The Scheme Programming Language

Alan Borning
University of Washington, Seattle
CSE 341, Spring 2000
Stolen from Greg Badros

Scheme philosophy

"Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary."

Scheme

- R5RS = Revised^5 Report on Scheme
  (R4RS is the older standard that Dr. Scheme and Guile use)
  - Only 50 pages describe the whole language
- Descends from Lisp and Algol
- Strong theoretical foundations
- A favorite introductory language in quite a few CS departments

Locally available versions

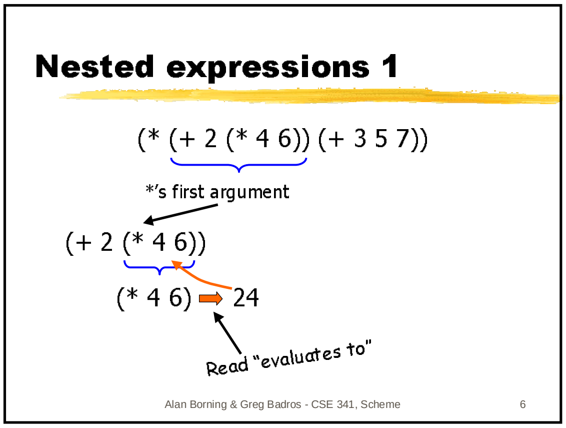
- MIT scheme: on NT machines and instructional unix servers
- DrScheme (from Rice University)
- See the 341 web page for directions on how to use

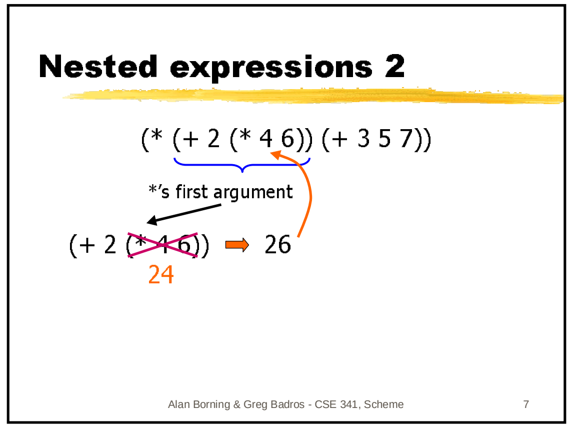
C vs. Scheme expressions

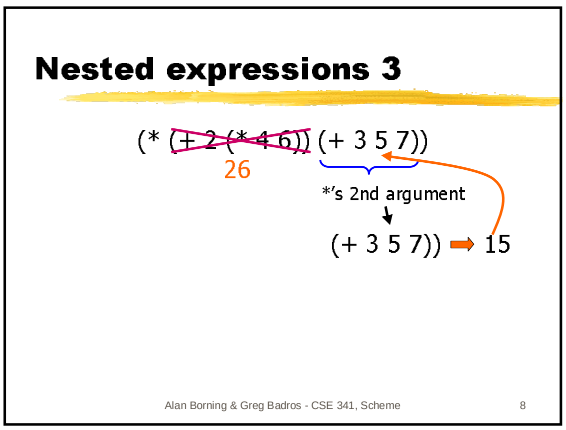
C	Scheme
`factorial(9)`	`(factorial 9)`
`1 + 2`	`(+ 1 2)`
`1 + 3 + 5`	`(+ 1 3 5)`
`(low < x)&&(x < high)`	`(< low x high)`
`f(g(2,-1), 7)`	`(f (g 2 -1) 7)`
`(6+3)*4`	`(* (+ 6 3) 4)`

