Topic 21
Streams

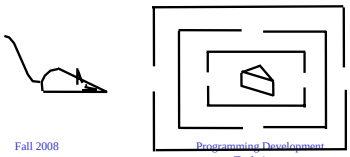
Section 3.5

Modeling State

- Previously – looked at assignment as a tool in modeling, and looked at difficulties raised with time and assignment
- Alternative approach – using streams

Streams

- Streams are data objects for representing long or infinite sequences – in many ways looking like lists but…
- Use delayed (lazy) evaluation
- Example: paths in maze sorted by length



Palindrome numbers

- A palindrome number is a number that remains the same if its digits are reversed
- Examples: 1253521, 53677635
- Suppose we want to compute the sum of all palindrome numbers in a certain range

Recognizing palindrome numbers

```
; takes a number and returns a list of digits in
; the number in reverse order
(define (rev-digits n)
  (if (< n 10)
      (list n)
      (let ((d (remainder n 10)))
        (cons d (rev-digits (/ (- n d) 10))))))

; takes a number and returns #t if that number is a
; palindrome
(define (palindrome? n)
  (let ((dd (rev-digits n)))
    (equal? dd (reverse dd))))
```


Summing palindrome numbers – Implementation 1

```
; takes two integers a < b and returns a sum of all
; palindromes in the interval from a to b
(define (sum-palindromes-1 a b)
  (define (sum-iter count sum)
    (cond ((> count b) sum)
          ((palindrome? count)
           (sum-iter (+ count 1) (+ count sum)))
          (else (sum-iter (+ count 1) sum))))
  (sum-iter a 0))
```

Summing palindrome numbers – Implementation 2

```
; takes two integers a < b and returns a sum of all
; palindromes in the interval from a to b
; uses sequence operations (clear but significantly
    less
; efficient)
(define (sum-palindromes-2 a b)
  (accumulate + 0 (filter palindrome?
                          (enumerate-interval a b))))
```


Problem with conventional interface approach

- Sum-palindromes-2 is easier to understand
- But it is grossly inefficient if b - a is large, e.g, a = 100 and b = 100,000,000
- Finding the first palindrome number can also be inefficient:

```
; return first palindrome in interval
(define (first-palindrome a b)
   (car (filter palindrome?
          (enumerate-interval a b)))

(first-palindrome 123 10000000000)
```


Key property of streams

- Streams allow use of sequence manipulations without incurring the costs
- Streams allow us to generate the elements of a list incrementally as they are needed
- We use the technique of delayed evaluation to represent very large sequences as streams

Streams are delayed lists

- We saw earlier how sequences could serve as standard interfaces for combining program modules
  - map
  - filter
  - accumulate
- But if we represent streams as lists, the elegance is bought at the prices of inefficiency – instead we want to delay evaluation with streams
- Basic idea – arrange to construct a stream only partially, and to pass the partial construction that consumes the stream – construct just as much of the stream as is needed for computation

On the surface, streams look like lists

- We want to think of streams as lists, so we'll need the equivalent of (), null?, car, cdr, cons and other procedures for building and manipulating lists
- With these, we can build most functions we built for lists with streams...

Some basics

```
; empty stream is a special element
(define the-empty-stream empty)

; can think of stream-null? just like null?
(define (stream-null? x) (null? x))
```

We need a cons:

```
(cons-stream <value>
             <procedure-call promise>)
```

Can define most functions on streams – what is this procedure-call promise?

Delay and force

- **Stream implementation based on special form call delay.**
- **Delay macro generates promises**
- **Force converts promises into procedure calls**

```
(define test-promise
        (delay (display 'hi)))
(force test-promise) --> hi
```


Implementing cons-stream

```
; macro for cons – creates new object with promise
; of delayed function call for cdr
(define-syntax cons-stream
  (syntax-rules ()
    ((cons-stream value-expr call-expr)
     (cons value-expr (delay call-expr)))))
```

Expression `(cons-stream <value> <call-expr>)` **expands into** `(cons <value> (delay <call-expr>))` **which is then evaluated**

