

## COMPUTER SCIENCE 51

Spring 2009

cs51.seas.harvard.edu

Greg Morrisett & Ramin Zabih

## Topics

- Last Time:
  - objects, classes, inheritance
- This Time:
  - macros
    - see *PLT Guide*
  - lazy lists or streams
    - see *SICP*

## Macros

- Used to extend a language.
  - see G.Steele's "Growing a Language"
  - avoid making the language bigger
  - provide convenient notation
- Many languages have support
  - C/C++ via the C-pre-processor
    - works at level of *concrete syntax*
    - very error prone; abused frequently
  - Scheme
    - works at level of *abstract syntax*
    - understands the structure of the language

## An Example

Suppose "or" wasn't pre-defined.

We could try to write:

```
(define (or x y)
  (if x true y))
```

But this isn't ideal.

```
(or (empty? x) (= 0 (car x)))
```

We want `or` to be *lazy* evaluating its second argument, but the default is to evaluate all arguments.

## Macros to the Rescue

So we can instead define a macro:

```
(define-syntax-rule (or x y)
  (if x true y))
```

This looks just like a function, but:

- it operates over syntax, not values
- the compiler expands the macro each time it is called.
- So writing `(or e1 e2)` expands into `(if e1 true e2)`.

## Another Example

Why doesn't this work?

```
(define (inc! x)
  (set! x (+ 1 x)))
```

```
(define z 0)
(inc! z)
z ==> 0!!!
```

## With a Macro

```
(define-syntax-rule (inc! x)
  (set! x (+ 1 x)))

(define z 0)
(inc! z)
; same as (set! z (+1 z))
z ==> 1
```

## One More Example

```
(define-syntax-rule (while t b)
  (letrec ([loop
            (lambda ()
              (if t
                  (begin b (loop))
                  (void)))]])
    (loop)))

(let ([x 0]
      [y 0])
  (while (<= x 10)
    (begin (set! y (+ y x)) (inc! x))))
```

## A Subtlety

```
(define-syntax-rule (swap x y)
  (let ([temp x])
    (begin (set! x y)
           (set! y temp))))

(define temp 42)
(define z 3)
(swap temp z)???
```

## Bad Expansion

```
(define-syntax-rule (swap x y)
  (let ([temp x])
    (begin (set! x y)
           (set! y temp))))

(swap temp z) expands to

(let ([temp temp])
  (begin (set! temp z)
         (set! z temp)))???
```

## Fortunately

Scheme picks fresh variables for us:

```
(swap temp z) expands to

(let ([temp182 temp])
  (begin (set! temp z)
         (set! z temp182)))
```

## Further Uses

Scheme macros are very powerful.

- essentially arbitrary programs
- that manipulate/generate programs

Used to keep language small:

- (let ([x<sub>1</sub> e<sub>1</sub>]...[x<sub>n</sub> e<sub>n</sub>]) e) ==> ((lambda (x<sub>1</sub>...x<sub>n</sub>) e<sub>1</sub>...e<sub>n</sub>)
- (cond ([t<sub>1</sub> e<sub>1</sub>] [t<sub>2</sub> e<sub>2</sub>]... [t<sub>n</sub> e<sub>n</sub>] [else e])) ==> (if t<sub>1</sub> e<sub>1</sub> (if t<sub>2</sub> e<sub>2</sub> ... (if t<sub>n</sub> e<sub>n</sub> e)...))

## Lazy Lists (aka Streams)

One of the coolest applications of macros is building lazy data structures.

The lazy-cons operation

`(lcons e1 e2)` works just like `cons`, except that it only evaluates `e2` when you ask for the lazy-cdr.

## One Way to Define LLs

```
; Wrapping e2 in a lambda causes
; its evaluation to be delayed.
(define-syntax-rule (lcons e1 e2))
  (cons e1 (lambda () e2))
; Lazy-car works the same as car.
(define (lcar lcell) (car lcell))
; Lazy-cdr needs to evaluate the
; lambda that lcons put in.
(define (lcdr lcell)
  ((lcdr lcell)))
```

## Why So Cool?

We can build infinite lists!

```
(define ones (lcons 1 ones))

(lcar ones) ==> 1
(lcar (lcdr (lcdr ones))) ==> 1
(lcar (lcdr (lcdr (...
  (lcdr ones)...))) ==> 1
```

## In Fact:

```
(define ones (lcons 1 ones))

(define (take n xs)
  (if (= n 0) (lcar xs)
      (nth (- n 1) (lcdr xs))))

(take 100 ones) ==> 1
```

