Recursion Revisited

- ⟨⟨PLAI Chapter 9⟩⟩
- `with` is too weak to do recursion as the name is **not** bound in the named expression. This fails because one of the `x`'s is **free**:

```
{with {x {+ x 1}} x}
```

- Similarly, in Racket this won't work if 'let' (or let*) is the only tool you use to create bindings:

2

```
(let ([fact (lambda (n)
              (if (zero? n) 1
                  (* n (fact (- n 1)))))])
  (fact 5))
```

- the above 'fact' definition is similar to writing:

3

```
fact := (lambda (n)
          (if (zero? n) 1
              (* n (fact (- n 1)))))
(fact 5)
```

- which is not a well-formed definition. What we'd really want, is to **solve** the **equation** (using '=' instead of ':=')

4
```
fact = (lambda (n)
         (if (zero? n) 1
             (* n (fact (- n 1)))))
```

- We know that Racket's `define` and `letrec` do allow us to implement recursion
- We saw that it was relatively easy to implement something like `letrec` (we called it `rec` in lecture 9) using **mutation**.
- A natural question is if recursion somehow requires mutation. Among other things, that would mean we can no longer claim the semantics are defined by substitution.

Recursion without Mutation

- The Little Schemer:
  http://www.ccs.neu.edu/home/matthias/BTLS/sample.pdf
- "The Why of Y", by Richard Gabriel.
  http://www.drdobbs.com/web-development/the-why-of-y/199200394
- To start, note that in a dynamically-scoped language `let` is enough to define recursive functions

```
5   (let ([fact (lambda (n)
                  (if (zero? n) 1
                      (* n (fact (- n 1)))))])
      (fact 5))
```

- By the time we evaluate the body of the function, ‘fact’ is already bound to itself in the current dynamic scope.
- In the lexical scope case when ‘fact’ is called, it does have a value, but the binding is inaccessible.
- We can make the calling scope value available by passing a parameter. The following *almost* works

6

```
(define (fact self n)  ;***
  (if (zero? n) 1 (* n (self (- n 1)))))
(fact fact 5)          ;***
```

2019-03-11

- By the time we evaluate the body of the function, ‘fact’ is already bound to itself in the current dynamic scope.
- In the lexical scope case when ‘fact’ is called, it does have a value, but the binding is inaccessible.
- We can make the calling scope value available by passing a parameter. The following *almost* works

6

```
(define (fact self n)  ;***
  (if (zero? n) 1 (* n (self (- n 1)))))
(fact fact 5)          ;***
```

1. This is another reason why dynamic scope is perceived as a convenient approach in new languages.

- except that now the recursive call should still send itself along:

```
(let ([fact (lambda (self n)
              (if (zero? n) 1
                  (* n (self self
                             (- n 1)))))])
  (fact fact 5))
```

- In some sense, we are done. We have recursion, using only `let`, and could do the same in FLANG
- The problem with the parameter passing method is that it requires changing the calls.

- Eventually, we should be able to get a working `fact` definition that uses just

8

```
(lambda (n) (if (zero? n) 1
                (* n (fact (- n 1)))))
```

- The first step is to curry the `fact` definition.

9

```
(let [(fact
       (lambda (self)
         (lambda (n)
           (if (zero? n)
               1
               (* n ((self self) (- n 1)))))))]
  ((fact fact) 5))
```

- ‘fact’ is a function that constructs constructs the real factorial function.
- So call it ‘make-fact’, and bind ‘fact’ to the actual factorial function.

10

```
(let* [(make-fact
         (lambda (self)
           (lambda (n)
             (if (zero? n)
                 1
                 (* n ((self self) (- n 1)))))))
        (fact (make-fact make-fact))]
(fact 5))
```

We can try to do the same thing in the body of the factorial function: instead of calling (self self), just bind 'fact' to it:

11

```
(let* ([make-fact
        (lambda (self)
          (lambda (n)
            (let ([fact (self self)])
              (if (zero? n)
                  1
                  (* n (fact (- n 1)))))))]
       [fact (make-fact make-fact)])
  (fact 5))
```

This works fine, but if we consider our original goal, we need to get that local ‘fact’ binding outside of the (lambda (n) ...)

12

```
(let* ([make-fact
        (lambda (self)
          (let ([fact (self self)])  ;***
            (lambda (n)              ;***
              (if (zero? n) 1
                  (* n (fact (- n 1)))))))]
       [fact (make-fact make-fact)])
  (fact 5))
```

- This *looks* good, but loops infinitely.

