The next feature we consider is recursion. One way to get recursion is through a self-application. In the \(\lambda\)-calculus, the self-application of the self-applier yields an infinite loop:

\[
(\lambda x.(xx))(\lambda x.(xx)) \rightarrow (\lambda x.(xx))(\lambda x.(xx)) \rightarrow \ldots
\]

Similarly, we could write the factorial function with a self-application like this:

\[
(let ((fac2
  (lambda (n p)
    (if (eq? n 0)
      1
      (* n (p (- n 1) p))))))
(fac2 5 fac2))
\]

A different approach uses assignment to make a closure refer to itself:

\[
(let ((ref (lambda (x) (error "don’t call me"))))
(let ((fac
  (lambda (n)
    (if (eq? n 0)
      1
      (* n (ref (- n 1))))))
(set! ref fac)
(fac 5))
\]

To add this recursion to the interpreted language, we need an additional variant \(\text{letrec-exp}\) in the abstract syntax. The interpreter is extended with the following clause to handle this case:

\[
(\text{letrec-exp} (\text{proc-names idss bodies letrec-body})
  (\text{eval-expression} \text{letrec-body}
    (\text{extend-env-recursively} \text{proc-names idss bodies env})))
\]

The crucial ingredient here is the procedure \(\text{extend-env-recursively}\).

For example, the closure created as the binding for \(\text{fac}\) should contain an environment that maps the variable \(\text{fac}\) to that closure (so that the procedure can call itself recursively). If one draws the environments and closures, then it becomes clear that we have a circular data structure.