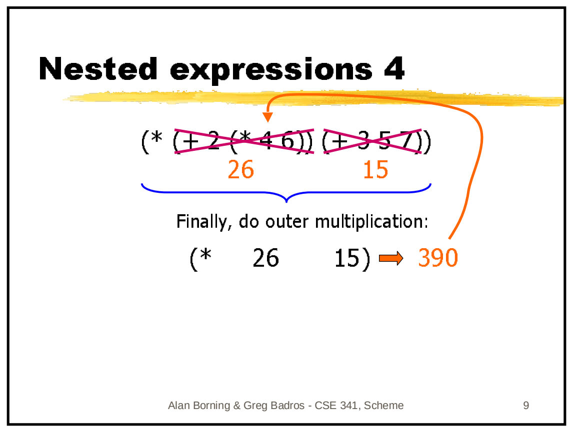
Prefix vs. infix

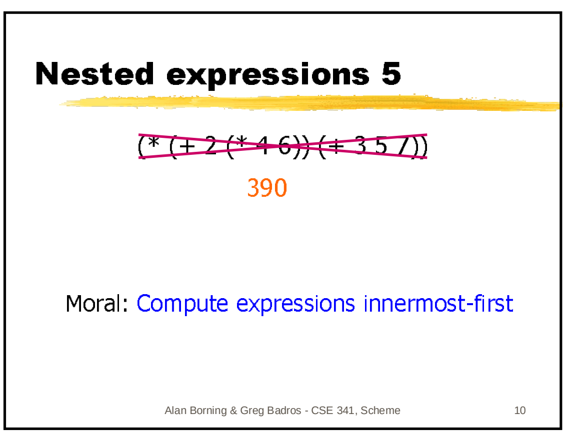
- Infix: 2 * (1 + 2)
  - Group explicitly to override precedence
  - Operator must be repeated when applied to more than 2 arguments
- Prefix: (* 2 (+ 1 2))
  - Lots of parentheses **required**
  - (+ 1 2 3 4 5) ⇒ 15
- **Same execution order!**











Evaluating arguments

- (+ 1 2)
  - 1 ⇒ 1
  - 2 ⇒ 2

So whole expression: (+ 1 2) ⇒ 3

- 1 and 2 are literal numbers that evaluate to themselves!

Alan Borning & Greg Badros - CSE 341, Scheme 11

Types

- Numbers: 1 3.1415 9/5 -2
- Strings: "Hello world"
- Characters: #\A #\space #\newline
- Booleans: #t #f
  (represent TRUE and FALSE)

More types

- Symbols: + a-list x lambda
- Procedures: #<primitive:+>
  #<procedure:factorial>
- Lists: (1 "str" #\g factorial +)
- Vectors: #(1 "str" #\g factorial +)
- Ports for I/O #<input-port>

What's in a symbol?

- Operators are just procedures, so identifiers must allow them
- Valid characters include
  – + * / < > ^ ~ _ % ! ? $: =
- Digits not allowed at start of symbol

Some "literals" evaluate to themselves

- Numbers: 2 ⇒ 2
- Strings: "Hello world" ⇒ "Hello world"
- Characters: #\g ⇒ #\g
- Booleans: #t ⇒ #t
- Vectors: #(a 2 3.14) ⇒ #(a 2 3.14)

Symbols evaluate by variable lookup

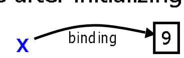
Symbols:
x ⇒ 9
mylist ⇒ ("squid" "clam" "barnacle")
factorial ⇒ #<procedure:factorial>
\+ ⇒ #<primitive:+>

➨ So how do we give variables a value?

define special form

(define x 9)

- define is not a procedure
  it is a "special form" which does not necessarily evaluate all of its arguments
- Allocates space, and binds the symbol x to that space after initializing it to the value:



Lists evaluate by procedure application*

*Except special forms!

Lists:
(factorial 4) ⇒ 24
(+ 1 2) ⇒ 3

("squid" 2 3) ⇒
Error: procedure application:
expected procedure, given: "squid";
arguments were: 2 3

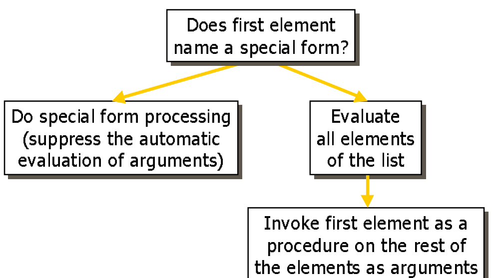
Special forms

Must memorize them!

define	lambda	let, let*, letrec
quote	quasiquote	set!
if	case	cond
begin	do	and, or
let-syntax	letrec-syntax	delay

+ any macros (also must be learned)

List evaluation



Creating a symbol's value

- Symbols evaluate by variable lookup:
  factorial ⇒ #<procedure:factorial>
- How do we enter a symbol as a value?
  e.g., suppose we want:
  x ⇒ octopus
  What do we write?
  (define x ☐)

Suppressing evaluation

(define x octopus)
Error: reference to undefined identifier: octopus

- octopus is evaluated
  Since symbols are evaluated by variable lookup we get the error
- But we want the symbol octopus
- Solution: must *suppress the evaluation* of the symbol octopus

quote special form

(define x (quote octopus))

Gets evaluated to initialize space x is bound to

(quote octopus) ⇒ octopus

Because quote is a special form, octopus does not get evaluated when evaluating the list (quote octopus).

quote just returns its argument un-evaluated!

