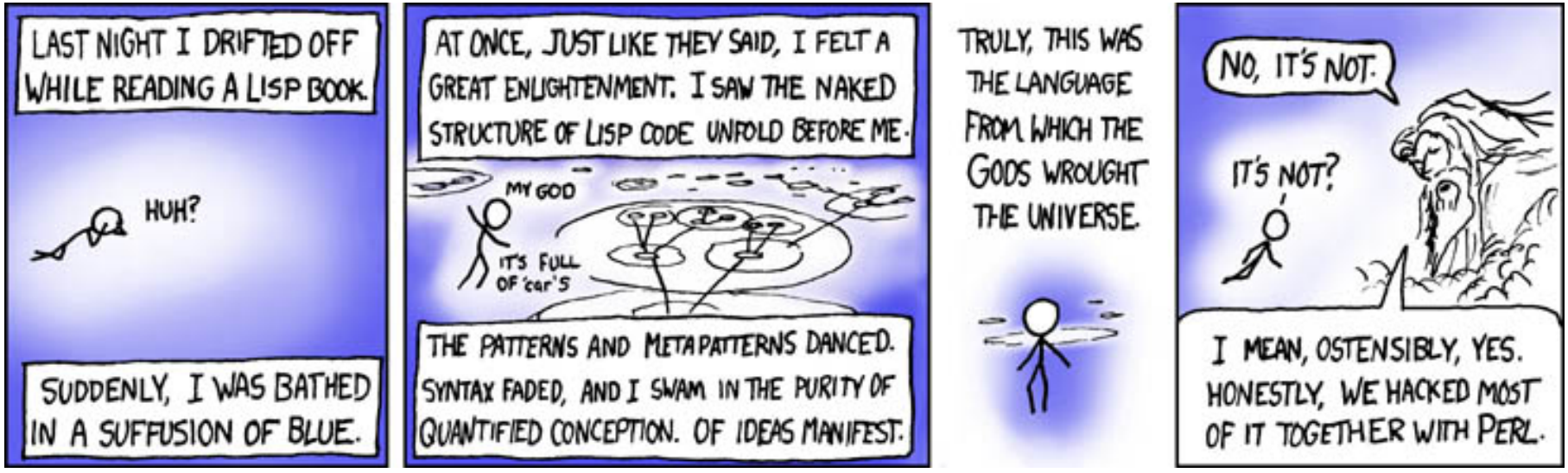
(LISP ≈ Scheme ≈ Racket)



How Scheme (Racket) Works

September 6, 2011

CS 42: Principles and Practice of Computer Science

Sample Solution: duperreverse

```
(define (dR L)     ;; "dR" fits on the slide better
  (cond
    [ (not (list? L))   L ]
    [ (null? L)         L ]
    [ else  (append (dR (rest L))
                    (list (dR (first L)))) ]
  ))
```