Selector functions

```
; takes a stream and returns its first element
(define (stream-car stream) (car stream))

; takes a stream and returns the rest after first
; element – this means the next element is forced, but
; remaining elements remain implicit
(define (stream-cdr stream) (force (cdr stream)))

; it is possible to define most list functions for streams
; takes a stream and a positive number and returns the nth
; element of the stream
(define (stream-ref stream n)
  (if (= n 0)
      (stream-car stream)
      (stream-ref (stream-cdr stream) (- n 1))))
```


Stream version of enumerate-interval

```
; takes two integers where low < high and
; returns a stream containing the numbers
; from low to high
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream
        low
        (stream-enumerate-interval (+ low 1)
                                   high))))
```


Equivalent to writing …

```
; takes two integers with low < high and
; generates a stream containing the numbers
; from low to high – here regular cons is used
; Along with delay for the cdr part
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons
        low
        (delay (stream-enumerate-interval (+ low 1)
                                          high)))))
```


Stream-filter

```
; takes a predicate and a stream and returns
; a stream whose elements are the original
; elements of stream for which pred returns #t
(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
        ((pred (stream-car stream))
         (cons-stream (stream-car stream)
                      (stream-filter
                        pred
                        (stream-cdr stream))))
        (else (stream-filter
                pred
                (stream-cdr stream)))))
```

Finding palindrome numbers efficiently

```
; efficient palindrome function using streams
(define palnums
        (stream-filter
          palindrome?
          (stream-enumerate-interval
            123
            100000000)))
(stream-car palnums) --> 131
(stream-ref palnums 99) --> 2222
(stream-ref palnums 1000) --> 92329
```


Some useful stream procedures

```
; takes a procedure of one argument and a stream
; returns the stream that is the procedure applied
; to the first element of the stream, and whose
; remaining elements are the procedure applied
; to the next and so on
(define (stream-map proc stream)
  (if (stream-null? stream)
      the-empty-stream
      (cons-stream
        (proc (stream-car stream))
        (stream-map proc (stream-cdr stream)))))
```


Stream-for-each

```
; takes a procedure that has a side effect
; and a stream.  Applies the procedure
; to each element of the stream.
(define (stream-for-each proc stream)
  (if (stream-null? stream)
      'done
      (begin
        (proc (stream-car stream))
        (stream-for-each proc
                         (stream-cdr stream)))))
```


Display-stream

```
; a function that takes a stream and
; displays it
(define (display-stream stream)
  (display "[")
  (stream-for-each
    (lambda (x)
      (display x)
      (display " "))
    stream)
  (display "]"))
```


Implementing delay

- Delay needs to produce something that can evaluate to a procedure call later
- Lambda expressions do this
- We can expand `(delay <procedure call>)` **to** `(lambda () <procedure call>)`

Simple implementation: delay and force

```
; delays the evaluation of its argument -- macro
(define-syntax delay
  (syntax-rules ()
    ((delay expr) (lambda () expr))))

; forces the evaluation of a delayed procedure call
(define (force delayed-proc-call)
  (delayed-proc-call))
```

Example of delay and force

(define s (stream-enumerate-interval 1 3))
(stream-car (stream-cdr s))

Finding s:

(cons-stream 1
(stream-enumerate-interval 2 3))

(cons 1 (delay (stream-enumerate-interval 2 3)))

(cons 1 (lambda () (stream-enumerate-interval 2 3)))

Evaluating (stream-cdr s)

(stream-cdr (cons 1 (lambda () (stream-enumerate-interval 2 3))))
(force (cdr (cons 1 (lambda () (stream-enumerate-interval 2 3)))))
(force (lambda () (stream-enumerate-interval 2 3)))
(stream-enumerate-interval 2 3)
(cons-stream 2 (stream-enumerate-interval 3 3))
(cons 2 (delay (stream-enumerate-interval 3 3)))
(cons 2 (lambda () (stream-enumerate-interval 3 3)))

Making delay more efficient

It would be more efficient if the delayed procedure call were evaluated only once and the result stored for future reuse

