Scheme (circa 1975)

Son of LISP (circa 1960):

- applicative programming
  “define a function” vs “write a program”
  “evaluate a function” vs “run a program”
  (interactivity)
- the ultimate simple syntax: parenthesized prefix
  (N.B. no precedence)
- recursion as the standard control structure
- recursive types as the standard data type
  (S-expressions)
- programs as data
- automatic memory mgmt (garbage collection)

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Scheme values

Values are S-expressions, where an S-expression is

- a symbol (name), e.g., `'a`
- an integer literal, e.g., `99`
- a Boolean `#t` or `#f`
- a list $S_1 \ldots S_n$ of 0 or more S-expressions
  - list of 0 elements is `'()`
- (this characterization of lists is a lie :-)

©Copyright 2002 Norman Ramsey. All Rights Reserved.

S-expressions

Like any other abstract data type

- constructors create new values of the type
- observers examine values of the type
- mutators change values of the type

(No mutators in pure subset)

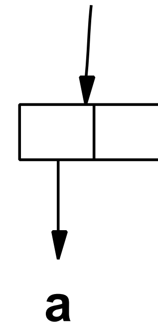
©Copyright 2002 Norman Ramsey. All Rights Reserved.

S-expression Constructors

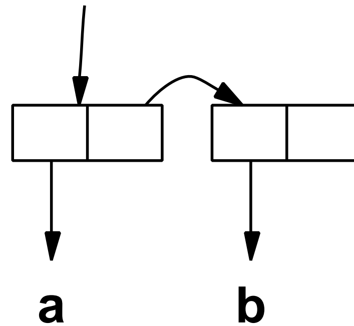
`'()` is the empty list

`cons` creates new list: `(cons S' S)` is the list `S' S`

`S` must be a list: `(cons 'a '())` is



`(cons 'a '(b))` is



`'(b)` is a literal 1-element list

©Copyright 2002 Norman Ramsey. All Rights Reserved.

S-expression observers

Observers defined only on lists:

`(car S)` **where** `S` **is** $(S_1 \ldots S_n)$, $n > 0$, **is** S_1

`(cdr S)` **where** `S` **is** $(S_1 \ldots S_n)$, $n > 0$, **is** $S_2 \ldots S_n$

`(car (cons 'a L))` $\Rightarrow$ `'a`

`(cdr (cons 'a L))` $\Rightarrow$ `L`

N.B. `(cdr (cons 'a '()))` $\Rightarrow$ `'()`

© Copyright 2002 Norman Ramsey. All Rights Reserved.

more S-expression observers

Predicates applying to all types

return `#f` for false, `#t` for true

`(number? x)`	`#t` if `x` is a number
`(symbol? x)`	`#t` if `x` is a symbol
`(pair? x)`	`#t` if `x` is non-empty list
`(null? x)`	`#t` if `x` is `nil`

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Constructor/observer correspondences

symbol literal	`symbol?`	—
integer literal	`number?`	—
`cons`	`pair?`	`car`, `cdr`
`'()`	`null?`	—

Notes

- atomic vs structured values
- primitive vs general constructors

©Copyright 2002 Norman Ramsey. All Rights Reserved.

More constructors and observers

`(< x y)` `#t` if numbers `x < y`

`(> x y)` `#t` if numbers `x > y`

`(= s y)` `#t` if numbers `x, y` are same number, symbol, Boolean, or `'()`

Also the usual arithmetic operators

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Programming with S-expressions

Use recursive functions for a recursive type:

List is `'()` or `(cons x list)`

E.g., length is 0 in base case, 1+length in recursive case:

```
(define length (l)
  (if (null? l) 0 (+1 (length (cdr l)))))
```

`length` applies only to lists.

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Polymorphic equality testing

```
(define atom? (x)
  (or (number? x)
  (or (symbol? x)
  (or (boolean? x)
      (null? x)))))

(define equal? (s1 s2)
  (if (or (atom? s1) (atom? s2))
    (= s1 s2)
    (and (equal? (car s1) (car s2))
         (equal? (cdr s1) (cdr s2)))))
```

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Costs of Cons Cells

Every call to `cons` must allocate space to hold the car and cdr.

To append two lists, exploit $(xX)Y = x(XY)$

```
(define append (x y)
  (if (null? x) y
      (cons (car x) (append (cdr x) y))))
```

Allocates as many cons cells as elements of `x`

What if we want to reverse a list?

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Counting cons cells — reversal

Exploit $rev(xX) = (rev\,X)(rev\,x) = (rev\,X)x$

```
(define reverse (x)
  (if (null ? x) '()
    (append (reverse (cdr x))
            (list1 (car x)))))
```

Cost is $\frac{1}{2}N^2$ cons cells!

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Cheaper reversal

Reduce cost to N cells with new identities:

$$(rev(xX))Z = ((rev\,X)(rev\,x))Z = ((rev\,X)x)Z = (rev\,X)(xZ)$$

```
(define revapp (l z)
  (if (null? l) z
    (revapp (cdr l) (cons (car l) z))))
```

And now:

```
(define reverse (x) (revapp x '()))
```

Using extra arguments to build results:

