OO PROGRAMMING

Continue: Abstracting list recursion: list-recur

OOP in Racket

Persistent Local Variables

```
(define make-weird-counter
  (let ([supply 5])
    (lambda ()
      (let ([num 0])
        (lambda ()
          (if (zero? supply)'supply_empty
              (begin
                (set! num (add1 num))
                (set! supply (sub1 supply))
                (list supply num)))))))))
; (define wc1 (make-weird-counter))
; (define wc2 (make-weird-counter))
```

OOP

```
(define make-stack
  (lambda ()
    (let ([stk '()])
      (lambda (msg  . args )
       (case msg
         [(empty?) (null? stk)]
         [(push)   (set! stk (cons (car args) stk))]
         [(pop)    (if (not (null? Stk))
                       (let ([top (car stk)])
                         (set! stk (cdr stk))
                         top)
                       (error 'stack "stack is empty"))]
         [else (error 'stack "illegal message to stack
object: ~a" msg)])))))
```

Interacting with the stack

```
> (define s1 (make-stack))
> (define s2 (make-stack))
> (s1 'push 'a)
> (s2 'push 'z)
> (s1 'push 'b)
> (s1 'pop)
b
> (s1 'empty?)
#f
> (s2 'push (s1 'pop))
> (s1 'empty?)
#t
> (s2 'pop)
a
> (s2 'pop)
z
> (s2 'pop)
stack: stack is empty
```

Lazy Evaluation

- Consider:

```
(define lazy_evaluation
  (lambda (t)
    (let ([val #f] [flag #f])
      (lambda () (if (not flag)
                     (begin (set! val (t))
                            (set! flag #t)
                            (write "Yoohoo")
                            val)
                     val)))))
```

- Let's call it like so:

```
(define foo (lazy_evaluation (lambda () 3)))
```

- How do we invoke it and what does it do?

Lazy Evaluation

- Notice that foo is bound to a procedure, the one in red.
- This procedure has associated with it an environment, a context if you wish, in which we have two variables: **`val`** and **`flag`**, as well as their associated values:
  - **val = #f**
  - **flag = #f**
- When we invoke foo like so: **`(foo)`** Racket will evaluate the if expression in the body of **`foo`**.
- Since **`val`** and flag are associated with the procedure **`foo`**, we have access to them and will modify them as indicated.
- The first time we call **`foo`**, we will evaluate **`t`**, store it in **`val`** and return it.
- From then on, we will simply return the stored **`val`**

HW 8 Preview

- On HW 8, you will be asked to implement a lazy iterator for s-lists.
- The iterator returns all non-empty leaf nodes, in the order in which they appear in the s-list.
- Since the iterator has to be lazy, you cannot flatten the list.
- You want to use the stack to process the list.
- Please have a look at the assignment for additional details.

HW 8 Preview

- Below is an example of the iterator in use.

```
> (define iter (make-slist-leaf-iterator '((a (b c) () d) () e)))
> (iter 'next)
a
> (iter 'next)
b
> (iter 'next)
c
> (iter 'next)
d
> (iter 'next)
e
> (iter 'next)
#f
> (iter 'next)
#f
```