Structure and Interpretation of Computer Programs

COMP200
ELEMENTS OF DATA ABSTRACTION

- Constructors
- Selectors
- Operations
- Contract
ELEMENTS OF DATA ABSTRACTION

+ Mutation

- Constructors
- Selectors
- Mutators
- Operations
- Contract
• Primitive and Compound Data Mutators
• Stack Example
  (non-mutating vs. mutating)
• Queue Example
  (non-mutating vs. mutating)
ELEMENTS OF DATA ABSTRACTION

• Constructors
• Selectors
• Mutators
• Operations
• Contract

+Mutation makes a new structure
ELEMENTS OF DATA ABSTRACTION

+Mutation

- Constructors
- Selectors
- Mutators
- Operations
- Contract
ELEMENTS OF DATA ABSTRACTION

+ Mutation

- Constructors
- Selectors
- Mutators
- Operations
- Contract
ELEMENTS OF DATA ABSTRACTION

+Mutation

- Constructors
- Selectors
- Mutators
- Operations
- Contract
ELEMENTS OF DATA ABSTRACTION

+Mutation

- Constructors
- Selectors
- Mutators
- Operations
- Contract
ELEMENTS OF DATA ABSTRACTION

+ Mutation

• Constructors
• Selectors
• Mutators changes an existing structure
• Operations
• Contract
MUTATION
Primitive Data

(define x 10)  creates a new binding for name; special form
x            returns the value bound to name
**MUTATION**

*Primitive Data*

```scheme
(define x 10)
```

creates a new binding for `name`; special form

```scheme
x
```

returns the value bound to `name`

```scheme
(set! x "foo")
```

changes the binding for `name`; special form
• The Substitution Model — functional programming
**MUTATION**

Assignment `set!`

- The Substitution Model — *functional programming*

```scheme
(define x 10)
(+ x 5) ;==> 15
...
(+ x 5) ;==> 15
```

The expression has same value each time it is evaluated. (in the same scope as binding)
• The Substitution Model — *functional programming*

```lisp
(define x 10)
(+ x 5) ;===> 15
...
(+ x 5) ;===> 15
```

The expression has same value each time it is evaluated.

(in the same scope as binding)

• With assignment
The expression has same value each time it is evaluated. (in the same scope as binding)

With assignment

The value of expression depends on when it is evaluated.
COMPOUND DATA

+Mutation

• Constructors

(cons x y) creates a new pair p
COMPOUND DATA

+Mutation

- Constructors
  \[ (\text{cons } x \ y) \]
  creates a new pair \( p \)

- Selectors
  \[ (\text{car } p) \]
  \[ (\text{cdr } p) \]
  returns \text{car} part of the pair
  returns \text{cdr} part of the pair
COMPOUND DATA

Constructors
(cons x y) creates a new pair \( p \)

Selectors
(car p) returns car part of the pair
(cdr p) returns cdr part of the pair

Mutators
(set-car! p new-x) changes car pointer of the pair
(set-cdr! p new-y) changes cdr pointer of the pair
COMPOUND DATA

Constructors
(cons x y) creates a new pair p

Selectors
(car p) returns car part of the pair
(cdr p) returns cdr part of the pair

Mutators
(set-car! p new-x) changes car pointer of the pair
(set-cdr! p new-y) changes cdr pointer of the pair
; Pair, anytype -> undef — side effect only!
EXAMPLE 1
Pair/List Mutation

(define a (list 1 2))

;a ==> (1,2)
EXAMPLE 1
Pair/List Mutation

(define a (list 1 2))
(define b a)

;a ==> (1,2)
;b ==> (1,2)
(define a (list 1 2))
(define b a)
;a ==> (1,2)
;b ==> (1,2)

(set-car! a 10)
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)
```
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)

(define a (list 1 2))
(define b (list 1 2))
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)
```

```
(define a (list 1 2))
(define b (list 1 2))
(set-car! a 10)
;b ==> (1 2)
```
EXAMPLE 2
Pair/List Mutation

\[
\text{(define } x \text{ (list 'a 'b))}
\]
EXAMPLE 2
Pair/List Mutation

(define x (list 'a 'b))

How to mutate to achieve the result at right?
EXAMPLE 2
Pair/List Mutation

(\texttt{define x (list 'a 'b)})

How to mutate to achieve the result at right?

(\texttt{set-car! (cdr x) (list 1 2)})

1. Eval \texttt{(cdr x)} to get a pair object
2. Change \texttt{car} pointer of that pair object
SHARING, EQUVALENCE AND IDENTITY

How to tell if the two things are equivalent?
SHARING, EQUIVALENCE AND IDENTITY

How to tell if the two thing are equivalent?