the method of *accumulating parameters*

a powerful, general programming technique

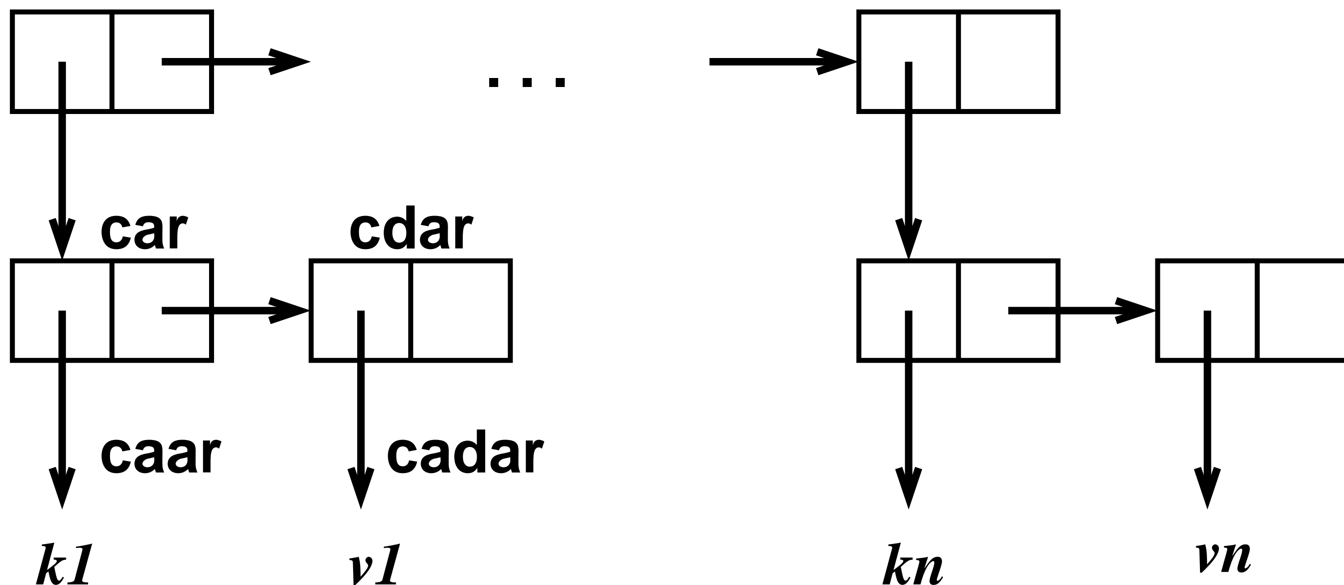
©Copyright 2002 Norman Ramsey. All Rights Reserved.

Association lists

Abstractly, a mapping from keys to values

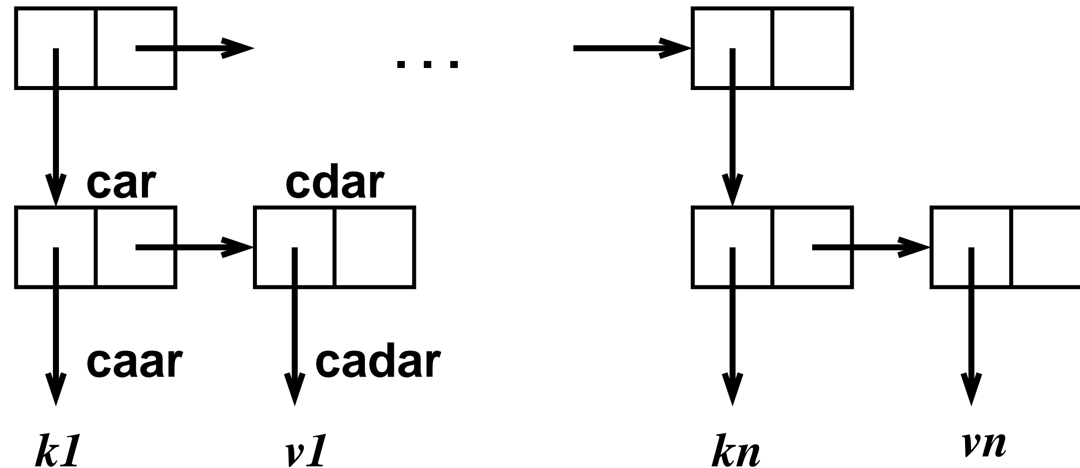
Implementation: list of key-value pairs

$((k_1\ v_1)\ (k_2\ v_2)\ \ldots\ (k_n\ v_n))$



©Copyright 2002 Norman Ramsey. All Rights Reserved.

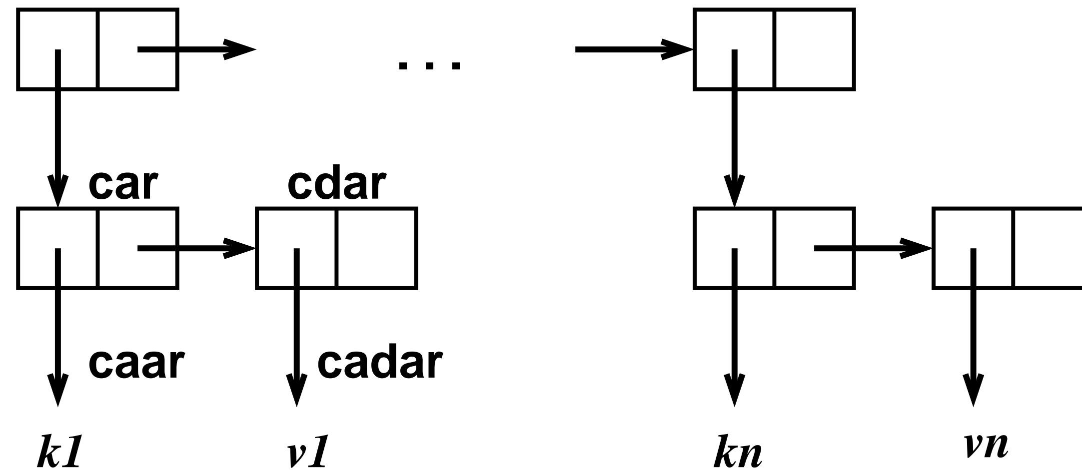
A-list observer: `find`




```
(define caar (l) (car (car l)))
(define cdar (l) (cdr (car l)))
(define cadar (l) (car (cdar l)))
(define find (x alist)
  (if (null? alist) '()
    (if (equal? x (caar alist))
      (cadar alist)
      (find x (cdr alist)))))
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

A-list constructor: `bind`



No side effects:

```
(define bind (x y alist)
  (if (null? alist)
    (list1 (list2 x y))
    (if (= x (caar alist))
      (cons (list2 x y) (cdr alist))
      (cons (car alist) (bind x y (cdr alist))))))
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

A-list example

```
-> (find 'Room '((Course 152)
                 (Instructor Ramsey) (Room (MD G135))))
(MD G135)
-> (val nr (bind 'Office '(MD 231)
           (bind 'Course 152
           (bind 'Email 'cs152@fas '()))))
((Email cs152@fas) (Course 152) (Office (MD 231)))
-> (find 'Office nr)
(MD 231)
-> (find 'Favorite-Food nr)
()
```

Attributes can be lists, not just symbols

“Not found” $\equiv$ “nil”

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Truth about S-expressions

S-expression is symbol, number, Boolean, or *pair* of S-expressions

So, `(cons 2 3)` is legal

- `'()` terminates list just by convention!

