Delay and force: streams

`(delay expr)` returns a value that freezes evaluation of *expr*

`(force expr)` thaws *expr* and memoizes the result so future thaws just return the thawed value

Can use these to implement streams and unbounded data structures:

```
(define (ints-from n)
  (cons n (delay (ints-from (+ n 1)))))

(define nats (ints-from 0))

(define (first-n n strm)
  (if (= n 0) nil
    (cons (car strm)
          (first-n (- n 1)
                   (force (cdr strm))))))

(define (map-stream f strm)
  (cons (f (car strm))
        (delay (map-stream f
                   (force (cdr strm))))))
```

This is handled more cleanly by lazy evaluation in Haskell.

Implementing Delay and Force

We can't implement delay in Scheme as an ordinary function. (Why not?)

But we can come close:

```
(define (my-delay fn)
  (let ((run-once #f)
        (value ()))
    (lambda ()
      (cond ((eqv? run-once #f)
             (set! run-once #t)
             (set! value (fn))))
     value)))

(define (my-force d)
  (d))
```

Trivial example of use:

```
(define test(my-delay
    (lambda () (horrible 1))))

(my-force test) ; or just (test)
```


Using Lambda to Delay Evaluation

We can use lambda to delay evaluation in Scheme (although as with my-delay we have to do something special to cache the value after it is first computed).

This version evaluates both arguments:

```
(define (my-and x y
  (if x y #f)
```

`(and (= 1 2) (= 1 (/ 2 0)))` => error

This version always evauates the first argument, and evaluates the second argument only if necessary:

```
(define (my-and x fn)
  (if x (fn) #f)

  (and (= 1 2) (lambda () (= 1 (/ 2 0)))) => #f
```


How Other Languages Address this Issue

Ada:

two version of and

- and
- and then

Haskell: uses lazy evaluation, so problem doesn't arise

Smalltalk:

two versions of and -- both use call-by-value, but Smalltalk has a lightweight syntax for creating blocks (lambda expressions)

This version evaluates the argument:

`(1=2) & (2/0=1)` => error

[ ] creates a block (lambda):

`(1=2) and: [2/0=1]` => false

Continuations

This is a powerful and sometimes confusing construct.

Suppose that we call a function and never return. Then this is like a goto, but with parameters. (Recall that Scheme supports tail recursion, so this can be made efficient.)

Continuation-passing version of factorial:

```
(define (fact n c) ; c is the continuation
  (if (zero? n) (c 1)
      (fact (- n 1) (lambda (x) (c (* n x))))))
```

We can get the value out by passing the identity function as the continuation function:

```
(fact 3 (lambda (x) x))
  => 6
```

or by passing `write` as the continuation:

`(fact 3 write)`

=> `no value` (prints 6 as a side effect)

Note that we care about the value of `fact` only if we care about the value of its continuation.

Continuations continued

We could also do arithmetic this way.

Let's define continuation versions of +, -, *, and expt:

```
(define (cplus j k f)
  (f (+ j k)))

(define (cminus j k f)
  (f (- j k)))

(define (ctimes j k f)
  (f (* j k)))

(define (cexpt j k f)
  (f (expt j k)))
```


Continuations (3)

Then we can rewrite b^2 - 4ac as follows:

Standard Scheme version:

```
(- (expt b 2) (* 4 a c))
```

Continuation passing version:

```
(cexpt b 2
  (lambda (x) (ctimes 4 a
    (lambda (y)(ctimes y c
      (lambda (z)(cminus x z
                  the-continuation)))))))
```


Continuations (4)

This is an obscure way to write an algebraic expression, and we would be unlikely to write code this way in general.

(Although in the IO language one does just that, albeit with plenty of syntactic sugar to make it more palatable!)

However ...

Operations to be performed appear in eactly the order in which they must be performed. This will make continuations a useful tool when we come to I/O in Haskell.

Continuations are also useful in compiler writing. For example Steele's Rabbit compiler for Scheme was continuation-based.

The same example, reformatted to emphasize how the Scheme code could be converted to 3-register instructions (and a procedure call for exponentiation):

```
(cexpt b 2 (lambda (x)
(ctimes 4 a (lambda (y)
(ctimes y c (lambda (z)
(cminus x z the-continuation)))))))
```

Continuations in More Standard Use in Scheme Programs

Can be used to implement a wide variety of control structures.

- Most common use is loop exits.
- Can also implement catch/throw, coroutines, scheduling, backtracking, etc.

Continuations are first-class citizens in Scheme.

To use continuations in Scheme:

```
(call-with-current-continuation proc)
```

This packages up the current continuation (i.e. the rest of the computation) and calls proc with the continuation as an argument.

Sometimes named `call/cc` (but not in our implementation of Scheme)

```
(define (fast-prod lst)
  (call/cc (lambda (exit)
    (reduce
      (lambda (x y)
        (if (zero? x) (exit 0) (* x y)))
      1 lst))))
```