• What do we mean by equivalent?
SHARING, EQUIVALENCE AND IDENTITY

How to tell if the two things are equivalent?

• What do we mean by equivalent?

  1. The same object: test with eq?
      (eq? a b) ;===> #t
SHARING, EQUIVALENCE AND IDENTITY

How to tell if the two thing are equivalent?

• What do we mean by equivalent?

1. The same object: test with eq?
   \[(eq\?\ a\ b)\;\Longrightarrow\;\texttt{t}\]

2. Objects that "look" the same: test with equal?
   \[(\text{equal}\?\ (\text{list}\ 1\ 2)\ (\text{list}\ 1\ 2))\;\Longrightarrow\;\texttt{t}\]
   \[(eq\?\ (\text{list}\ 1\ 2)\ (\text{list}\ 1\ 2))\;\Longrightarrow\;\texttt{f}\]
SHARING, EQUIVALENCE AND IDENTITY

How to tell if the two thing are equivalent?

• What do we 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
SHARING, EQUIVALENCE AND IDENTITY

How to tell if the two things are equivalent?

• What do we mean by equivalent?

1. The same object: test with eq?
   
   ```scheme
   eq? a b) ===> #t
   ```

2. Objects that "look" the same: test with equal?
   
   ```scheme
   (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
QUIZ
Find the value of $x$

; $x \implies (3\ 4)$
; $y \implies (1\ 2)$

(set-car! x y)
; $x \implies \boxed{}$

followed by:

(set-cdr! y (cdr x))
; $x \implies \boxed{}$
QUIZ

Find the value of $x$

; $x \Longrightarrow (3 \ 4)$

; $y \Longrightarrow (1 \ 2)$

(set-car! x y)
; $x \Longrightarrow ((1,2), \ 4)$

followed by:

(set-cdr! y (cdr x))
; $x \Longrightarrow ((1,4), \ 4)$
WHAT HAVE WE LEARNT?

Summary

- Scheme provides built-in mutators
  - `set!` to change a binding
  - `set-car!` and `set-cdr!` to change a pair
WHAT HAVE WE LEARNT?

Summary

• 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 the behavior
STACK DATA ABSTRACTION
STACK DATA ABSTRACTION

- Constructor
  \((\text{make-stack})\) returns an empty stack
STACK DATA ABSTRACTION

• **Constructor**
  
  `(make-stack)`

  returns an empty stack

• **Selectors**
  
  `(top stack)`

  returns current top element from a stack
STACK DATA ABSTRACTION

- **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
STACK DATA ABSTRACTION

- **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
STACK DATA ABSTRACTION

- **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 DATA ABSTRACTION

Contract
STACK DATA ABSTRACTION

Contract

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

1. If $j = i$ then `(empty-stack? s)` is true, and `(top s)` and `(delete s)` are errors.
2. If $j < i$ then `(top (insert s val)) = val` for any $val$.
3. If $j \leq i$
STACK DATA ABSTRACTION

Contract

If $s$ is a stack, created by `(make-stack)` and subsequent 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
If \( s \) is a stack, created by \((\text{make-stack})\) and subsequent procedures, where \( i \) is the number of insertions and \( j \) is the number of deletions, then:

1. \( j > i \) then it is an error
2. \( j = i \) then \((\text{empty-stack? } s)\) is true, and \((\text{top } s)\) and \((\text{delete } s)\) are errors.
STACK DATA ABSTRACTION

Contract

If $s$ is a stack, created by `(make-stack)` and subsequent 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)`
STACK DATA ABSTRACTION

Contract

If $s$ is a stack, created by (make-stack) and subsequent 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 $j \leq i$ then (top (insert $s$ val)) = val for any val
STACK DATA ABSTRACTION
The First Attempt

• implement stack as a list

```
  a  b  c
```
STACK DATA ABSTRACTION
The First Attempt

• implement stack as a list

• insert and delete items off the front of the stack
STACK DATA ABSTRACTION
The First Attempt — Implementation

(define (make-stack) nil)

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

(define (insert stack elt)
  (cons elt stack))
STACK DATA ABSTRACTION
The First Attempt — Implementation

(define (make-stack) nil)

(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)))
STACK DATA ABSTRACTION
The First Attempt — Limitations

(define (make-stack) nil)
(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)))

(define s (make-stack))
  ;s ==> ()
(insert s 'a)
  ;==> (a) s ==> ()
STACK DATA ABSTRACTION

The First Attempt — Limitations