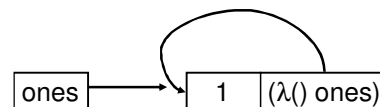
## How Does this Work?

```
(define ones (lcons 1 ones))
```

expands to:

```
(define ones
  (cons 1 (lambda () ones)))
```

## In Pictures



## Map over Lazy Lists

```
(define (lmap f xs)
  (lcons (f (lcar xs))
        (lmap f (lcdr xs))))

(define (incr x) (+ 1 x))

(define twos (lmap incr ones))

(take 42 twos) ==> 2
(take 179 twos) ==> 2
```

## Hmmm....

```
(define nats
  (lcons 0 (lmap incr nats)))

(nth 0 nats) ==> 0
(nth 1 nats) ==> 1
(nth 2 nats) ==> 2
...
(nth 42 nats) ==> 42
...
```

## Ziping Lazy Lists

```
(define (lzip f xs ys)
  (lcons (f (lcar xs) (lcar ys))
        (lzip f (lcdr xs) (lcdr ys))))

(define threes (lzip + ones twos))

(take 74 threes) ==> 3
```

## Whoa...

```
(define fibs
  (lcons 0
        (lcons 1
              (lzip + fibs (lcdr fibs)))))

(take 1 fibs) ==> 1
(take 2 fibs) ==> 1
(take 3 fibs) ==> 2
(take 4 fibs) ==> 3
(take 5 fibs) ==> 5
(take 6 fibs) ==> 8
```

## Filter

```
(define (lfilter p? xs)
  (if (p? (lcar xs))
      (lcons (lcar xs)
            (lfilter p (lcdr xs)))
      (lfilter p (lcdr xs))))

(define odds (lfilter odd? nats))

(take 0 odds) ==> 1
(take 1 odds) ==> 3
(take 2 odds) ==> 5
```

## Sieve of Eratosthenes

```
(define (from n)
  (lcons n (from (+ 1 n))))

(define (not-div n)
  (lambda (m) (not (= 0 (modulo m n)))))

(define (sieve ns)
  (lcons (lcar ns)
        (sieve (filter (not-div (lcar ns))
                       (lcdr ns)))))

(define primes (sieve (from 2)))
```

### Memoization

If you ask for the 54th prime, and then ask for the 32nd prime, we end up re-computing all of the primes.

It would be better if we remembered the result of a computation instead of re-computing it each time.

This is the role of *memoization*.

### Memoizing Lazy Lists

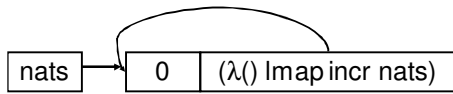
```
(define-syntax-rule (lcons e1 e2))
  (mcons e1 (lambda () e2))

(define (lcar lcell) (mcar lcell))

(define (lcdr lcell)
  (let* ([v ((lcdr lcell))])
    ([f (lambda () v)])
    (begin
      (set-mcdr! lcell f)
      v)))
```

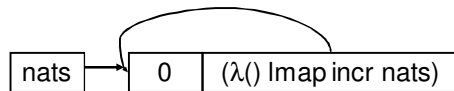
### How This Works

```
(define nats
  (lcons 0 (lmap incr nats)))
```



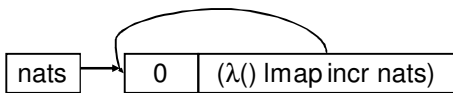
### How This Works

```
(lcar (lcdr nats)) =
(lcar (let* ([v ((lcdr lcell))])
  ([f (lambda () v)])
  (begin
    (set-mcdr! nats f)
    v)))
```



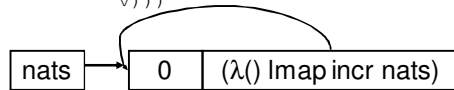
### How This Works

```
(lcar (let* ([v (lmap incr nats)])
  ([f (lambda () v)])
  (begin
    (set-mcdr! nats f)
    v)))
```



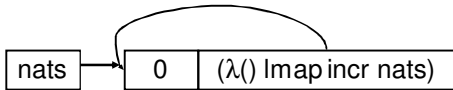
### How This Works

```
(lcar (let* ([v (lcons
  (incr (lcar nats))
  (lmap incr (lcdr nats)))]
  ([f (lambda () v)])
  (begin
    (set-mcdr! nats f)
    v)))
```



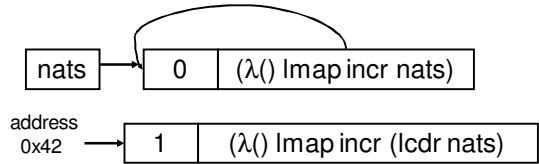
### How This Works

```
(lcar (let* ([v (lcons 1
                    (lmap incr (lcdr nats)))]
            ([f (lambda () v)])
  (begin
    (set-mcdr! nats f)
    v)))
```