Quoting

(quote octopus) ⇒ octopus

- ' is shorthand because quote is so useful
  'octopus ⇒ octopus
- Suppose we want a value that is a list?
  (1 2 3) ⇒ ERROR (1 is not a procedure)
  ´(1 2 3) ⇒ (1 2 3)

Forcing evaluation with eval

(+ 1 2 3) ⇒ 6
'(+ 1 2 3) ⇒ (+ 1 2 3)
(eval ´(+ 1 2 3)) ⇒ 6

- eval evaluates its single argument
- eval is implicitly called to evaluate each expression you enter into the repl (read eval print loop)
- **Note**: in MIT Scheme, eval takes a second argument (the environment). So instead write:
  (eval ´(+ 1 2 3) user-initial-environment)

The Lambda Calculus

- Typical function in mathematics:
  f(x,y) = 2x + y
- Alonzo Church's lambda calculus lets us talk about un-named (anonymous) functions:

expression to compute

λx,y: 2x + y   λa,b: 2a + b

lambda — arbitrary names of arguments — same function!

Creating procedures with the lambda special form

(lambda (x y)
  (+ (* 2 x) y)) ⇒ #<procedure>

(☐ 4 5) ⇒ 13

i.e.,

((lambda (x y) (+ (* 2 x) y)) 4 5) ⇒ 13

A moment for syntax

((lambda (x y) (+ (* 2 x) y)) 4 5)
([lambda (x y) (+ (* 2 x) y)] 4 5)
;;; can use [and] as grouping constructs
;;; in some versions of Scheme
;;; (including Dr. Scheme)

Naming a procedure

- No new rule!
  (define f [lambda (x y) (+ (* 2 x) y)])
  (f 4 5) ⇒ 13

  f —binding→ #<procedure>

- Remember list evaluation rule:
  we evaluate f by doing variable lookup!

Shorthand for procedure definition

(define f [lambda (x y) (+ (* 2 x) y)])

⇕

(define (f x y)
(+ (* 2 x) y))

Procedures vs. variables

- (define f 3)
  f ⇒ 3
  (f) ⇒ ERROR
- (define (f) 3)
  f ⇒ #<procedure:f>
  (f) ⇒ 3

Parentheses around name make this the same as:
(define f (lambda () 3))
which is a procedure that, when called, returns 3.

Conditionals: if special form

(if *TEST* *TRUE-EXPR* *FALSE-EXPR*)

- if is an expression—evaluates to either TRUE-EXPR or FALSE-EXPR depending on the value of TEST
- (if #t 3 4) ⇒ 3
- (if (> 5 6) 3 4) ⇒ 4

C vs. Scheme

C

```
if (operator == PLUS)
  return add(x,y);
else
  return subtract(x,y);
```

Scheme

```
(if (eq? operator 'PLUS)
    (add x y)
    (subtract x y))
```


C vs. Scheme

C

```
if (operator == PLUS)
  return add(x,y);
else
  return subtract(x,y);
```

Repeated arguments

Scheme

```
([if (eq? operator 'PLUS)
    add
    subtract]
 x y)
```

Evaluates to either #<procedure:subtract> or #<procedure:add>

eq? procedure tests for identity equality

- (eq? 'foo 'foo) ⇒ #t
- (eq? 'foo 'bar) ⇒ #f
- Procedures ending in ? are predicates—they return a boolean value, #t or #f
- Symbols are only useful as identifiers and for eq? comparison testing

Recursion

(define (factorial n)
  (if (<= n 1)
    1 ; base case
    [* n (factorial (– n 1))])) ; inductive step

Procedure is calling itself

Linear recursive process

(factorial 4)
(* 4 (factorial 3))
(* 4 (* 3 (factorial 2)))
(* 4 (* 3 (* 2 (factorial 1))))
(* 4 (* 3 (* 2 1)))
(* 4 (* 3 2))
(* 4 6)
24

expansion

Base case hit here

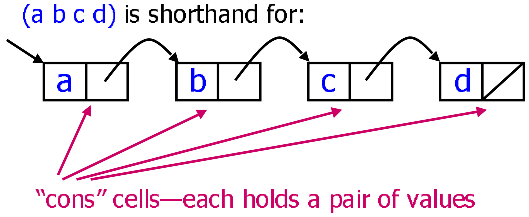
contraction

Adapted from: Abelson & Sussman, Fig. 1.3

Lists are made of *cons* cells

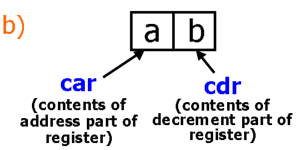
- Lists are a recursive data structure!

(a b c d) is shorthand for:



cons cells and the cons procedure

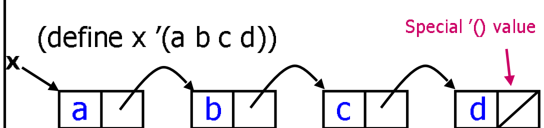
- (cons 'a 'b) ⇒ (a . b)



Axioms:
(car (cons α β)) ⇒ α
(cdr (cons α β)) ⇒ β

List syntax shorthand

(define x '(a b c d))



(cons 'a (cons 'b (cons 'c (cons 'd '()))))
(car x) ⇒ a
(cdr x) ⇒ (b c d) ; another list!

