

Today's topics

- Abstractions
 - Procedural
 - Data
- Relationship between data abstraction and procedures that operate on it
- Isolating use of data abstraction from details of implementation

6.001 SICP

1/38

Procedural abstraction

• Process of procedural abstraction

- Define formal parameters, capture process in body of procedure
- Give procedure a name
- Hide implementation details from user, who just invokes name to apply procedure

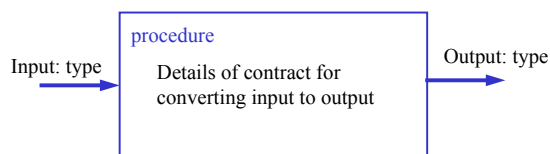
6.001 SICP

2/38

Procedural abstraction

• Process of procedural abstraction

- Define formal parameters, capture process in body of procedure
- Give procedure a name
- Hide implementation details from user, who just invokes name to apply procedure



6.001 SICP

3/38

Procedural abstraction example: sqrt

To find an approximation of square root of x:

- Make a guess G
- Improve the guess by averaging G and x/G
- Keep improving the guess until it is good enough

```

(define try (lambda (guess x)
  (if (good-enuf? guess x)
      guess
      (try (improve guess x) x))))

(define improve (lambda (guess x)
  (average guess (/ x guess))))

(define average (lambda (a b) (/ (+ a b) 2)))

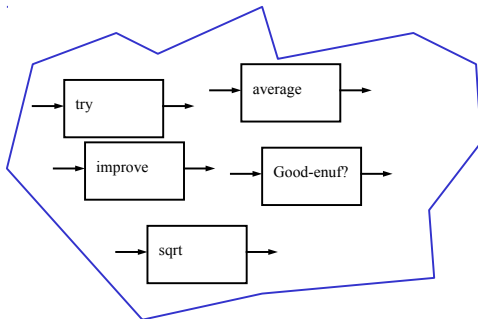
(define good-enuf? (lambda (guess x)
  (< (abs (- (square guess) x)) 0.001)))

(define sqrt (lambda (x) (try 1 x)))
  
```

6.001 SICP

4/38

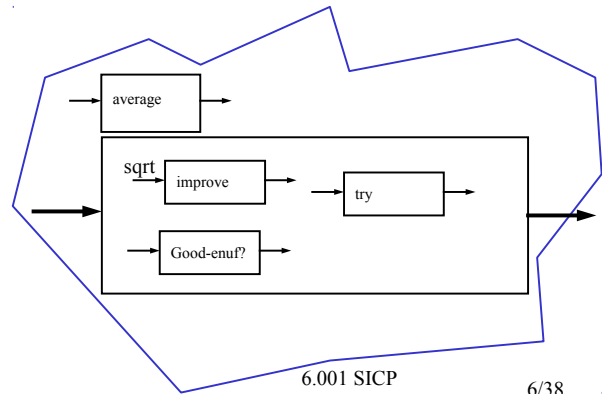
The universe of procedures for sqrt



6.001 SICP

5/38

The universe of procedures for sqrt



6.001 SICP

6/38

sqrt - Block Structure

```
(define sqrt
  (lambda (x)
    (define good-enuf? (lambda (guess)
      (< (abs (- (square guess) x))
        0.001)))
    (define improve (lambda (guess)
      (average guess (/ x guess))))
    (define sqrt-iter (lambda (guess)
      (if (good-enuf? guess)
          guess
          (sqrt-iter (improve guess)))))
    (sqrt-iter 1.0)))
```



6.001 SICP

7/38

Summary of part 1

- Procedural abstractions
 - Isolate details of process from its use
 - Designer has choice of which ideas to isolate, in order to support general patterns of computation

6.001 SICP

8/38

Language Elements

- **Primitives**
 - prim. data: numbers, strings, booleans
 - primitive procedures
- **Means of Combination**
 - procedure application
 - **compound data (today)**
- **Means of Abstraction**
 - naming
 - compound procedures
 - block structure
 - higher order procedures (next time)
 - **conventional interfaces – lists (today)**
 - **data abstraction**

6.001 SICP

9/38

Compound data

- Need a way of gluing data elements together into a unit that can be treated as a simple data element
- Need ways of getting the pieces back out
- Need a contract between the “glue” and the “unglue”
- Ideally want the result of this “gluing” to have the property of **closure**:
 - “the result obtained by creating a compound data structure can itself be treated as a primitive object and thus be input to the creation of another compound object”

6.001 SICP

10/38

Pairs (cons cells)

- `(cons <x-exp> <y-exp>) ==> <P>`
 - Where `<x-exp>` evaluates to a value `<x-val>`, and `<y-exp>` evaluates to a value `<y-val>`
 - Returns a **pair** `<P>` whose **car-part** is `<x-val>` and whose **cdr-part** is `<y-val>`
- `(car <P>) ==> <x-val>`
 - Returns the car-part of the pair `<P>`
- `(cdr <P>) ==> <y-val>`
 - Returns the cdr-part of the pair `<P>`

6.001 SICP

11/38

Compound Data

- Treat a PAIR as a single unit:
 - Can pass a pair as **argument**
 - Can return a pair as a **value**