- This *looks* good, but loops infinitely.

1. Things might be different in a lazy language, but here we call '(self self)' when binding fact

- In essence our last `make-fact` works something like

13

```
(let ([make-fact (lambda (self) (self self))])
  (make-fact make-fact))

--replace-definition-->
((lambda (self) (self self))
 (lambda (self) (self self)))

--rename-identifiers-->
((lambda (x) (x x)) (lambda (x) (x x)))
```

- this expression (related to the "Y combinator") has an interesting property: it reduces to itself, so evaluating it gets stuck in an infinite loop.
- `(self self)` **should** be a one argument function, we just need to delay evaluation until we have an argument.
- Consider the example `(lambda (n)(add1 n))`
- This is really the same function as 'add1'; there is a delay from lambda, but it doesn't seem to matter because add1 is always defined.
- The difference in `make-fact` is that there is a function-call in the function position, so the delay is important

```
(lambda (n) ((self self) n))
```

- Applying this trick to our code produces a version that does not get stuck in an infinite loop:

15

```
(let* ([make-fact
        (lambda (self)
          (let ([fact
                 (lambda (n) ((self self) n))])
            (lambda (n)
              (if (zero? n)
                  1
                  (* n (fact (- n
                      1))))))))]
       [fact (make-fact make-fact)])
  (fact 5))
```

- Again, if you followed this far, congratulations. We have a recursive function with a nice definition (the inner lambda).
- Still, it would be be nice to have more than one recursive function.
- First, we need to figure out how to make this reusable, then we'll worry about making it prettier.

- We know that

  16 `(let ([x v]) e)`

  `is the same as`

  `((lambda (x) e) v)`

2019-03-11

- We know that

```
(let ([x v]) e)
```

is the same as

```
((lambda (x) e) v)
```


1. remember how we derived 'fun' from a 'with'

- we can turn that 'let' into the equivalent function application form:


```
(let* ([make-fact
        (lambda (self)
          ((lambda (fact)
             (lambda (n)
               (if (zero? n)
                   1
                   (* n (fact (- n 1))))))
           (lambda (n) ((self self) n))))]
       [fact (make-fact make-fact)])
  (fact 5))
```

- Our (lambda (fact) ...) is a "normal" recursive function definition, except we have to pass the function in.
- Let's refactor this into a separate definition for 'fact-core':

18

```
(let*
    ([fact-core
      (lambda (fact)
        (lambda (n)
          (if (zero? n) 1
              (* n (fact (- n 1))))))]
     [make-fact
      (lambda (self)
        (fact-core
         (lambda (n) ((self self) n))))]
     [fact  (make-fact make-fact)])
  (fact 5))
```

We can now proceed by moving the (make-fact make-fact) self-application into its own function which is what creates the real factorial:

19

```
(let* (
        ⋮
      [make-fact
       (lambda (self)
         (fact-core
          (lambda (n) ((self self) n))))]
      [make-real-fact
       (lambda () (make-fact make-fact))]
      [fact (make-real-fact)])
(fact 5))
```

We can fold the functionality of ‘make-fact’ and ‘make-real-fact’ into a single ‘make-fact’ function by just using the value of ‘make-fact’ explicitly instead of through a definition:

20

```
(let* (
            ⋮
     [make-real-fact
      (lambda ()
        (let ([make (lambda (self)            ;***
                      (fact-core              ;***
                       (lambda (n)
                         ((self self) n))))]) ;***
          (make make)))]
     [fact (make-real-fact)])
  (fact 5))
```

Note that ‘make-real-fact’ has nothing that is specific to factorial – we can make it take a ”core function” as an argument:

21

```
(let* (
             ⋮
     [make-real-fact
      (lambda (core)
        (let ([make (lambda (self)
                      (core                     ;***
                       (lambda (n)
                         ((self self) n))))])
          (make make)))]
     [fact (make-real-fact fact-core)])
  (fact 5))
```

And call it ‘make-recursive’:

22

```
(let* (
              ⋮
     [make-recursive
      (lambda (core)
        (let ([make (lambda (self)
                       (core
                        (lambda (n)
                          ((self self) n))))])
          (make make)))]
     [fact (make-recursive fact-core)])
  (fact 5))
```

