CSc 335 Recursion via the Y-Combinator

; We now consider how one can define recursive functions in tls-scheme.

; Thus far, in drscheme, one would use the special form 'define'. For example

```
(define (fact n)
  (if (= n 0) 1 (* n (fact (- n 1)))))
```

; By some magic which we will understand later, the second occurrence of fact refers
; to the first one, and fact is called repeatedly on a diminishing argument.

; You may have noticed, however, that tls-scheme does not include 'define'. It
; sets up special forms quote, cond, and lambda, but no define.

; So if we can in fact define recursive functions in tls-scheme, it must be that
; we can do so without using 'define' itself.

; Indeed, using a device known as the 'Y-combinator', it is possible to do precisely
; this.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; computing without define

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;; We note first that we do not need define for non-recursive code.  For example, in
;
;; (add1 2)
;
;; we can eliminate the reference to add1 using in-line coding
;
;; ((lambda (x) (+ x 1)) 2)
```

strongly recommended exercise

```
;; why can't we do something similar for recursion?  for example, why not

((lambda (f n)
   (f n))
 (lambda (n)
   (if (= n 0) 1 (* n (f (- n 1)))))
 5)

;; (here I assume that the special form if has been added to tls-scheme)

;; to figure out why this is not a way to define the factorial function, you will
;; want to work through its evaluation.  The main question is this: when evlis is
;; called to evaluate the arguments which yield the bindings for f and n, what
;; environment (table) does it use?  Does _that_ table know about _this_ f?


;; So we need another approach.  We don't have time this term to present Friedman and Felleis
;; derivation of the applicative y-combinator, let alone to discuss it's relation to fixed points, bu
;; at least show you what it looks like.

;; Here is an implementation of the length function without define:
```

wannabe recursive call

;; Here is an implementation of the length function without define:

;;;

body of length function with lambda (length) wrapper

; let's agree that we will call

```
(lambda (length)
  (lambda (l)
    (cond
      ((null? l) 0)
      (else (add1 (length (cdr l)))))))
```

; 'the function that looks like length',

; and that the operator

```
(lambda (le)
   ((lambda (mk-length)
      (mk-length mk-length))
    (lambda (mk-length)
      (
       le
       (lambda (x)
         ((mk-length mk-length) x)))))
   )
```

; will be described as 'the function that makes length from the function
; that look like length'

```
; the function that makes length from the function that looks like length is called the
; (applicative) y-combinator, and we have (roughly)

;   length = (y-combinator the-function-which-looks-like-length)

; which you may check by just computing with

((lambda (le)
   ((lambda (mk-length)
      (mk-length mk-length))
    (lambda (mk-length)
      (le (lambda (x)
            ((mk-length mk-length) x))))))

 (lambda (length)
   (lambda (l)
     (cond ((null? l) 0)
           (else (add1 (length (cdr l)))))))
 )


; for example, try

(((lambda (le)
    ((lambda (mk-length)
       (mk-length mk-length))
     (lambda (mk-length)
       (le (lambda (x)
             ((mk-length mk-length) x))))))

  (lambda (length)
    (lambda (l)
      (cond ((null? l) 0)
            (else (add1 (length (cdr l)))))))
  )

 '(a b c d e f))
```

; but the real surprise comes up when we apply 'the function that makes length
; from the function that looks like length' (ie, the y-combinator) to
; a function which looks like factorial, as in

```
(((lambda (le)
   ((lambda (mk-length)
      (mk-length mk-length))
    (lambda (mk-length)
      (le (lambda (x)
            ((mk-length mk-length) x))))))
  (lambda (f)
   (lambda (x)
     (cond ((= 0 x) 1)
           (else (* x (f (- x 1)))))))
  )
 5)
```

the body of factorial with a lambda (f) wrapper, where f is the wannabe recursive call

; and find that this returns 5!, or 120.

; as they say - 'whoa!'. where does that 120 come from? clearly, something
; quite general is going on here.

; it seems that the function which makes length from the function that looks
; like length is also the
; function that makes factorial from the function that looks like factorial.

; you might also notice that the function that looks like factorial itself
; looks a lot like the
; the function which looks like length

```
; we must of course at this point exclaim that these computations can be carried out
; in tls-scheme: we have recursive functions without using define!

; to compute the length of '(a b c d e f), for example, we could write

(value '(((lambda (le)
            ((lambda (mk-length)
               (mk-length mk-length))
             (lambda (mk-length)
               (le (lambda (x)
                     ((mk-length mk-length) x))))))

          (lambda (length)
            (lambda (l)
              (cond ((null? l) 0)
                    (else (add1 (length (cdr l)))))))
          )

         '(a b c d e f)))


; to use the function which looks like factorial with tls-scheme, you will need to
; add a primitive or two to the base language


; so we don't need define.
```

```scheme
; aside: you can explore the y-combinator in R5RS more conveniently, however, by
; using define.


