Chapter 9

Understanding Recursion

Can we write the factorial function in FAE? We currently don't have subtraction or multiplication, or a way of making choices in our FAE code. But those two are easy to address. At this point adding subtraction and multiplication is trivial, while to make choices, we can add a simple conditional construct, leading to this language:

```
<CFAE> ::= <num>
         | {+ <CFAE> <CFAE>}
         | {- <CFAE> <CFAE>}
         | {* <CFAE> <CFAE>}
         | <id>
         | {fun {<id>} <CFAE>}
         | {<CFAE> <CFAE>}
         | {if0 <CFAE> <CFAE> <CFAE>}
```

An `if0` evaluates its first sub-expression. If this yields the value `0` it evaluates the second, otherwise it evaluates the third. For example,

```
{if0 {+ 5 -5}
     1
     2}
```

evaluates to 1.

Given CFAE, we're ready to write factorial (recall from Section 6.3 that `with` can be handled by a pre-processor or by the parser):

```
{with {fac {fun {n}
             {if0 n
                  1
                  {* n {fac {+ n -1}}}}}}
  {fac 5}}
```

What does this evaluate to? 120? No. Consider the following simpler expression, which you were asked to contemplate when we studied substitution:

```
{with {x x} x}
```

In this program, the `x` in the named expression position of the `with` has no binding. Similarly, the environment in the closure bound to `fac` binds no identifiers. Therefore, only the environment of the first invocation (the body of the `with`) has a binding for `fac`. When `fac` is applied to `5`, the interpreter evaluates the body of `fac` in the closure environment, which has no bound identifiers, so interpretation stops on the intended recursive call to `fac` with an unbound identifier error. (As an aside, notice that this problem disappears with *dynamic* scope! This is why dynamic scope persisted for as long as it did.)

Before you continue reading, please pause for a moment, study the program carefully, write down the environments at each stage, step by hand through the interpreter, even run the program if you wish, to convince yourself that this error will occur. Understanding the error thoroughly will be essential to following the remainder of theis section.

9.1 A Recursion Construct

It's clear that the problem arises from the scope rules of `with`: it makes the new binding available only in its body. In contrast, we need a construct that will make the new binding available to the named expression also. Different intents, so different names: Rather than change `with`, let's add a new construct to our language, `rec`.

```
<RCFAE> ::= <num>
          | {+ <RCFAE> <RCFAE>}
          | {- <RCFAE> <RCFAE>}
          | {* <RCFAE> <RCFAE>}
          | <id>
          | {fun {<id>} <RCFAE>}
          | {<RCFAE> <RCFAE>}
          | {if0 <RCFAE> <RCFAE> <RCFAE>}
          | {rec {<id> <RCFAE>} <RCFAE>}
```

RCFAE is CFAE extended with a construct for recursive binding. We can use `rec` to write a description of factorial as follows:

```
{rec {fac {fun {n}
            {if0 n
                 1
                 {* n {fac {+ n -1}}}}}}
  {fac 5}}
```

Simply defining a new syntactic construct isn't enough; we must also describe what it means. Indeed, notice that syntactically, there is nothing but the keyword distinguishing `with` from `rec`. The interesting work lies in the interpreter. But before we get there, we'll first need to think hard about the semantics at a more abstract level.

9.2 Environments for Recursion

It's clear from the analysis of our failed attempt at writing `fac` using `with` that the problem has something to do with environments. Let's try to make this intuition more precise.

One way to think about constructs such as `with` is as *environment transformers*. That is, they are functions that consume an environment and transform it into one for each of their sub-expressions. We will call the environment they consume—which is the one active outside the use of the construct—the *ambient* environment.

There are two transformers associated with `with`: one for its named expression, and the other for its body. Let's write them both explicitly.

$$\rho_{\texttt{with},named}(e) = e$$

In other words, whatever the ambient environment for a `with`, that's the environment used for the named expression. In contrast,

$$\rho_{\texttt{with},body}(e) = (\mathsf{aSub}\ \textit{bound-id}\ \textit{bound-value}\ e)$$

where *bound-id* and *bound-value* have to be replaced with the corresponding identifier name and value, respectively.

