Chapter 1

Modeling Languages

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 but in the end, they don’t tell us very much about a program’s behavior. For instance, the following code fragment

\[
\text{a [25] + 5}
\]

has more in common with

\[
(+ \text{(vector-ref a 25) 5})
\]

than with

\[
\text{a [25] + 5}
\]

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.

\[1\] 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.
That said, syntax does matter, at least inasmuch as its brevity can help programmers express and understand more by saying less. For the purpose of our study, 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. Our supplementary materials will include literature on this distinction.

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 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.

1.1 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. 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 only briefly gloss over these topics, in section[19]) We will instead use a method that is a first cousin of operational semantics, which some people

---

2Alfred 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.
1.2. 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

- \(3 + 4\) (infix),
- \(3\ 4\ +\) (postfix), or
- \((+\ 3\ 4)\) (parenthesized prefix),

we always mean the idealized operation of adding the idealized numbers (represented by) “3” and “4”.

Indeed, there exists at least one programming language that uses each of these syntaxes.

If we ignore syntactic details, the essence of the input is a tree with the addition operation at the root and two leaves, the left leaf representing the number 3 and the right leaf the number 4. With the right data definition, we can describe this in Scheme as the expression

\((add\ (num\ 3)\ (num\ 4))\)

and similarly, the expression

- \((3 - 4) + 7\) (infix),
• 3 4 - 7 + (postfix), or
• (+ (− 3 4) 7) (parenthesized prefix)

would be represented as

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

One data definition that supports these representations is the following:

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

1.3 A Primer on Parsers

Ideally, our interpreter should consume terms of this type, thereby avoiding the syntactic details of the source language. For the user, however, it becomes onerous to construct terms of this type. Ideally, there should be a program that translates terms in concrete syntax into values of this type. 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.

As we’ve seen, there are many concrete syntaxes that we could use for arithmetic expression programs (which we will henceforth abbreviate AE). 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. Here are three programs employing this concrete syntax:

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
1.3. A PRIMER ON PARSERS

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

(+ (list '− 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 tokenizing (or scanning) and parsing, into three different phases: tokenizing, reading and parsing. Furthermore, it provides a single primitive that does the first and second, so all that’s left to do is the second. read returns a value known as an s-expression.

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, the BNF captures both the concrete syntax (the brackets, the operators representing 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 succinctly, 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 (specifically, either ‘+’ or ‘−’) and whose second and third values are AE sub-expressions that need further parsing. Thus, the entire parser looks like this:

\[
\text{;; parse : sexp → AE}
\]
\[
\text{;; to convert s-expressions into AEs}
\]
\[
\begin{align*}
\text{(define (parse sexp)} \\
\text{\hspace{1cm} (cond)} \\
\text{\hspace{1.5cm} [(number? sexp) (num sexp)]} \\
\text{\hspace{1.5cm} [(list? sexp)} \\
\text{\hspace{2cm} (case (first sexp)} \\
\text{\hspace{2.5cm} [(+) (add (parse (second sexp))} \\
\text{\hspace{3cm} (parse (third sexp)))]} \\
\text{\hspace{2.5cm} [(-) (sub (parse (second sexp))} \\
\text{\hspace{3cm} (parse (third sexp)))]])})
\end{align*}
\]

Here’s the parser at work. The first line after each invocation of \((\text{parse (read)})\) is what the user types; the second line is the result of parsing. This is followed by the next prompt:

Welcome to DrScheme, version 208p1.
Language: PLAI - Advanced Student.
> (parse (read))
3
#(struct:num 3)
> (parse (read))
{+ 3 4}
#(struct:add #(struct:num 3) #(struct:num 4))
> (parse (read))
{+ {- 3 4} 7}
#(struct:add #(struct:sub #(struct:num 3) #(struct:num 4)) #(struct:num 7))

This raises a practical problem, however: to test the programs, we need to enter our inputs manually each time (or pre-convert them into abstract syntax). The problem arises because \text{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 \text{read}. That is, we can write

Welcome to DrScheme, version 208p1.

---

3This 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 according to our BNF specification, the parser silently ignores the 3 instead of signaling an error. You must write more robust parsers than this one.

4Note that for most of our study, we will use the Advanced Student language level. The Beginning Student and Intermediate Student levels are present only for symmetry with other textbooks.
1.4 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 an 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, 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?

---

5Two houses.
Chapter 2

Interpreting Arithmetic

We’ll now 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 is indeed a very rudimentary exercise, but that’s the 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.

The interpreter has the following contract and purpose:

```plaintext
;; calc : AE → number
;; consumes an AE and computes the corresponding number
```

which leads to these test cases:

```plaintext
(test (calc (parse ’3)) 3)
(test (calc (parse ’{+ 3 4}))) 7)
(test (calc (parse ’{+ {- 3 4} 7}))) 6)
```

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

```plaintext
(define (calc an-ae)
  (type-case 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:

```plaintext
(define (calc an-ae)
  (type-case 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.