1 Writing a Recursive Function

Can we write the procedure factorial in Rip? We currently don’t have a way of making choices in our Rip code. Let’s add a simple construct called if0 which decides which of two expressions to evaluate based on whether or not its first argument is 0. For example,

{if0 {+ 5 -5} 1 2}

returns 1. Now we’re ready to write factorial:

{let {fac {proc {n}  {if0 n 1 1 {n {fac {+ n -1}}}}}  {fac 5}}}

What does this return? 120? No. Recall the example {let {x x} x} in which the second x is unbound. Similarly, rip is unbound on the RHS of the let. When we apply fac to 5, fac is bound to a procedure. But when we try to evaluate the expression {fac {+ n -1}}, we get an unbound identifier error.

To make this clearer, consider the environments available to each part of the Rip program. The initial environment is empty. The procedure declaration inherits this environment (due to the semantics of let), so it closes over an empty environment. Only the invocation {fac 5} sees a binding for fac.

In short, let seems insufficient for constructing recursive functions. We may yet try more complex approaches using procedures (and indeed we will, in the coming weeks), but for now we could take an easy way out: add recursion explicitly to the language.

2 A Construct for Recursion

We extend the data definition for Rip programs:
A Rip Program (RP) is either

- everything it was before
- \((\text{make-recE} \; \text{var-name} \; \text{RP} \; \text{RP})\)

with the corresponding structure definition

\(\text{(define-struct recE (var-name val body))}\)

with the corresponding concrete syntax

\({\text{rec \{\text{var val} \\text{body}}\})\)

Using this, we should be able to write factorial as follows:

\(\{\text{rec \{\text{fac \{\text{proc \{n}}\} \\{\text{if0} \; \text{n} \\{1 \{\* \; \text{n \{\text{fac \{\* \; n \; -1\}}}}\}}}\} \{\text{fac 5}}\}\}

3 Environments for Recursion

Merely writing the syntax of a new term isn’t enough to specify its meaning, much less implement it. To understand recursion, let’s consider what we really need. Recall that we explained the problem in terms of environments, above. It seemed that if we could just get the right environment in place when creating the closure, the interpretation of \text{fac} inside the procedure’s body would be recursive. So let’s reduce the difficult problem of defining recursive procedures to the (hopefully) somewhat simpler task of defining recursive environments. Environments are data structures—more concrete entities—so maybe this will be easier.

The essence of our problem is the following. We can associate, with constructs like let and rec, an environment transformer (something that consumes an existing environment and produces a new one), which builds new environments from old ones. The right environment transformer captures the essence of the construct by describing its scope. What is the environment transformer for the failed attempt at defining \text{fac}?

\(\rho_{\text{let}}(E_0) = (\text{extend-env} \ E_0 \ \text{'fac} \ (\text{make-closure} \ (\text{procE}) \ E_0))\)

That is, we begin with some ambient environment, \(E_0\), which is the one active at the beginning of the expression. We extend this environment with a binding for the Rip variable \text{fac}, represented here by the Scheme symbol \text{'fac}. We bind it to a closure, which contains some procedure’s source, as well as the environment.
it closes over. In the case of \texttt{let}, this environment would be the same as the ambient environment, namely $E_0$.

Now let’s examine \texttt{rec}. We expect that

\[
\rho_{\text{rec}}(E_0) = (\text{extend-env } E_0 \\
\text{\texttt{\textbackslash{f}ac}} \\
(\text{make-closure (procE) } E_1))
\]

Clearly, $E_1$ can’t be the same as $E_0$. But it’s hard to work with a function that has free variables in it. We can, however, bind this environment easily enough. We’ll just make $\rho_{\text{rec}}$ return a function:

\[
\rho_{\text{rec}}(E_0) = \lambda E_1. (\text{extend-env } E_0 \\
\text{\texttt{\textbackslash{f}ac}} \\
(\text{make-closure (procE) } E_1))
\]

($\lambda E_1, \cdots$ signifies a function of one argument.) Let’s set $F_\rho = \rho_{\text{rec}}(E_0)$ for some ambient $E_0$.

Let’s study $F_\rho$. This is a function ready to consume $E_1$, the environment we put in the closure. What do we know about $E_1$? We want it to have \texttt{fac} bound (so we can look up \texttt{fac} in the body of the \texttt{rec} expression). We want the bound value to be a closure (so we can apply it). The closure’s environment needs to have \texttt{fac} bound (so we can refer to \texttt{fac} inside the procedure’s body). The closure’s environment should extend $E_0$ (because we don’t want any other names than \texttt{fac} and those bound in $E_0$ bound in the body of the closure). In particular, \texttt{fac} must be bound to a closure (so we can apply it inside the procedure’s body). This closure’s environment must have . . .

In short, the environment $E_1$ we need to feed $F_\rho$ needs to be the same as the environment we will get from applying $F_\rho$ to it. This looks pretty tricky: we’re being asked to pass in the very environment we want to get back out! Strange as it may seem, it’s not an unreasonable request. We call such a value a \textit{fixed-point} of a function.

Let’s summarize what we’ve learned. We want $E_1 = F_\rho(E_1)$. In particular, the right-hand side of this equality has the left-hand side buried deep within it. In short, $E_1$ contains a reference to itself: it’s \textit{cyclic}.

\section*{An Aside on Recursion}

Note that we’ve seen two kinds of recursion in this course. We’ve seen and written lots of recursive \textit{functions}, not least of all \textit{interp}. We’ve also seen how these functions arise: by following the structure of the data. That is, they processed instances of data definitions that referred to themselves, such as the extended definition of Rip programs we saw above.