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Let and local variables

```
(let ((x1 e1)
      (x2 e2)
        ⋮
      (xn en)) e)
```

"Evaluate $e_1, \ldots, e_n$, bind answers to $x_1, \ldots, x_n$"

- Name intermediate results (avoid recomputation)
- Creates new environment
  $\rho\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$, used to evaluate `e`

Also

`let*`	bind one at a time
`letrec`	local recursive functions

©Copyright 2002 Norman Ramsey. All Rights Reserved.

More about let

`lambda` can do the job, but seems unnatural

`(let ((x1 e1)` $\cdots$ `(xn en)) e)`

Exactly equivalent to

```
((lambda (x1 x2 ... xn) e) e1 e2 ... en)
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

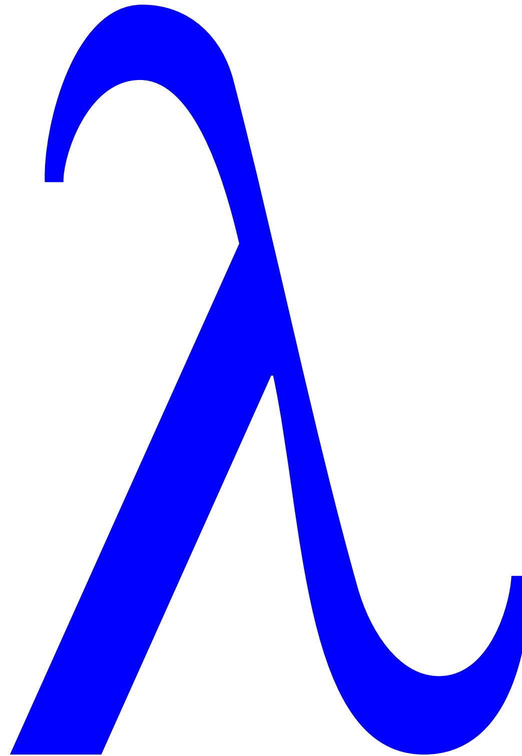
Functional Programming in Scheme

Things that should offend you about Impcore:

- different interface for looking up function vs variable
- have to walk through 2 or 3 environments for variables
- can't create a function without giving it a name
  - means high overhead for using functions
  - sign of 2nd-class citizenship in general (structs)

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Solution to all problems



© Copyright 2002 Norman Ramsey. All Rights Reserved.

Lambda Abstraction

Taken from Church's λ-calculus

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

"The function that maps x to x plus x"

At top level, just like `define`

In general, $\lambda x.E$, also written `(lambda (x) E)`

- **x is *bound* in E**
- **other variables are *free* in E**

Free variables make things interesting

```
(lambda (x) (+ x y))
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Nested functions

Inner funs use parameters, variables of outer funs

- **implementation uses “static links” or “displays”**
- **count different in nesting depth**
- **maintain stack at run time**

  different from the call stack!
- **can identify at compile time which variables are used**

  hence “static scoping”

  (compile-time name resolution)

©Copyright 2002 Norman Ramsey. All Rights Reserved.

History—functions as arguments

Begin to treat functions as values

Example: general zero-finder

```
int findzero(int (*f)(int)) {
  int lo=0, hi=1000, k;
  while (lo + 1 < hi) {
    k = (lo + hi) / 2;
    if (f(k) < 0)  lo = k;
    else           hi = k;
  }
  return hi;
}
```

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Finding roots

Nth root of k by finding zero of $x^n - k$

Can do for any n—use nested function, find its root:

```
-> (define power (x n)
      (if (= n 0) 1 (* x (power x (- n 1)))))
-> (define nth-root-of-k (n k)
      (let
        ((x-to-the-n-minus-k (lambda (x)
                                (- (power x n) k))))
        (findzero x-to-the-n-minus-k)))
-> (nth-root-of-k 3 27)
3
```

So-called "downward funargs"—down the call stack

free variables are in calling context, which is always live

Possible in Ada, Clu, Modula, Pascal ...

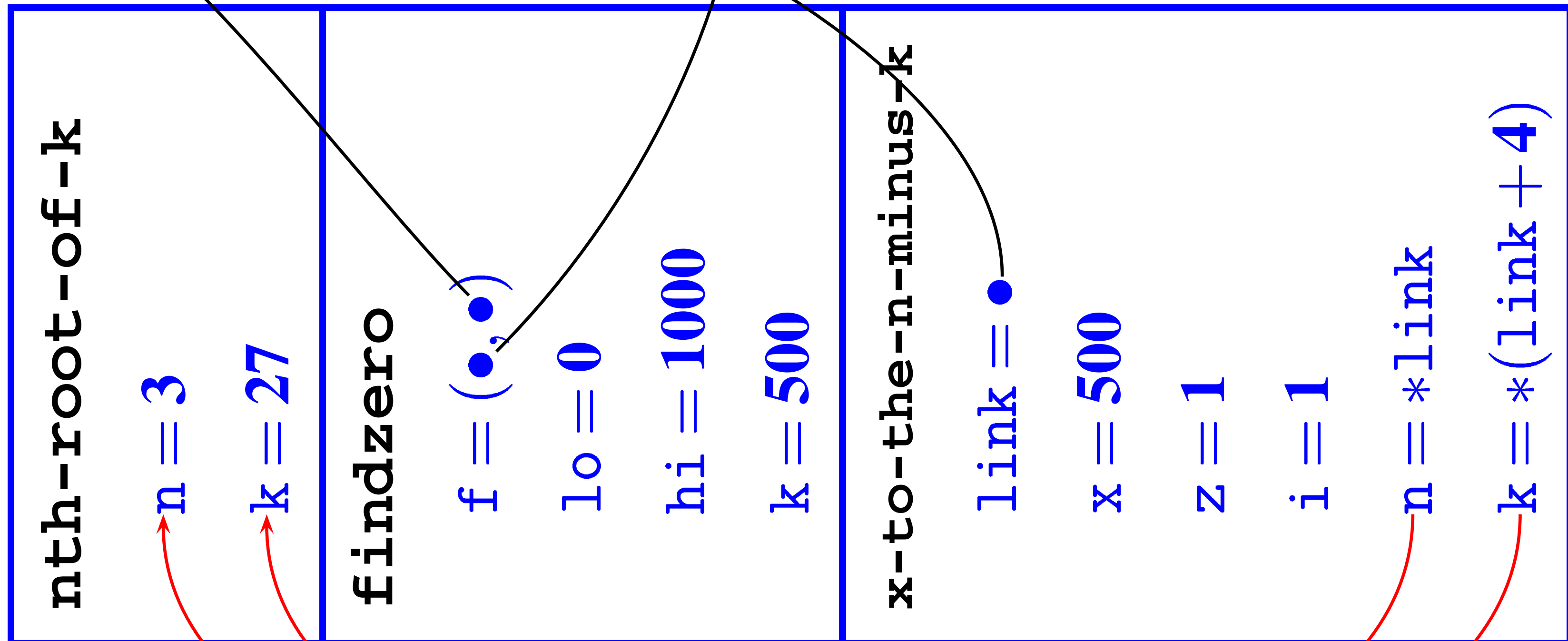
©Copyright 2002 Norman Ramsey. All Rights Reserved.