Sample Solution: duperreverse

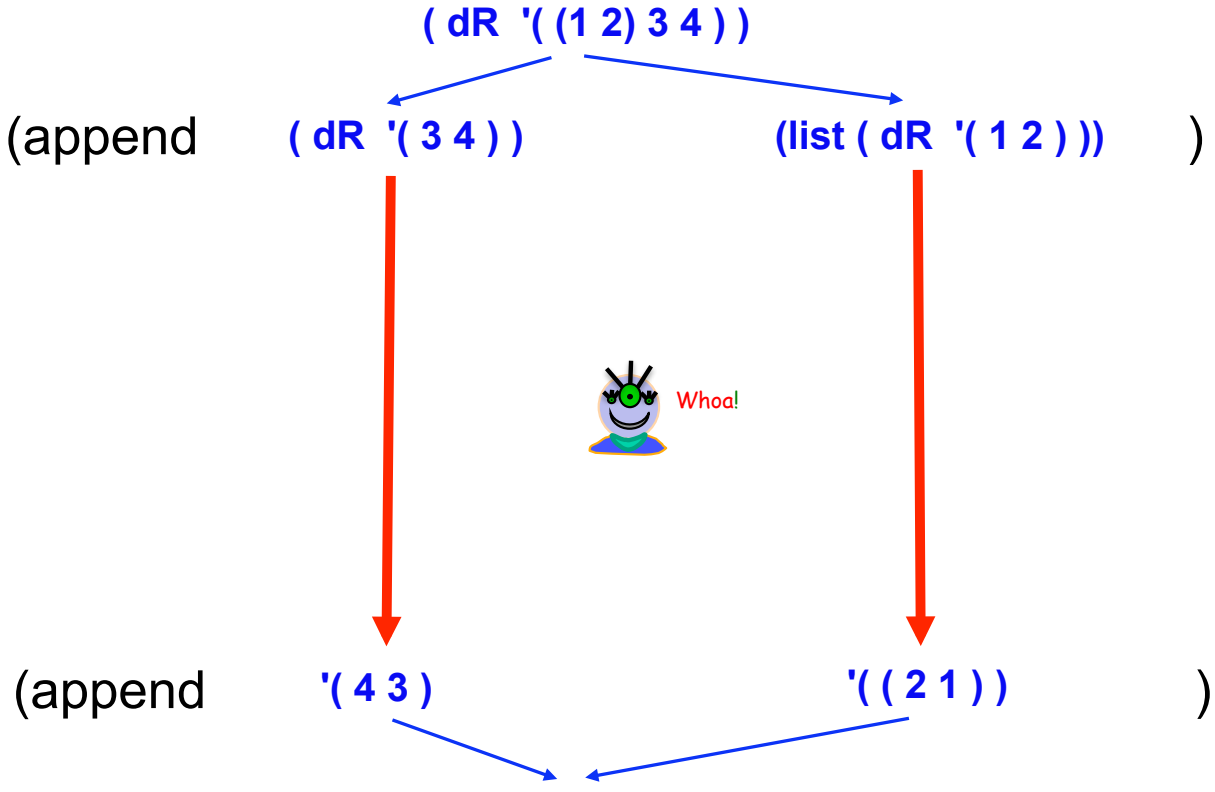
```
(define (dR L)      ;; "dR" fits on the slide better
  (cond
    [ (not (list? L))   L ]
    [ (null? L)         L ]
    [ else  (append (dR (rest L))
                    (list (dR (first L)))) ]
  ))
```

```
Welcome to DrRacket, version 5.1.3 [3m].
Language: racket; memory limit: 128 MB.
> (dR '((1 2) 3 4))
'(4 3 (2 1))
>
```

