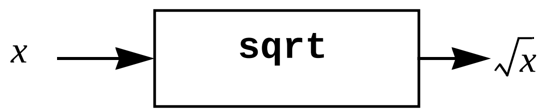
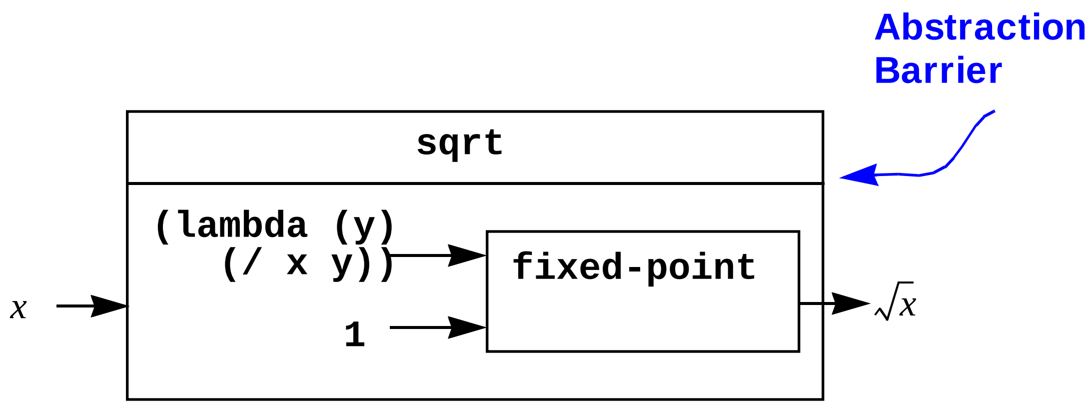
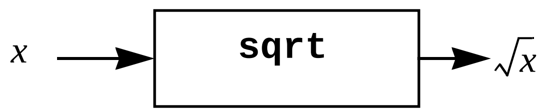
# Procedures: Black-Box Abstraction

