6.001
Data Mutation

• Primitive and Compound Data Mutators
• Stack Example
  • non-mutating
  • mutating
• Queue Example
  • non-mutating
  • mutating

Elements of a Data Abstraction

• A data abstraction consists of:
  • constructors  -- makes a new structure
  • selectors
  • mutators  -- changes an existing structure
  • operations
  • contract

Primitive Data

(define x 10) creates a new binding for name;
special form
x returns value bound to name
• To Mutate:
  (set! x "foo") changes the binding for name;
special form

Assignment -- set!

• Substitution model -- functional programming:
  (define x 10)
  (+ x 5) ==> 15  - expression has same value
each time it evaluated (in
... (+ x 5) ==> 15 same scope as binding)
• With assignment:
  (define x 10)
  (+ x 5) ==> 15  - expression "value" depends
... (set! x 94) on when it is evaluated
... (+ x 5) ==> 99

Compound Data

• constructor:
  (cons x y) creates a new pair p
• selectors:
  (car p) returns car part of pair
  (cdr p) returns cdr part of pair
• mutators:
  (set-car! p new-x) changes car pointer in pair
  (set-cdr! p new-y) changes cdr pointer in pair
; Pair, anytype -> undef -- side-effect only!

Example 1: Pair/List Mutation

(define a (list 1 2))
(define b a)
(a (1 2))
b (1 2)
(set-car! a 10)
b (10 2)

Compare with:
(define a (list 1 2))
(define b (list 1 2))
(set-car! a 10)
b (1 2)
Example 2: Pair/List Mutation

```
(define x (list 'a 'b))
```

- How to mutate to achieve the result at right?

```
(set-car! (cdr x)
  (list 1 2))
```

1. Eval `(cdr x)` to get a pair object
2. Change car pointer of that pair object

---

Sharing, Equivalence and Identity

- How can we tell if two things are equivalent?
  -- Well, what do you mean by "equivalent"?
    1. The same object: test with `eq?`
       ```
       (eq? a b) ==> #t
       ```
    2. Objects that "look the same": test with `equal?`
       ```
       (equal? (list 1 2) (list 1 2)) ==> #t
       (eq? (list 1 2) (list 1 2)) ==> #f
       ```

- If we change an object, is it the same object?
  -- Yes, if we retain the same pointer to the object

- How tell if part of an object is shared with another?
  -- If we mutate one, see if the other also changes

---

Stack Data Abstraction

- constructor:
  ```
  (make-stack)
  ```
  returns an empty stack

- selectors:
  ```
  (top stack)
  ```
  returns current top element from a stack

- operations:
  ```
  (insert stack elt)
  (delete stack)
  (empty-stack? stack)
  ```
  returns a new stack with the element added to the top of the stack
  returns a new stack with the top element removed from the stack
  returns #t if no elements, #f otherwise

---

End of part 1

- Scheme provides built-in mutators
  - `set!` to change a binding
  - `set-car!` and `set-cdr!` to change a pair

- Mutation introduces substantial complexity
  - Unexpected side effects
  - Substitution model is no longer sufficient to explain behavior

---

Your Turn

```
x ==> (3 4)
y ==> (1 2)
```

```
(set-car! x y)
x ==> ((1 2) 4)
```

followed by

```
(set-cdr! y (cdr x))
x ==> ((1 4) 4)
```
Stack Contract

- If s is a stack, created by (make-stack) and subsequent stack procedures, where i is the number of insertions and j is the number of deletions, then

1. If j > i then it is an error
2. If j = i then (empty-stack? s) is true, and (top s) and (delete s) are errors.
3. If j < i then (empty-stack? s) is false and (top (delete (insert s val))) = (top s)
4. If j ≤ i then (top (insert s val)) = val for any val

Stack Implementation Strategy

- implement a stack as a list

Limitations in our Stack

- Stack does not have identity

Alternative Stack Implementation – pg. 1

- Attach a type tag – defensive programming
- Additional benefit:
  - Provides an object whose identity remains even as the object mutates

Alternative Stack Implementation – pg. 2

- Note: This is a change to the abstraction! User should know if the object mutates or not in order to use the abstraction correctly.
Alternative Stack Implementation – pg. 3

(define (insert! stack elt)
  (cond
    ((not (stack? stack))
      (error "object not a stack:" stack))
    (else
      (set-cdr! stack (cons elt (cdr stack)))
      stack)))

(define (delete! stack)
  (if (empty-stack? stack)
      (error "stack underflow – delete")
      (set-cdr! stack (cddr stack)))
  stack)

(define (top stack)
  (if (empty-stack? stack)
      (error "stack underflow – top")
      (cadr stack)))

Queue Data Abstraction (Non-Mutating)

• constructor:
  (make-queue) returns an empty queue

• accessors:
  (front-queue q) returns the object at the front of the queue. If queue is empty signals error

