quite as attractive as it may sound, because now the callee faces a symmetric problem, not knowing whether its parameters are aliased or not. In traditional, sequential programs this is less of a concern, but if the procedure is *reentrant*, the callee faces precisely the same predicaments.

At some point, therefore, we should consider whether any of this fuss is worthwhile. Instead, callers who want the callee to perform a mutation could simply send a boxed value to the callee. The box signals that the caller accepts—indeed, invites—the callee to perform a mutation, and the caller can extract the value when it's done. This does obviate the ability to write a simple swapper, but that's a small price to pay for genuine software engineering concerns.

9 Recursion and Cycles: Procedures and Data

Recursion is the act of self-reference. When we speak of recursion in programming languages, we may have one of (at least) two meanings in mind: recursion in data, and recursion in control (i.e., of program behavior—that is to say, of functions).

9.1 Recursive and Cyclic Data

Recursion in data can refer to one of two things. It can mean referring to something of the same *kind*, or referring to the same *thing* itself.

Recursion of the same kind leads to what we traditionally call *recursive data*. For instance, a tree is a recursive data structure: each vertex can have multiple children, each of which is itself a tree. But if we write a procedure to traverse the nodes of a tree, we expect it to terminate without having to keep track of which nodes it has already visited. They are finite data structures.

In contrast, a graph is often a *cyclic* datum: a node refers to another node, which may refer back to the original one. (Or, for that matter, a node may refer directly to itself.) When we traverse a graph, absent any explicit checks for what we have already visited, we should expect a computation to *diverge*, i.e., not terminate. Instead, graph algorithms need a memory of what they have visited to avoid repeating traversals.

Adding recursive data, such as lists and trees, to our language is quite straightforward. We mainly require two things:

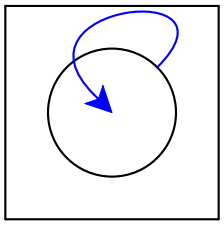
1. The ability to create compound structures (such as nodes that have references to children).

2. The ability to bottom-out the recursion (such as leaves).

Exercise

> Add lists and binary trees as built-in datatypes to the programming language.

Adding cyclic data is more subtle. Consider the simplest form of cyclic datum, a cell referring back to itself:



Let's try to define this in Racket. Here's one attempt:

```
(let ([b b])
  b)
```

But this doesn't work: `b` on the right-hand side of the `let` isn't bound. It's easy to see if we desugar it:

```
((lambda (b)
   b)
  b)
```

and, for clarity, we can rename the `b` in the function:

```
((lambda (x)
   x)
 b)
```

Now it's patently clear that `b` is unbound.

Absent some magical Racket construct we haven't yet seen, it becomes clear that we can't create a cyclic datum in one shot. Instead, we need to first create a "place" for the datum, then refer to that place within itself. The use of "then"—i.e., the introduction of time—should suggest a mutation operation. Indeed, let's try it with boxes.

That construct would be `shared`, but virtually no other language has this notational mechanism, so we won't dwell on it here. In fact, what we are studying is the main idea behind how `shared` actually works.

Our plan is as follows. First, we want to create a box and bind it to some identifier, say `b`. Now, we want to mutate the content of the box. What do we want it to contain? A reference to itself. How does it obtain that reference? By using the name, `b`, that is already bound to it. In this way, the mutation creates the cycle:

```
(let ([b (box 'dummy)])
  (begin
    (set-box! b b)
    b))
```

Note that this program will *not* run in Typed PLAI as written. We'll return to typing such programs later [REF]. For now, run it in the untyped (`#lang plai`) language.

When the above program is Run, Racket prints this as: `#0='#&#0#`. This notation is in fact precisely what we want. Recall that `#&` is how Racket prints boxes. The `#0=` (and similarly for other numbers) is how Racket names pieces of cyclic data. Thus, Racket is saying, "`#0` is bound to a box whose content is `#0#`, i.e., whatever is bound to `#0`, i.e., itself".

Exercise

Run the equivalent program through your interpreter for boxes and make sure it produces a *cyclic* value. How do you check this?

The idea above generalizes to other datatypes. In this same way we can also produce cyclic lists, graphs, and so on. The central idea is this two-step process: first name an vacant placeholder; then mutate the placeholder so its content is itself; to obtain "itself", use the name previously bound. Of course, we need not be limited to "self-cycles": we can also have mutually-cyclic data (where no one element is cyclic but their combination is).

9.2 Recursive Functions

In a shift in terminology, a recursive function is not a reference to a same *kind* of function but rather to the same function *itself*. It's useful to first ensure we've first extended our language with conditionals (even of the kind that only check for `0`, as described earlier: section 5), so we can write non-trivial programs that terminate.

