# Scheme

## a functional programming language

© Copyright 2007 Norman Ramsey. All Rights Reserved.

# Data!

**Two new kinds of data**

**The automatically managed cons cell**

- **Mother of heap-allocated data structures**

**The function closure**

- **Not just a datum: a new feature**

**The setting: Scheme**

© Copyright 2007 Norman Ramsey. All Rights Reserved.

# Scheme (circa 1975)

**Child 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**
  - **(so no operator precedence)**
- **recursive types as the standard data type**
  - **(S-expressions)**
- **recursion as the standard control structure**
- **programs as data**
- **automatic memory mgmt (garbage collection)**

© Copyright 2007 Norman Ramsey. All Rights Reserved.

# What's new in 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 `'( )` (or "nil")

(This characterization of S-expressions is accurate enough for our current purpose.)

© Copyright 2007 Norman Ramsey. All Rights Reserved.

# S-expressions

Like any other abstract data type

- creators create new values of the type
- producers make new values from existing values
- observers examine values of the type
- mutators change values of the type

So creators + producers = constructors

No mutators in $\mu$Scheme

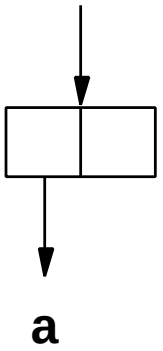
©Copyright 2007 Norman Ramsey. All Rights Reserved.

# S-expression Creators and Producers

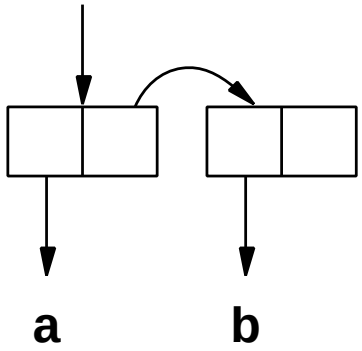
`'( )` is the empty list

`cons` creates new list: `(cons X Y)` is the list `X Y`

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




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




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

©Copyright 2007 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`

**So** `(cdr (cons 'a '( )))` $\Rightarrow$ `'( )`

© Copyright 2007 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 |
| `(boolean? x)` | `#t` if `x` is `#t` or `#f` |
| `(symbol? x)` | `#t` if `x` is a symbol |
| `(pair? x)` | `#t` if `x` is non-empty list |
| `(null? x)` | `#t` if `x` is empty list `'( )` |

© Copyright 2007 Norman Ramsey. All Rights Reserved.

# Constructor/observer correspondences

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

## Notes

- atomic vs structured values
- only producer is `cons`

© Copyright 2007 Norman Ramsey. All Rights Reserved.

# More constructors and observers

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

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

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

Usual arithmetic operators are producers

© Copyright 2007 Norman Ramsey. All Rights Reserved.

# S-expression literals

**Write S-expression literals using either `'` or the `quote` operator**

**`'(a b c)` is shorthand for `(quote (a b c))`**

© Copyright 2007 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 2007 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)))))
```

**Later this semester we will see that ML provides a more elegant syntax for this style: pattern matching**

© Copyright 2007 Norman Ramsey. All Rights Reserved.

# Costs of Cons Cells

**Allocation is a big cost center**

- **Every `cons` allocates space to hold car and cdr**

**To append lists $x$ and $y$, either $x$ is empty or $x = x_a x_d$, in which case exploit $(x_a x_d)y = x_a(x_d y)$**

```
(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 2007 Norman Ramsey. All Rights Reserved.

# Counting cons cells — reversal

**Exploit** $rev(x_a x_d) = (rev\ x_d)(rev\ x_a) = (rev\ x_d)x_a$

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

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

© Copyright 2007 Norman Ramsey. All Rights Reserved.

