

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 (cdr data)))))
      (display (car sum+restoflist))
      (newline)
      (display "more? ")
      (if (equal? (read) 'y)
          (if (= 1
                (length sum+restoflist))
              (loop (moredata))
              (loop (cdr sum+restoflist)))
          #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? (cdr 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))))))
  ) )
```

Reading Assignment

- MULTILISP: a language for concurrent symbolic computation,
by Robert H. Halstead
(linked from class web page)