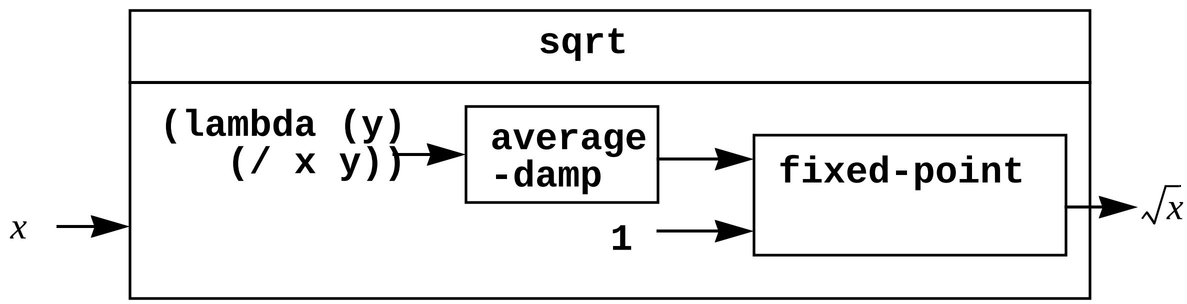


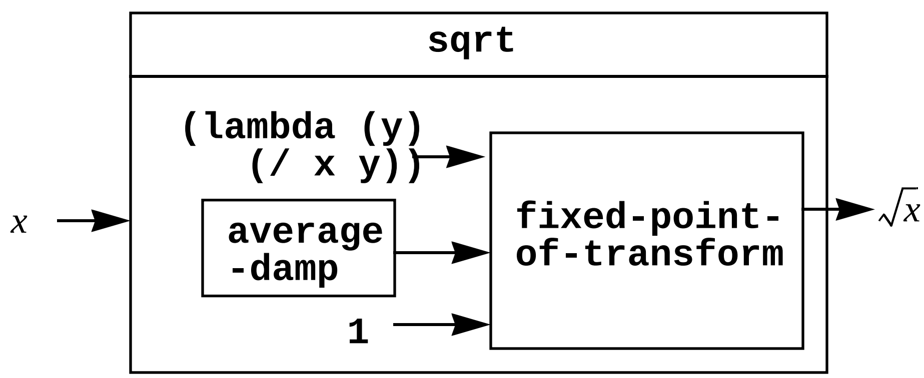



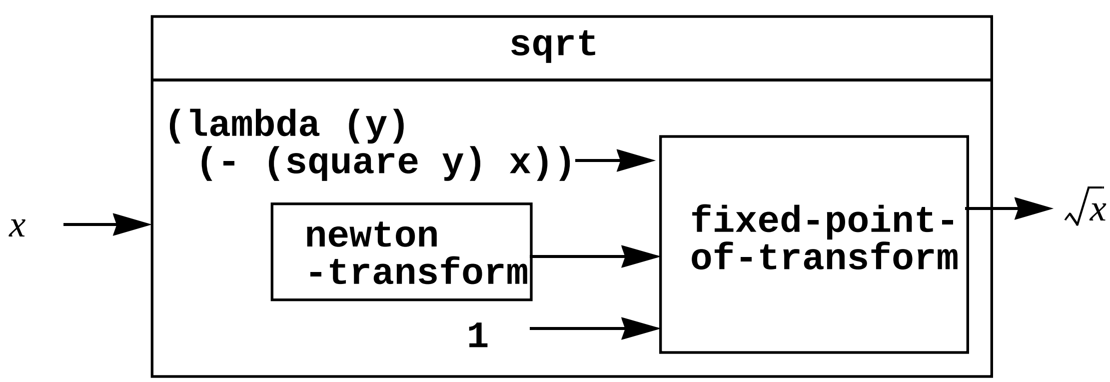

Modularity: small pieces which
may be COMBINED

# Black-Box Abstraction - Managing Complexity



# Point Implementation (PS #2)

```
; make-point: Num, Num -> Point
(define (make-point x y)
  (lambda (bit)
    (if (zero? bit) x y)))

; x-of: Point -> Num
(define (x-of point)
  (point 0))

; y-of: Point -> Num
(define (y-of point)
  (point 1))
```

Constructor

Accessors

# Check with Substitution Model

```
(x-of (make-point 10 20))
(x-of (lambda (bit) (if (zero? bit) 10 20)))
(x-of [proc (bit) (if (zero? bit) 10 20)])
([proc (bit) (if (zero? bit) 10 20)]   0)
(if (zero? 0) 10 20)
10
```

# Point Data Abstraction (PS #2)

## 1. Constructor

```
(make-point <x> <y>) -> given x & y coordinates,
                        create a new Point object
```

## 2. Accessors

```
(x-of <Point>)
(y-of <Point>)
```

## 3. Contract

```
(x-of (make-point <x> <y>)) = <x>
(y-of (make-point <x> <y>)) = <y>
```

## 4. Abstraction Barrier

**Say nothing about representation or implementation of Point!**

# Pair Abstraction

## 1. Constructor

```
; cons: T, T -> Pair
(cons <x> <y>) -> given x & y parts,
                  create a new Pair object
```

## 2. Accessors

```
; car, cdr: Pair -> T
(car <Pair>) -> the first part of the pair
(cdr <Pair>) -> the second part of the pair
```

## 3. Contract

```
(car (cons <x> <y>)) = <x>
(cdr (cons <x> <y>)) = <y>
```

## 4. Abstraction Barrier

**Say nothing about representation or implementation of pairs!**

# Rational Number Abstraction

## 1. Constructor

```
; make-rat: Int, Int -> RepRat
(make-rat <n> <d>) -> <RepRat>
```

## 2. Accessors

```
; numer, denom: RepRat -> Int
(numer <RepRat>)
(denom <RepRat>)
```

## 3. Contract

```
(numer (make-rat <n> <d>)) = <n>
(denom (make-rat <n> <d>)) = <d>
```

## 4. Abstraction Barrier

**Say nothing about representation or implementation of RepRat!**

## 5. Representation & Implementation

```
; RepRat = Int X Int
(define (make-rat n d) (cons n d))
(define (numer r) (car r))
(define (denom r) (cdr r))
```

# Layered Rational Number Operations

```
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

(define (*rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
```

# “Rationalizing” Implementation

```
(define (numer r)
  (let ((g (gcd (car r) (cdr r))))
    (/ (car r) g)))

(define (denom r)
  (let ((g (gcd (car r) (cdr r))))
    (/ (cdr x) g)))

(define (make-rat n d)
  (cons n d))

(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b))))
```