Now let's try to construct the intended transformers for `rec` in the factorial definition above. Since `rec` has two sub-expressions, just like `with`, we will need to describe two transformers. The body seems easier to tackle, so let's try it first. At first blush, we might assume that the body transformer is the same as it was for `with`, so:

$$\rho_{\texttt{rec},body}(e) = (\mathsf{aSub}\ \text{'}\mathsf{fac}\ (\mathsf{closureV} \cdots)\ e)$$

Actually, we should be a bit more specific than that: we must specify the environment contained in the closure. Once again, if we had a `with` instead of a `rec`, the closure would close over the ambient environment:

```
ρ_rec,body(e) = (aSub 'fac
                      (closureV 'n        ;; bound id
                                (if0 ···)  ;; body
                                e)
                      e)
```

But this is no good! When the `fac` procedure is invoked, the interpreter is going to evaluate its body in the environment bound to *e*, which doesn't have a binding for `fac`. So this environment is only good for the first invocation of `fac`; it leads to an error on subsequent invocations.

Let's understand how this closure came to have that environment. The closure simply closes over whatever environment was active at the point of the procedure's definition. Therefore, the real problem is making sure we have the right environment for the named-expression portion of the `rec`. If we can do that, then the procedure in that position would close over the right environment, and everything would be set up right when we get to the body.

We must therefore shift our attention to the environment transformer for the named expression. If we evaluate an invocation of `fac` in the ambient environment, it fails immediately because `fac` isn't bound. If we evaluate it in (aSub 'fac (closureV $\cdots$ *e*) *e*), we can perform one function application before we halt with an error. What if we wanted to be able to perform two function calls (i.e., one to initiate the computation, and one recursive call)? Then the following environment would suffice:

```
ρ_rec,named(e) =
  (aSub 'fac
        (closureV 'n
                  (if0 ···) ;; body
                  (aSub 'fac
                        (closureV 'n
                                  (if0 ···) ;; body
                                  e)
                        e))
        e)
```

That is, when the body of `fac` begins to evaluate, it does so in the environment

```
(aSub 'fac
      (closureV 'n
                (if0 ···) ;; body
                e)
      e)
```

which contains the "seed" for one more invocation of `fac`. That second invocation, however, evaluates its body in the environment bound to *e*, which has no bindings for `fac`, so any further invocations would halt with an error.

Let's try this one more time: the following environment will suffice for one initial and *two* recursive invocations of `fac`:

```
ρ_rec,named(e) =
  (aSub 'fac
        (closureV 'n
                  (if0 ···) ;; body
                  (aSub 'fac
                        (closureV 'n
                                  (if0 ···) ;; body
                                  (aSub 'fac
                                        (closureV 'n
                                                  (if0 ···) ;; body
                                                  e)
                                        e))
                        e))
        e)
```

There's a pattern forming here. To get true recursion, we need to not "bottom out", which happens when we run out of extended environments. This would not happen if the "bottom-most" environment were somehow to refer back to the one enclosing it. If it could do so, we wouldn't even need to go to three levels; we only need one level of environment extension: the place of the boxed *e* in

$\rho_{\texttt{rec},named}(e) =$
```
(aSub 'fac
      (closureV 'n
                (if0 ···) ;; body
                [e])
      e)
```

should instead be a reference to the entire right-hand side of that definition. The environment must be a *cyclic* data structure, or one that refers back to itself.

We don't seem to have an easy way to represent this environment transformer, because we can't formally just draw an arrow from the box to the right-hand side. However, in such cases we can use a variable to name the box's location, and specify the constraint externally (that is, once again, name and conquer). Concretely, we can write

$\rho_{\texttt{rec},named}(e) =$
```
(aSub 'fac
      (closureV 'n
                (if0 ···) ;; body
                E)
      e)
```

But this has introduced an unbound (free) identifier, E. It's easy to make this bound, by introducing a new (helper) function:

$\rho'(e) =$
```
λ E .
  (aSub 'fac
        (closureV 'n
                  (if0 ···) ;; body
                  E)
        e)
```

We'll call this function, ρ', a *pre-transformer*, because it consumes both *e*, the ambient environment, and E, the environment to put in the closure. For some ambient environment e_0, let's set

$$F_{e_0} = \rho'(e_0)$$

Observe that F_{e_0} is a procedure ready to consume an E, the environment to put in the closure. What does it return? It returns an environment that extends the ambient environment. If we feed the right environment for E, then recursion can proceed forever. What E will enable this?

