CSSE 304 Day 31

`More call/cc` examples

continuations as a datatype

CALL/CC DEFINITION (FOR REFERENCE)

Two more call/cc examples

g)
```
(let ([f 0] [i 0])
  (call/cc (lambda (k) (set! f k)))
  (printf "~a~n" i)
  (set! i (+ i 1))
  (if (< i 10) (f "ignore")))
```

h)
```
(define strange1
  (lambda (x)
    (display 1)
    (call/cc x)
    (display 2)))

(strange1
  (call/cc
    (lambda (k) k)))
```

"mondo bizarro" example

i)

```
(define strange2
  (lambda (x)
    (display 1)
    (call/cc x)
    (display 2)
    (call/cc x)
    (display 3)))
```

We probably will not do this one in class; good practice for you.

```
(strange2 (call/cc (lambda (k) k)))
```

Recap: Environment representations

```
(define apply-env
  (lambda (env sym)
    (env sym)))
```

```
(define empty-env
 (lambda ()
  (lambda (sym)
    (eopl:error 'apply-env
                "No binding for ~s"
                 sym))))
```

```
(define extend-env
  (lambda (syms vals env)
    (lambda (sym)
      (let ([pos (list-find-position
                    sym
                    syms)])
        (if (number? pos)
            (list-ref vals pos)

      (apply-env env sym))))))
```

1. Use Scheme procedures as environments
2. Use environment datatype

```
(define-datatype environment environment?
  [empty-env-record]
  [extended-env-record
    (syms (list-of symbol?))
    (vals (list-of scheme-value?))
    (env  environment?)])

(define empty-env
  (lambda ()
    (empty-env-record)))

(define extend-env
  (lambda (syms vals env)
    (extended-env-record syms
                         vals
                         env)))
```

```
(define apply-env
  (lambda (env sym)
    (cases environment env
      [empty-env-record ()
        (errorf 'apply-env
                "No binding for ~s" sym)]
      [extended-env-record (syms vals env)
        (let ([pos
               (list-find-position sym syms)]
          (if (number? pos)
            (list-ref vals pos)
            (apply-env env sym)))])))
```

This time we represent continuations by our variant-record datatypes

BACK TO WRITING CPS CODE

Continutaion representations

Two possibilities

1. Use Scheme procedures as your continuations (as we have done previously)
2. Use the continuation datatype

    With many variants and a complex **`apply-k`** procedure

You should understand both, but you only have to use the continuation datatype in your A18 interpreter (and, yes, you **must** use it)

Advantages of continuation datatype

- You can "see into" the continuations
  - Thus easier to debug. "trace" will let you see "what's inside" the continuations.
  - And easier to use this exercise as a means of understanding what continuations are all about.
- You can implement continuations in a language that does not have first-class procedures. And more efficiently in a language that does have them.

Advantage of Scheme Procedure Continuations

- It's more like what we did with CPS before.
- All of the information needed for the continuation is in the procedure definitions, so understanding the code requires less mental "jumping around".

Starting code

```
(define read-flatten-print
  (lambda ()
    (display "enter slist to flatten: ")
    (let ([slist (read)])
      (unless (eq? slist 'exit)
        (flatten-cps slist
                     (make-k (lambda (val)
                                (pretty-print val)
                                (read-flatten-print)))))))))
(define flatten-cps
 (lambda (ls k)
  (if (null? ls)
      (apply-k k ls)
      (flatten-cps (cdr ls)
         (make-k
          (lambda (v) (if (list? (car ls))
                          (flatten-cps (car ls)
                                 (make-k (lambda (u) (append-cps u v k))))
                          (apply-k k (cons (car ls) v)))))))))
(define append-cps
  (lambda (L1 L2 k)
    (if (null? L1)
        (apply-k k L2)
        (append-cps (cdr L1)
                    L2
                    (make-k (lambda (appended-cdr)
                              (apply-k k (cons (car L1)
                                               appended-cdr)))))))))
```

Transformations:
Live coding