Downward funargs

While calling `x-to-the-n-minus-k`,

`nth-root-of-k` remains active:

can get to `n` and `k` on the call stack



© Copyright 2002 Norman Ramsey. All Rights Reserved.

More history—functions as results

Functions as values

suppose you don't want zero-finder mixed in?

```
-> (define to-the-n-minus-k (n k)
     (let
       ((x-to-the-n-minus-k (lambda (x)
                              (- (power x n) k))))
       x-to-the-n-minus-k))
-> (val x-cubed-minus-27 (to-the-n-minus-k 3 27))
-> (x-cubed-minus-27 2)
-19
```

`x-to-the-n-minus-k` "*escapes*" its original context

©Copyright 2002 Norman Ramsey. All Rights Reserved.

The “upward funarg problem”

How functions escape:

- return a function
- assign function to global
- store in heap-allocated data structure

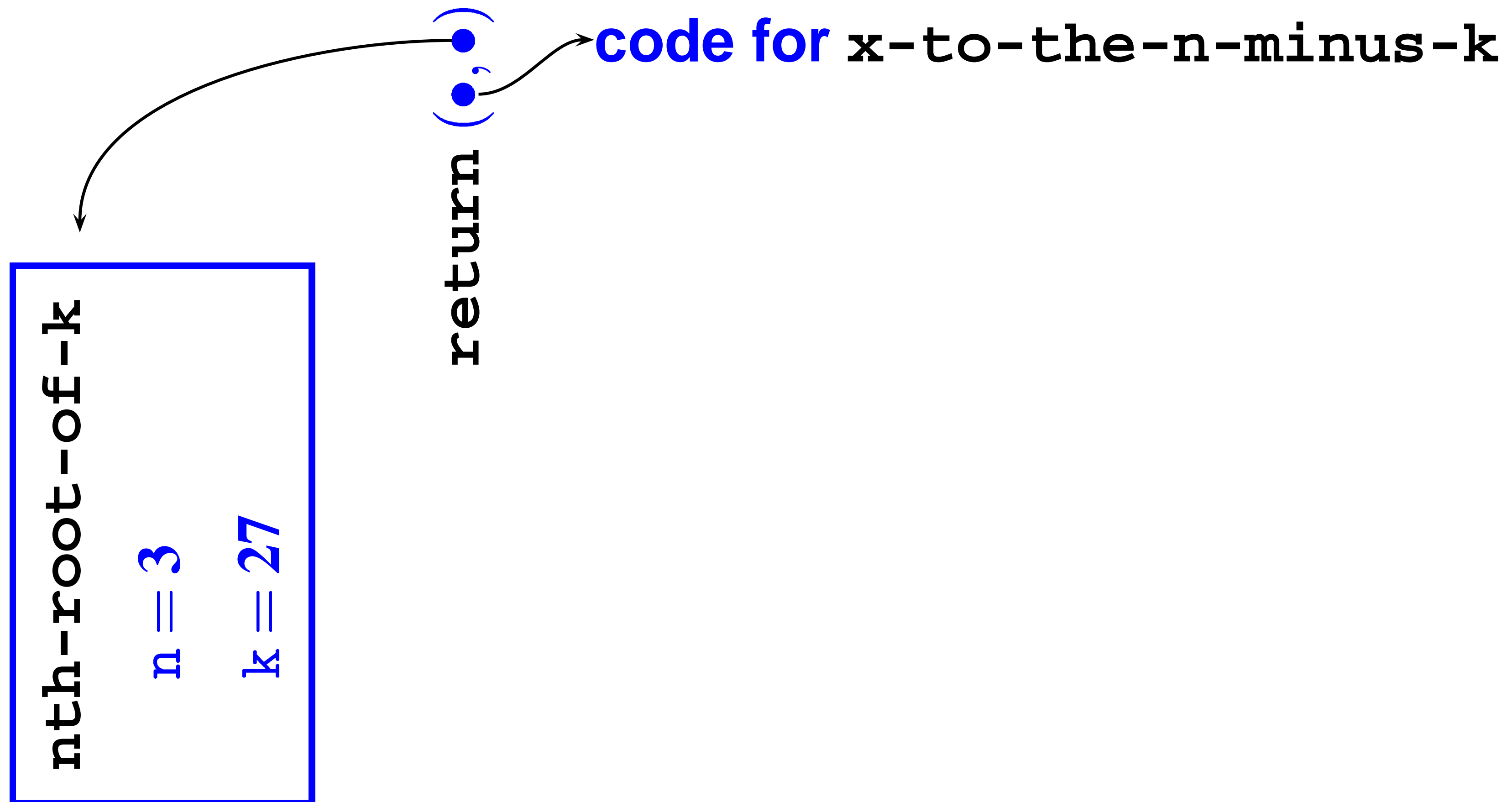
To see problem, imagine implementation:

when call to `to-the-n-minus-k` returns,

where are `n` and `k` ?

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Upward funargs



©Copyright 2002 Norman Ramsey. All Rights Reserved.