# Alternative "Rationalizing" Implementation

```
(define (numer r) (car r))

(define (denom r) (cdr r))

(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g)
          (/ d g))))

(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b))))
```

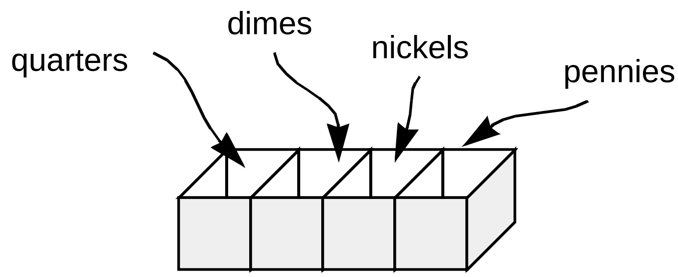
# Alternative +rat Operations

```
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
```

```
(define (+rat x y)
  (cons (+ (* (car x) (cdr y))
           (* (car y) (cdr x)))
        (* (cdr x) (cdr y))))
```


```
(define (+rat x y)
  (let ((n (+ (* (car x) (cdr y))
              (* (car y) (cdr x))))
        (d (* (cdr x) (cdr y))))
    (let ((g (gcd n d)))
      (cons (/ n g)
            (/ d g)))))
```

# Cash Register




## 1. Constructor

```
  ; make-register: Int, Int, Int, Int -> Reg
  (make-cash-register q d n p)
```

## 2. Accessors

```
  ; num-quarters: Reg -> Int
  (num-quarters <Reg>)
  (num-dimes <Reg>)  ...  etc.
```

## Layered Operations

```
(define (register-value reg)
  (+rat
    (+rat (*rat (make-rat (num-quarters reg) 1)
                (make-rat 1 4))
          (*rat (make-rat (num-dimes reg) 1)
                (make-rat 1 10)))
    (+rat (*rat (make-rat (num-nickels reg) 1)
                (make-rat 1 20))
          (*rat (make-rat (num-pennies reg) 1)
                (make-rat 1 100)))))
```

## Implementation

```
(define (make-cash-register q d n p)
  (list q d n p))
(define (num-quarters reg) (car reg)) ... etc.
```

# Bag of Coins

## 1. Constructor

```
; make-coin-bag: Int, RepRat -> CoinBag
(make-coin-bag <count> <coin-value>)
```

## 2. Accessors

```
(num-coins <CoinBag>)
(coin-value <CoinBag>)
```

## Layered Operations

```
(define (bag-value bag)
  (*rat (make-rat (num-coins bag) 1)
        (coin-value bag)))
```

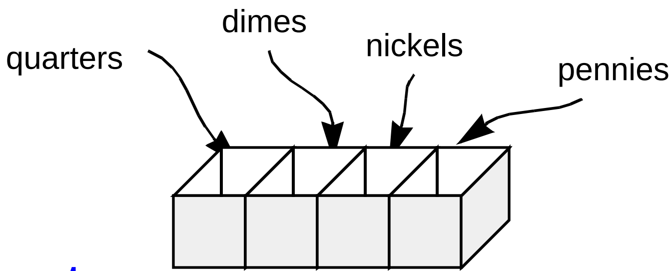
## Implementation

```
; CoinBag = Int X RepRat

(define (make-coin-bag count coin-value)
  (cons count coin-value))

(define (num-coins bag) (car bag))
(define (coin-value bag) (cdr bag))
```

# Cash Register - New Implementation




## 1. New Constructor

```
(define (make-cash-register q d n p)
  (list (make-coin-bag q (make-rat 1 4))
        (make-coin-bag d (make-rat 1 10))
        (make-coin-bag n (make-rat 1 20))
        (make-coin-bag p (make-rat 1 100))))
```

## Operations as Part of Implementation

```
(define (coins-in-register reg)
  (define (helper bag-list)
    (cond ((null? bag-list) 0)
          (else (+ (num-coins (car bag-list))
                   (helper (cdr bag-list))))))
  (helper reg))


(define (register-value reg)
  (define (helper bag-list)
    (cond ((null? bag-list) (make-rat 0 1))
          (else (+rat (bag-value (car bag-list))
                      (helper (cdr bag-list))))))
  (helper reg))
```