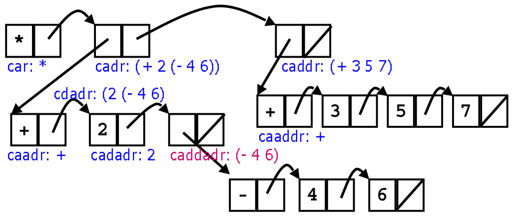
car, cdr, and friends

(define x '(a b c d))
(car (cdr x)) ⇒ b   (cadr x) ⇒ b
(car (cdr (cdr x))) ⇒ c   (caddr x) ⇒ c

All combinations of up to four a/d characters are standard; e.g.:
caaaar, cddddr, cdadar, cdar, cadadr, etc.

Nested lists

(* (+ 2 (- 4 6)) (+ 3 5 7))



Do not try this at home

```
(define (caddadr a-list)
  (car (cddadr a-list)))
```

- Rely instead on recursive processing of the data structure

our-list-ref procedure

```
(define (our-list-ref items n)
  "Return element N of list ITEMS."
  (if (= n 0)
      (car items)
      (our-list-ref (cdr items) (- n 1))))
```


our-list-ref trace Linear iterative process

```
(our-list-ref '(a b c d) 3)
(our-list-ref '(b c d) 2)
(our-list-ref '(c d) 1)
(our-list-ref '(d) 0)
d
```

base case n=0, so return (car '(d))

- No expansion and contraction!

Contrast the inductive steps

(* n [factorial (– n 1)])

[our-list-ref (cdr items) (- n 1)]

- factorial recurses and the result is used in further computation:
  (* n □)

Tail-recursion

our-list-ref is "tail-recursive"

- Recursive call's return value is the final result
- Thus, the recursion can be replaced with iteration (i.e., a "goto" the top of the procedure with variables re-bound)
- Standard Scheme implementations are **required** to eliminate tail recursions!

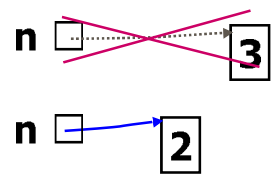
our-list-ref tail recursion

```
(define (our-list-ref items n)
  (if (= r
    (ca
    (our-list-ref (cdr items) (- n 1))))
```

Bind items to (cdr items),

Bind n to (- n 1)

Re-binding is NOT assignment



Thinking recursively

- Describe the answer—
  - do not worry about how to compute it directly
  - decompose the problem into a simpler problem
  - state the incremental step from the simpler problem to the general problem
- Consider the base case(s)—
  - get an intuition using the base cases
  - test solution on base cases to ensure they give the right answers

Recursion is about being a smart-aleck!

- What's the sum of the first 9 integers?
  ⇨9 + sum of the first 8 integers!
- What is the length of a list?
  ⇨1 + length of the rest of the list!

Recursion templates

- Augmenting recursion: build up the answer bit by bit
- Conditional augmentation (variant of augmenting recursion)
- Simultaneous recursion on several variables
- Tail recursion

Augmenting recursion examples

- already saw one: factorial

```
;; double each number in a list (map –
;; defined later – is a cleaner solution)
(define (double-each s)
  (if (null? s) ()
      (cons (* 2 (car s))
            (double-each (cdr s)))))
```

More augumenting recursion

```
;; simple version of built-in append function

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


Reducing functions

- Yet another kind of augumenting recursion – reduce a list of elements to a single element

```
;; sum the elements of a list
(define (sumlist x)
  (if (null? x) 0
     (+ (car x) (sumlist (cdr x)))))
```


Tail recursive version of factorial

```
(define (factorial n)
  (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
  (if (> counter max-count)
     product
     [fact-iter   (* counter product)
                  (+ counter 1)
                  max-count]))
```


Tail recursive factorial trace

(factorial 4) ;;	product	counter	max
(fact-iter	1	1	4)
(fact-iter	1	2	4)
(fact-iter	2	3	4)
(fact-iter	6	4	4)
(fact-iter	24	5	4)

counter > max, so terminate

Conditional augmentation

```
(define (positive-numbers x)
  (cond
    ((null? x) ())
    ((> (car x) 0)
      (cons (car x)
        (positive-numbers (cdr x))))
    (else (positive-numbers (cdr x)))))
```

("filter" will provide a cleaner solution)

Insertion sort

```
(define (insert x s)
  (cond
    ((null? s) (list x))
    ((< x (car s)) (cons x s))
    (else (cons (car s) (insert x (cdr s))))))

 (define (isort s)
  (if (null? s) ()
  (insert (car s) (isort (cdr s)))))
```

Nested procedure defines