We’re almost done now – there’s no real need for a separate ‘fact-core’ definition, just use the value for the definition of ‘fact’:

23

```
(let* ([make-recursive
                 ⋮
       [fact
        (make-recursive
         (lambda (fact)                     ;***
           (lambda (n)
             (if (zero? n)
                 1
                 (* n (fact (- n 1))))))))])
       (fact 5))
```

Turn the ‘let’ into a function form:

24

```
(let* ([make-recursive
        (lambda (core)
          ((lambda (make) (make make))
           (lambda (self)
             (core
              (lambda (n) ((self self) n))))))]
              ⋮
       (fact 5))
```

Do some renamings to make things simpler – ‘make’ and ‘self’ turn to ‘x’, and ‘core’ to ‘f’:

25

```
(let* ([make-recursive
         (lambda (f)
           ((lambda (x) (x x))
            (lambda (x)
              (f (lambda (n) ((x x) n))))))]
               ⋮
       (fact 5))
```

We can manually expand that first (lambda (x) (x x)) application to make the symmetry more obvious:

26

```
(let* ([make-recursive
        (lambda (f)
          ((lambda (x) (f (lambda (n) ((x x) n))))
           (lambda (x) (f (lambda (n) ((x x)
              n))))))]
              ⋮
       (fact 5))
```

And we finally got what we were looking for: a general way to define **any** recursive function using only ‘let’ and ‘lambda’.

2019-03-11

We can manually expand that first (lambda (x) (x x)) application to make the symmetry more obvious:

26
```
(let* ([make-recursive
        (lambda (f)
          ((lambda (x) (f (lambda (n) ((x x) n))))
           (lambda (x) (f (lambda (n) ((x x)
              n))))))]
              ⋮
       (fact 5))
```

And we finally got what we were looking for: a general way to define **any** recursive function using only ‘let’ and ‘lambda’.

1. The symmetry is not really surprising because it started with a ‘let’ whose purpose was to do a self-application

Of course it's a bit tedious to type the make-recursive definition over and over again.

- Instead, let's use the macro facility discussed in Tutorial 4 (and in cs2613).

27

```
(define-syntax-rule (lambda/rec (fun) def)
  ((lambda (f)
     ((lambda (x) (f (lambda (n) ((x x) n))))
      (lambda (x) (f (lambda (n) ((x x) n))))))
   (lambda (fun) def)))

(let
    ([fact
      (lambda/rec (fact)
         (lambda (n)
           (if (zero? n) 1
               (* n (fact (- n 1))))))])
  (fact 5))
```

2019-03-11

Of course it's a bit tedious to type the make-recursive definition over and over again.

- Instead, let's use the macro facility discussed in Tutorial 4 (and in cs2613).

```
(define-syntax-rule (lambda/rec (fun) def)
  ((lambda (f)
     ((lambda (x) (f (lambda (n) ((x x) n))))
      (lambda (x) (f (lambda (n) ((x x) n))))))
   (lambda (fun) def)))

(let
    ([fact
      (lambda/rec (fact)
         (lambda (n)
           (if (zero? n) 1
               (* n (fact (- n 1))))))])
  (fact 5))
```

1. We could just use ‘define‘ to define it as a function, but that might make us suspicious about where the recursion is coming from.

This also works for other recursive functions:

28

```
(let ([fib
    (lambda/rec (fib)
      (lambda (n) (if (<= n 1) n
                      (+ (fib (- n 1))
                         (fib (- n 2))))))])
(fib 8))
```

29

```
(let ([length
    (lambda/rec (length)
      (lambda (l) (if (empty? l) 0
                      (add1
                        (length (rest l))))))])
(length '(x y z)))
```

Syntax Rules revisited

- We already know that certain kinds of syntax are not easily done with functions, e.g. short-circuiting `if`, `and`, `or`
- We also used preprocessors to add new syntax to our target language
- One of the strengths of scheme/racket/lisp is the ability to do this kind of preprocessing in the host language.
- A familiar example:

```
(define-syntax-rule (with (x e) b)
  ((lambda (x) b) e))

(with (x 1) (with (y x) (+ y y)))
```

- a simple kind of laziness

31

```
(define-syntax-rule (delay exp) (lambda ()
    exp))
(define-syntax-rule (force exp) ((exp)))

(define crash (delay (error 'help "my
    hovercraft is full of eels")))
(display "everything is copacetic")
(force crash)
```