# Cheaper reversal

**Reduce cost to $N$ cells with new identities:**

$$(rev(x_a x_d))z = ((rev\, x_d)(rev\, x_a))z = ((rev\, x_d)x_a)z = (rev\, x_d)(x_a z)$$

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

**And now:**

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

**Using extra arguments to build results:**

**the method of *accumulating parameters***

**a powerful, general programming technique**

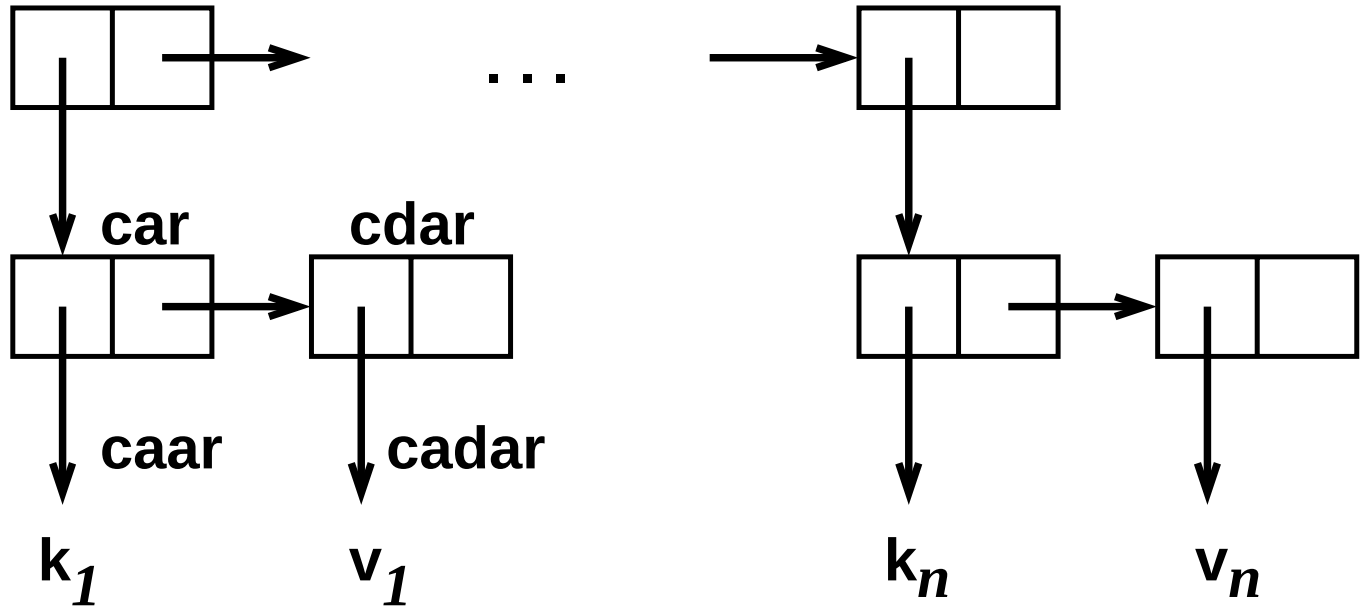
© Copyright 2007 Norman Ramsey. All Rights Reserved.

# Association lists

**Abstractly, a mapping from keys to values**

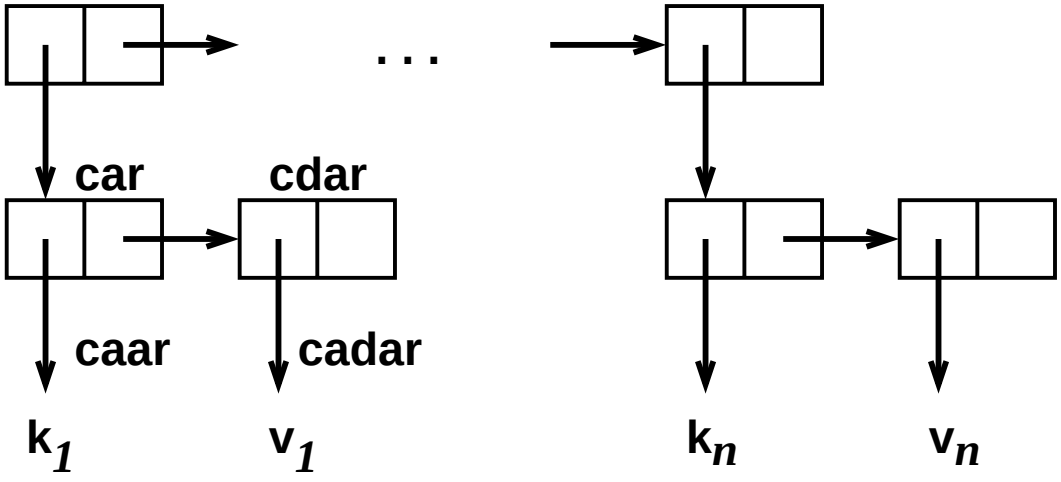
**Implementation: list of key-value pairs**

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




© Copyright 2007 Norman Ramsey. All Rights Reserved.

