Chapter 1
Modeling Languages

A student of programming languages who tries to study a new language can be overwhelmed by details. Virtually every language consists of:

- a peculiar syntax,
- some behavior associated with each syntax,
- numerous useful libraries, and
- 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, consider the following four fragments:

1. \{a \[25\]
2. (vector-ref \ a \[25\]
3. \[a \[25\]
4. \a \[25\]

Which two are most alike? The first and second, obviously! Why? 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, silently ignores the vector’s size, leading to unspecified behavior, even though its syntax is exactly the same as that of the Java code. The fourth, in ML or Haskell, is an application of a function to a list containing just one element: \2; that is, it’s not an array dereference at all, it’s a function application!

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

We will therefore use a uniform syntax for all the languages we implement.

1.2 Modeling Syntax

Exercise 1.2.1

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))

One data definition that supports these representations is the following:

(define-type AE
  (num n number?)
  (add (AE?) (AE?)
  (sub (AE?) (AE?)

Where AE stands for “Arithmetic Expression”.

Exercises

1.2.1. Why are the two and the sub-expressions of type AE rather than of type num? Provide sample expressions permitted by the former and rejected by the latter, and argue that our choice is reasonable.

1.3 A Primer on Parsers

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: operational, axiomatic and denotational. 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 2.1. We will instead use a method that is a first cousin of operational semantics, which some people 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 resides on paper, whereas we can run the interpreter to study its effect on sample programs. We might incrementally modify the interpreter 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 accomplish a task, 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 topic.

Finally, the benefit of a language are useful as a sociological exercise (“How do the natives of this lan
guage traditionally cook up a Web script?”), but it’s dangerous to glean too much from them. Idioms are funda
dentally human, and therefore bear 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 principle use of objects as callbacks—they are likely to invent its take on 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 sometimes tells us more about the programmers than about the language. Therefore, we should be careful to not read too much into one.

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 jocular 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. In this study, therefore, 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. But how? To study meaning, we need a language for describing meaning. Human language is, however, notoriously slippery, and as such is a poor means for communicating what are very precise concepts. But what can we use?

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: operational, axiomatic and denotational. 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 2.1. We will instead use a method that is a first cousin of operational semantics, which some people 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 resides on paper, whereas we can run the interpreter to study its effect on sample programs. We might incrementally modify the interpreter would be represented as

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

One data definition that supports these representations is the following:

(define-type AE
  (num n number?)
  (add (AE?) (AE?)
  (sub (AE?) (AE?)

Where AE stands for “Arithmetic Expression”.

Exercises

1.2.1. Why are the two and the sub-expressions of type AE rather than of type num? Provide sample expressions permitted by the former and rejected by the latter, and argue that our choice is reasonable.

1.3 A Primer on Parsers

Our interpreter should consume terms of type AE, thereby avoiding the syntactic details of the source lan
guage. For the user, however, it becomes onerous to construct terms of this type. Ideally, there should be a collection of idioms that programmers of that language use.

A careful reader should, however, be either confused or enraged (or both). We’re going to describe

The word

%McBride Fixity: “Syntax is the not Name of programming languages.”

Irrespective of whether we write

\(\{+ (-3 4) 7\}

or

\(\{+{ -34 }7\}

we get the same result. 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 topic.

Finally, the "benefit" of a language are useful as a sociological exercise ("How do the natives of this lan
guage traditionally cook up a Web script?"). But it’s dangerous to glean too much from them. Idioms are funda
dentally human, and therefore bear 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 principle use of objects as callbacks—they are likely to invent the same technique to mean the same thing, but it will almost certainly be weaker, less robust, and less informative to use the idiom than to just use callbacks. In this course, and indeed in general, the idiom sometimes tells us more about the programmers than about the language. Therefore, we should be careful to not read too much into one.

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 jocular 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. In this study, therefore, 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. But how? To study meaning, we need a language for describing meaning. Human language is, however, notoriously slippery, and as such is a poor means for communicating what are very precise concepts. But what can we use?