Recursive Mantra

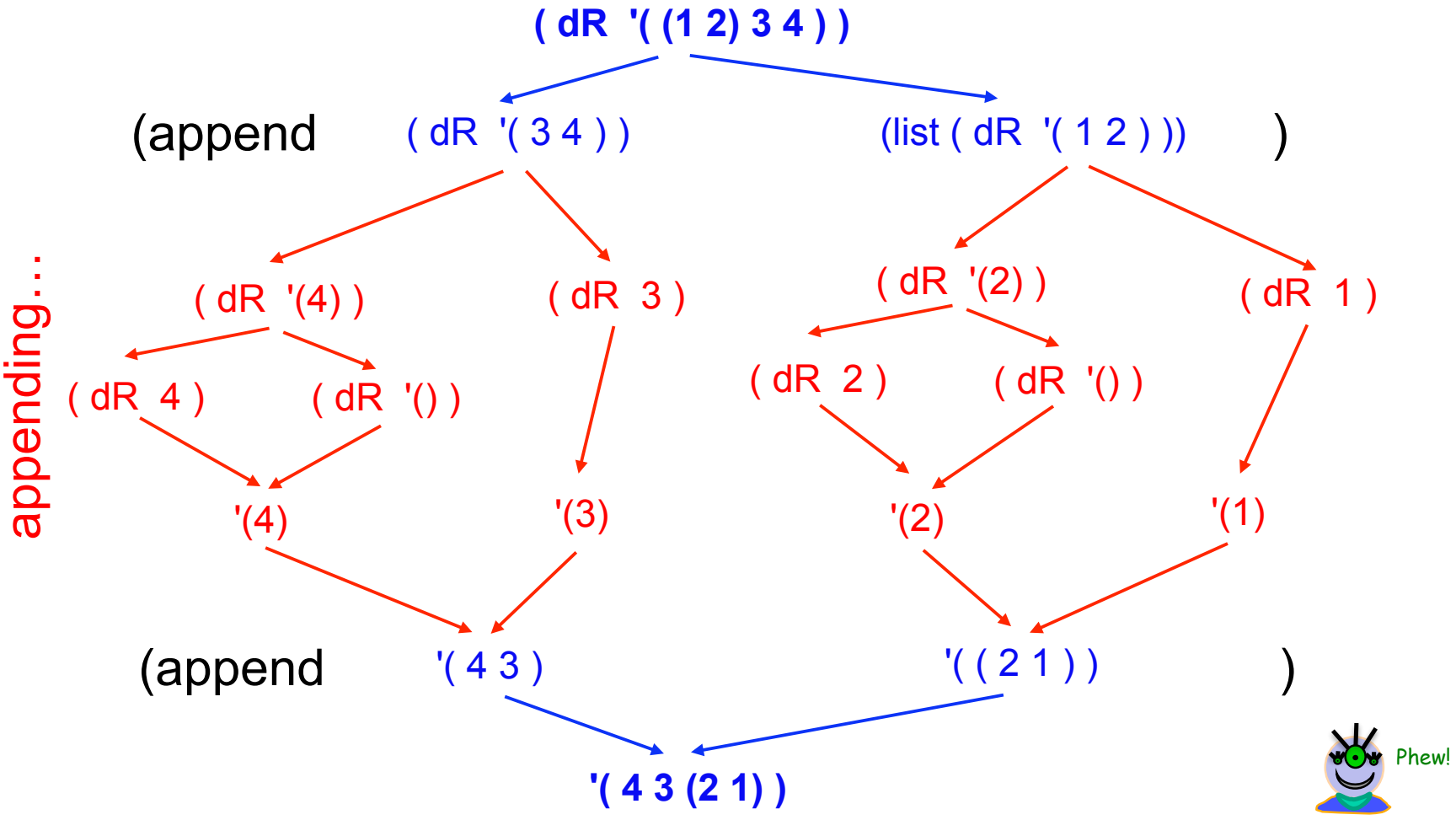
Let recursion *do* the work for you!

Recursive Intuition



done!

Recursive Reality



Functions on a Stack: (f 4)

```
(define (f x)
  (g (expt x 3) x))

(define (g x y)
  (h (- x y)))

(define (h x)
  (* x x))
```

Recursion on a Stack: `(len '(a b c))`

```
(define (len L)
  (cond
    [ (null? L) 0                       ]
    [  else     (+ 1 (len (rest L))) ]
  ))
```

Tail Calls

A function call is called a **tailcall** if it is the last thing a function does before returning. Which are tailcalls?

```
(define (f x)
  (g (expt x 3) x))
```

```
(define (g x y)
  (h (- x y)))
```

```
(define (h x)
  (* x x)
```

```
(define (len L)
  (cond
    [ (null? L) 0                          ]
    [  else     (+ 1 (len (rest L))) ]
  ))
```

Tail-Recursive Functions on a Stack: Compute

```
(len '(a b c))
```

```
(define (len L)  (len-helper L 0))

;; Returns length of L, plus n.
(define (len-helper L n)
  (cond
    [ (null? L) n                                  ]
    [  else     (len-helper (rest l) (+ n 1)) ]
  ))
```

n *here is called an "accumulator argument."*

Optimizing Tail Calls

Key insight: if a function is *about* to make a tailcall, we can *immediately* discard its stack frame!

```
(define (len L)  (len-helper L 0))
(define (len-helper L n)
  (cond
    [ (null? L) n                                  ]
    [  else     (len-helper (rest l) (+ n 1)) ]
  ))

(len '(a b c))
```