Closures

To have a function value, we need the equivalent of $\langle\langle \lambda x.e, \rho \rangle\rangle$, where ρ binds all the free variables of e

This agglutination is called a *closure*

In a compiled sysem, a record containing

- **pointer to the code**
- **all the free variables**

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Operational semantics of closures

$$\frac{}{\langle \mathsf{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle \langle\!\langle \mathsf{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho \rangle\!\rangle, \sigma \rangle} \quad \textbf{(MkClosure)}$$

$$\frac{\begin{array}{c} \ell_1, \ldots, \ell_n \notin \operatorname{dom} \sigma \\ \langle e, \rho, \sigma \rangle \Downarrow \langle \langle\!\langle \mathsf{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e_c), \rho_c \rangle\!\rangle, \sigma_0 \rangle \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e_c, \rho_c\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \Downarrow \langle v, \sigma' \rangle \rangle \end{array}}{\langle \mathsf{APPLY}(e, e_1, \ldots, e_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad \textbf{(ApplyClosure)}$$

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Closures change the game

Higher-order functions

- Functions that take [existing] functions as arguments (mildly amusing)
- Functions that return [new] functions (surprisingly powerful)

```
-> (define o (f g) (lambda (x) (f (g x))))
-> (define even? (n) (= 0 (mod n 2)))
-> (val odd? (o not even?))
-> (odd? 3)
#t
-> (odd? 4)
#f
```

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Examples in Scheme: Currying

```
-> (val positive? (lambda (y) (< 0 y)))
-> (positive? 3)
#t
-> (val <-curried (lambda (x) (lambda (y) (< x y))))
-> (val positive? (<-curried 0))
-> (positive? 0)
#f
-> (val curry ; binary function -> value -> function
     (lambda (f)
       (lambda (x)
         (lambda (y) (f x y)))))
-> (val positive? ((curry <) 0))
-> (positive? -3)
#f
-> (positive? 11)
#t
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

λ as program structuring tool

Global variables are vulnerable:

```
-> (set seed 1)
-> (set rand (lambda ()
        (set seed (mod (+ (* seed 9) 5) 1024))))
-> (rand)
14
-> (rand)
131
-> (set seed 1)
1
-> (rand)
14
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

λ: the ultimate protection

Idea: Hide internal variable `seed` inside λ
(nobody else can touch)

```
-> (set init-rand (lambda (seed)
     (lambda ()
       (set seed (mod (+ (* seed 9) 5) 1024)))))
-> (set rand (init-rand 1))
-> (rand)
14
-> (rand)
131
-> (set seed 1)
1
-> (rand)
160
```

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Standard higher-order functions

Common computations on lists:

`exists?`	Does any element satisfy predicate?
`filter`	Select elements satisfing predicate
`map`	Apply function to elements
fold	General list computations

©Copyright 2002 Norman Ramsey. All Rights Reserved.

List fun. `exists?`: does an element exist?

```
-> (define exists? (p? l)
     (if (null? l)
       #f
       (if (p? (car l))
         #t
         (exists? p? (cdr l)))))
-> (exists? pair? '(1 2 3))
#f
-> (exists? pair? '(1 2 (3)))
#t
-> (exists? ((curry =) 0) '(1 2 3))
#f
-> (exists? ((curry =) 0) '(0 1 2 3))
#t
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

List fun. `filter`: select some elements

```
-> (define filter (p? l)
     (if (null? l)
       '()
       (if (p? (car l))
         (cons (car l) (filter p? (cdr l)))
         (filter p? (cdr l)))))
-> (filter (lambda (n) (>  n 0)) '(1 2 -3 -4 5 6))
(1 2 5 6)
-> (filter (lambda (n) (<= n 0)) '(1 2 -3 -4 5 6))
(-3 -4)
-> (filter ((curry <)  0) '(1 2 -3 -4 5 6))
(1 2 5 6)
-> (filter ((curry >=) 0) '(1 2 -3 -4 5 6))
(-3 -4)
```

© Copyright 2002 Norman Ramsey. All Rights Reserved.

List filtering: composition revisited

```
-> (val positive? ((curry <) 0))
<procedure>
-> (filter positive?          '(1 2 -3 -4 5 6))
(1 2 5 6)
-> (filter (o not positive?) '(1 2 -3 -4 5 6))
(-3 -4)
```

© Copyright 2002 Norman Ramsey. All Rights Reserved.

List function `map`: apply function to list

```
-> (define map (f l)
     (if (null? l)
       '()
       (cons (f (car l)) (map f (cdr l)))))
-> (map number? '(3 a b (5 6)))
(#t #f #f #f)
-> (map ((curry *) 100) '(5 6 7))
(500 600 700)
-> (val square* ((curry map) (lambda (n) (* n n))))
<procedure>
-> (square* '(1 2 3 4 5))
(1 4 9 16 25)
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Grand-daddy of list functions: fold

Idea is: $\lambda + . \lambda 0 . x_1 + \cdots + x_n + 0$

Need the identity element of `+` (call it `zero`):

```
-> (define foldr (plus zero l)
     (if (null? l)
       zero
       (plus (car l) (foldr plus zero (cdr l)))))
-> (val sum  (lambda (l) (foldr + 0 l)))
-> (val prod (lambda (l) (foldr * 1 l)))
-> (sum '(1 2 3 4))
10
-> (prod '(1 2 3 4))
24
```

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Another view of operator folding

```
'(1 2 3 4)   =   (cons 1 (cons 2 (cons 3 (cons 4 '())))))
(foldr + 0 '(1 2 3 4))
             =   (+    1 (+    2 (+    3 (+    4 0   ))))
(foldr f z '(1 2 3 4))
             =   (f    1 (f    2 (f    3 (f    4 z   ))))
```

`foldr` is a member of a class of transformations: **catamorphisms**

Works with *any* recursive datatype—often useful

Another catamorphism: `foldl` associates to left

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Sets revisited

Higher-order functions lead to more compact code:

```
-> (val emptyset '())
-> (define member?      (x s)
     (exists? ((curry equal?) x) s))
-> (define add-element (x s)
     (if (member? x s) s (cons x s)))
-> (define union     (s1 s2) (foldl add-element s1 s2))
-> (define set-from-list (l) (foldl add-element '() l))
-> (union '(1 2 3 4) '(2 4 6 8))
(8 6 1 2 3 4)
```

A few higher-order functions go a long way

Example: Quicksort in 10 lines

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Generalized equality for alists

"Built-in" equal won't do for association lists

A-lists equal if each has same associations as the other:

$$al_1 = al2 \textbf{ iff } al_1 \subseteq al_2 \wedge al_2 \subseteq al_1$$

```
(define sub-alist? (al1 al2)
  (not (exists?
          (lambda (pair)
             (not (equal? (cadr pair)
                          (find (car pair) al2))))
          al1)))
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Equality from subset

```
-> (define =alist? (al1 al2)
      (if (sub-alist? al1 al2) (sub-alist? al2 al1) #f))
-> (=alist? '() '())
#t
-> (=alist? '((E coli)(I Magnin)(U Thant))
            '((E coli)(I Ching)(U Thant)))
#f
-> (=alist? '((U Thant)(I Ching)(E coli))
            '((E coli)(I Ching)(U Thant)))
#t
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Sets of alists

Where to put the equality function?