Memoization

```
; takes a procedure and if it has already run
; saves its value - otherwise, the value is
; kept implicit
(define (memo-proc proc)
  (let ((already-run? #f) (result #f))
    (lambda ()
      (if already-run?
          result
          (begin (set! result (proc))
                 (set! already-run? #t)
                 result)))))
```


Memoization – saves on evaluation

(define (factorial n)
(if (eq? n 1) 1 (* n (factorial (- n 1)))))

(define fact-3 (lambda () (factorial 3)))
(fact-3)
(fact-3)
(define memoized-fact-3 (memo-proc fact-3))
(memoized-fact-3)
(memoized-fact-3)

A better delay

```
; more efficient delay
(define-syntax delay
  (syntax-rules ()
    ((delay expr)
     (memo-proc (lambda () expr)))))
```

Infinite streams

We can use streams to represent sequences that are infinitely long.

```
; infinite stream of integers starting at n
(define (integers-from n)
  (cons-stream n (integers-from (+ n 1))))

; an infinite stream of integers starting from 100
(define large-integers (integers-from 100))

; filter to leave just palindromes
(define large-palindromes
  (stream-filter palindrome? large-integers))

; pull the 11th palindrome from the infinite sequence
(stream-ref large-palindromes 11) --> 212
```


Stream of Fibonacci numbers

```
; generates a stream of Fibonacci Numbers
; given two numbers in the sequence, creates
; a sequence of the rest starting from a
(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b))))

; fibs is a pair whose car is 0 and whose
; cdr is a promise to evaluate (fibgen 1 1)
(define fibs (fibgen 0 1))

(stream-ref fibs 10) --> 55
```


Recursively defined streams (defining streams implicitly)

```
; generate an infinite stream of 7's
(define lucky (cons-stream 7 lucky))
```

Note: lucky is a pair whose car is 7 and whose cdr is the promise to evaluate lucky.

Stream addition

```
; takes two streams and returns a stream
; that is the pairwise addition of the
; individual streams
(define (add-stream stream1 stream2)
  (stream-map + stream1 stream2))
```

(See Exercise 3.50 for generalized `stream-map`)

Defining Streams Implicitly...

```
(define ones (cons-stream 1 ones)

(define integers
      (cons-stream 1 (add-streams ones
                                  integers)))

integers =  1  2  3  4  5  6  7 ...
    ones =  1  1  1  1  1  1  1 ...
```


Recursive Fibonacci stream (theory)

- Fib(n+2) = Fib(n+1) + Fib(n)
- In terms of fibs stream:

```
(stream-cdr (stream-cdr fibs)) =
  (add-stream (stream-cdr fibs) fibs)
```

- **(In general, when the index is `(n+k)`, `stream-cdr` has to be applied `k` times)**

In other words ...

```
                fibs = 0  1  1  2  3  5  8 13 21 ...
+ (stream-cdr fibs) = 1  1  2  3  5  8 13 21 34 ...
                    = 1  2  3  5  8 13 21 34 55 ...
                    = (stream-cdr (stream-cdr fibs))
```

hence

```
fibs =
 (cons-stream
   0
   (cons-stream
     1
     (stream-cdr (stream-cdr fibs))))
```


Recursive definition

```
; a recursive version of fibs
(define rfibs
  (cons-stream 0
               (cons-stream 1
                            (add-stream
                              (stream-cdr rfibs)
                              rfibs))))
```


Why it works

- By the time `(add-streams (stream-cdr fibs) fibs)` **is evaluated, the first two values in `fibs` are already explicitly in the stream (memoized), so first value in the stream addition of `(stream-cdr fibs)` and `fibs` can be calculated**
- **Once the first value in the addition stream is calculated, it is made the third value in `fibs`**
- **The second and third values in `fibs` are then available to calculate the second value in the addition stream**
- **This second value in the addition stream is made the fourth value in `fibs`**
- **And so forth**

Scalar multiplication of streams

```
; returns a stream that is the original
; stream with each element multiplied by
; factor
(define (scale-stream stream factor)
  (stream-map (lambda (x) (* x factor))
              stream))
```


Recursive stream of powers

```
; generates the powers of 3
(define triple
  (cons-stream 1
               (scale-stream triple
                             3)))
```