Data Mutation

- Primitive and compound data mutators
  - `set!` for names
  - `set-car!`, `set-cdr!` for pairs
- 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 (value is undefined)

Assignment -- `set!`

- Substitution model -- functional programming:
  ```
  (define x 10)
  (+ x 5) ==> 15
  ... 
  ```
  expression has same value each time it evaluated (in same scope as binding)

- With mutation:
  ```
  (define x 10)
  (+ x 5) ==> 15
  ... 
  (set! x 94)
  ... 
  (+ x 5) ==> 99
  ```

Compound Data

- constructor:
  ```
  (cons x y)
  ```
creates a new pair `p`

- selectors:
  ```
  (car p)
  (cdr p)
  ```
returns car part of pair `p`
returns cdr part of pair `p`

- mutators:
  ```
  (set-car! p new-x)
  (set-cdr! p new-y)
  ```
changes car part of pair `p`
changes cdr part of pair `p`

; Pair, anytype -> undef -- side-effect only!

Example 1: Pair/List Mutation

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

(define x (list 'a 'b))

- How can we use mutation to achieve the result at right?

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

1. Evaluate (cdr x) to get a pair object
2. Change car part 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

Your Turn

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

(set-car! x y)

1 2
x ==> 3 4

followed by

(set-cdr! y (cdr x))

x ==> 1 2

Stack Data Abstraction

- constructor: (make-stack) returns an empty stack
- selectors: (top-stack s) returns current top element from a stack s
- operations: (insert-stack s elt) returns a new stack with the element added to the top of the stack
  (delete-stack s) returns a new stack with the top element removed from the stack
  (empty-stack? s) 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
Stack Contract

- If \( s \) is a stack, created by \( \text{make-stack} \) and subsequent stack procedures, where \( i \) is the number of inserts and \( j \) is the number of deletes, then

1. If \( j > i \) then it is an error
2. If \( j = i \) then \( (\text{empty-stack? } s) \) is true and \( (\text{top-stack } s) \) and \( (\text{delete-stack } s) \) are errors.
3. If \( j < i \) then \( (\text{empty-stack? } s) \) is false and
   \[ \begin{align*}
   (\text{top-stack } s) &= (\text{top-stack } s) \\
   (\text{delete-stack } s) &= (\text{insert-stack } s \text{ val}))
   \end{align*} \]
   for any \( \text{val} \)
4. If \( j \leq i \) then
   \[ \text{top-stack } (\text{insert-stack } s \text{ val}) = \text{top-stack } s \]
   for any \( \text{val} \)

Stack Implementation Strategy

- Implement a stack as a list

```
(define (make-stack) '())
(define (empty-stack? s) ; Stack<A> -> boolean
  (null? s))
(define (insert-stack s elt) ; Stack<A>, A -> Stack<A>
  (cons elt s))
(define (delete-stack s) ; Stack<A> -> Stack<A>
  (if (not (empty-stack? s))
    (cdr s)
    (error "stack underflow – delete")))
(define (top-stack s) ; Stack<A> -> A
  (if (not (empty-stack? s))
    (car s)
    (error "stack underflow – top")))
```

Limitations in our Stack

- Stack does not have identity

```
(define s (make-stack))
(s ==> ())
(insert s 'a) ==> (a)
(set! s (insert s 'b))
(s ==> (b))
```

Alternative Stack Implementation – pg. 1

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

```
(stack s)
```

```
(stack! s)
```