Tail-Recursive Factorial

```
(define (fac N) (tail-fac N

(define (tail-fac N A)
  (cond
    [
    [
  )
)
```

Tail-Recursive Factorial

```
(define (fac N)   (tail-fac N 1)

(define (tail-fac N A)
  (cond
    [ (= N 0)   A
    [ else      (tail-fac (- N 1) (* N A)) ]
  )
)
```

Recall: Reverse

How efficient is it? (How efficient is append?)

```
(define (rev L)
 (cond
   [ (null? L)  '() ]
   [  else      (append (rev (rest L))
                        (list (first L)))  ] ))
```

Tail-Recursive Reverse

How efficient is it? (How efficient is `cons`?)

```
(define (rev L)
  (tail-rev

;; Returns a list containing the
;; reverse of L followed by the elements of A.
(define (tail-rev L A)
 (cond
   [
   [  else
  )
)
```

WARNING: Not every function gets more efficient if you make it tail-recursive!

Good News, Bad News

Recursion is very common in Racket, but you can often avoid it completely!

Good News, Bad News

Recursion is very common in Racket, but you can often avoid it completely! How are these similar?

```
;; find squares of numbers
(define (squares L)
  (if (equal? L '())
    '()
     (cons (square (first L))
           (squares (rest L)))
  )
)

;; (squares '(1 2 3))
;;          ===> '(1 4 9)
```

```
;; find factorials of numbers
(define (facts L)
  (if (equal? L '())
    '()
    (cons (fac (first L))
          (facts (rest L)))
  )
)

;; (facts '(1 2 3))
;;        ===> '(1 2 6)
```

map: A Recursion Alternative

```
;; apply function f to all the elements in L
(define (map f L)
  (if (null? L)
      '()
      (cons (f (first L)) (map f (rest L)))))
```

map: A Recursion Alternative

```
;; apply function f to all the elements in L
(define (map f L)
  (if (null? L)
      '()
      (cons (f (first L)) (map f (rest L)))))

(define (facts L) (map fac L))
```

map: A Recursion Alternative

```
;; apply function f to all the elements in L
(define (map f L)
  (if (null? L)
      '()
      (cons (f (first L)) (map f (rest L)))))

(define (facts L) (map fac L))

(define (squares L)
```

Some Anonymous Functions in Racket

- ✓ The "successor function" (Does the name `x` matter?)

```
(lambda (x) (+ x 1))
```

- ✓ The "geometric mean" function

```
(lambda (x y) (sqrt (* x y)))
```

- ✓ The "is-greater-than-5 function"

```
(lambda (N) (> N 5))
```

- ✓ The "is-a-list-of-length-two function"

```
(lambda (L) (and (list? L) (= (len L) 2)))
```

- ✓ The squaring function?

Syntactic Sugar Makes Racket Sweeter

The function definitions we saw earlier:

```
(define (square x)  (* x x))
```

are just abbreviations for:

```
(define square  (lambda (x) (* x x)))
```

Of course, you don't *have* to give functions names before using them:

```
( (lambda (x) (+ x x))  12 )
```

Higher-Order Functions

Functions that take functions as arguments (like `map`) or return functions as results...

```
(define (wrap tag)
  (lambda (text) (list tag text tag)))
```

What is `( (wrap 'w) 'o))` ?

1. Error
2. `#procedure`
3. '(w o)
4. '(w o w)
5. '(o w o)

What does `foldr` do?

```
(foldr  +  0  '(3 4 5))      ==>   12

(foldr cons '() '(1 2 3 4)) ==>  '(1 2 3 4)
```

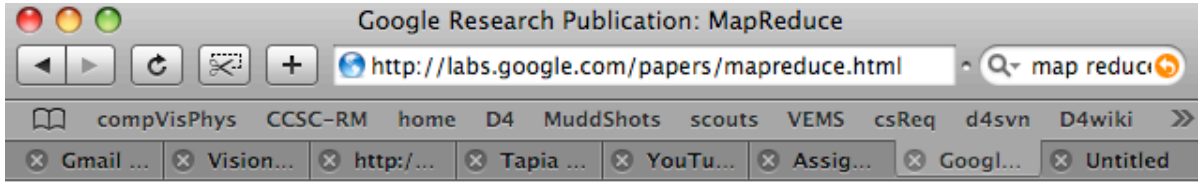
Notes:

- ✓ Sometimes called `reduce`
- ✓ There's also a `foldl`

MapReduce: Google's solution to large-scale parallel processing:

map: processes (in parallel)

reduce: combines



Google™ **Research Publications** Google Labs Home

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

Appeared in:
OSDI'04: Sixth Symposium on Operating System Design and Implementation,
San Francisco, CA, December, 2004.

Download: PDF Version

Slides: HTML Slides

Hiding From Recursion

Write this function without using (explicit) recursion:

```
;; (nest '(a b (c d) e))) ==> '( (a) (b) ((c d)) (e) )
;; (nest '(-1 -2))        ==> '( (-1) (-2) )
```

Hiding From Recursion

Write this function without using (explicit) recursion:

```
;; (nest '(a b (c d) e))) ==> '( (a) (b) ((c d)) (e) )
;; (nest '(-1 -2))        ==> '( (-1) (-2) )
```

Which is right?

```
(define (nest L)
  (map cons L))
```

```
(define (nest L)
  (foldr list '() L))
```

```
(define (nest L)
  (map list L))
```

```
(define (nest L)
  (map (lambda (x) (cons x '())
```

```
(define (nest L)
  (foldr append '() L))
```

Hiding From Recursion (2)

```
;; (bestNum '(32 38 44 50 56)) ===> 44
;; (bestNum '(0 10 100 1000))  ===> 10
```

Hiding From Recursion (2)

```
;; (bestNum '(32 38 44 50 56)) ===> 44
;; (bestNum '(0 10 100 1000))  ===> 10

(define (bestNum L)
 (map max L))

(define (bestNum L)
 (foldr max -inf.0 L))

(define (bestNum L)
 (foldr (lambda (x y) (if (< (abs (- x 42)) (abs (- y 42)) x y))
        +inf.0 L))

(define (bestNum L)
 (foldr (lambda (x y) (if (< (abs (- x 42)) (abs (- y 42)) x y))
        0 L))
```

Name:

"Quiz" (Don't use explicit recursion)

```
; Given a list of lists L, combine them
;   all into a single list.
;
; (smush  '((t h i) (s i s) (s o c o) (o l) )
;     ===> '( t h i s i s s o c o o l )
(define (smush L)
```

```
; Add k to each element of L
; (all will be numeric)
;
; (addk 60 '(-18 101 7940))
;        ===> '(42 161 8000)
(define (addk k L)
```

```
; Given lists of entrees and deserts, return all possible combinations.
;
; (combos '(steak pasta turkey) ('pie 'mousse))
;   ===> '((steak pie) (pasta pie) (turkey pie)
;          (steak mousse) (pasta mousse) (turkey mousse))
;        Possibly in a different order!
(define (combos E D)
```

`filter`

What *does the* higher-order function `filter` *do?*

```
(filter even? '(1 2 3 4 5 6 8))  ===>  '(2 4 6 8)
```

filter

What *does the higher-order function* `filter` *do?*

```
(filter even? '(1 2 3 4 5 6 8))  ===>  '(2 4 6 8)
```

```
;; return L without elements > k.
;; (drop-more 60 '(52 60 65 101 133))  ==> '(52 60)
(define (drop-more L k) ...)

;; return elements in L that *aren't* in M
(define (list-difference L M) ...)
```

sort

`sort` works like you'd expect, but it requires a two-argument comparison function:

```
(sort '(3 2 9 12) <)  ===>
(sort '(3 2 9 12) >)  ===>
(sort '("works" "way" "no" "this") string<?)   ==>

(sort '("works" "way" "no" "this")
      (lambda (s1 s2) (< (string-length s1)
                         (string-length s2)))) ==>
```