1. **Extra argument — awkward**

```
(define member? (x s eqfun)
   (exists? ((curry eqfun) x) s))
(define add-element (x s eqfun)
   (if (member? x s eqfun) s (cons x s)))
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Sets of alists, continued

2. Make set pair `(eqfun . elems)` of equality function, elements:

```
(define mk-set (eqfun elements) (cons eqfun elements))
(define eqfun-of    (set) (car set))
(define elements-of (set) (cdr set))
(val emptyset (lambda (eqfun) (mk-set eqfun '())))
(define member? (x s)
   (exists? ((curry (eqfun-of s)) x) (elements-of s)))
(define add-element (x s)
   (if (member? x s) s
       (mk-set (eqfun-of s) (cons x (elements-of s)))))
```

Works, but costs an extra cons cell per instance

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Sets of alists continued

3. Curry! Equality function as part of operations:

```
(val mk-set-ops (lambda (eqfun)
   (list2
      (lambda (x s) (exists? ((curry eqfun) x) s))
      (lambda (x s)
          (if (exists? ((curry eqfun) x) s) s
                      (cons x s))))))
(val al-nullset '())
(val list-of-al-ops (mk-set-ops =alist?))
(val al-member?     (car list-of-al-ops))
(val al-add-element (cadr list-of-al-ops))
```

Must create new ops for every new equality test

best w/few types, static checking

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Continuations—what to do next

Direct style: functions finish by returning a value

Continuation-passing style (CPS): functions finish by "throwing" value to continuation

- Not like a call, because it never returns
- "Goto with arguments"

Can simulate with ordinary tail call

Direct:	`return answer;`
True CPS:	`throw k answer;`
Simulated CPS:	`return k(answer);`

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Application of continuations

`find` can’t distinguish “unbound” from “bound to nil”

```
(define find (x alist)
  (if (null? alist) '()
    (if (equal? x (caar alist))
      (cadar alist)
      (find x (cdr alist)))))
```

Could use different kinds of return values, e.g.,

- if found, `(cons #t (caar alist))`
- if not found, `(cons #f 'irrelevant)`

But this is clunky and costs extra cons cells.

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Success and failure continuations

```
(define find-c (key alist success-cont failure-cont)
  (letrec
     ((search (lambda (alist)
                 (if (null? alist)
                     (failure-cont)
                     (if (equal? key (caar alist))
                         (success-cont (cadar alist))
                         (search (cdr alist)))))))
    (search alist)))
```

Example: table with default

```
(define find-default (key table default)
     (find-c key table (lambda (x) x)
                       (lambda () default)))
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Continuations for search problems

start	Begin with partial solution
`fail`	Partial solution won't work
`succeed`	Pass on improved solution
`resume`	If improved solution won't work, keep trying

A composable unit!

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Moral: functions are cheap

Use lots of them

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Semantics and implementation of μScheme

Key changes from Impcore:

- **New language constructs: let, lambda, application**
- **New values, including functions (closures)**
- **Single environment**
- **Environments get copied**
- **Environment maps names to mutable locations (not values)**

©Copyright 2002 Norman Ramsey. All Rights Reserved.

μScheme vs Impcore

New abstract syntax:

- **LET (keyword, names, expressions, body)**
- **LAMBDAX (formals, body)**
- **APPLY (exp, actuals)**

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Evaluation rules

Judgment $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma \rangle$

σ **is the store**

$$\frac{x \in \operatorname{dom} \rho \qquad \rho(x) \in \operatorname{dom} \sigma}{\langle \mathsf{VAR}(x), \rho, \sigma \rangle \Downarrow \langle \sigma(\rho(x)), \sigma \rangle} \quad \textbf{(Var)}$$

$$\frac{x \in \operatorname{dom} \rho \qquad \rho(x) = \ell \qquad \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \mathsf{SET}(x, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma'\{\ell \mapsto v\} \rangle} \quad \textbf{(Assign)}$$

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Implementation of closures

Key issue: values of free variables

Static scoping:

- at the location of `lambda`, "look outward" for ρ
- keep that ρ until we need it

$$\frac{}{\langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle \langle\!\langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho \rangle\!\rangle, \sigma \rangle} \text{(MkClosure)}$$

So, create closure in `eval` by

```
case LAMBDAX:
    return mkClosure(e->u.lambdax, env);
```

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Applying closures

Saved environment for free variables

Arguments for bound variables (≡ formal parameters)

$$\frac{\begin{array}{c}\ell_1, \ldots, \ell_n \notin \operatorname{dom} \sigma \\ \langle e, \rho, \sigma \rangle \Downarrow \langle \langle\!\langle \mathsf{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e_c), \rho_c \rangle\!\rangle, \sigma_0 \rangle \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e_c, \rho_c\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \Downarrow \langle v, \sigma' \rangle \rangle\end{array}}{\langle \mathsf{APPLY}(e, e_1, \ldots, e_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\textsc{ApplyClosure})$$

```
nl = f.u.closure.lambda.formals;
return eval(f.u.closure.lambda.body,
            bindalloclist(nl, vl, f.u.closure.env));
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Locations in closures

Key is shared mutable state

```
-> (val resettable-counter-from
     (lambda (n)
        (list2
          (lambda () (set n (+ n 1)))
          (lambda () (set n 0)))))
-> (val twenty  (resettable-counter-from 20))
-> ((car twenty))
21
-> ((car twenty))
22
-> ((cadr twenty))
0
-> ((car twenty))
1
```

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Real closures

As in Kamin, closures stored on the heap.

BUT:

- contain only free vars needed, not whole ρ
- name lookup done at compile time

Example: SML/NJ has closure register, arg register.

© Copyright 2002 Norman Ramsey. All Rights Reserved.

SML/NJ closures