```
(define (factorial n)
  [define (iter product counter)
     (if    (> counter n)
            product
            [iter (* counter product)
                  (+ counter 1)])]
  (iter 1 1))
```


Factoring out common sub-expressions

f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)

a = 1+xy
b = 1-y
f(x,y) = xa^2 + yb + ab

let special form

a = 1+xy
b = 1-y
f(x,y) = xa^2 + yb + ab

```
(define (f x y)
  (let ([a (+ 1 (* x y))]
        [b (- 1 y)])
    (+ [* x (square a)] [* y b] [* a b])))
```

list of bindings

expression to evaluate and return

Scope is visibility

```
(define (f x y)
  (let ([a (+ 1 (* x y))]
        [b (- 1 y)])
    (+ [* x (square a)] [* y b] [* a b])))
```

Scope of a and b only in this block

a ⇒ ERROR (a not visible at "top level")

let bindings happen in parallel

(define x 'a)
(define y 'b)
(list x y) ⇒ (a b)
(let ([x y] [y x]) (list x y)) ⇒ (b a)

- Remember, it is still **not** assignment— only re-binding

Bad let bindings

(let ([x 1] [y (+ x 1)]) (list x y))

Reference to undefined identifier

- So what if we do want y to be computed in terms of x?

let* special form

(let* ([x 1] [y (+ x 1)]) (list x y)) ⇒ (1 2)

Syntactic sugar for nested lets

(let ([x 1])
  (let ([y (+ x 1)]) (list x y)))

More about conditionals: cond special form

(if *TEST EXPR-TRUE EXPR-FALSE*)

- cond evalutates to EXPR-K, where K is the lowest of the TEST-Ks that evaluates to #t

(cond [*TEST-1 EXPR-1*]
      [*TEST-2 EXPR-2*]
      …
      [*TEST-N EXPR-N*]
      [else *ELSE-EXPR*]) ; or [#t *ELSE-EXPR*]

cond example

```
(define (sign number)
  (cond [(> number 0) 'positive]
        [(< number 0) 'negative]
        [else 'zero]))
```

(sign -1) ⇒ negative
(sign 0) ⇒ zero
(sign 'a) ⇒ ERROR!

Short-circuiting and, or special forms

(and (> 1 0) (> 2 3) (/ 5 0)) ⇒ #f

#f, so never evaluates this

(or (> 0 1) (> 2 1) (/ 5 0)) ⇒ #t

#t, so never evaluates this

Boolean values & and, or

- All values except #f are treated as #t in conditionals
- and evaluates to the last value in sequence if it succeeds (not necessarily #t)
  (and (+ 2 3) (- 3 1)) ⇒ 2
- or evaluates to the first true value in sequence if it succeeds (not necessarily #t)
  (or (> 4 5) 'a (/ 5 0)) ⇒ a

Procedures are first-class values

- lambda special form returns a procedure
- Procedures can take procedures as arguments
- Procedures can return procedures
- Procedures are treated no differently from other kinds of values
  (e.g., numbers, lists, strings, symbols, etc.)

map, a higher-order function

(define (negation x) (- x))

(map negation '(1 2 -5 7)) ⇒ (-1 -2 5 -7)

(map + '(1 2 3) '(4 5 6)) ⇒ (5 7 9)

the list of return values

two lists, so give two arguments to each + call

(map [lambda (x) (+ x 2)]
  '(-2 0 2)) ⇒ 0 2 4

- A higher-order function is one that operates on other functions

Filter procedure

(filter number? '(a 4 "testing" +)) ⇒ (4)

(filter [lambda (x) (>= x 0)]
  '(-2 3 -1 1 4)) ⇒ (3 1 4)

(filter [lambda (x) (>= x 2)]
  '(-2 3 -1 1 4)) ⇒ (3 4)

Typechecking predicates

(number? 4) ⇒ #t

(string? "foo") ⇒ #t

(symbol? 'foo) ⇒ #t

(pair? (cons 'a 'b)) ⇒ #t

(pair? '(a b c d e f)) ⇒ #t

(pair? '(1)) ⇒ #t

(pair? '()) ⇒ #f   (pair? 6) ⇒ #f

(null? '()) ⇒ #t   (null? 0) ⇒ #f

All names end in ?

Procedure factories

(filter [lambda (x) (>= x 0)]
  '(-2 3 -1 1 4)) ⇒ (3 1 4)

(filter [lambda (x) (>= x 2)]
  '(-2 3 -1 1 4)) ⇒ (3 4)

(define (greater-than-n?? n)
  (lambda (x) (>= x n)))

Building procedures

(define (greater-than-n?? n)
  (lambda (x) (>= x n)))

greater-than-n?? ⇒ #<procedure>

(greater-than-n?? 2) ⇒ #<procedure>

((greater-than-n?? 2) 3) ⇒ #t

(define greater-than-2 (greater-than-n?? 2))

(greater-than-2 3) ⇒ #t

A factory procedure

predicate

Dr. Scheme Graphics

(define v (open-viewport ...))

(define red (make-rgb 1 0 0))

((draw-pixel v) (make-posn 100 120) red)

Returns a procedure that is specialized for drawing a pixel on the viewport v

(define draw-on-v (draw-pixel v))

(draw-on-v (make-posn 100 120) red)

lambdas and closures

(define (greater-than-n?? n)
  (lambda (x) (>= x n)))

(lambda (x) (>= x n))

- which n? Looks like an undefined reference!
- lambda "remembers" the value n had when the procedure was created—it "closes" over its environment and gives a "closure"

Alan Borning & Greg Badros - CSE 341, Scheme 78

lambdas and their environments

(define (greater-than-n-by-k?? n k)
  (lambda (x) (>= x (+ n k))))
(greater-than-n-by-k?? 2 5) ⇒

λx: x >= n +k

environment

n ↔ 2
k ↔ 5

Alan Borning & Greg Badros - CSE 341, Scheme 79

Free variables and Lexical Scoping

(define (greater-than-n-by-k?? n k)
  (lambda (x) (>= x (+ n k))))
(define pred (greater-than-n-by-k?? 2 5))

(let ([n 0] [k 0])
  (pred 1)) ⇒ #f

These bindings are unused

Alan Borning & Greg Badros - CSE 341, Scheme 80

Dynamic Scoping

These args are unused

(define (greater-than-n-by-k?? n k)
  (lambda (x) (>= x (+ n k))))
(define pred (greater-than-n-by-k?? 2 5))

(let ([n 0] [k 0])
  (pred 1)) ⇒ #t

Alan Borning & Greg Badros - CSE 341, Scheme 81

Lexical vs. Dynamic Scope

- Lexical
  - Fewer surprises
  - Easier to compile—all mentions of a variable refer to the same variable
- Dynamic
  - Historical LISPs—considered an implementation bug by McCarthy
  - Can result in confusing "name capture"
    (Sometimes (but rarely) this is what you want)

Alan Borning & Greg Badros - CSE 341, Scheme 82

Lexical scope and variable hiding

