

CSC 533: Organization of Programming Languages

Spring 2007

Advanced Scheme programming

- memory management: structure sharing, garbage collection
- structuring data: association list, trees
- let expressions
- non-functional features: set!, read, display, begin
- OOP in Scheme

1

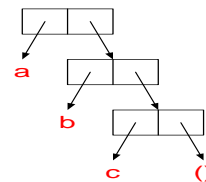
Memory management in Scheme

all data is dynamically allocated (heap-based)

- variables (from function definitions, let-expressions) may be stored on stack

underlying lists is the dotted-pair structure

$(a\ b\ c) \equiv (a . (b . (c . ())))$



this structure demonstrates

- non-contiguous nature of lists (non-linear linked-lists)
- behavior of primitive operations (car, cdr, cons)

$(\text{car } '(a\ b\ c)) \equiv (\text{car } '(a . (b . (c . ()))))) \rightarrow a$

$(\text{cdr } '(a\ b\ c)) \equiv (\text{cdr } '(a . (b . (c . ()))))) \rightarrow (b . (c . ())) \equiv (b\ c)$

$(\text{cons } 'x\ '(a\ b\ c)) \equiv (\text{cons } 'x\ '(a . (b . (c . ())))))$
 $\rightarrow (x . (a . (b . (c . ()))))) \equiv (x\ a\ b\ c)$

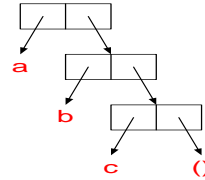
2

Structure sharing

since destructive assignments are rare, Scheme makes extensive use of structure-sharing

```
(define (my-length lst)
  (if (null? lst)
      0
      (+ 1 (my-length (cdr lst)))))
```

```
> (my-length '(a b c))
3
```



- each recursive call shares a part of the list
- other code that uses a, b, c or () can share as well

problems caused by destructive assignments? solutions?

3

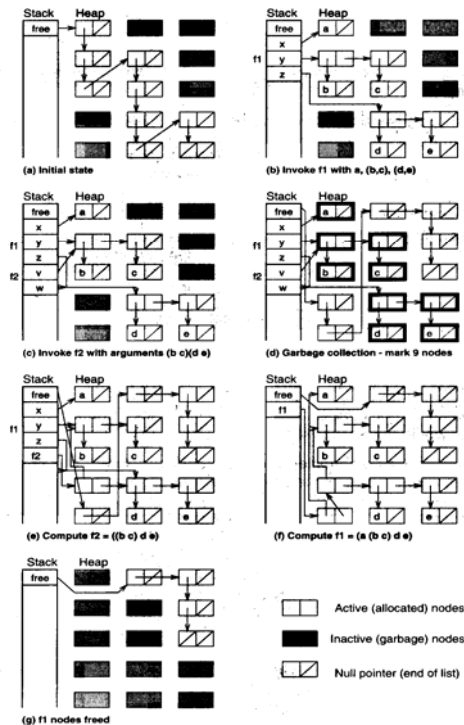
Garbage collection

garbage collection is used to reclaim heap memory

```
(define (f1 x y z)
  (cons x (f2 y z)))
```

```
(define (f2 v w)
  (cons v w))
```

```
> (f1 'a '(b c) '(d e))
(a (b c) d e)
```



4

Structuring data

an association list is a list of "records"

- each record is a list of related information, keyed by the first field
- i.e., a Map

```
(define NAMES '((Smith Pat Q)
                (Jones Chris J)
                (Walker Kelly T)
                (Thompson Shelly P)))
```

note: can use define to
create "global constants"
(for convenience)

- can access the record (sublist) for a particular entry using `assoc`

```
> (assoc 'Smith NAMES)      > (assoc 'Walker NAMES)
(Smith Pat Q)              (Walker Kelly T)
```

- `assoc` traverses the association list, checks the `car` of each sublist

```
(define (my-assoc key assoc-list)
  (cond ((null? assoc-list) #f)
        ((equal? key (caar assoc-list)) (car assoc-list))
        (else (my-assoc key (cdr assoc-list)))))
```

5

Association lists

to access structured data,

- store in an association list with search key first
- access via the search key (using `assoc`)
- use `car/cdr` to select the desired information from the returned record

```
(define MENU '((bean-burger 2.99)
              (tofu-dog 2.49)
              (fries 0.99)
              (medium-soda 0.79)
              (large-soda 0.99)))
```

```
> (cadr (assoc 'fries MENU))
0.99
```

```
> (cadr (assoc 'tofu-dog MENU))
2.49
```

```
(define (price item)
  (cadr (assoc item MENU)))
```