```
(val map (lambda (f)
  (lambda (l)
    (if (null? l) '()
      (cons (f (car l)) ((map f) (cdr l)))))))
```

Translation of `map f` uses

`l`	→ `ARG`
`f`	→ `CLOSURE[1]`
`(map f)`	→ `CLOSURE`

body becomes machine code for

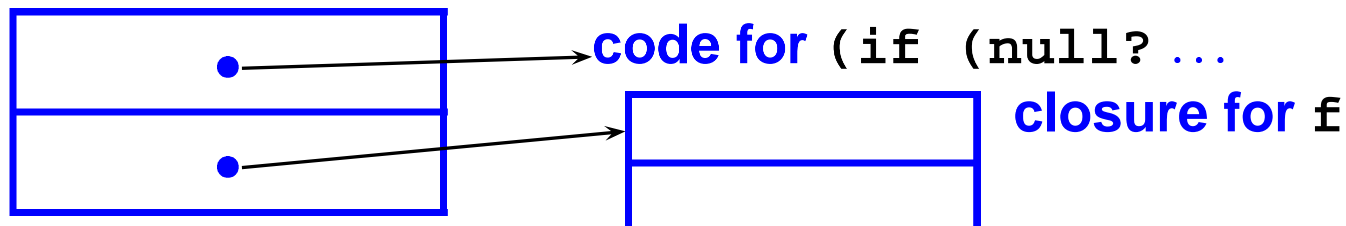
```
  if ARG is null then
    return NIL
  else
    return (cons (CLOSURE[1] (car ARG))
                 (CLOSURE (cdr ARG)))
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Closures in pictures

```
(val map (lambda (f)
  (lambda (l)
    (if (null? l) '()
      (cons (f (car l)) ((map f) (cdr l)))))))
```

closure for `(map f)`



©Copyright 2002 Norman Ramsey. All Rights Reserved.

Closure optimizations

Major issue in making functional programs efficient

Heavy static analysis for:

- **keeping closures on the stack**
  - **(when used as downward funargs)**
- **sharing closures**
  - **(e.g., mutually recursive functions)**
- **eliminating closures**
  - **(e.g., when functions never escape)**

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Scoping

Problem also called *name resolution*:

given an occurrence of variable `x`, which declaration of `x` is meant?

or

what is denoted by this occurrence of `x`?

Scoping rules come in two categories

static—can answer for each x at compile time

dynamic—can't tell until run time

(might change during program execution!)

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Static Scoping

What we all know and love

Works by examining source text.

Scope of a declaration **of x**

> **region of source text in which occurrences of x resolve to that declaration**

Typical scoping strategy:

- **divide source regions into "blocks"**
- **scope runs from declaration to end of block**
- **used in Algol, Pascal, C, Modula, Scheme, ML, ...**

© Copyright 2002 Norman Ramsey. All Rights Reserved.

More Static Scoping

Sometimes blocks can be nested:

- inner declaration can “hide” outer declaration
- outer declaration now suffers from “hole in the scope”

Scope rules easy to express as computations on environments

- like our interpreters, but done at compile time
- we’ll see more when we get denotational semantics

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Scoping—block structure

(Misleading) term **block structure** refers to nested functions

- without block structure, no funarg problems (since all nonlocals must be global & live forever)
- therefore, functions are perfectly good values
- C, C++, Icon fit this category

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Static Scoping with Multiple Name Spaces

One name, one region, many meanings

Example, C:

- variable names
- typedef names
- struct and union tags

Each “name space” is simply an environment

More C:

- members of `struct` define their own name space
- typical compilers keep a tiny environment with the declaration of the structure type

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Multiple Name Spaces, continued

Recall Impcore: functions in one name space, variables in another

```
-> (define f (x) (+ x 1))
-> (set f 3)
-> (f f)
4
```

Multiple name spaces have good and bad points:

- permits more natural use of names
- too many name spaces can confuse the user
  (my opinion: C is pushing the limit)

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Dynamic Scoping

Has origins in an implementation bug in LISP

- resolve free variables using “currently active” functions
  (walk up the call stack looking for parameter names)
- makes `lambda` essentially useless
- but environments are never copied (no closures)

Other languages have other dynamic scope rules

©Copyright 2002 Norman Ramsey. All Rights Reserved.

PostScript Dynamic Scoping

Example, PostScript

- "environment frame" is a first-class value ("dictionary")
- current environment determined by "dictionary stack"
- primitive operators to add, remove frames from stack
- can play some wonderful dirty tricks

  e.g., use exactly the same code to print values, trace pointers for garbage collection
- BUT, creating local variables is a lot of work!

© Copyright 2002 Norman Ramsey. All Rights Reserved.

TeX Dynamic Scoping

Example, TeX

- has both compile-time and run-time binding
- compile time is much like closure formation
- run time is much like PostScript,
  except frames are not first class
  (can only modify current environment
  frame)

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Object-oriented Dynamic Scoping

Example, object orientation

- **Can't tell what "method" will be invoked at compile time**
- **But, can sometimes get some static checking anyway**
- **Most O-O languages have static scoping for variables**

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Applying Scheme: Programs as data

Natural representation for Scheme programs as S-expressions

Classic example is Scheme interpreter in Scheme

"metacircular evaluator"

```
-> (read-eval-print '(
       (+ 3 (* 4 5))
       (val abs (lambda (x) (if (< x 0) (- 0 x) x)))
       (abs -77)
       (cdr (quote (a b c)))))
23
77
(b c)
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Metacircular evaluator, part 1

To begin, simple arithmetic and constants:

```
(define eval (exp)
   (if (number? exp) exp
       (apply-op (car exp)
                 (eval (cadr exp))
                 (eval (caddr exp)))))
(define apply-op (f x y)
        (if (= f '+) (+ x y)
        (if (= f '-) (- x y)
        (if (= f '*) (* x y)
        (if (= f '/) (/ x y) 'error!)))))
-> (eval '(+ 3 (* 4 5)))
23
```

; applications are always binary

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Evaluation with variables

For variables, need environments:
variable/value pairs in association list `rho`

```
(define eval (exp rho)
  (if (number? exp) exp
  (if (symbol? exp) (assoc exp rho)
    (apply-op   (car exp)
        (eval (cadr exp) rho)
        (eval (caddr exp) rho)))))
```

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Evaluating quotation, unary ops

```
(define eval (exp rho)
  ...
 (if (= (car exp) 'quote) (cadr exp)
 (if (= (length exp) 2)

(apply-unary-op (car exp) (eval (cadr exp) rho))
   (apply-binary-op (car exp) (eval ...)))))