(define y
  (lambda (le)
    ((lambda (f) (f f))
     (lambda (f)
       (le (lambda (x) ((f f) x)))))))


; so now we might write (in full scheme)

(define length
  (y
   (lambda (len)
     (lambda (l)
       (cond
         ((null? l) 0)
         (else (add1 (len (cdr l)))))))))


; and

(define factorial
  (y
   (lambda (f)
     (lambda (n)
       (cond
         ((zero? n) 1)
         (else (* n (f (- n 1)))))))))
```

```scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; let, let* and letrec

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; We have seen many examples of let over the course of this semester; we know
; that let can always
; be replaced by an equivalent lambda


; for instance

(let ((a 1)
      (b 2))
  (+ a b))

; the equivalent lambda form is

((lambda (a b) (+ a b))
 1 2)


;;

; for

(let ((a 1))
  (let ((b 2))
    (+ a b)))

; the equivalent lambda form is

((lambda (a)
   ((lambda (b) (+ a b))
    2))
 1)

;;
```

```
; for

(let ((a 1))
  (let ((b 2))
    (+ a b)))

; the equivalent lambda form is

((lambda (a)
   ((lambda (b) (+ a b))
    2))
 1)

;;

; for

(let ((a 1) (b 2))
  (+ a (let ((a 3) (c 4))
         (* a b c))
     ))

; the equivalent lambda form is

((lambda (a b)
   (+ a ((lambda (a c)
           (* a b c))
         3 4)))
 1 2)
```

?

;for an error

```
(let ((a 1)
      (b (+ a 2)))
  (+ a b))
```

; this is expected once we consider the equivalent lambda formulation:

```
((lambda (a b)
   (+ a b))
 1
 (+ a 2))
```

; let* (iterated let) can do what we want here

```
(let* ((a 1)
       (b (+ a 2)))
  (+ a b))
     )
```

should ~~think~~ think of ~~let*~~ let* as

nested let

```
(let ((a 1))
  (let ((b (+ a 2)))
    (+ a b)))
```

list of bindings — each binding is a 2-elt list

; we have noted previously that let can be used to bind closures, as in

```
(let ((fact (lambda (n) (add1 n))))
  (fact 4))
```

; but that there is a problem when we attempt to name a recursive procedure this way

```
(let ((fact ((lambda (n) (add1 n)))))
  (let ((fact (lambda (n)
                (cond ((zero? n) 1)
                      (else (* n (fact (- n 1)))))))
    (fact 5))))
```

; to do so we need letrec

```
(letrec ((f (lambda (n)
              (if (= n 0)
                  1
                  (* n (f (- n 1)))))))
  (f 4))

;

(letrec ((a 1)
         (b (+ a 2)))
  (+ a b))
```

letrec can in fact do it all, BUT using it when no recursion is present is bad form.

; Interestingly, letrec can be based on assignment (set! in scheme). Before
; considering its implementation,
; however, it makes sense to get a better idea of what it is used for.
; (The following is again from
; Friedman and Felleisen - this time their sequel text 'The Seasoned Schemer')

As for let and let*, letrec may appear anywhere. The restrictions imposed on define do not apply.

; Recall the code for multirember

```
(define multirember
  (lambda (a lat)
    (cond
      ((null? lat) ())
      ((eq? (car lat) a) (multirember a (cdr lat)))
      (else (cons (car lat) (multirember a (cdr lat)))))))
```

; we could remove the requirement that the parameter 'a' be carried each time by explo
; Y-combinator, as follows - without using define -

```
(define multirember
  (lambda (a lat)
    ((Y (lambda (mr)
          (lambda (lat)
            (cond
              ((null? lat) (quote ()))
              ((eq? a (car lat)) (mr (cdr lat)))
              (else (cons (car lat)
                          (mr (cdr lat))))))))
     lat)))
```

; here the argument to Y reminds us of the argument we passed to Y
; when we wanted to compute factorial

; that is, it is 'almost' multirember

```
; here the argument to Y reminds us of the argument we passed to Y
; when we wanted to compute factorial

; that is, it is 'almost' multirember

; Y being something of an inconvenience, we can use letrec
; to accomplish the same thing


(define multirember
  (lambda (a lat)
    (letrec (
             (mr (lambda (lat)
                   (cond
                     ((null? lat) (quote ()))
                     ((eq? a (car lat))
                      (mr (cdr lat)))
                     (else
                      (cons (car lat)
                            (mr (cdr lat)))))))
             )
      (mr lat))))


; as you see from these examples, we can use (letrec ...) and scope to remove argume
; change for recursive applications: mr is a function of just the one parameter, lat -- th
; a is held constant, and not passed directly to the recursive calls to mr.
```

; a similar use of letrec is indicated in the context of currying

```
(define multirember-f
  (lambda (test?)
    (letrec
        (
         (m-f
          (lambda (a lat)
            (cond
              ((null? lat) (quote ()))
              ((test? (car lat) a)
               (m-f a (cdr lat)))
              (else
               (cons (car lat)
                     (m-f a (cdr lat)))))))
         )
      m-f)))
```

; eg,

```
(define myfunc (multirember-f eq?))

(my-func 'a '(a b c a b c))
```

```
; it is always good to see examples!


; letrec applied to union

; we had previously designed the function to pass set2,
; even though this parameter never changes

(define union
  (lambda (set1 set2)
    (cond
      ((null? set1) set2)
      ((member? (car set1) set2)
       (union (cdr set1) set2))
      (else (cons (car set1)
                  (union (cdr set1) set2))))))


; a version using letrec solves the awkwardness

(define union
  (lambda (set1 set2)
    (letrec
        ((U (lambda (set)
              (cond
                ((null? set) set2)
                ((member? (car set) set2) (U (cdr set)))
                (else (cons (car set)
                            (U (cdr set))))))))
      (U set1))))
```

; worried that you may not recall the precise (library) definition of member?
; write a private
; member? function enclosed in union

```
(define union
  (lambda (set1 set2)
    (letrec (
             (U
              (lambda (set)
                (cond
                  ((null? set) set2)
                  ((member? (car set) set2) (U (cdr set)))
                  (else (cons (car set)
                              (U (cdr set)))))))

             (member?
              (lambda (a lat)
                (cond
                  ((null? lat) #f)
                  ((eq? (car lat) a) #t)
                  (else (member? a (cdr lat)))))))

      (U set1))))
```

; thus letrec provides a way of hiding functions (think: modules)