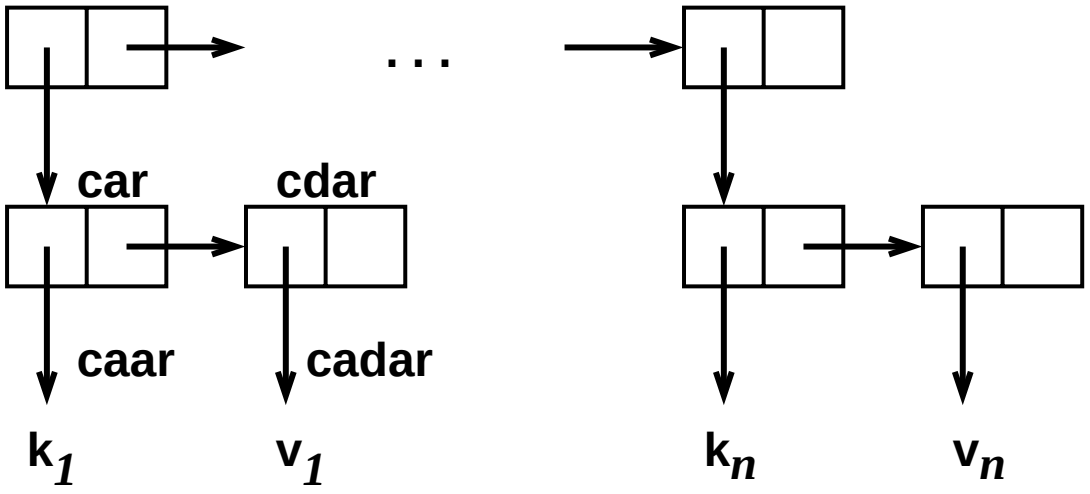
# A-list observer: `find`




```
(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 2007 Norman Ramsey. All Rights Reserved.

# A-list producer: `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 2007 Norman Ramsey. All Rights Reserved.

# A-list example

```
-> (find 'Room '((Course 603)
                 (Instructor Borie) (Room (HO 108))))
(HO 108)
-> (val rb (bind 'Office '(HO 116)
           (bind 'Course 603
           (bind 'Email 'borie@cs '( )))))
((Email borie@cs) (Course 603) (Office (HO 116)))
-> (find 'Office rb)
(HO 116)
-> (find 'Phone rb)
( )
```

**Attributes can be lists, not just symbols**

**"Not found" ≡ "( )"**

© Copyright 2007 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 2007 Norman Ramsey. All Rights Reserved.

# What's new in Scheme: Let

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

"Evaluate $\texttt{e}_1, \ldots, \texttt{e}_n$, bind answers to $\texttt{x}_1, \ldots, \texttt{x}_n$"

- Name intermediate results (improve structure)
- Creates new environment for local use only:

  $\rho\{\texttt{x}_1 \mapsto \texttt{e}_1, \ldots, \texttt{x}_n \mapsto \texttt{e}_n\}$, used to evaluate `e`

Also discussed in the textbook:

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

©Copyright 2007 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 (like C structs)**

© Copyright 2007 Norman Ramsey. All Rights Reserved.

# Solution to all the previous problems

$\lambda$

**pronounced: lambda**

© Copyright 2007 Norman Ramsey. All Rights Reserved.

# What's new in Scheme: Lambda

Taken from Church's $\lambda$-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 2007 Norman Ramsey. All Rights Reserved.

# Examples using Lambda

## *Higher-order functions*

```
-> (val square (lambda (x) (* x x)))
-> (square 3)
9
-> (define twice (f) (lambda (y) (f (f y))))
-> (val 4th_power (twice square))
-> (4th_power 3)
81
-> ((twice square) 3)
81
```

© Copyright 2007 Norman Ramsey. All Rights Reserved.