Do Now!

What happens if you mis-apply functions to the wrong kinds of values? For instance, what if you give the `caml` constructor a string? What if you send a number into each version of `good?` above?

2 Everything (We Will Say) About Parsing

Parsing is the act of turning an input character stream into a more structured, internal representation. A common internal representation is as a tree, which programs can recursively process. For instance, given the stream

```
23 + 5 - 6
```

we might want a tree representing addition whose left node represents the number `23` and whose right node represents subtraction of `6` from `5`. A *parser* is responsible for performing this transformation.

Parsing is a large, complex problem that is far from solved due to the difficulties of ambiguity. For instance, an alternate parse tree for the above input expression might put subtraction at the top and addition below it. We might also want to consider whether this addition operation is commutative and hence whether the order of arguments can be switched. Everything only gets much, much worse when we get to full-fledged programming languages (to say nothing of natural languages).

2.1 A Lightweight, Built-In First Half of a Parser

These problems make parsing a worthy topic in its own right, and entire books, tools, and courses are devoted to it. However, from our perspective parsing is mostly a distraction, because we want to study the parts of programming languages that are *not* parsing. We will therefore exploit a handy feature of Racket to manage the transformation of input streams into trees: `read`. `read` is tied to the parenthetical form of the language, in that it parses fully (and hence unambiguously) parenthesized terms into a built-in tree form. For instance, running `(read)` on the parenthesized form of the above input—

```
(+ 23 (- 5 6))
```

—will produce a list, whose first element is the symbol `'+`, second element is the number `23`, and third element is a list; this list's first element is the symbol `'-`, second element is the number `5`, and third element is the number `6`.

2.2 A Convenient Shortcut

As you know you need to test your programs extensively, which is hard to do when you must manually type terms in over and over again. Fortunately, as you might expect, the parenthetical syntax is integrated deeply into Racket through the mechanism of *quotation*. That is, `'<expr>`—which you saw a moment ago in the above example—acts as if you had run `(read)` and typed `<expr>` at the prompt (and, of course, evaluates to the value the `(read)` would have).

2.3 Types for Parsing

Actually, I've lied a little. I said that `(read)`—or equivalently, using quotation—will produce a list, etc. That's true in regular Racket, but in Typed PLAI, the type it returns a distinct type called an *s-expression*, written in Typed PLAI as `s-expression`:

```
> (read)
- s-expression
[type in (+ 23 (- 5 6))]
'(+ 23 (- 5 6))
```

Racket has a very rich language of s-expressions (it even has notation to represent cyclic structures), but we will use only the simple fragment of it.

In the typed language, an s-expression is treated distinctly from the other types, such as numbers and lists. Underneath, an s-expression is a large recursive datatype that consists of all the base printable values—numbers, strings, symbols, and so on—and printable collections (lists, vectors, etc.) of s-expressions. As a result, base types like numbers, symbols, and strings are *both* their own type and an instance of s-expression. Typing such data can be fairly problematic, as we will discuss later [REF].

Typed PLAI takes a simple approach. When written on their own, values like numbers are of those respective types. But when written inside a complex s-expression—in particular, as created by `read` or quotation—they have type `s-expression`. You have to then *cast* them to their native types. For instance:

```
> '+
- symbol
'+
> (define l '(+ 1 2))
> l
- s-expression
'(+ 1 2)
> (first l)
. typecheck failed: (listof '_a) vs s-expression in:
  first
  (quote (+ 1 2))
  l
  first
> (define f (first (s-exp->list l)))
> f
- s-expression
'+
```

This is similar to the casting that a Java programmer would have to insert. We will study casting itself later [REF].

Observe that the first element of the list is still not treated by the type checker as a symbol: a list-shaped s-expression is a list of *s-expressions*. Thus,

```
> (symbol->string f)
. typecheck failed: symbol vs s-expression in:
  symbol->string
  f
```

```
  symbol->string
  f
  first
  (first (s-exp->list l))
  s-exp->list
```
whereas again, casting does the trick:
```
> (symbol->string (s-exp->symbol f))
- string
"+"
```
The need to cast s-expressions is a bit of a nuisance, but some complexity is unavoidable because of what we're trying to accomplish: to convert an *untyped* input stream into a *typed* output stream through robustly *typed* means. Somehow we have to make explicit our assumptions about that input stream.

Fortunately we will use s-expressions only in our parser, and our goal is to *get away from parsing as quickly as possible*! Indeed, if anything this should be inducement to get away even quicker.

2.4 Completing the Parser

In principle, we can think of `read` as a complete parser. However, its output is generic: it represents the token structure without offering any comment on its intent. We would instead prefer to have a representation that tells us something about the *intended meaning* of the terms in our language, just as we wrote at the very beginning: "representing addition", "represents a number", and so on.

To do this, we must first introduce a datatype that captures this representation. We will separately discuss (section 3.1) how and why we obtained this datatype, but for now let's say it's given to us:

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

We now need a function that will convert s-expressions into instances of this datatype. This is the other half of our parser:

```
(define (parse [s : s-expression]) : ArithC
  (cond
    [(s-exp-number? s) (numC (s-exp->number s))]
    [(s-exp-list? s)
     (let ([sl (s-exp->list s)])
       (case (s-exp->symbol (first sl))
         [(+) (plusC (parse (second sl)) (parse (third sl)))]
         [(*) (multC (parse (second sl)) (parse (third sl)))]
         [else (error 'parse "invalid list input")]))]
    [else (error 'parse "invalid input")]))
```

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