Recursion without circularity
(or using `let` to define `letrec`)

Dan-Adrian German

Recall the recursive definition of factorial

```
(define fun
  (lambda (n)
    (if (zero? n)
        1
        (* n (fun (sub1 n))))))
```

One way to make this recursive procedure anonymous is by making it aware of itself (and introducing a keyword, `this`)

```
(define nuf
  (lambda (n)
    (if (zero? n)
        1
        (* n (this (sub1 n))))))) ; but what's this?
```

Such a keyword can be easily introduced with the help of a `letrec`

```
(define fact
  (letrec ((this (lambda (n)
                   (if (zero? n)
                       1
                       (* n (this (sub1 n)))))))
    this))
```

But this takes us back to where we started, except that all `lambda`-expressions would now be called `this` and could refer to themselves as such.

Remember the basic method of `cps`-ing our friend the factorial function

```
(define fun-cps
  (lambda (n k)
    (if (zero? n)
        (k 1)
        (fun-cps (sub1 n) (lambda (v) (k (* n v)))))))
```

This new definition requires a specific testing strategy

```
(define test-fun-cps
  (lambda ()
    (fun-cps 3 (lambda (v) v))))

(define fact-cps
  (lambda (n)
    (fun-cps n (lambda (v) v))))
```

Two points are now worth bringing up:

1. How does one prove that `fun`, `fact` and `fact-cps` are equivalent?
2. If we can pack the future and pass it along as an action to be performed later, what other kind of information can we carry around during our computations and what usefulness could this ability entail?

Let me first answer the second question with a version of the factorial which passes itself as an argument:

```
(let ((nuf (lambda (n fun)
             (if (zero? n)
                 1
                 (* n (fun (sub1 n) fun))))))
  (nuf n nuf)) ; won't work because this n is not bound
```

Let's clean up this code so it can actually be used:

```
(define turing
  (lambda (fun)
    (lambda (n)
      (fun n fun))))

(define fact-dag (turing
                   (lambda (n fun)
                     (if (zero? n)
                         1
                         (* n (fun (sub1 n) fun))))))
```

We again bring two points up:

1. No more circularities are present in the definition of `fact` hence the suffix `dag` (that usually stands for direct acyclic graph—which is what we can use now to describe the structure of our computations). Can this result be made more general?
2. How does one prove that `fact-dag` is equivalent to `fact` (and `fact-cps`)?

The answer to the first question is affirmative: it appears that every `letrec` can indeed be expressed as a `let` by following a strategy as described below:

Transformation Strategy:

1. Let `fun`$_0$, `fun`$_1$, ..., `fun`$_n$ be n mutually recursive procedures defined through the same `letrec`, and let `<formals`$_i$`>` be the list of formal parameters of `fun`$_i$, $\forall i \in \{1, \ldots, n\}$
2. Then the original `letrec` can be rewritten as a `let` if all invocations of `fun`$_i$ are replaced with `(fun`$_i$ `<formals`$_i$`> fun`$_0$ `fun`$_1$ `...fun`$_n$`)`

I include an example:

```
(letrec ((odd?  (lambda (n)
                  (if (zero? n) #f
                      (even? (sub1 n)))))
         (even? (lambda (n)
                  (if (zero? n) #t
                      (odd?  (sub1 n))))))
  (list (odd?  5)
        (even? 6)
        (odd?  8)
        (even? 7)))
```

The transformation described above yields in this case:

```
(let ((odd?  (lambda (n odd? even?)
               (if (zero? n)
                   #f
                   (even? (sub1 n) odd? even?))))
      (even? (lambda (n odd? even?)
               (if (zero? n)
                   #t
                   (odd?  (sub1 n) odd? even?)))))
  (list (odd?  5 odd? even?)
        (even? 6 odd? even?)
        (odd?  8 odd? even?)
        (even? 7 odd? even?)))
```

One should notice that the transformation removes all circularities, for the names in the body of the `let` don't have anything in common with the names used inside the `lambda`-expressions.

In other words this is exactly the same thing:

```
(let ((alpha (lambda (n odd? even?)
               (if (zero? n) #f (even? (sub1 n) odd? even?))))
      (beta  (lambda (n odd? even?)
               (if (zero? n) #t (odd?  (sub1 n) odd? even?)))))
  (list (alpha 5 alpha beta)
        (beta  6 alpha beta)
        (alpha 8 alpha beta)
        (beta  7 alpha beta)))
```

Two more points could be brought up:

1. How can one prove that the transformation is correct in general?
2. Does it cover circular structures not encoded as `lambda`-expressions?

The second point could be given an immediate answer:

```
(letrec ((a (cons 1 a))) a)
```

becomes (by turning data into *thunks* and then applying the same rules):

```
(let    ((a (lambda (a) (cons 1 (a a))))) (a a))
```

which is reminiscent of lazy evaluation and infinite streams.

As for the question of correctness (or proof of equivalence) same derivation trees or a notion of bisimilarity as in π-calculus would probably offer a uniform answer to questions 1, 4, and 5 (my choice was to use diagrams).

Replication is used a lot in π-calculus. It is replication that provides a replacement for circularity in the recursive definitions above in our transformation.