• mutators:
  (insert-queue q elt) returns a new queue with elt at the rear of the queue
  (delete-queue q) returns a new queue with the item at the front of the queue removed

• operations:
  (empty-queue? q) tests if the queue is empty

Queue Contract

• If q is a queue, created by (make-queue) and subsequent queue procedures, where i is the number of insertions, j is the number of deletions, and x_i is the ith item inserted into q, then
  1. If j > i then it is an error
  2. If j = i then (empty-queue? q) is true, and (front-queue q) and (delete-queue q) are errors.
  3. If j < i then (front-queue q) = x_{j+1}

Simple Queue Implementation – pg. 1

• Let the queue simply be a list of queue elements:

• The front of the queue is the first element in the list
• To insert an element at the tail of the queue, we need to "copy" the existing queue onto the front of the new element:

Simple Queue Implementation – pg. 2

(define (make-queue) nil)
(define (empty-queue? q) (null? q))
(define (front-queue q)
  (if (empty-queue? q)
      (error "front of empty queue:" q)
      (car q)))
(define (delete-queue q)
  (if (empty-queue? q)
      (error "delete of empty queue:" q)
      (cdr q)))
(define (insert-queue q elt)
  (if (empty-queue? q)
      (cons elt nil)
      (cons (car q) (insert-queue (cdr q) elt))))

Simple Queue - Orders of Growth

• How efficient is the simple queue implementation?
  • For a queue of length n
    – Time required – number of cons, car, cdr calls?
    – Space required – number of new cons cells?
  • front-queue, delete-queue:
    • Time: $T(n) = O(1)$ that is, constant in time
    • Space: $S(n) = O(1)$ that is, constant in space
  • insert-queue:
    • Time: $T(n) = O(n)$ that is, linear in time
    • Space: $S(n) = O(n)$ that is, linear in space
Queue Data Abstraction (Mutating)

• constructor: (make-queue) returns an empty queue

• accessor: (front-queue q) returns the object at the front of the queue. If queue is empty signals error

• mutator: (insert-queue! q elt) inserts the elt at the rear of the queue and returns the modified queue

• mutator: (delete-queue! q) removes the elt at the front of the queue and returns the modified queue

• operations: (queue? q) tests if the object is a queue

EMPTY-QUEUE? q) tests if the queue is empty

Better Queue Implementation – pg. 1

• We’ll attach a type tag as a defensive measure

• Maintain queue identity

• Build a structure to hold:
  • a list of items in the queue
  • a pointer to the front of the queue
  • a pointer to the rear of the queue

Queue Helper Procedures

• Hidden inside the abstraction

(define (front-ptr q) (cadr q))
(define (rear-ptr q) (cddr q))
(define (set-front-ptr! q item) (set-car! (cdr q) item))
(define (set-rear-ptr! q item) (set-cdr! (cdr q) item))

Queue Implementation – pg. 2

(define (make-queue) (cons 'queue (cons nil nil)))
(define (queue? q) (and (pair? q) (eq? 'queue (car q))))
(define (empty-queue? q) (if (not (queue? q)) ; defensive
 (error "object not a queue:" q) ; programming
 (null? (front-ptr q))))

(define (front-queue q) (if (empty-queue? q) (error "front of empty queue:" q) (car (front-ptr q))))

Queue Implementation – pg. 3

(define (insert-queue! q elt) (let ((new-pair (cons elt nil))) (cond ((empty-queue? q) (set-front-ptr! q new-pair) (set-rear-ptr! q new-pair) q)
 (else (set-cdr! (rear-ptr q) new-pair) (set-rear-ptr! q new-pair) q)))))

Queue Implementation – pg. 4

(define (delete-queue! q) (cond ((empty-queue? q) (error "delete of empty queue:" q))
 (else (set-front-ptr! q (cdr (front-ptr q))))))
**Mutating Queue - Orders of Growth**

- How efficient is the mutating queue implementation?
  - For a queue of length $n$
    - Time required -- number of cons, car, cdr calls?
    - Space required -- number of new cons calls?

- `front-queue, delete-queue!`
  - Time: $T(n) = O(1)$ that is, constant in time
  - Space: $S(n) = O(1)$ that is, constant in space

- `insert-queue!`
  - Time: $T(n) = O(1)$ that is, constant in time
  - Space: $S(n) = O(1)$ that is, constant in space

**Summary**

- Built-in mutators which operate by side-effect
  - `set!` (special form)
  - `set-car!` ; Pair, anytype -> undef
  - `set-cdr!` ; Pair, anytype -> undef

- Extend our notion of data abstraction to include mutators

- Mutation is a powerful idea
  - enables new and efficient data structures
  - can have surprising side effects
  - breaks our “functional” programming (substitution) model