```
(define (make-stack) nil)

(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)))
```

```
(define s (make-stack))
  ;s ==> ()

(insert s 'a)
  ;==> (a) s ==> ()

(set! s (insert s 'b))
  ;s ==> (b)
```
STACK DATA ABSTRACTION
The First Attempt — Limitations

(define (make-stack) nil)
(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)))

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

Stack does not have an identity
STACK DATA ABSTRACTION
Alternative Implementation with Tagging

- Attach a type tag — *defensive programming*

![Diagram](image-url)
STACK DATA ABSTRACTION
Alternative Implementation with Tagging

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

(s) -> stack -> a -> b -> c
(delete! s)
STACK DATA ABSTRACTION
Alternative Implementation with Tagging

- Attach a type tag — *defensive programming*

- Additional benefit:
  Provides an object whose identity remains even as the object *mutates*

*Note:* This is a change to the abstraction! User should know if the object mutates or not in order to use the abstraction correctly.
STACK DATA ABSTRACTION

Alternative Implementation with Tagging

(define (make-stack)
  (cons 'stack nil))
STACK DATA ABSTRACTION
Alternative Implementation with Tagging

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

(define (stack? stack)
  (and (pair? stack)
       (eq? 'stack (car stack)))))
(define (make-stack)
  (cons 'stack nil))

(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))))
STACK DATA ABSTRACTION
Alternative Implementation with Tagging

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

Alternative Implementation with Tagging

(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")

(define (top stack)
  (if (empty-stack? stack)
   (error "stack underflow - top")
STACK DATA ABSTRACTION

Alternative Implementation with Tagging

(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
QUEUE DATA ABSTRACTION

The First Attempt — Non-mutating

- **Constructor**
  
  `(make-queue)`

  Returns an empty queue

- **Selectors**
  
  `(front-queue q)`

  Returns the object at the front of the queue. If queue is empty, signals error.
QUEUE DATA ABSTRACTION

The First Attempt — Non-mutating

- **Constructor**
  
  `(make-queue)`
  
  returns an empty queue

- **Selectors**
  
  `(front-queue q)`
  
  returns the object at the front of the queue. If queue is empty, signals error.

- **Operations**
  
  `(insert-queue q elt)`
  
  returns a new queue with elt at the rear of the queue
QUEUE DATA ABSTRACTION

The First Attempt — Non-mutating

• Constructor
  (make-queue) returns an empty queue

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

• Operations
  (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
QUEUE DATA ABSTRACTION

The First Attempt — Non-mutating

- **Constructor**
  - `(make-queue)`
  - returns an empty queue

- **Selectors**
  - `(front-queue q)`
  - returns the object at the front of the queue.
  - If queue is empty, signals error.

- **Operations**
  - `(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
If $q$ is a queue, created by (make-queue) and subsequent procedures, where $i$ is the number of insertions and $j$ is the number of deletions, and $x_i$ is the i’th item inserted into $q$, then:
 QUEUE DATA ABSTRACTION

Contract

If q is a queue, created by (make-queue) and subsequent procedures, where i is the number of insertions and j is the number of deletions, and x_i is the i’th item inserted into q, then:

1. if j > i then it is an error
**QUEUE DATA ABSTRACTION**

**Contract**

If \( q \) is a queue, created by \((\text{make-queue})\) and subsequent procedures, where \( i \) is the number of insertions and \( j \) is the number of deletions, and \( x_i \) is the \( i \)'th item inserted into \( q \), then:

1. **if** \( j > i \) **then** it is an error

2. **if** \( j = i \) **then** \((\text{empty-queue?} \ q)\) is **true**, and \((\text{front-queue} \ q)\) and \((\text{delete-queue} \ q)\) are errors.
If q is a queue, created by (make-queue) and subsequent procedures,
where i is the number of insertions and j is the number of deletions,
and \( x_i \) is the i’th 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) = \( x_{j+1} \)
• Implement queue as a list
The First Attempt

- implement queue as a list

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 to the front of the new element:
The First Attempt

(define (make-queue) nil)

(define (empty-queue? q)
  (null? q))
(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 (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)))))
The First Attempt: Orders of Growth
How efficient is the simple queue implementation? For a queue of length $n$:

- **time**: number of `cons`, `car`, `cdr` calls
