

Principles of Programming Languages Handout

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

$$(\lambda x.(xx)) (\lambda x.(xx)) \rightarrow (\lambda x.(xx)) (\lambda x.(xx)) \rightarrow \dots$$

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 `letrec-exp` in the abstract syntax. The interpreter is extended with the following clause to handle this case:

```
(letrec-exp (proc-names idss bodies letrec-body)
  (eval-expression letrec-body
    (extend-env-recursively proc-names idss bodies env)))
```

The crucial ingredient here is the procedure `extend-env-recursively`.

For example, the closure created as the binding for `fac` should contain an environment that maps the variable `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.