```
(define x 4)
(display x)            ; writes 4
(let ([x "hello"])
  (display x)          ; writes "hello"
  (let ([x '(1 2)])
     (display x))      ; writes (1 2)
  (display x))         ; writes "hello" again
```

Alan Borning & Greg Badros - CSE 341, Scheme 83

When the arguments don't fit

- Suppose we have args ⇒ (1 2 3) and we want to compute the sum:
  (+ args) ⇒ Error; expects args of type number
- We need a list with + as first element, then the elements of args: (+ 1 2 3)
- Create the list and evaluate it:
  (cons + args) ⇒ (+ 1 2 3)
  (eval (cons + args)) ⇒ 6

Alan Borning & Greg Badros - CSE 341, Scheme 84

A more direct approach: the apply procedure

(apply + args)

Procedure   Arguments to use

(apply [lambda (x y) (+ x y)] '(1 2))

Alan Borning & Greg Badros - CSE 341, Scheme 85

Procedure arity

- (define (proc0) 'foo)
  (arity proc0) ⇒ 0
- (define (proc1 arg) 'foo)
  (arity proc1) ⇒ 1
- (define (procn . args) 'foo))
  (arity procn) ⇒ #(struct arity-at-least 0)
- (define (proc reqd1 reqd2 . rest) 'foo))
  (arity proc) ⇒ #(struct arity-at-least 2)

Alan Borning & Greg Badros - CSE 341, Scheme 86

Rest arguments

(define (skip-first ignore . rest) rest)
(skip-first 1 2 3) ⇒ (2 3)
(define (skip-first . args) (cdr args))

(define skip-first (lambda (. args) (cdr args))
Error! Can't put "." as first element in list

(define skip-first (lambda args (cdr args)))

Alan Borning & Greg Badros - CSE 341, Scheme 87

Controlling evaluation inside of lists

Evaluated
Unevaluated

(define a "Hello")
'(1 a (+ 2 3)) ⇒ (1 a (+ 2 3))
(list 1 a (+ 2 3)) ⇒ (1 "Hello" 5)
(list 1 'a (+ 2 3)) ⇒ (1 a 5)
\`(1 a (+ 2 3)) ⇒ (1 a (+ 2 3))
\`(1 a ,(+ 2 3)) ⇒ (1 a 5)
\`(1 ,a ,(+ 2 3)) ⇒ (1 "Hello" 5)

Alan Borning & Greg Badros - CSE 341, Scheme 88

quasiquote and unquote

- (quasiquote (α)) ⇔ \`(α)
  Only evaluate parts of α that are unquoted
- (unquote β) ⇔ ,β

(quasiquote (1 a (unquote (+ 2 3)))
⇒ (1 a 5)

Alan Borning & Greg Badros - CSE 341, Scheme 89

Forcing and suppressing evaluation are fundamental

- Not Scheme's fault—
  it just gives you finer control!
- C/C++ lack such complexity in the general case because they are more restricted languages
- Analogous to dereferencing or taking address of variables when dealing with pointers

Comparisons

- C/C++:
  (4 == x) && ('\n' == ch)
- Scheme:
  (and (= 4 x) (char=? ch #\newline))
- Other comparison procedures:
  string=?
  eq? eqv? equal?

eq?, eqv?, equal?

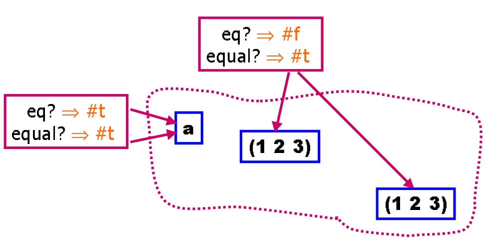
(eq? 'a 'a) ⇒ #t (eq? 'a 'b) ⇒ #f
(eq? 9 9) ⇒ #t (eq? 1234567890 1234567890) ⇒ #f