- **space**: number of new `cons` cells
How efficient is the simple queue implementation? For a queue of length $n$:

- **time**: number of cons, car, cdr calls
- **space**: number of new cons cells

- **front-queue, delete-queue**
  - **time**: $\Theta(1)$ - constant
  - **space**: $\Theta(1)$ - constant
Queue Data Abstraction

The First Attempt: Orders of Growth

• How efficient is the simple queue implementation? For a queue of length $n$:
  • **time**: number of cons, car, cdr calls
  • **space**: number of new cons cells

• `front-queue`, `delete-queue`
  • **time**: $\Theta(1)$ - constant
  • **space**: $\Theta(1)$ - constant

• `insert-queue`
How efficient is the simple queue implementation? For a queue of length \( n \):

- **time**: number of \texttt{cons}, \texttt{car}, \texttt{cdr} calls
- **space**: number of new \texttt{cons} cells

- \texttt{front-queue}, \texttt{delete-queue}
  - **time**: \( \Theta(1) \) - constant
  - **space**: \( \Theta(1) \) - constant

- \texttt{insert-queue}
  - **time**: \( \Theta(n) \) - linear
  - **space**: \( \Theta(n) \) - linear
QUEUE DATA ABSTRACTION
A Better Implementation – Mutating

• **Constructor**
  
  \[(\text{make-queue})\]
  
  returns an empty queue

• **Selectors**
  
  \[(\text{front-queue } q)\]
  
  returns the object at the front of the queue.
  
  if queue is empty, signals error.
**QUEUE DATA ABSTRACTION**

**A Better Implementation — Mutating**

- **Constructor**
  
  \(\text{(make-queue)}\)

  returns an empty queue

- **Selectors**
  
  \(\text{(front-queue q)}\)

  returns the object at the front of the queue.
  if queue is empty, signals error.

- **Mutators**
  
  \(\text{(insert-queue! q elt)}\)

  inserts the elt at the rear of the queue
  and returns the \text{modified} queue
QUEUE DATA ABSTRACTION

A Better Implementation – Mutating

- **Constructor**
  
  `(make-queue)`
  
  returns an empty queue

- **Selectors**
  
  `(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
QUEUE DATA ABSTRACTION

A Better Implementation — Mutating

• Constructor
  (make-queue)  returns an empty queue

• Selectors
  (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
• attach a type tag as a defensive measure

• Maintain the queue *identity*
 QUEUE DATA ABSTRACTION
A Better Implementation – Mutating

- attach a type tag as a defensive measure
- Maintain the 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 DATA ABSTRACTION

Helper Procedures

(define (front-ptr q))

(define (rear-ptr q))
(define (front-ptr q) (cadr q))

(define (rear-ptr q) (cddr q))
(define (front-ptr q) (cadr q))

(define (rear-ptr q) (cddr q))

(define (set-front-ptr! q item) (set-car! (cdr q) item))
(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))
(define (make-queue)
  (cons 'queue (cons nil nil)))
(define (make-queue)
  (cons 'queue (cons nil nil)))

(define (queue? q)
  (and (pair? q)
       (eq? 'queue (car q))))
QUEUE DATA ABSTRACTION

A Better Implementation — Mutating

(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-ptr q) (cadr q))
(define (rear-ptr q) (cddr q))
(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))))
(define (insert-queue! q elt)
  (let ((new-pair (cons elt nil)))
    ...))
(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)))
(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-rear-ptr! q new-pair)
           (set-front-ptr! q new-pair))))
(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)))

(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))
(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)))))
(define (delete-queue! q)
  (cond ((empty-queue? q)
            (error "delete of empty queue:" q)))
(define (delete-queue! q)
  (cond ((empty-queue? q)
         (error "delete of empty queue:" q))
        (else ...)))
(define (delete-queue! q)
  (cond ((empty-queue? q)
           (error "delete of empty queue:" q))
        (else
         (set-front-ptr! q
                         (cdr (front-ptr q)))
         q)))
WHAT HAVE WE LEARNT?

Summary

• Built-in mutators which operate by \textit{side-effect}

  \begin{verbatim}
  set! ;(special form)
  set-car! ;Pair, anytype -> undef
  set-cdr! ;Pair, anytype -> undef
  \end{verbatim}
WHAT HAVE WE LEARNT?

Summary

• Built-in mutators which operate by \texttt{side-effect}
  
  \begin{verbatim}
  set! ;(special form)
  set-car! ;Pair, anytype \rightarrow \texttt{undef}
  set-cdr! ;Pair, anytype \rightarrow \texttt{undef}
  \end{verbatim}

• Extend our notion of data abstraction to include \texttt{mutators}

• \textbf{Mutation} is a powerful idea:
  
  • enables new and efficient data structures
  
  • can have surprising side effects
  
  • breaks our "functional" programming (substitution) model