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: operational, axiomatic and denotational. 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 2.1. We will instead use a method that is a first cousin of operational semantics, which some people 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 resides on paper, whereas we can run the interpreter to study its effect on sample programs. We might incrementally modify the interpreter would be represented as

\(\{+ (-3 4) 7\}

\(\{+{ -34 }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 current 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

\(\{2 3 4 <eof>\)
1.3. A PRIMER ON PARSERS

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:

```scheme
(read ("c x 1 ? 3"))
```

(again, the spaces are only for effect) evaluates to the Scheme symbol `'c'.

Thus, for instance,

```scheme
(read ("a b c"))
```

returns a list of Scheme values, each the result of invoking read recursively. Thus, for instance, read applied to the stream

```scheme
([1 a])
```

returns (list 1 'a), to

```scheme
(+ 3 4)
```

returns (list + 3 4), and to

```scheme
(+ [- 3 4])
```

returns (list + (list [- 3 4]) 7).

### 1.3.1. A PRIMER ON PARSERS

The read primitive is a crown jewel of Lisp and Scheme. It reduces what are conventionally two quite elaborate phases, called lexing or scanning and parsing, into three different phases: lexing, reading and parsing. Furthermore, it provides a single primitive that does the first and second, so all that’s left to do is the third; read returns a value known as an `e-expression`.

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

```plaintext
<AE> ::= <num>
| (+ <AE> <AE>)
| (- <AE> <AE>)
| (\<num\>)
```

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. (To keep the description simple, we assume that there’s a corresponding definition for `\<num\>`, but leave its precise definition to your imagination.) The `<AE>`s in the productions are references back to the `<AE>` non-terminal.

Notice the strong similarity between the BNF and the abstract syntax representation. In one stroke, the we capture both for concrete syntax (the brackets) and the operators representing addition and subtraction, and a default abstract syntax. Indeed, the only thing that the 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.

### 1.4. PRIMITIV INTER PARSERS

Most languages do not use this form of parenthesized syntax. Writing parsers for languages that don’t is much more complex, to learn more about that, study a typical text from a compilers course. Before we drop the matter of syntax entirely, however, let’s discuss our choice—parenthesized syntax—on a little more depth.

I said above that read is a crown jewel of Lisp and Scheme. In fact, 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 document which is a term usually used to describe legislators 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 room to pass through knowing that the wisdom of the upper house will prevent excesses. In turn, the upper house can focus on a smaller and more important set of problems. In a bicameral syntax, the reader is, in American terms, the House of Representatives: it rejects the input

```scheme
(+ 1 2)
```

(mismatched delimiters) but permits both of

```scheme
(+ 1 2)
(+ 1 2 3)
```

the first of which is legal, the second of which isn’t in our arithmetic language. It’s the parser’s (semantic) job to eliminate the latter, more refined form of invalid input.

**Exercise 1.4.1** Based on this discussion, examine X3.1. What are 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?[^1]

[^1]: References.
Chapter 2

Interpreting Arithmetic

Having established a handle on parsing, which addresses syntax, we now begin to study semantics. We will study a language with 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 well, we can focus on the mechanics. Once you have a feel for the mechanics, we can use the same methods to explore languages you have never seen before.

The interpreter has the following contract and purpose:

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

which leads to these test cases:

```
(test (calc (parse "3")) 3)
(test (calc (parse "[ + 3 4 ]")) 7)
(test (calc (parse "[ + [ 3 4 ] ]")))
```

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

```
(define (calc an-ae)
  (type-case AE an-ae
    [num (n) n]
    [add (lr) (calc l) (calc r)]
    [sub (lr) (calc l) (calc r)])
```

Running the test suite helps validate our interpreter.

What we have seen is actually quite remarkable, though its full power may not yet be apparent. We have shown that a programming language with just the ability to represent structured data can represent one of the most interesting forms of data, namely programs themselves. That is, we have just written a program that consumes programs, perhaps we can even write programs that generate programs. The former is the foundation for an interpreter semantics, while the latter is the foundation for a compiler. 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 Gődel’s Theorem.