6

assoc example

consider a more general problem: determine price for an entire meal

- represent the meal order as a list of items,
e.g., (tofu-dog fries large-soda)
- use recursion to traverse the meal list, add up price of each item

```
(define (meal-price meal)
  (if (null? meal)
      0.0
      (+ (price (car meal)) (meal-price (cdr meal)))))
```

- alternatively, could use map & apply

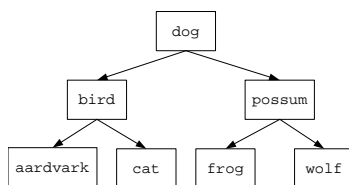
```
(define (meal-price meal)
  (apply + (map price meal)))
```

7

Non-linear data structures

note: can represent non-linear structures using lists

e.g. trees



```
(dog
 (bird (aardvark () ()) (cat () ()))
 (possum (frog () ()) (wolf () ())))
```

- empty tree is represented by the empty list: ()
- non-empty tree is represented as a list: (ROOT LEFT-SUBTREE RIGHT-SUBTREE)
- can access the the tree efficiently

```
(car TREE)    → ROOT
(cadr TREE)   → LEFT-SUBTREE
(caddr TREE)  → RIGHT-SUBTREE
```

8

Tree routines

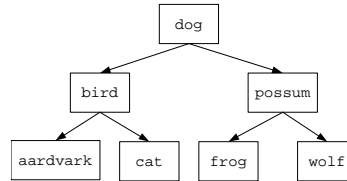
```
(define TREE1
  ' (dog
    (bird (aardvark () ()) (cat () ()))
    (possum (frog () ()) (wolf () ())))
```

```
(define (empty? tree)
  (null? tree))
```

```
(define (root tree)
  (if (empty? tree)
      'ERROR
      (car tree)))
```

```
(define (left-subtree tree)
  (if (empty? tree)
      'ERROR
      (cadr tree)))
```

```
(define (right-subtree tree)
  (if (empty? tree)
      'ERROR
      (caddr tree)))
```



9

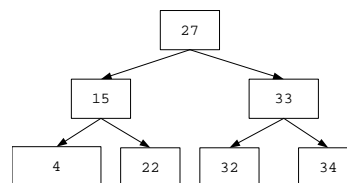
Tree searching

note: can access root & either subtree in constant time

→ can implement binary search trees with $O(\log N)$ access

binary search tree: for each node, all values in left subtree are \leq value at node
all values in right subtree are $>$ value at node

```
(define (bst-contains? bstree sym)
  (cond ((empty? tree) #f)
        ((= (root tree) sym) #t)
        ((> (root tree) sym) (bst-contains? (left-subtree tree) sym))
        (else (bst-contains? (right-subtree tree) sym))))
```



note: recursive nature of trees makes them ideal for recursive traversals

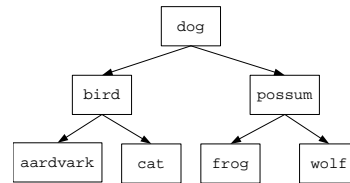
10

Tree traversal

```
(define (pre-order tree)
  (if (null? tree)
      '()
      (append (list (car tree))
              (pre-order (cadr tree))
              (pre-order (caddr tree)))))
```

```
(define (in-order tree)
  (if (null? tree)
      '()
      (append (in-order (cadr tree))
              (list (car tree))
              (in-order (caddr tree)))))
```

```
(define (post-order tree)
  (if (null? tree)
      '()
      (append (post-order (cadr tree))
              (post-order (caddr tree))
              (list (car tree)))))
```



- pre-order traversal?
- in-order traversal?
- post-order traversal?

11

Finally, variables!

Scheme does provide for variables and destructive assignments

```
> (define x 4)           define creates and initializes a variable
>
> x
4
> (set! x (+ x 1))      set! updates a variable
>
> x
5
```

since Scheme is statically scoped, can have global variables

YUCK: destroys functional model, messes up structure sharing

12

Let expression

fortunately, Scheme provides a "clean" mechanism for creating variables to store (immutable) values

```
(let ((VAR1 VALUE1)
      (VAR2 VALUE2)
      . . .
      (VARn VALUEn))
  EXPRESSION)
```

let expression introduces a new environment with variables (i.e., a block)
good for naming a value (don't need `set!`!)

a `let` expression has the same effect as creating a help function & passing value

as long as destructive assignments are not used, the functional model is preserved

