Continuations

In our Scheme implementation of *list, we’d like a way to delay doing any multiplies until we know no zeros appear in the list. One approach is to build a continuation—a function that represents the context in which a function’s return value will be used:

```
(define (*listC L con)
  (cond
    ((null? L) (con 1))
    ((= 0 (car L)) 0)
    (else
     (*listC (cdr L)
              (lambda (n)
                 (* n (con (car L))))))
  )
)
```

The top-level call is

```
(*listC L (lambda (x) x))
```

For ordinary lists *listC expands to a series of multiplies, just like *list did.

```
define (id x) x
(*listC '(1 2 3) id) ⇒
(*listC '(2 3)
  (lambda (n) (* n (id 1)))) ≡
(*listC '(2 3)
  (lambda (n) (* n 1))) ⇒
(*listC '(3)
  (lambda (n) (* n (* 2 1)))) ≡
(*listC '(3)
  (lambda (n) (* n 2))) ⇒
(*listC ()
  (lambda (n) (* n (* 3 2)))) ≡
(*listC () (lambda (n) (* n 6))) ⇒ (* 1 6) ⇒ 6
```

But for a list with a zero in it, we get a different execution path:

```
(*listC '(1 0 3) id) ⇒
(*listC '(0 3)
  (lambda (n) (* n (id 1)))) ⇒ 0
```

No multiplies are done!

Another Example of Continuations

Let’s redo our list multiply example so that if a zero is seen in the list we return a function that computes the product of all the non-zero values and a parameter that is the “replacement value” for the unwanted zero value. The function gives the caller a chance to correct a probable error in the input data.

We create

```
(*list2 L) ≡
Product of all integers in L if no zero appears
else
  (lambda (n) (* n product-of-all-nonzeros-in-L))
```
(define (*list2 L) (*listE L id))

(define (*listE L con)
  (cond
    ((null? L) (con 1))
    ((= 0 (car L))
      (lambda (n)
        (* (con n)
          (*listE (cdr L) id))))
    (else
      (*listE (cdr L)
        (lambda (m)
          (* m (con (car L)))))))
)
)

In the following, we check to see if *list2 returns a number or a function. If a function is returned, we call it with 1, effectively removing 0 from the list

(let ( (V (*list2 L)) )
  (if (number? V) V
     (V 1))
)

For ordinary lists *list2 expands to a series of multiplies, just like *list did.

(*listE '(1 2 3) id) ⇒
(*listE '(2 3)
  (lambda (m) (* m (id 1)))) ≡
(*listE '(2 3)
  (lambda (m) (* m 1))) ⇒
(*listE '3)
  (lambda (m) (* m (id 1))) ≡
(*listE '3)
  (lambda (m) (* m 1))) ⇒
(*listE ()
  (lambda (m) (* m (id 1)))) ≡
(*listE () (lambda (n) (* n 6)))
⇒ (* 1 6) ⇒ 6

But for a list with a zero in it, we get a different execution path:

(*listE '(1 0 3) id) ⇒
(*listE '0 3)
  (lambda (m) (* m (id 1))) ⇒
(*listE '3)
  (lambda (n) (* m 1))
  (lambda (n) (* n 1))
  (lambda (n) (* n 1 3))

This function multiplies n, the replacement value for 0, by 1 and 3, the non-zero values in the input list.
But note that only one zero value in the list is handled correctly!
Why?

(define (*listE L con)
  (cond
   ((null? L) (con 1))
   ((= 0 (car L))
    (lambda (n)
     (* (con n)
        (*listE (cdr L) id))))
   (else
    (*listE (cdr L)
     (lambda (m)
      (* m (con (car L)))))))
)

Continuations in Scheme
Scheme provides a built-in mechanism for creating continuations. It has a long name: call-with-current-continuation
This name is usually abbreviated as call/cc
(Perhaps using define).
call/cc takes a single function as its argument. That function also takes a single argument. That is, we use call/cc as
(call/cc funct) where
funct ≡ (lambda (con) (body))
call/cc calls the function that it is given with the “current continuation” as the function’s argument.

Current Continuations
What is the current continuation?
It is itself a function of one argument. The current continuation function represents the execution context within which the call/cc appears. The argument to the continuation is a value to be substituted as the return value of call/cc in that execution context.
For example, given
(+ (fct n) 3)
the current continuation for (fct n) is (lambda (x) (+ x 3))
Given (* 2 (+ (fct z) 10))
the current continuation for (fct z) is (lambda (m) (* 2 (+ m 10)))

To use call/cc to grab a continuation in (say) (+ (fct n) 3) we make (fct n) the body of a function of one argument. Let’s call that argument return. We therefore create
(lambda (return) (fct n))
Then
(call/cc
  (lambda (return) (fct n)))
binds the current continuation to return and executes (fct n).
We can ignore the current continuation bound to return and do a normal return
Or
we can use return to force a return to the calling context of the call/cc.
The call (return value) forces value to be returned as the value of call/cc in its context of call.

Example:

```scheme
(* (call/cc (lambda(return) (/ (g return) 0))) 10)
```

```
(define (g con) (con 5))
```

Now during evaluation no divide by zero error occurs. Rather, when (g return) is called, 5 is passed to con, which is bound to return. Therefore 5 is used as the value of the call to call/cc, and 50 is computed.

---

Continuations are Just Functions

Continuations may be saved in variables or data structures and called in the future to “reactive” a completed or suspended computation.

```scheme
(define CC ())
(define (F)
  (let ((v (call/cc (lambda(here)
    (set! CC here)
    1))))
    (display "The ans is: ")
    (display v)
    (newline))
)
```

This displays

```
The ans is: 1
```

At any time in the future, (CC 10) will display

```
The ans is: 10
```

---

List Multiplication Revisited

We can use call/cc to reimplement the original *list to force an immediate return of 0 (much like a throw in Java):

```scheme
(define (*listc L return)
  (cond
    ((null? L) 1)
    ((= 0 (car L)) (return 0))
    (else (* (car L)
             (*listc (cdr L) return)))))
)
```

```scheme
(define (*list L)
  (call/cc
    (lambda (return)
      (*listc L return))))
)
```

A 0 in L forces a call of (return 0) which makes 0 the value of call/cc.