(eqv? 9 9) ⇒ #t (eqv? 1234567890 1234567890) ⇒ #t
(eq? "foo" "foo") ⇒ #f
(eqv? "foo" "foo") ⇒ #f
(equal? "foo" "foo") ⇒ #t
(eq? (list 1 2 3) (list 1 2 3)) ⇒ #f
(equal? (list 1 2 3) (list 1 2 3)) ⇒ #t

More eq?, eqv?, equal?

(define a "foo")
(eq? a "foo") ⇒ unspecified (likely #f)
(eq? a a) ⇒ #t
(equal? a "foo") ⇒ #t

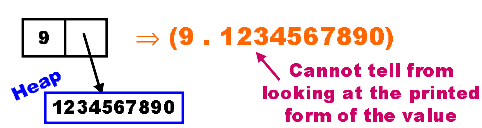
- Values can look the same, but be different objects
- eq? tests for object-identity equality
  - returns #t only when they are same object
  - symbols are "intern"ed to make them the same

Objects/values in the Scheme Heap



Do not use eq? for numbers

- Some numbers compare eq? to themselves, others do not
- Answer depends on how Scheme stores numbers (implementation-dependent):



Making new lists

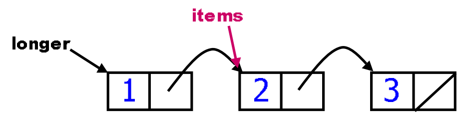
(define items '(2 3))

- Already saw how to get a list with a new first element:
  (cons 1 items) ⇒ (1 2 3)
  items ⇒ (2 3) ; items is unchanged
- How can we get (2 3 4)?
  (cons items 4) ⇒ ((2 3) . 4) ; wrong!
  (append items 4) ⇒ (2 3 . 4) ; closer…
  (append items '(4)) ⇒ (2 3 4) ; Yes!

Alan Borning & Greg Badros - CSE 341, Scheme 96

Sharing of list structure

(define items '(2 3))
(define longer (cons 1 items))

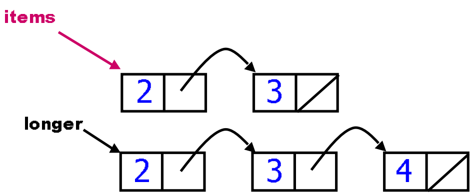


longer ⇒ (1 2 3)
items ⇒ (2 3) (eq? items (cdr longer)) ⇒ #t

Alan Borning & Greg Badros - CSE 341, Scheme 97

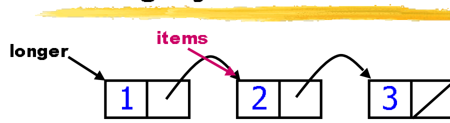
append must duplicate the list

(define items '(2 3))
(define longer (append items '(4)))

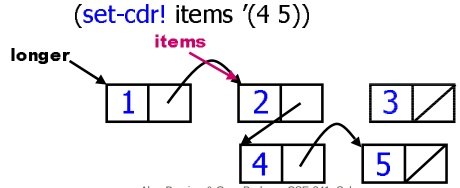


Alan Borning & Greg Badros - CSE 341, Scheme 98

List surgery



(set-cdr! items '(4 5))



Alan Borning & Greg Badros - CSE 341, Scheme 99

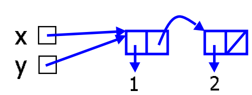
set-cdr! and set-car!

- Change what a cons cell contains
- They side-effect the values passed in!
- Exclamation point (!) tells you that an argument is being altered
- Non-functional feature—useful mostly for efficiency

Alan Borning & Greg Badros - CSE 341, Scheme 100

Assignment

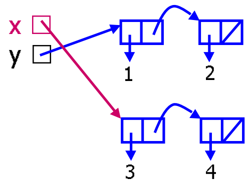
(define x '(1 2))
(define y x)
(eq? x y) ⇒ #t
(set! x '(3 4))



Alan Borning & Greg Badros - CSE 341, Scheme 101

set! procedure

(define x '(1 2))
(define y x)
(eq? x y) ⇒ #t
(set! x '(3 4))
x ⇒ (3 4)
y ⇒ (1 2)



Other side-effects

- display procedure
  input/output in general
  (display 1) ⇒ #unspecified
- (define x #\A) ⇒ #unspecified
- (/ 5 0) ⇒ Run-time error
- (iconify-window (get-window))
  ⇒ #unspecified

Sequencing and the begin special form

