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. Call that argument **return**. We therefore build

```
(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:

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

return

```
(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.

```
(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):

```
(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.

```
(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))
       )
    )
  )
)
```



```
(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.

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 no 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.

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)

Lazy Evaluation

Lazy evaluation is sometimes called “call by need.” We do an evaluation when a value is used; not when it is defined.

Scheme provides for lazy evaluation:

```
(delay expression)
```

Evaluation of **expression** is delayed. The call returns a “promise” that is essentially a lambda expression.

```
(force promise)
```

A promise, created by a call to **delay**, is evaluated. If the promise has already been evaluated, the value computed by the first call to **force** is reused.

Example:

Though **and** is predefined, writing a correct implementation for it is a bit tricky.

The obvious program

```
(define (and A B)
  (if A
     B
     #f
  )
)
```

is incorrect since **B** is always evaluated whether it is needed or not. In a call like

```
(and (not (= i 0)) (> (/ j i) 10))
```

unnecessary evaluation might be fatal.

An argument to a function is *strict* if it is always used. Non-strict arguments may cause failure if evaluated unnecessarily.

With lazy evaluation, we can define a more robust **and** function:

```
(define (and A B)
  (if A
     (force B)
     #f
  )
)
```

This is called as:

```
(and (not (= i 0))
     (delay (> (/ j i) 10)))
```

Note that making the programmer remember to add a call to **delay** is unappealing.

Delayed evaluation also allows us a neat implementation of suspensions.

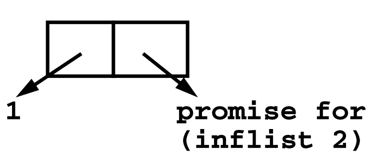
The following definition of an infinite list of integers clearly fails:

```
(define (inflist i)
   (cons i (inflist (+ i 1))))
```

But with use of delays we get the desired effect in finite time:

```
(define (inflist i)
   (cons i
     (delay (inflist (+ i 1)))))
```

Now a call like **(inflist 1)** creates



We need to slightly modify how we explore suspended infinite lists. We can't redefine **car** and **cdr** as these are far too fundamental to tamper with.

Instead we'll define **head** and **tail** to do much the same job:

```
(define head car)
(define (tail L)
   (force (cdr L)))
```

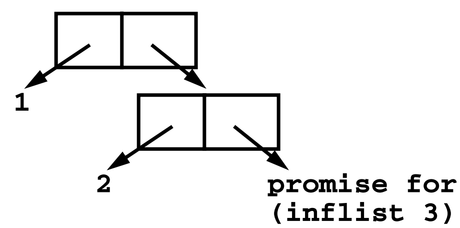
head looks at **car** values which are fully evaluated.

tail forces one level of evaluation of a delayed **cdr** and saves the evaluated value in place of the suspension (promise).

Given

```
(define IL (inflist 1))
```

(head (tail IL)) returns **2** and expands **IL** into



Exploiting Parallelism

Conventional procedural programming languages are difficult to compile for multiprocessors.

Frequent assignments make it difficult to find independent computations.

Consider (in Fortran):

```
   do 10  I = 1,1000
      X(I) = 0
      A(I) = A(I+1)+1
      B(I) = B(I-1)-1
      C(I) = (C(I-2) + C(I+2))/2
10 continue
```

This loop defines 1000 values for arrays **X**, **A**, **B** and **C**.

Which computations can be done in parallel, partitioning parts of an array to several processors, each operating independently?

- **X(I) = 0**
  Assignments to **X** can be readily parallelized.
- **A(I) = A(I+1)+1**
  Each update of **A(I)** uses an **A(I+1)** value that is not yet changed. Thus a whole array of new **A** values can be computed from an array of "old" **A** values in parallel.
- **B(I) = B(I-1)-1**
  This is less obvious. Each **B(I)** uses **B(I-1)** which is defined in terms of **B(I-2)**, etc. Ultimately all new **B** values depend only on **B(0)** and **I**. That is, **B(I)** = **B(0) - I**. So this computation can be parallelized, but it takes a fair amount of insight to realize it.

- `C(I) = (C(I-2) + C(I+2))/2`
  It is clear that even and odd elements of **C** don't interact. Hence two processors could compute even and odd elements of **C** in parallel. Beyond this, since both earlier and later **C** values are used in each computation of an element, no further means of parallel evaluation is evident. Serial evaluation will probably be needed for even or odd values.

Exploiting Parallelism in Scheme

Assume we have a shared-memory multiprocessor. We might be able to assign different processors to evaluate various independent subexpressions.

For example, consider

```
(map (lambda(x) (* 2 x))
  '(1 2 3 4 5))
```

We might assign a processor to each list element and compute the lambda function on each concurrently: