'll use *Backus-Naur Form* (BNF), named for two early programming language pioneers. A BNF description of rudimentary arithmetic looks like this:

```
<AE> ::= <num>
       | {+ <AE> <AE>}
       | {- <AE> <AE>}
```

The <AE> in the BNF is called a *non-terminal*, which means we can rewrite it as one of the things on the right-hand side. Read ::= as "can be rewritten as". Each | presents one more choice, called a *production*. Everything in a production that isn't enclosed in <···> is literal syntax. <num> is a *terminal*: we cannot expand it further. The <AE>s in the productions are references back to the <AE> non-terminal. (We call <num> a terminal because the process of expanding out <AE>s *terminates* with it; non-terminals are so named because we can expand further, so we needn't terminate; and each production *produces* a bigger <AE>.)

Notice the strong similarity between the BNF and the abstract syntax representation. In one stroke, it captures *both* the concrete syntax (the brackets, the notation for addition and subtraction) and a default abstract syntax. Indeed, the only thing that the actual abstract syntax data definition contains that's not in the BNF is names for the fields. Because BNF tells the story of concrete and abstract syntax so succintly, it has been used in definitions of languages ever since Algol 60, where it first saw use.

Assuming all programs fed to the parser are valid AE programs, the result of reading must be either a number, or a list whose first value is a symbol—either '+ or '−—and whose second and third values are AE sub-expressions that need further parsing. Thus, the entire parser looks like this:[3]

```
;; parse : sexp ⟶ AE
;; to convert s-expressions into AEs

(define (parse sexp)
  (cond
    [(number? sexp) (num sexp)]
    [(list? sexp)
     (case (first sexp)
       [(+) (add (parse (second sexp))
                 (parse (third sexp)))]
       [(-) (sub (parse (second sexp))
                 (parse (third sexp)))])]))
```

Here's the parser at work. The first line after each invocation of (*parse* (*read*)) is what the user types; the second line is the result of parsing. This is followed by the next prompt.

```
Welcome to DrScheme, version 205.
Language: Beginner.
> (parse (read))
3
#(struct:num 3)
> (parse (read))
{+ 3 4}
#(struct:add #(struct:num 3) #(struct:num 4))
> (parse (read))
```

[3]This is a parser for the whole language, but it is not a *complete* parser, because it performs very little error reporting: if a user provides the program {+ 1 2 3}, which is not syntactically legal, the parser silently ignores the 3 instead of signaling an error. You must write more robust parsers than this one.

```
{+ {- 3 4} 7}
#(struct:add #(struct:sub #(struct:num 3) #(struct:num 4)) #(struct:num 7))
```

Therefore, sure enough,

```
> (calc (parse (read)))
{+ {- 3 4} 7}
6
```

There is still one practical problem: to test the interpreter, we need to enter our inputs manually each time (or pre-convert them into abstract syntax). The problem arises because *read* demands input each time it runs. Fortunately, Scheme provides a handy notation that lets us avoid this problem: we can use the quote notation to simulate *read*. That is, we can write

```
Welcome to DrScheme, version 205.
Language: Beginner.
> (parse '3)
#(struct:num 3)
> (parse '{+ 3 4})
#(struct:add #(struct:num 3) #(struct:num 4))
> (parse '{+ {- 3 4} 7})
#(struct:add #(struct:sub #(struct:num 3) #(struct:num 4)) #(struct:num 7))
> (calc (parse '{+ {- 3 4} 7}))
6
```

6 Appendix: Primus Inter Parsers

Naturally, most languages do not use this form of parenthesized syntax. Writing parsers for those languages is much more complex; to learn more about that, study a typical text from a compilers course. What I want to do here is say a little more about parenthetical syntax.

I said above that *read* is a crown jewel of Lisp and Scheme. I think it's actually one of the great ideas of computer science. It serves as the cog that helps decompose a fundamentally difficult process—generalized parsing of the input stream—into two very simple processes: reading the input stream into an intermediate representation, and parsing that intermediate representation. Writing a reader is relatively simple: when you see a opening bracket, read recursively until you hit a closing bracket, and return everything you saw as a list. That's it. Writing a parser using this list representation, as we've seen above, is also a snap.

I call these kinds of syntaxes *bicameral*,[4] which is a term usually used to describe legislatures such as that of the USA. No issue becomes law without passing muster in both houses. The lower house establishes a preliminary bar for entry, but allows some rabble to pass through, knowing that the wisdom of the upper house will prevent it. In turn, the upper house can focus on a smaller and more important set of problems. In a bicameral syntax, the reader is the House of Representatives: it rejects the input

```
{+ 1 2)
```

(mismatched delimiters) but permits both of

```
{+ 1 2}
{+ 1 2 3}
```

the first of which is legal, the second of which isn't. It's the parser's (Senate's) job to eliminate the latter, more refined form of invalid input.

Look closely at XML sometime. What do the terms *well-formed* and *valid* mean, and how do they differ? How do these requirements relate to bicameral syntaxes such as that of Scheme?

[4] Two houses.