What we’re studying now is a special kind of recursive datum: it refers not only to other data \textit{like} it, it actually refers to \textit{itself}. In other words, it is not
merely recursive, it’s actually cyclic. Every cyclic datum must have a recursive
data definition, but not every recursive data definition results in cyclic data (for
instance, consider a tree, or even the representations of Rip programs).

An Aside on Fixed-Points

You’ve already seen fixed points in mathematics. For example, consider func-
tions over the real numbers. The function \( f(x) = 0 \) has a fixed point at 0.
Functions do not always have a fixed point, as in \( f(x) = x + 1 \). The fixed point
is simply a point where the graph of the function intersects the line \( y = x \). From
this, we can immediately see that a function can have more than one fixed point:
\( f(x) = x \) has infinitely many fixed points.

4 Implementing Recursion

We have now reduced the problem of creating recursive functions to that of
creating circular environments. We can express this in our interpreter by simply
writing
\[
[(\text{recE} \ expr) \ (\text{interp} \ (\text{recE-body} \ expr)) \ (\text{circularly-bind} \ env \ (\text{recE-var-name} \ expr) \ (\text{recE-val} \ expr))]
\]

where the function \text{circularly-bind} is responsible for creating circular environ-
ments.

The implementation of \text{circularly-bind} depends on our implementation of
environments. Let’s first consider our implementation of environments as pro-
cedures:

\[
(\text{define} \ (\text{circularly-bind} \ env \ var-name \ valE) \\
(\text{let} \ ((\text{new-env}) \ (\lambda \ (\text{want}) \ (\text{cond} \ ((\text{symbol=?} \ (\text{varE-name} \ want) \ var-name) \ (\text{interp} \ valE ??)) \ [\text{else} \ (\text{lookup-env} \ want \ env)]))) \text{new-env}))))
\]

We have essentially copied the \text{extend-env} code. Which environment do we fill
in for the question marks? We want to interpret \text{valE} in the environment that
we are creating. Thus, the question marks should be replaced by \text{new-env}.

There’s a problem here. We are trying to define a function \text{new-env} that
refers to itself. This is the problem we are trying to solve in Rip! Fortunately,
Scheme gives us a way to declare recursive procedures: \text{letrec}. The code be-
comes
(define (circularly-bind env var-name valE)
  (letrec ([new-env (lambda (want)
            (cond
               [(symbol=? (var-name want) var-name)
                (interp valE new-env)]
               [else
                (lookup-env want env)])
            new-env)])

This is somewhat unsatisfying. We have learned that the problem of creating recursive procedures reduces to creating circular environments, but just using Scheme’s letrec does not tell us how we actually create such an environment.

Let’s consider our implementation of environments as a list of structures. Before we can create a circular environment, we must first extend it with the new variable. We don’t yet know what it will be bound to, so we stick a dummy value into the environment:

(define (circularly-bind env var-name valE)
  (let ([new-env (cons (make-binding var-name 'undefined-identifier) env)])

Now that we have this extended environment, we can interpret the RHS expression in it:

(define (circularly-bind env var-name valE)
  (let ([new-env (cons (make-binding var-name 'undefined-identifier) env)])
    (let ([valV (interp valE new-env)])

This has the happy benefit that if the RHS expression is a closure, it will close over the extended environment. Notice that this environment is half-right and half-wrong: it has the right names bound, but the newest addition is bound to the wrong value (indeed, it is bound to something that isn’t a Rip value at all!).

Now comes the critical step. Recall that the value we get from evaluating the RHS is the same value we want to get on all subsequent references to the name contained in var-name in the environment new-env. Therefore, the dummy value ought to be replaced with the new value:

(define (circularly-bind env var-name valE)
  (let ([new-env (cons (make-binding var-name 'undefined-identifier) env)])
    (let ([valV (interp valE new-env)])
      (begin
        (set-binding-val! (first new-env) valV)
        new-env))))

Since any closures in the value expression share the same binding, they automatically avail of this update. As a result, the environment that circularly-bind returns has the correct bindings.
Looking at this definition, you can see that there is a period of time when
the variable has `undefined-identifier` as a value. It's not until the `set-binding-val` call that the variable gets its correct value. We just temporarily leave the
environment in an unstable state and hope that nobody notices.

A Hazard

When programming with `rec` (in Rip) or `letrec` (in Scheme), we have to be extremely careful about using values before their time. Specifically, consider this Rip program:

```
{rec {f f}
   f}
```

The problem with this program is that it violates precisely the hope that nobody will notice the illegal value in the environment while evaluating the RHS expression.

What should this evaluate to? The `f` in the body has whatever value the `f` did on the RHS of the `rec`—whose value is unclear, because we’re in the midst of a recursive definition. What you get really depends on the implementation strategy. An implementation could give you some sort of strange, internal value, referring to the initial value in the interpreter (where we use `undefined` above). It could even result in an infinite loop, as `f` tries to look up the definition of `f`, which depends on the definition of `f`, which . . .

There is a (excessively) safe way of avoiding this pickle. The problem arises because the RHS expression tries to compute a value. This wouldn’t happen if that expression already were a value itself. The values in Rip are currently numbers and closures. Therefore, if we guarantee that the RHS expressions were all `syntactically` values (that is, they don’t require any evaluation to reduce to a value), we would be safe: all evaluation would happen only in the body, by which time `circularly-bind` would be done producing its circular environment.