- in particular, structure sharing is safe

```
(let ((x 5) (y 10))
  (let (z (x + y))
    )
  )
```

environment where z = 15

environment where x = 5 and y = 10

13

Craps example

consider a game of craps:

- if first roll is 7, then WINNER
- if first roll is 2 or 12, then LOSER
- if neither, then first roll is "point"
 - keep rolling until get 7 (LOSER) or point (WINNER)

```
(define (craps)
  (define (roll-until point)
    (let ((next-roll (+ (random 6) (random 6) 2)))
      (cond ((= next-roll 7) 'LOSER)
            ((= next-roll point) 'WINNER)
            (else (roll-until point)))))
  (let ((roll (+ (random 6) (random 6) 2)))
    (cond ((or (= roll 2) (= roll 12)) 'LOSER)
          ((= roll 7) 'WINNER)
          (else (roll-until roll)))))
```

14

Craps example with I/O

to see the results of the rolls, could append rolls in a list and return

or, bite the bullet and use non-functional features

- `display` displays S-expr (`newline` yields carriage return)
- `read` reads S-expr from input
- `begin` provides sequencing (for side effects)

```
(define (craps)
  (define (roll-until point)
    (let ((next-roll (+ (random 6) (random 6) 2)))
      (begin (display "Roll: ") (display next-roll) (newline)
             (cond ((= next-roll 7) 'LOSER)
                   ((= next-roll point) 'WINNER)
                   (else (roll-until point))))))
  (let ((roll (+ (random 6) (random 6) 2)))
    (begin (display "Point: ") (display roll) (newline)
           (cond ((or (= roll 2) (= roll 12)) 'LOSER)
                 ((= roll 7) 'WINNER)
                 (else (roll-until roll))))))
```

15

OOP in Scheme

map & apply showed that functions are first-class objects in Scheme

- can be passed as inputs to other functions
- can be returned as the output of other functions

can use this feature to provide object-oriented programming

example: bank account

data:	account balance
operations:	initialize with some amount
	deposit some amount
	withdraw some amount

16

Naïve (imperative) solution

- use global variable to represent the balance
- initialize and update the balance using `set!`

```
(define balance 100)

(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount)) balance)
      "Insufficient funds"))

(define (deposit amount)
  (begin (set! balance (+ balance amount)) balance))

> (withdraw 25)
75
> (deposit 50)
125
> (withdraw 200)
"Insufficient funds"
```

DRAWBACKS

- no encapsulation
- no data hiding
- not easily extended to multiple accounts

17

OOP behavior

following OOP principles, would like the following behavior

<code>(define savings (account 100))</code>	creates an account called savings, initialized to \$100
<code>(savings 'deposit 50)</code>	updates the savings account by depositing \$50
<code>(savings 'withdraw 50)</code>	updates the savings account by withdrawing \$50

want balance to be inaccessible except through deposit & withdraw

SOLUTION: make an account object be a *function*

- contains the balance as local data (as a parameter)
- recognizes deposit and withdraw commands as input

18

OOP solution

```
(define (account balance)

  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
        "Insufficient funds"))

  (define (deposit amount)
    (begin (set! balance (+ balance amount)) balance))

  (define (menu message arg)
    (if (member message '(withdraw deposit))
        ((eval message) arg)
        (else "Unknown operation")))

  menu)
```

```
(define savings (account 100))
  ↓
(define (menu message arg)
  (if (member message '(withdraw deposit))
      ((eval message) arg)
      (else "Unknown operation")))
```

since the returned function is in the scope of the balance parameter, that value is maintained along with the function

(savings 'deposit 50) applies the menu function to the arguments

19

OOP analysis

this implementation provides

- encapsulation: balance & operations are grouped together
- data hiding: balance is hidden in an account object, accessible via ops

can have multiple objects – each has its own private balance

```
(define checking (account 100))
(define savings (account 500))

(checking 'withdraw 50)
(savings 'deposit 50)
```

note: this notation can become a bit awkward

- most Schemes provide an OOP library that insulates the user from details
- allows more natural definitions, inheritance, . . .

20

Scheme recap

simple & orthogonal

- code & data are S-expressions
- computation via function application, composition

symbolic & list-oriented

- can manipulate words, flexible & abstract data structure
- efficient (but less flexible) data structures are available

functional style is very natural

- supports imperative & OOP styles if desired

first-class functions

- leads to abstract, general functions (e.g., map, apply)
- code = data → flexibility

memory management is hidden

- dynamic allocation with structure sharing, garbage collection
- tail-recursion optimization is required