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))))
)

(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.

Interactive Replacement of Error Values

Using continuations, we can also redo `*listE` so that zeroes can be replaced interactively! Multiple zeroes (in both original and replacement values) are correctly handled.

```scheme
(define (*list L)
  (let ((V (call/cc
            (lambda (here)
             (*liste L here))))
       (if (number? V)
           V
           (begin
             (display "Enter new value for 0")
             (newline) (newline)
             (V (read))
           )))
)

Implementing Coroutines with `call/cc`

Coroutines are a very handy generalization of subroutines. A coroutine may suspend its execution and later resume from the point of suspension. Unlike subroutines, coroutines do not have to complete their execution before they return. Coroutines are useful for computation of long or infinite streams of data, where we wish to compute some data, use it, compute additional data, use it, etc.

Subroutines aren't always able to handle this, as we may need to save a lot of internal state to resume with the correct next value.

(define (*liste L return)
  (if (null? L) 1
      (let loop ((value (car L)))
        (if (= 0 value)
            (loop
             (call/cc
              (lambda (x) (return x)))
             (* value (*liste (cdr L) return)))
            )))

If a zero is seen, `*liste` passes back to the caller (via `return`) a continuation that will set the next value of `value`. This value is checked, so if it is itself zero, a substitute is requested. Each occurrence of zero forces a return to the caller for a substitute value.
Producer/Consumer using Coroutines

The example we will use is one of a consumer of a potentially infinite stream of data. The next integer in the stream (represented as an unbounded list) is read. Call this value n. Then the next n integers are read and summed together. The answer is printed, and the user is asked whether another sum is required. Since we don’t know in advance how many integers will be needed, we’ll use a coroutine to produce the data list in segments, requesting another segment as necessary.

```
(define (consumer)
  (next 0) ; reset next function
  (let loop ((data (moredata)))
    (let ((sum+restoflist
            (sum-n-elems (car data)
            (cons 0 (cddr data)))))
      (display (car sum+restoflist))
      (newline)
      (display "more? ")
      (if (equal? (read) 'y)
        (if (= 1
            (length sum+restoflist))
          (loop (cadr sum+restoflist))
          (loop (moredata)))
        #t ; Normal completion)
    )
  )
)
```

Next, we’ll consider sum-n-elems, which adds the first element of list (a running sum) to the next n elements on the list. We’ll use moredata to extend the data list as needed.

```
(define (sum-n-elems n list)
  (cond
    ((= 0 n)    list)
    ((null? (cddr list))
      (sum-n-elems n
       (cons (car list) (moredata))))
    (else
      (sum-n-elems (- n 1)
       (cons (+ (car list)
          (cadr list))
          (cddr list)))))
)
```

The function moredata is called whenever we need more data. Initially a producer function is called to get the initial segment of data. producer actually returns the next data segment plus a continuation (stored in producer-cc) used to resume execution of producer when the next data segment is required.
(define moredata
  (let ((producer-cc () ) )
    (lambda ()
      (let ((data+cont
        (if (null? producer-cc)
          (call/cc (lambda (here)
            (producer here)))
          (call/cc (lambda (here)
            (producer-cc here))))
        )))
      (set! producer-cc
        (cdr data+cont))
      (car data+cont)
      )
    )
  )
)

Function (next z) returns the next z integers in an infinite sequence that starts at 1. A value z=0 is a special flag indicating that the sequence should be reset to start at 1.

(define next
  (let ((i 1))
    (lambda (z)
      (if (= 0 z)
        (set! i 1)
        (let loop
          ((cnt z) (val i) (ints () ))
            (if (> cnt 0)
              (loop (- cnt 1)
                (+ val 1)
                (append ints
                  (list val)))
              (begin
                (set! i val)
                ints
              )
            )
          )
        )
      )
    )
  )
)

The function producer generates an infinite sequence of integers (1,2,3,...). It suspends every 5/10/15/25 elements and returns control to moredata.

(define (producer initial-return)
  (let loop
    ( (return initial-return) )
      (set! return
        (call/cc (lambda (here)
          (return (cons (next 5)
            here))))))
    (set! return
      (call/cc (lambda (here)
        (return (cons (next 10)
          here))))))
    (set! return
      (call/cc (lambda (here)
        (return (cons (next 15)
          here))))))
    (loop
      (call/cc (lambda (here)
        (return (cons (next 25)
          here))))))
  )