Whatever we get by feeding some E_0 to F_{e_0}—that is, $F_{e_0}(E_0)$—is precisely the environment that will be bound in the closure in the named expression of the `rec`, by definition of F_{e_0}. We also want that the environment we get back one that extends the ambient environment with a suitable binding for `fac`, i.e., the *same* environment. In short, we want

$$E_0 = F_{e_0}(E_0)$$

That is, the environment E_0 we need to feed F_{e_0} needs to be the same as the environment we will get from applying F_{e_0} to it. We're being asked to supply the very answer we want to produce!

We call such a value—one such that the function's output is the same as its input—a *fixed-point* of a function. In this particular case, the fixed-point of ρ' is an environment that extends the ambient environment with a binding of a name to a closure whose environment is . . . itself.

Exercise 9.2.1 *This discussion about recursion has taken place in the context of environments, i.e., deferred substitutions. How would it differ if we were performing explicit substitution (i.e., without deferral)?*

Fixed-Points

Consider functions over the real numbers. The function $f(x) = 0$ has exactly one fixed point, because $f(n) = n$ only when $n = 0$. But not all functions over the reals have fixed points: consider $f(x) = x + 1$. A function can have two fixed points: $f(x) = x^2$ has fixed points at 0 and 1 (but not, say, at -1). And because a fixed point occurs whenever the graph of the function intersects the line $y = x$, the function $f(x) = x$ has infinitely many fixed points.

The study of fixed points over topological spaces is fascinating, and yields many rich and surprising theorems. One of these is the Brouwer fixed point theorem. The theorem says that every continuous function from the unit *n*-ball to itself must have a fixed point. A famous consequence of this theorem is the following result. Take two instances of the same map, align them, and lay them flat on a table. Now crumple the upper copy, and lay it atop the smooth map any way you like (but entirely fitting within it). No matter how you place it, at least one point of the crumpled map lies directly above its equivalent point on the smooth map!

The mathematics we must use to carefully define this fixed-point is not trivial. Fortunately for us, we're using programming instead of mathematics! In the world of programming, the solution will be to generate a *cyclic* environment.

Recursiveness and Cyclicity

It is important to distinguish between *recursive* and *cyclic* data. A recursive object contains references to instances of objects of the same kind as it. A cyclic object doesn't just contains references to objects of the same *kind* as itself: it contains references to *itself*.

To easily recall the distinction, think of a typical family tree as a canonical recursive structure. Each person in the tree refers to two more family trees, one each representing the lineage of their mother and father. However, nobody is (usually) their own ancestor, so a family tree is never cyclic. Therefore, structural recursion over a family tree will always terminate. In contrast, the Web is not merely recursive, it's cyclic: a Web page can refer to another page which can refer back to the first one (or, a page can refer to itself). Naïve recursion over a cyclic datum will potentially not terminate: the recursor needs to either not try traversing the entire object, or must track which nodes it has already visited and accordingly prune its traversal. Web search engines face the challenge of doing this efficiently.

9.3 An Environmental Hazard

When programming with `rec`, we have to be extremely careful to avoid using bound values prematurely. Specifically, consider this program:

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

What should this evaluate to? The `f` in the body has whatever value the `f` did in the named expression of the `rec`—whose value is unclear, because we're in the midst of a recursive definition. An implementation could give you an internal value that indicates that a recursive definition is in progress; it could even go into an infinite loop, as `f` tries to look up the definition of `f`, which depends on the definition of `f`, which. . . .

There is a safe way out of this pickle. The problem arises because the named expression can be any complex expression, including the identifier we are trying to bind. But recall that we went to all this trouble to create recursive *procedures*. If we merely wanted to bind, say, a number, we have no need to write

```
{rec {n 5}
  {+ n 10}}
```

when we could write

```
{with {n 5}
  {+ n 10}}
```

just as well instead. Therefore, instead of the liberal syntax for RCFAE above, we could use a more conservative syntax that restricts the named expression in a `rec` to *syntactically* be a procedure (i.e., the programmer may only write named expressions of the form `{proc ...}`). Then, interpretation of the named expression immediately constructs a closure, and the closure can't be applied until we interpret the body—by which time the environment is in a stable state.

Exercise 9.3.1 *Are there any other expressions we can allow, beyond just syntactic procedures, that would not compromise the safety of this conservative recursion regime?*

Exercise 9.3.2 *Can you write a useful or reasonable program that is permitted in the liberal syntax, and that safely evaluates to a legitimate value, that the conservative syntax prevents?*