```
(define hw (lambda ()
              (display "hello ")
              (display "world")))
(hw) ⇒ #unspecified ; prints "hello world"
(begin
  (display "hello ") 2 "foo" (display "world")
  "last"))) ⇒ "last" ; prints "hello world"
```


Side effects and Scwm

- Scwm, the Scheme Constraints Window Manager, relies on side-effects to do much of its work
- (iconify-window (get-window))

```
(for-each iconify-window
  (list-all-windows [lambda (w)
                        (list-all-windows #:only
                (string=? (window-class w) "XTerm"))]))
```


for-each procedure

- Similar to map, but does not return list of returned values from the applications
- The applications are only used for their side effects; e.g.
  - iconifying a window
  - displaying a string
- Like begin, for-each is unnecessary if writing in a purely functional style

Commenting example

```
;;; Code by Greg J. Badros
(define (foo . args)
  "Describe procedure foo here."
  ;; this comment is indented but a full line
  (for-each display args)) ; write out args
```

Final close parentheses do **not** go on separate lines

Commenting style

- ;;; comments flush-left on own line
- ;; comments indented on own line
- ; short comments after code segment
- Use editor's mode-specific indentation facilities
- Use editor that matches parentheses for you

Memory management

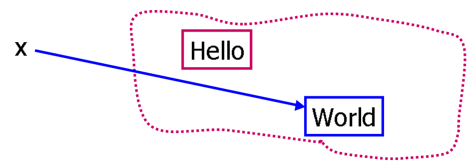
- Scheme has no delete
- Lots of new values getting constructed on the scheme heap
- Where do all the values go?
  - (define x "Hello")
  - (define x "World")
  - ⇨ What happens to the value "Hello"?

Scheme memory model

- Provides an abstract machine that has infinite memory
- Implementation can choose to re-use real (physical) memory whenever it can without breaking the abstraction
- Process used: "Garbage collection"
  Re-claims memory consumed by values that are no longer usable (i.e., that are garbage)

A view of garbage

(define x "Hello")
(define x "World")



Mark and sweep garbage collection

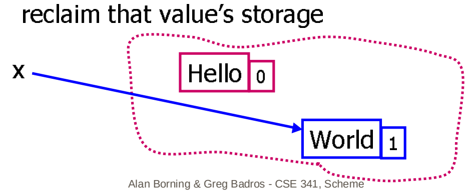
Simple GC algorithm is "mark and sweep"

mark:
- for each top-level binding
  - mark the values that the variables are bound to; and
  - recursively mark the memory that those values use

sweep:
- reclaim all not-marked memory

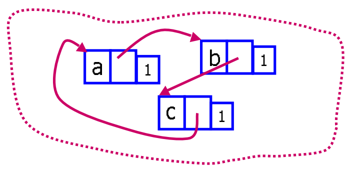
Reference counting

- Let each value keep track of how many other values or bindings point at it
- When reference count drops to zero, reclaim that value's storage



Reference counting never reclaims cycles

- Nothing is bound to the cyclic list, (a b c a b c a b c …), but counts are still > 0



Alan Borning & Greg Badros - CSE 341, Scheme 114

Mark and sweep vs. Reference counting

- Mark and sweep can take a while
  - Sometimes noticeable pauses in interaction
  - Various improvements on the basic algorithm can help dramatically (including generation-based garbage collection, incremental garbage collection, etc)
  - No extra overhead managing reference counts
- Reference counting
  - Incremental, so no noticeable pauses, but extra overhead throughout computation
  - Cyclic structures are not reclaimed

Alan Borning & Greg Badros - CSE 341, Scheme 115