CSC 533: Programming Languages

Spring 2018

Scheme structures

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

1

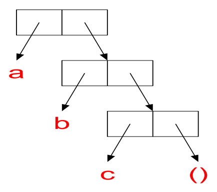
Memory management in Scheme

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)  ≡  (a . (b . (c . ())))
```



this structure demonstrates

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

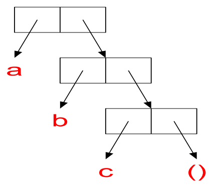
```
(car '(a b c)) ≡ (car '(a . (b . (c . ())))) → a

(cdr '(a b c)) ≡ (cdr '(a . (b . (c . ())))) → (b . (c . ())) ≡ (b c)

(cons 'x '(a b c)) ≡ (cons 'x '(a . (b . (c . ()))))
                 → (x . (a . (b . (c . ())))) ≡ (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

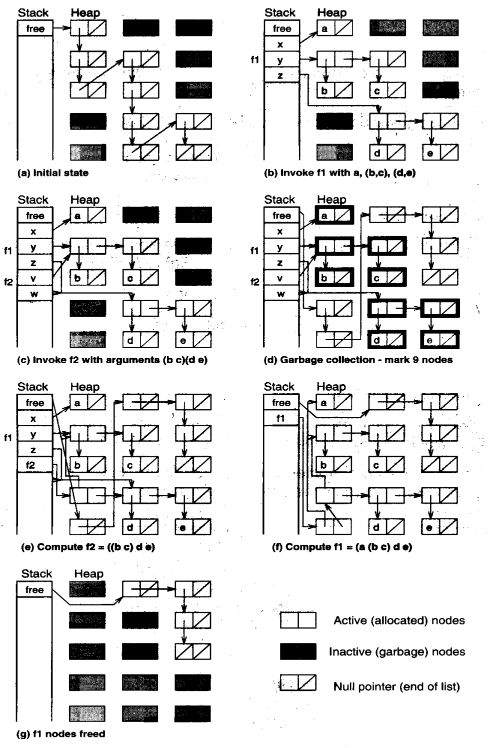
problems caused by destructive assignments? solutions?

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)
```


Stack
Heap
free
x
y
z
v
w
f1
f2
a
b
c
d
e
(a) Initial state
(b) Invoke f1 with a, (b,c), (d,e)
(c) Invoke f2 with arguments (b c)(d e)
(d) Garbage collection - mark 9 nodes
(e) Compute f2 = ((b c) d e)
(f) Compute f1 = (a (b c) d e)
(g) f1 nodes freed
Active (allocated) nodes
Inactive (garbage) nodes
Null pointer (end of list)

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

In-class exercise

given a menu structure:

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

define a function to count how many items are below a certain price

```
(define (numPricedBelow price menu)
    ??? )


(numPricedBelow 1.00 MENU) → 3

(numPricedBelow 0.99 MENU) → 1
```

8

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

9

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
```

10

Example: circle of friends

suppose we want to compute a person's circle of friends

- level 1: direct friends
- level 2: direct friends + friends of direct friends

  …
- level n : level (n-1) friends + friends of level (n-1) friends

```
(define FRIENDS
  '((amy (bob dan elle)) (bob (amy dan)) (chaz (dan elle))
    (dan (chaz)) (elle (amy bob chaz dan)) (fred (dan))))

(define (get-friends person)
  (cadr (assoc person FRIENDS)))

(define (get-circle person distance)
  (if (= distance 1)
      (get-friends person)
      (let ((circle (get-circle person (- distance 1))))
        (append circle (apply append (map get-friends circle))))))
```

11

Example: circle of friends (cont.)

consider Amy's circle of friends

- level 1: (bob dan elle)
- level 2: (bob dan elle) + (amy dan) + (chaz) + (amy bob chaz dan) →
  (bob dan elle amy dan chaz amy bob chaz dan)

```
(require scheme/list)

(define FRIENDS
  '((amy (bob dan elle)) (bob (amy dan)) (chaz (dan elle))
    (dan (chaz)) (elle (amy bob chaz dan)) (fred (dan))))

(define (get-friends person)
  (cadr (assoc person FRIENDS)))

(define (get-circle person distance)
  (if (= distance 1)
      (get-friends person)
      (let ((circle (get-circle person (- distance 1))))
        (remove person
                (remove-duplicates (append circle
                                           (apply append
                                                  (map get-friends circle))))))))
```

don't list self in circle of friends

don't list duplicates in circle of friends

12

Example: craps simulation

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)))))
```

13

Example: craps with history list

as is, all you see from `craps` is WINNER or LOSER

- would like to see the actual rolls to confirm proper behavior

the "functional way" is to construct a list of the rolls & return it

```
(define (craps)

    (define (roll-until point)
      (let ((next-roll (+ (random 6) (random 6) 2)))
        (cond ((= next-roll 7) (list next-roll 'LOSER))
              ((= next-roll point) (list next-roll 'WINNER))
              (else (cons next-roll (roll-until point))))))

    (let ((roll (+ (random 6) (random 6) 2)))
      (cond ((or (= roll 2) (= roll 12)) (list roll 'LOSER))
            ((= roll 7) (list roll 'WINNER))
            (else (cons roll (roll-until roll))))))
```

14

Example: craps with I/O

alternatively, can 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)

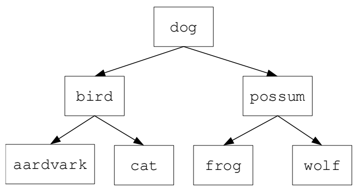
```
(define (craps)

  (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

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

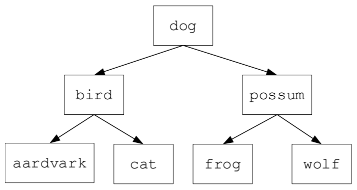
```
(car TREE)      ➔  ROOT
(cadr TREE)     ➔  LEFT-SUBTREE
(caddr TREE)    ➔  RIGHT-SUBTREE
```

16

Tree routines

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

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

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



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

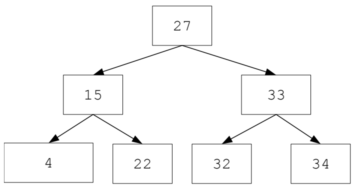
```
(define (right-subtree tree)
  (if (empty-tree? tree)
      'ERROR
      (caddr tree)))
```

17

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 <= value at node
all values in right subtree are > value at node



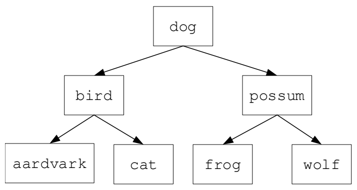
```
(define (bst-contains? bstree sym)
  (cond ((empty-tree? bstree) #f)
        ((= (root bstree) sym) #t)
        ((> (root bstree) sym) (bst-contains? (left-subtree bstree) sym))
        (else (bst-contains? (right-subtree bstree) sym))))
```

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

18

Tree traversal

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

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

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



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

19

In class exercises

```
(define (num-nodes tree)
    ??? )

(define (sum-values numtree)
    ??? )

(define (max-value numtree)
    ??? )
```

20