6.001 SICP
Data Mutation

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

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

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

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

Elements of a Data Abstraction

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

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!
Example 1: Pair/List Mutation

\[
\text{(define } a \text{(list 1 2))} \\
\text{(define } b \text{ a)} \\
\text{a } \Rightarrow \text{ (1 2)} \\
b \Rightarrow \text{ (1 2)} \\
\text{(set-car! a 10)} \\
b \Rightarrow \text{ (10 2)}
\]

Compare with:

\[
\text{(define } a \text{(list 1 2))} \\
\text{(define } b \text{(list 1 2))} \\
\text{(set-car! a 10)} \\
b \Rightarrow \text{ (1 2)}
\]

Example 2: Pair/List Mutation

\[
\text{(define } x \text{(list 'a 'b))} \\
\text{x } \Rightarrow \text{ ('a 'b)} \\
\text{(set-car! (cdr x) (list 1 2))} \\
\text{x } \Rightarrow \text{ (3 4)} \\
\text{y } \Rightarrow \text{ (1 2)} \\
\text{(set-car! x y)} \\
x \Rightarrow \text{ ('3 '4)} \\
\text{followed by} \\
\text{(set-cdr! y (cdr x))} \\
x \Rightarrow \text{ ('1 '2)}
\]

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?
    \[
    (\text{eq? a b}) \Rightarrow \text{ #t}
    \]
    2. Objects that "look the same": test with equal?
    \[
    (\text{equal? (list 1 2) (list 1 2)}) \Rightarrow \text{ #t} \\
    (\text{eq? (list 1 2) (list 1 2)}) \Rightarrow \text{ #f}
    \]

Your Turn

\[
x \Rightarrow \text{ ('3 '4)} \\
y \Rightarrow \text{ (1 2)} \\
\text{(set-car! x y)} \\
x \Rightarrow \text{ ('3 '4)} \\
\text{followed by} \\
\text{(set-cdr! y (cdr x))} \\
x \Rightarrow \text{ ('1 '2)}
\]

mutating version: map!

\[
\text{(define (map! op L))} \\
\text{(define L1 '(1 2 3))} \\
\text{(define L2 L1)} \\
\text{(map! square L1)}
\]

- We have mutated the list, thereby affecting other bindings
- This is an important change in the data abstraction
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 \( s \) \( v \)))) = (top \( s \))
4. If \( j = i \) then (top (insert \( s \) \( v \))) = \( v \) for any \( v \)

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

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

(insert s 'a) ==> (a)
s2 ==> ()
```

Even if we do
```scheme
(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

```scheme
(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

```scheme
(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
- accessor:
  (front-queue q) returns the object at the front of the queue. If queue is empty signals error
- mutator:
  (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 xi 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) = \( x_{i+j} \)
Simple Queue Implementation – pg. 1

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

```
   b   c   d
   New
```

- 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:

```
   b   c   d   New
   New
```

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 \( \text{cons} \), \( \text{car} \), \( \text{cdr} \) calls?
    - Space required -- number of new \( \text{cons} \) cells?

  - \( \text{front-queue, delete-queue}: \)
    - Time: \( T(n) = O(1) \) that is, constant in time
    - Space: \( S(n) = O(1) \) that is, constant in space

  - \( \text{insert-queue}: \)
    - Time: \( T(n) = O(n) \) that is, linear in time
    - Space: \( S(n) = O(n) \) that is, linear in space

Queue Data Abstraction (Mutating)

- \( \text{constructor}: \)
  - \( \text{make-queue} \) returns an empty queue
- \( \text{accessors}: \)
  - \( \text{front-queue} q \) returns the object at the front of the queue. If queue is empty signals error
- \( \text{mutators}: \)
  - \( \text{insert-queue!} q \) inserts the elt at the rear of the queue and returns the modified queue
  - \( \text{delete-queue!} q \) removes the elt at the front of the queue and returns the modified queue
- \( \text{operations}: \)
  - \( \text{queue?} q \) tests if the object is a queue
  - \( \text{empty-queue?} q \) 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
   front-ptr
   b   c   d   rear-ptr
   a   b   c   d
```

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**

```scheme
(define (make-queue)
  (cons 'queue (cons nil nil)))
```

```scheme
(define (queue? q)
  (and (pair? q) (eq? 'queue (car q))))
```

```scheme
(define (empty-queue? q)
  (if (not (queue? q)) ; defensive
      (error "object not a queue:" q) ; programming
      (null? (front-ptr q))))
```

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

**Queue Implementation – pg. 3**

```scheme
(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**

```scheme
(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