- 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. 2

```
(define (make-stack) (cons 'stack '()))
(define (stack? s) ; anytype -> boolean
  (and (pair? s) (eq? 'stack (car s)))))
(define (empty-stack? s) ; Stack<A> -> boolean
  (if (stack? s)
    (null? (cdr s))
    (error "object not a stack: " s))))
```

```
Alternative Stack Implementation – pg. 3

(define (insert-stack! s elt); Stack<A>, A -> Stack<A>
  (if (stack? s)
    (set-cdr! s (cons elt (cdr s)))
    (error "object not a stack: s")
  stack)

(define (delete-stack! s); Stack<A> -> Stack<A>
  (if (not (empty-stack? s))
    (set-cdr! s (cddr s))
    (error "stack underflow – delete")
  stack)

(define (top-stack s); Stack<A> -> A
  (if (not (empty-stack? s))
    (cadr s)
    (error "stack underflow – top")))

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

• operation:
  (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
  (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 inserts, j is the number of deletes, 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

(define (make-queue) '())
(define (empty-queue? q) (null? q)); Queue<A> -> boolean
(define (front-queue q); Queue<A> -> A
  (if (not (empty-queue? q))
    (car q)
    (error "front of empty queue: q"))
(define (delete-queue q); Queue<A> -> Queue<A>
  (if (not (empty-queue? q))
    (cdr q)
    (error "delete of empty queue: q"))
(define (insert-queue q elt); Queue<A>, A -> Queue<A>
  (if (empty-queue? q)
    (cons elt '())
    (cons (car q) (insert-queue (cdr q) elt)))))

Simple Queue Implementation – pg. 2

(define (make-queue) '())
(define (empty-queue? q) (null? q)); Queue<A> -> boolean
(define (front-queue q); Queue<A> -> A
  (if (not (empty-queue? q))
    (car q)
    (error "front of empty queue: q"))
(define (delete-queue q); Queue<A> -> Queue<A>
  (if (not (empty-queue? q))
    (cdr q)
    (error "delete of empty queue: q"))
(define (insert-queue q elt); Queue<A>, A -> Queue<A>
  (if (empty-queue? q)
    (cons elt '())
    (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

- 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

```scheme
(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 '() '())))
(define (queue? q) ; anytype -> boolean
  (and (pair? q) (eq? 'queue (car q))))
(define (empty-queue? q) ; Queue<A> -> boolean
  (if (queue? q)
      (null? (front-ptr q))
      (error "object not a queue:" q)))
(define (front-queue q) ; Queue<A> -> A
  (if (not (empty-queue? q))
      (car (front-ptr q))
      (error "front of empty queue:" q)))
```

Queue Implementation – pg. 3

```scheme
(define (insert-queue! q elt); Queue<A>, A -> Queue<A>
  (let ((new-pair (cons elt '())))
    (cond ((empty-queue? q) (set-front-ptr! q new-pair)
      (set-rear-ptr! q new-pair))
      (else (set-cdr! (rear-ptr q) new-pair)
      (set-rear-ptr! q new-pair))
      q)))
```

Queue Implementation – pg. 4

```scheme
(define (delete-queue! q); Queue<A> -> Queue<A>
  (if (not (empty-queue? q))
      (set-front-ptr! q (cdr (front-ptr q)))
      (error "delete of empty queue:" q))
```

Better Queue Implementation – pg. 2

```scheme
(define (make-queue) (cons 'queue (cons () ())))
(define (queue? q) ; anytype -> boolean
  (and (pair? q) (eq? 'queue (car q))))
(define (empty-queue? q) ; Queue<A> -> boolean
  (if (queue? q)
      (null? (front-ptr q))
      (error "object not a queue:" q)))
(define (front-queue q) ; Queue<A> -> A
  (if (not (empty-queue? q))
      (car (front-ptr q))
      (error "front of empty queue:" q)))
```

Better Queue Implementation – pg. 2

```scheme
(define (make-queue) (cons 'queue (cons () ())))
(define (queue? q) ; anytype -> boolean
  (and (pair? q) (eq? 'queue (car q))))
(define (empty-queue? q) ; Queue<A> -> boolean
  (if (queue? q)
      (null? (front-ptr q))
      (error "object not a queue:" q)))
(define (front-queue q) ; Queue<A> -> A
  (if (not (empty-queue? q))
      (car (front-ptr q))
      (error "front of empty queue:" 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 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: \( T(n) = O(1) \) 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 model of "functional" programming (substitution model)