-> (eval '(cons 3 (cons (+ i j) (quote ())))
     (mkassoc 'i 5 (mkassoc 'j 3 '())))
(3 8)
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Evaluation with functions

For functions, pass 3rd association list, binding functions to bodies: body includes arguments

```
-> (eval '(double 4) '()
        '((double ((a) (+ a a)))))
8
```

Given fun `exp`, `(car exp)` is formals, `(cadr exp)` is body

```
(define eval (exp rho fundefs)
   ...
   (if (userfun? (car exp) fundefs)
       (apply-userfun (assoc (car exp) fundefs)
                      (evallist (cdr exp) rho fundefs)
                      fundefs)
   ... )
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Applying user-defined functions

Note similarity with code in basic evaluator:

```
(define apply-userfun (fundef args fundefs)
  (eval (cadr fundef) ; body of function
        (mkassoc* (car fundef) args '()) ; local env
        fundefs))
```

where

```
(define mkassoc* (keys values al) ; like mkEnv
  (if (null?  keys) al
    (mkassoc* (cdr keys) (cdr values)
      (mkassoc (car keys) (car values) al))))
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Top-level eval

Don't have `read` to get S-expression, so use quoting:

```
(r-e-p-loop '(
      (+ 3 4)
      (define double (a) (+ a a))
      (double (car (quote (4 5))))   )
```

`(7 double 8)` ; "results list"

Where

```
(define r-e-p-loop* (inputs fundefs)
    ...
   (if (= (caar inputs) 'define) ; function definition
    (process-
def (car inputs) (cdr inputs) fundefs)
    (process-
exp (car inputs) (cdr inptus) fundefs))
```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

. . .)

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Elements of top-level eval

`process-exp` conses onto result list:

```
(define process-exp (e inputs fundefs)
  (cons (eval e '() fundefs) ; cons value of e
        (r-e-p-loop* inputs fundefs)))
```

`process-def` adds to `fundefs`:

```
(define process-def (e inputs fundefs)
  (cons (cadr e) ; cons function name to results
        (r-e-p-loop* inputs
            (mkassoc (cadr e) (cddr e) fun-
defs))))
```

Evaluator is “meta-circular” — can evaluate itself

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Scheme as it really is

Conditional expressions avoid if-else parenthesis nightmare

`(cond (e1 e1')`	**if** `e1` **then** `e1'`
`(e2 e2'')`	**elsif** `e2` **then** `e2'`
⋮	⋮
`(en en'))`	**elsif** `en` **then** `en'`

Eval until one of the *guards* is true, then take corresponding expression. So `(if e1 e2 e3)` is really

```
(cond (e1 e2)
      (#t e3))
```

© Copyright 2002 Norman Ramsey. All Rights Reserved.

More real Scheme

Macros

- functions that manipulate S-expressions (at compile time)
- hygienic macros—name clashes impossible
- `let`, `and`, etc., implemented as macros

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Even More real Scheme

Mutation

```
(set-car!  '(a b c) 'd) => (d b c)
```

- modifies original list
- can create circular lists, sharing
- avoids allocation (`cons`)

Garbage collection: reclaim and reuse unreachable cons cells

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Real Scheme—continuations

Call with current continuation:

`(call/cc (lambda (k) ...` body `... ) )`

Continuation `k` is "what will be done with result of body"

E.g., can call `(k 1)` to return 1 "instantly"

activations in progress are abandoned

If `k` escapes, could return to body even after it finishes!

Like closures, need activation records on the heap

Building block for control flow:

- multithreading
- exception handling

Will revisit later in the term

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Real Scheme—tail calls

Imperative style list-reverse (in C):

```
List revimp(List l) {
  List r;
  for (r = NULL; l; l = l->cdr)
    r = cons(l->car, r);
  return r;
}
```

Uses constant stack space

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Tail calls, continued

Write functionally?

```
List revapp(List l, List r) {
  if (l) then
    return revapp(l->cdr, cons(l->car, r));
  else
    return r;
}
List rev(List l) { return revapp(l, NULL); }
```

Uses stack space proportional to length of `l`

call; call; call; .... call; return; return; return; ... ; return

But, the call is the last thing in the body. Try this:

call; return; call; return; ... call; return

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Optimized tail calls

call; return; call; return; ... call; return

Idea: when call is recursive, implement return; call; by assignment and goto

Total result is

call; return

OK even when tail call is *not* recursive (good exam question!)

True Scheme implementations *must* optimize tail calls

"proper tail recursion"

(Actually applies to all tail calls)

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Why tail calls matter

Recursive function becomes the same as a loop
consumes *constant space*, and also faster!

Function call becomes
“goto with arguments” or
“assignment plus goto”

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Tail recursion and factorial

Non-tail-recursive factorial

```
(set fact (lambda (n)
  (if (= n 1) 1
     (* n (fact (- n 1))))))
```

Requires `n` simultaneous activations of `fact`

Tail-recursive version:

```
(set fact (lambda (n)
  (letrec (
    (f (lambda (i product) ; product of 1..i − 1
          (if (> i n) product
             (f (+ i 1) (* i product)))))
    ) (f 1 1))))
```

Never more than 1 `fact` active at a time: compile into a loop!

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Real systems

Common Lisp:

- Big systems (> 1000 builtins)

Scheme:

- Big programming environments (MzScheme, DrScheme)
- Tiny embedded interpreters (libscheme)
- Everything in between

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Assessment

High-level data structures

- lists powerful for programming
- symbols give instant enumeration
- tables as powerful — can be efficiently implemented

Cheap, easy recursion

- a good fit for recursively defined types
- a natural for symbolic computing

© Copyright 2002 Norman Ramsey. All Rights Reserved.

Assessment

Safety and convenience of garbage collection

hard to overestimate

historical performance highly variable

- early systems embarrassing
- modern systems outperform hand allocation

Programs as data a remarkable paradigm

dynamic analysis

dynamic construction (e.g., tactics)

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Assessment

LISPers invented first (and best) programming environments

everything interactive and dynamic

Difficult to eliminate errors at compile time

- no compile-time checking in language
- everything represented as (exposed) list

  hence mind-boggling `caddr` and friends
- so you *need* a good programming environment

©Copyright 2002 Norman Ramsey. All Rights Reserved.

Assessment

`lambda` is a major win

- can't do it justice in this short time
- but we'll see it some more in ML
- has a real implementation cost

  heap-allocated closures

  (copying environments)

Before leaving the LISP family, some comments about parentheses

- a major barrier to many people
- but as many people find it elegant
- last word: enables programs as data

©Copyright 2002 Norman Ramsey. All Rights Reserved.

What LISP really stands for

Land of

Infinite

Stupid

Parentheses

©Copyright 2002 Norman Ramsey. All Rights Reserved.