call/cc

Abandon all hope ye who enter here...

Scheme is one of the few languages that gives programmers access to the current continuation. call/cc is by far the most difficult aspect of Scheme to get a handle on, but it is extraordinarly powerful as a tool for controlling program execution.

Continuation Passing Style gives you control by moving context from the runtime stack, where it is out of your control, to the environment, where you can decide how to make use of it.

Scheme also makes use of the actual continuations built up on the stack and allows you to invoke them through an expression known as call/cc

call/cc has the following format:

(call/cc (lambda (k) body))

When this is executed the body expression is evaluated in an environment in which k is bound to the current continuation. Within the body k can be applied as a function of one argument v. Doing so immediately returns from the call/cc expression with (k v) as the value.

If k is not called within the body then the value of the call/cc expression is the value of body.

For example, consider

```
(call/cc (lambda (k) (k 42)))
```

The context surrounding this expression just gets the expression's value and returns it. That is the continuation that is bound to k in the call/cc expression. This expression applies that continuation to the argument 42. So the result is that this expression returns 42.

In other words, it works exactly the same as

```
(call/cc (lambda (k) 42)
```

However, consider this one

```
(call/cc (lambda (k) (* 5 (* 3 (k 2)))))
```

This has the same continuation as before -- get the value and return it (or (lambda (x) x))

However, when we run it this returns control immediately to the top level -- the entire expression returns 2, not 30.

What do you think this will return?

```
(+ 1 (call/cc (lambda (k)
                ( (lambda (x) (* 20 (k x))) 3))))
```

This should show you one immediate application of call/cc -- escaping from recursions:

```
(define prod-cc (lambda (vec)
            (call/cc (lambda (k)
                    (letrec ([f (lambda (v)
                                (if (null? v)
                                   1
                                   (if (= (car v) 0)
                                      (k 99)
                                      (* (car v) (f (cdr v))))))])
                     (f vec))))))
```

You can also grab the top-level continuation and save it for other uses:

```
(define foo 0)
(+ 1 (call/cc (lambda (k)  (begin (set! foo k) 1))))
```

The continuation of the call/cc expression is "add 1". This expression returns 2, but has a side-effect of setting foo to this continuation. foo is now the same as (lambda (x) (+ 1 x)).

Indeed, if we apply (foo 25) the system gives the return value of 26.

Here is another example of continuation-grabbing. This creates an exit continuation:

```
(define myExit 0)
(call/cc (lambda (k) (set! myExit k)))

(define prod2 (lambda (vec)
            (if (null? vec)
                1
                (if (= 0 (car vec) (myExit 99))
                    (* (car vec) (prod2 (cdr vec)))))))(
```

(prod2 '(2 3 4 0 5 6)) returns 99
(+ 2 (prod2 '(2 3 4 0 5 6))) also returns 99
(define A (prod2 '(2 3 4 0 5 6))) leaves A undefined.

Here is yet another example of continuation-grabbing:

```
(define C 0)
(define grabC (lambda (vec)
              (call/cc (lambda (k)
                       (if (null? vec)
                           (begin
                             (set! C k)
                             1)
                           (* (car vec) (grabC (cdr vec))))))))
```

What does this do?

Explain this one:

```
(define A (call/cc (lambda (k) k)))
(define B A)
```

If I ask for the value of A, it is a continuation.
If I apply A to 9 -- (A 9) the value of A is changed to 9; A is no longer a continuation. But the effect of
(B 27) is to change the value of A to 27.

Here is another mystery:

```
(define bar 0)
(define foo (+ 1 (call/cc (lambda (k) (begin (set! bar k) 0)))))
```

foo evaluates to 1

However, if I call (bar 99)
foo evaluates to 100