Thus:

```
> (parse '(+ (* 1 2) (+ 2 3)))
- ArithC
(plusC
 (multC (numC 1) (numC 2))
 (plusC (numC 2) (numC 3)))
```

Congratulations! You have just completed your first *representation of a program*. From now on we can focus entirely on programs represented as recursive trees, ignoring the vagaries of surface syntax and how to get them into the tree form. We're finally ready to start studying programming languages!

Exercise

What happens if you forget to quote the argument to the parser? Why?

2.5 Coda

Racket's syntax, which it inherits from Scheme and Lisp, is controversial. Observe, however, something deeply valuable that we get from it. While parsing traditional languages can be very complex, parsing this syntax is virtually trivial. Given a sequence of tokens corresponding to the input, it is absolutely straightforward to turn parenthesized sequences into s-expressions; it is equally straightforward (as we see above) to turn s-expressions into proper syntax trees. I like to call such two-level languages *bicameral*, in loose analogy to government legislative houses: the lower-level does rudimentary well-formedness checking, while the upper-level does deeper validity checking. (We haven't done any of the latter yet, but we will [REF].)

The virtues of this syntax are thus manifold. The amount of code it requires is small, and can easily be embedded in many contexts. By integrating the syntax into the language, it becomes easy for programs to manipulate representations of programs (as we will see more of in [REF]). It's therefore no surprise that even though many Lisp-based syntaxes have had wildly different semantics, they all share this syntactic legacy.

Of course, we could just use XML instead. That would be much better. Or JSON. Because that wouldn't be anything like an s-expression at all.

3 A First Look at Interpretation

Now that we have a representation of programs, there are many ways in which we might want to manipulate them. We might want to display a program in an attractive way ("pretty-print"), convert into code in some other format ("compilation"), ask whether it obeys certain properties ("verification"), and so on. For now, we're going to focus on asking what value it corresponds to ("evaluation"—the reduction of programs to *values*).

Let's write an evaluator, in the form of an *interpreter*, for our arithmetic language. We choose arithmetic first for three reasons: (a) you already know how it works, so we can focus on the mechanics of writing evaluators; (b) it's contained in every language

we will encounter later, so we can build upwards and outwards from it; and (c) it's at once both small and big enough to illustrate many points we'd like to get across.

3.1 Representing Arithmetic

Let's first agree on how we will represent arithmetic expressions. Let's say we want to support only two operations—addition and multiplication—in addition to primitive numbers. We need to represent arithmetic *expressions*. What are the rules that govern nesting of arithmetic expressions? We're actually free to nest any expression inside another.

Do Now!

> Why did we not include division? What impact does it have on the remarks above?

We've ignored division because it forces us into a discussion of what expressions we might consider legal: clearly the representation of `1/2` ought to be legal; the representation of `1/0` is much more debatable; and that of `1/(1-1)` seems even more controversial. We'd like to sidestep this controversy for now and return to it later [REF].

Thus, we want a representation for numbers and arbitrarily nestable addition and multiplication. Here's one we can use:

```
(define-type ArithC
  [numC (n : number)]
  [plusC (l : ArithC) (r : ArithC)]
  [multC (l : ArithC) (r : ArithC)])
```

3.2 Writing an Interpreter

Now let's write an interpreter for this arithmetic language. First, we should think about what its type is. It clearly consumes a `ArithC` value. What does it produce? Well, an interpreter evaluates—and what kind of value might arithmetic expressions reduce to? Numbers, of course. So the interpreter is going to be a function from arithmetic expressions to numbers.

Exercise

> Write your test cases for the interpreter.

Because we have a recursive datatype, it is natural to structure our interpreter as a recursive function over it. Here's a first template:

Templates are explained in great detail in *How to Design Programs*.

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
    [numC (n) n]
    [plusC (l r) ...]
    [multC (l r) ...]))
```

You're probably tempted to jump straight to code, which you can:

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
    [numC (n) n]
    [plusC (l r) (+ l r)]
    [multC (l r) (* l r)]))
```

Do Now!

Do you spot the errors?

Instead, let's expand the template out a step:

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
    [numC (n) n]
    [plusC (l r) ... (interp l) ... (interp r) ...]
    [multC (l r) ... (interp l) ... (interp r) ...]))
```

and now we can fill in the blanks:

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
    [numC (n) n]
    [plusC (l r) (+ (interp l) (interp r))]
    [multC (l r) (* (interp l) (interp r))]))
```

Later on [REF], we're going to wish we had returned a more complex datatype than just numbers. But for now, this will do.

Congratulations: you've written your first interpreter! I know, it's very nearly an anticlimax. But they'll get harder—much harder—pretty soon, I promise.

3.3 Did You Notice?

I just slipped something by you:

Do Now!

What is the "meaning" of addition and multiplication in this new language?

That's a pretty abstract question, isn't it. Let's make it concrete. There are many kinds of addition in computer science:

- First of all, there's many different kinds of *numbers*: fixed-width (e.g., 32-bit) integers, signed fixed-width (e.g., 31-bits plus a sign-bit) integers, arbitrary precision integers; in some languages, rationals; various formats of fixed- and floating-point numbers; in some languages, complex numbers; and so on. After the numbers have been chosen, addition may support only some combinations of them.

- In addition, some languages permit the addition of datatypes such as matrices.
- Furthermore, many languages support "addition" of strings (we use scare-quotes because we don't really mean the mathematical concept of addition, but rather the operation performed by an operator with the syntax +). In some languages this always means concatenation; in some others, it can result in numeric results (or numbers stored in strings).

These are all different meanings for addition. *Semantics* is the mapping of syntax (e.g., +) to meaning (e.g., some or all of the above).

This brings us to our first game of:

Which of these is the same?

- `1 + 2`
- `1 + 2`
- `'1' + '2'`
- `'1' + '2'`

Now return to the question above. What semantics do we have? We've adopted whatever semantics Racket provides, because we map + to Racket's +. In fact that's not even quite true: Racket may, for all we know, also enable + to apply to strings, so we've chosen the restriction of Racket's semantics to numbers (though in fact Racket's + doesn't tolerate strings).

If we wanted a different semantics, we'd have to implement it explicitly.

Exercise

> What all would you have to change so that the number had signed- 32-bit arithmetic?

In general, we have to be careful about too readily borrowing from the host language. We'll return to this topic later [REF].

3.4 Growing the Language

We've picked a very restricted first language, so there are many ways we can grow it. Some, such as representing data structures and functions, will clearly force us to add new features to the interpreter itself (assuming we don't want to use Gödel numbering). Others, such as adding more of arithmetic itself, can be done without disturbing the core language and hence its interpreter. We'll examine this next (section 4).

4 A First Taste of Desugaring

We've begun with a very spartan arithmetic language. Let's look at how we might extend it with more arithmetic operations that can nevertheless be expressed in terms of existing ones. We'll add just two, because these will suffice to illustrate the point.