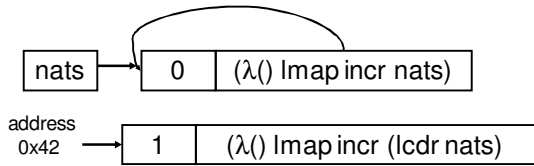
### How This Works

```
(lcar (let* ([v 0x42]
            ([f (lambda () v)])
  (begin
    (set-mcdr! nats f)
    v)))
```



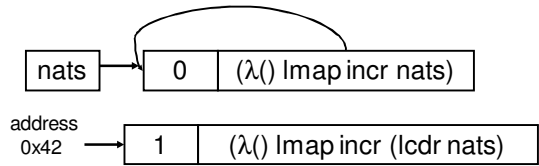
### How This Works

```
(lcar (let* ([f (lambda () 0x42)])
  (begin
    (set-mcdr! nats f)
    0x42)))
```



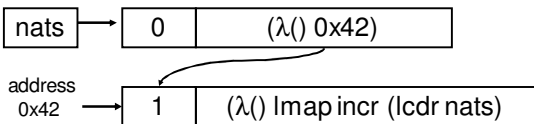
### How This Works

```
(lcar (begin
  (set-mcdr! nats (lambda () 0x42))
  0x42)))
```



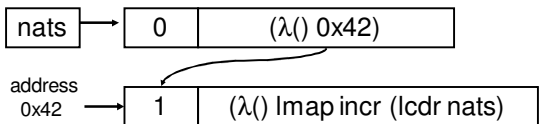
### How This Works

```
(lcar 0x42) ==> 1
```



### How This Works

Subsequent calls of (cdr nats) can return 0x42 immediately.



## Alternative Definitions

```
(define-syntax-rule (lcons e1 e2))
  (let*
    ([cell (mcons e1 empty)]
     [f (lambda ()
          (let ([v e2])
            (set-mcdr! (lambda () v)
                       v))]
          (set-mcdr! cell f)
          cell))

    (define (lcar lcell) (mcar lcell))
    (define (lcdr lcell) ((mcdr lcell))))
```

## Subtlety

Suppose we wish to generate an infinite list of moves to model a player for rock-papers-scissors.

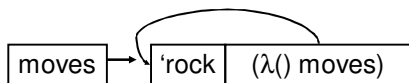
```
(define (gen-move)
  (let ([n (random 3)])
    (cond [(= n 0) 'rock]
          [(= n 1) 'paper]
          [else 'scissors])))

(define moves (lcons (move) moves))
```

## Doesn't Work!

Only one random move is generated.

```
(define moves
  (lcons (move) moves))
```



## Better

```
(define moves
  (lmap (lambda (x) (move)) ones))
```

Moral:

- side effects and laziness are very difficult to reason about.
- if we stick to pure expressions, then we don't run into as many problems.

## Cool Examples

Remember Taylor Series?

Recall that:

- $e = 1/1! + 1/2! + 1/3! + \dots$
- $\pi = 4/1 - 4/3 + 4/5 - 4/7 + \dots$

```
(define e-series
  (lzip / ones (lmap factorial nats)))
```

```
(define alt-fours
  (lcons 4 (lcons (-4 alt-fours))))
```

```
(define pi-series
  (lzip / alt-fours odds))
```

## Integration

*; an approximation of the area under f  
; from points [a..b]. Essentially, the  
; average height times the width.*

```
(define (approx f a b)
  (/ (* (+ (f a) (f b)) (- b a) 2))

(define (mid a b) (/ (+ a b) 2))
; series of better approximations
(define (integrate f a b)
  (lcons
    (approx f a b)
    (lzip + (integrate f a (mid a b))
          (integrate f (mid a b) b))))
```

## Convergence

```
(define (within eps s)
  (if (< (abs(- (lcar s)(lcdr s)))
        eps)
      (lcar s)
      (within eps (lcdr s))))

(define (integral f a b eps)
  (within eps
    (integrate f a b)))
```

## Lazy Data in Practice

- Numerical analysis
- Input/Output
  - see Unix IO and command line utilities
  - see Map/Reduce
- Modeling
  - see Peyton Jones "Financial Contracts"
  - graphics & animation: see Fran
    - e.g., "scale-free" description of scene
    - clip the scene to the viewport
- Fast, persistent data structures
  - see Okasaki's book
  - e.g., constant-time, fnl, double-ended queues