6.001 SICP
Data Mutation

- Mutation
- Stack Example
  - non-mutating
  - mutating
- Queue Example
  - non-mutating
  - mutating

1

Functional programming

- So far, procedures were simple input-output black boxes
- Procedures could not affect the rest of the world
- Given the same inputs, a procedure always returns the same output
- We call this **functional programming**

2

Non-mutating example: map

```
(define (map op L)
   (if (null? L) '()
      (cons (op (car L)) (map op (cdr L)))))
(define L1 '(1 2 3))
(define L2 L1)
(map square L1) ➔ (1 4 9)
L1              ➔
(define L1 (map square L1))
L2              ➔
```

- Map only returns a value, it does not modify anything else

3

Functional programming vs. mutation

- Today, we make a radical change: procedures will be able to affect outside bindings
- The order of evaluation ***does*** matter
- We call this **MUTATION**
- This has crucial consequences on data abstraction

4

Elements of a Data Abstraction

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

5

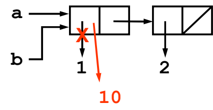
Compound data mutation

- 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!

6

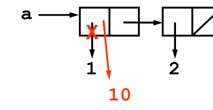
Example 1: Pair/List Mutation

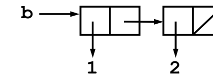
```
(define a (list 1 2))
(define b a)
a ➔ (1 2)
b ➔ (1 2)
(set-car! a 10)
b ==> (10 2)
```



Compare with:

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

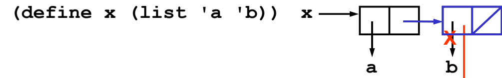




7

Example 2: Pair/List Mutation

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



- How mutate to achieve the result at right?

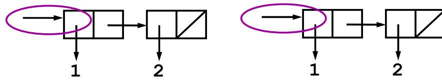
```
(set-car! (cdr x)
          (list 1 2))
```

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

8

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`



`(1 2)` `(1 2)`

9

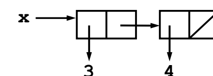
Sharing, Equivalence and Identity

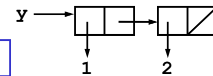
- How can we tell if two things are equivalent?
  - -- Well, what do you mean by "equivalent"?
    1. The *same object*: test with `eq?`
       `(eq? a b) ==> #t`
    2. Objects that *"look" the same*: test with `equal?`
       `(equal? (list 1 2) (list 1 2)) ==> #t`
       `(eq? (list 1 2) (list 1 2)) ==> #f`
- If we change an object, is it the same object?
  - -- Yes, if we retain the same pointer to the object
- How tell if parts of an object is *shared* with another?
  - -- If we mutate one, see if the other also changes

10

Your Turn

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






```
(set-car! x y)
x ==>
```

followed by

```
(set-cdr! y (cdr x))
x ==>
```

11

mutating version: map!

```
(define (map! op L)
  (if (not (null? L))
    (begin
      (set-car! L (op (car L)))
       (map! op (cdr L)))))

(define L1 '(1 2 3))
(define L2 L1)

(map! square L1)
L2 ➔
```

- We have mutated the list, thereby affecting other bindings
- This is an important change in the data abstraction

12

End of part 1

- Scheme provides built-in mutators
  - `set-car!` and `set-cdr!` to change a pair
- Functional vs. mutating version of map
  - map! modifies the list given as input
  - We use ! in the name to emphasize mutation
- Mutation introduces substantial complexity
  - Unexpected side effects
  - Substitution model is no longer sufficient to explain behavior

Stacks and Queues

- Store "stuff" waiting to be processed
- Queue: First-In-First-Out (FIFO)
- Stack: Last-In-First-Out (LIFO)

Stack Data Abstraction - functional

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

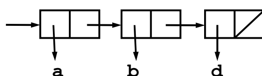
Stack Contract

- 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



- we will insert and delete items off the front of the stack

Stack Implementation

```
(define (make-stack) '())

(define (empty-stack? stack) (null? stack))

(define (insert stack elt) (cons elt stack))

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

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

Limitations in our Stack

- Stack does not have ***identity***

```
(define s (make-stack))
(define s2 s)
s ➔ ()

(insert s 'a) ==> (a)
s ➔ ()
```

Even if we do

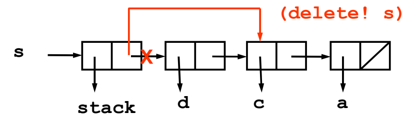
```
(define s (insert s 'a))
```

`s2` is not updated

Alternative Stack Implementation – mutation

- We want the ability to ***modify*** the stack
- Note: This is a change to the abstraction! User should know if the object mutates or not in order to use the abstraction correctly

- Problem: the head of the stack will be modified
- Solution: use a tag as first element
  - Additional advantage: defensive programming



Alternative Stack Implementation – pg. 2

```
(define (make-stack) (cons 'stack '()))

(define (stack? stack)
  (and (pair? stack) (eq? 'stack (car stack))))

(define (empty-stack? stack)
  (if (not (stack? stack))
      (error "object not a stack:" stack)
      (null? (cdr stack))))
```


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

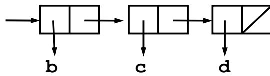
Queue Contract

- 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

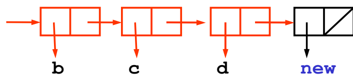
1. If $j>i$ then it is an error
2. If $j=i$ then **`(empty-queue? q)`** is true, and **`(front-queue q)`** and **`(delete-queue q)`** are errors.
3. If $j<i$ then **`(front-queue q)`** = x_{j+1}

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

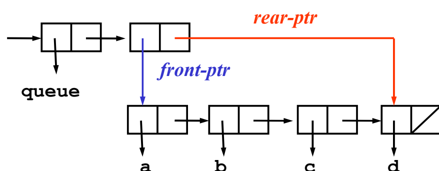
- How efficient is the simple queue implementation?
  - For a queue of length *n*
    - Time required -- number of `cons, car, cdr` calls?
    - Space required -- number of new `cons` cells?
- `front-queue, delete-queue:`
  - Time: $T(n) = O(1)$ that is, constant in time
  - Space: $S(n) = O(1)$ that 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
- accessors:
  - `(front-queue q)` returns the object at the front of the queue. If queue is empty signals error
- mutators:
  - `(insert-queue! q elt)` inserts the elt at the rear of the queue and returns the modified queue
  - `(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

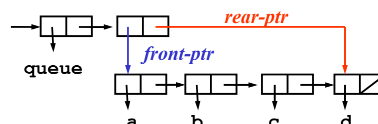
- Build a structure to hold:
  - a list of items in the queue
  - a pointer to the front of the queue
  - a pointer to the rear of the queue
- We'll attach a type tag as a defensive measure



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))
```

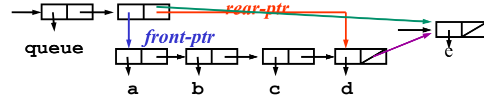


Better Queue Implementation – pg. 2

```
(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))))
```

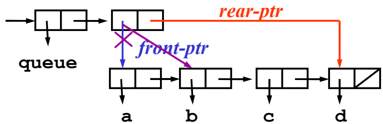

Queue Implementation – pg. 3

```
(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)))
         q)))
```



Summary

- Built-in mutators which operate by **side-effect**
  - `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