Let's now try to write a recursive factorial:

```
(let ([fact (lambda (n)
              (if0 n
                   1
                   (* n (fact (- n 1)))))])
  (fact 10))
```

But this doesn't work at all! The inner `fact` gives an unbound identifier error, just as in our cyclic datum example.

It is no surprise that we should encounter the same error, because it has the same cause. Our traditional binding mechanism does not automatically make function definitions cyclic (indeed, in some early programming languages, they were not: misguidedly, recursion was considered a special *feature*). Instead, if we want recursion—i.e., for a function definition to cyclically refer to itself—we must implement it by hand.

The means to do so is now clear: the problem is the same one we diagnosed before, so we can reuse the same solution. We again have to follow a three-step process: first create a placeholder, then refer to the placeholder where we want the cyclic reference, and finally mutate the placeholder before use. Thus:

```
(let ([fact (box 'dummy)])
  (let ([fact-fun
         (lambda (n)
           (if (zero? n)
               1
               (* n ((unbox fact) (- n 1)))))])
    (begin
      (set-box! fact fact-fun)
      ((unbox fact) 10))))
```

Because you typically write *top-level* definitions, you don't encounter this issue. At the top-level, every binding is implicitly a variable or a box. As a result, the pattern below is more-or-less automatically put in place for you. This is why, when you want a recursive local binding, you must use `letrec` or `local`, not `let`.

In fact, we don't even need `fact-fun`: I've used that binding just for clarity. Observe that because it isn't recursive, and we have identifiers rather than variables, its use can simply be substituted with its value:

```
(let ([fact (box 'dummy)])
  (begin
    (set-box! fact
              (lambda (n)
                (if (zero? n)
                    1
                    (* n ((unbox fact) (- n 1))))))
    ((unbox fact) 10)))
```

There is the small nuisance of having to repeatedly unbox `fact`. In a language with variables, this would be even more seamless:

```
(let ([fact 'dummy])
    (begin
      (set! fact
            (lambda (n)
              (if (zero? n)
                  1
                  (* n (fact (- n 1))))))
      (fact 10)))
```

Indeed, one use for variables is that they simplify the desguaring of the above pattern, instead of requiring every use of a cyclically-bound identifier to be unboxed. On the other hand, with a little extra effort the desugaring process could take care of doing the unboxing, too.

9.3 Premature Observation

Our preceding discussion of this pattern shows a clear temporal sequencing: create, update, use. We can capture it in a desugaring rule. Suppose we add the following new syntax:

```
(rec name value body)
```

As an example of its use,

```
(rec fact
     (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)))))
     (fact 10))
```

would evaluate to the factorial of `10`. This new syntax would desugar to:

```
(let ([name (box 'dummy)])
  (begin
    (set-box! name value)
    body))
```

Where we assume that all references to `name` in `value` and `body` have been rewritten to `(unbox name)`, or alternatively that we instead use variables:

```
(let ([name 'dummy])
  (begin
    (set! name value)
    body))
```

This naturally inspires a question: what if we get these out of order? Most interestingly, what if we try to use `name` before we're done updating its true value into place? Then we observe the state of the system right after creation, i.e., we can see the placeholder in its raw form.

The simplest example that demonstrates this is as follows:

```
(letrec ([x x])
  x)
```

or equivalently,

```
(local ([define x x])
  x)
```

In most Racket variants, this *leaks* the initial value given to the placeholder—a value that was never meant for public consumption. This is troubling because it is, after all, a legitimate *value*, which means it can probably be used in at least some computations. If a developer accesses and uses it inadvertently, however, they are effectively computing with nonsense.

There are generally three solutions to this problem:

1. Make sure the value is sufficiently obscure so that it can never be used in a meaningful context. This means values like `0` are especially bad, and indeed most common datatypes should be shunned. Instead, the language might create a new type of value just for use here. Passed to any other operation, this will result in an error.

2. Explicitly check every use of an identifier for belonging to this special "premature" value. While this is technically feasible, it imposes an enormous performance penalty on a program. Thus, it is usually only employed in teaching languages.

3. Allow the recursion constructor to be used only in the case of binding functions, and then make sure that the right-hand side of the binding is *syntactically* a function. Unfortunately, this solution can be a bit drastic because it precludes writing, for instance, structures to create graphs.

9.4 Without Explicit State

As you may be aware, there is another way to define recursive functions (and hence recursive data) that does not leverage explicit mutation operations.

Do Now!