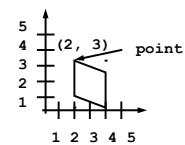
```
(define (make-point x y)
  (cons x y))

(define (point-x point)
  (car point))

(define (point-y point)
  (cdr point))

(define (make-seg pt1 pt2)
  (cons pt1 pt2))

(define (start-point seg)
  (car seg))
```



6.001 SICP

12/38

Pair Abstraction

• Constructor

```
; cons: A,B -> A X B
; cons: A,B -> Pair<A,B>
  (cons <x> <y>) ==> <P>
```

• Accessors

```
; car: Pair<A,B> -> A
  (car <P>) ==> <x>
; cdr: Pair<A,B> -> B
  (cdr <P>) ==> <y>
```

• Predicate

```
; pair? anytype -> boolean
  (pair? <z>)
    ==> #t if <z> evaluates to a pair, else #f
```

6.001 SICP

13/38

Pair abstraction

- Note how there exists a contract between the constructor and the selectors:

- $(car (cons <a>)) \rightarrow <a>$
- $(cdr (cons <a>)) \rightarrow $

- Note how pairs have the property of closure – we can use the result of a pair as an element of a new pair:

- $(cons (cons 1 2) 3)$

6.001 SICP

14/38

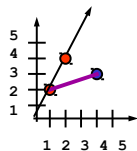
Using pair abstractions to build procedures

- Here are some data abstractions

```
(define p1 (make-point 1 2))
(define p2 (make-point 4 3))
(define s1 (make-seg p1 p2))

(define stretch-point
  (lambda (pt scale)
    (make-point (* scale (point-x pt))
                (* scale (point-y pt)))))

(stretch-point p1 2) → (2 . 4)
p1 → (1 . 2)
```



6.001 SICP

15/38

Using pair abstractions to build procedures

- Generalize to other structures

```
(define stretch-seg
  (lambda (seg sc)
    (make-seg (stretch-point (start-pt seg) sc)
              (stretch-point (end-pt seg) sc))))

(define seg-length
  (lambda (seg)
    (sqrt (+ (square (- (point-x (start-point seg))
                        (point-x (end-point seg))))
             (square (- (point-y (start-point seg))
                        (point-y (end-point seg))))))))
```

6.001 SICP

16/38

Grouping together larger collections

- Suppose we want to group together a set of points. Here is one way

```
(cons (cons (cons (cons p1 p2)
                  (cons p3 p4))
        (cons (cons p5 p6)
              (cons p7 p8)))
      p9)
```

- UGH!!** How do we get out the parts to manipulate them?

6.001 SICP

17/38

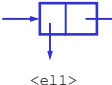
Conventional interfaces -- lists

- A list is a data object that can hold an arbitrary number of ordered items.
- More formally, a list is a sequence of pairs with the following properties:
 - Car-part of a pair in sequence – holds an item
 - Cdr-part of a pair in sequence – holds a pointer to rest of list
 - Empty-list `nil` – signals no more pairs, or end of list
- Note that lists are closed under operations of `cons` and `cdr`.

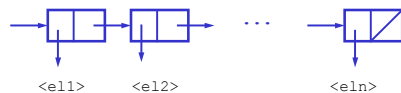
6.001 SICP

18/38

Conventional Interfaces - Lists

`(cons <e1> <e2>)` 

`(list <e1> <e2> ... <eln>)`



Predicate

`(null? <z>)`

\Rightarrow `#t` if `<z>` evaluates to empty list

6.001 SICP

19/38

... to be really careful

- For today we are going to create different constructors and selectors for a list
 - `(define first car)`
 - `(define rest cdr)`
 - `(define adjoin cons)`
- Note how these abstractions inherit closure from the underlying abstractions!

6.001 SICP

20/38

Common Pattern #1: cons'ing up a list

```
(define (enumerate-interval from to)
  (if (> from to)
      nil
      (adjoin from
               (enumerate-interval
                (+ 1 from)
                to)))))

(e-i 2 4)
(if (> 2 4) nil (adjoin 2 (e-i (+ 1 2) 4)))
(if #f nil (adjoin 2 (e-i 3 4)))
(adjoin 2 (e-i 3 4))
(adjoin 2 (adjoin 3 (e-i 4 4)))
(adjoin 2 (adjoin 3 (adjoin 4 (e-i 5 4))))
(adjoin 2 (adjoin 3 (adjoin 4 nil)))
```

```
(adjoin 2 (adjoin 3 (→ [ ])))
```

```
(adjoin 2 (→ [ ] → [ ] → [ ]))
```

```
→ [ ] → [ ] → [ ] ⇒ (2 3 4)
```

6.001 SICP

21/38

Common Pattern #2: cdr'ing down a list

```
(define (list-ref lst n)
  (if (= n 0)
      (first lst)
      (list-ref (rest lst)
                 (- n 1))))
```

joe → [] → [] → [] (list-ref joe 1)

```
(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (rest lst)))))
```

6.001 SICP

22/38

Cdr'ing and Cons'ing Examples

```
(define (copy lst)
  (if (null? lst)
      nil ; base case
      (adjoin (first lst) ; recursion
               (copy (rest lst)))))
```

```
(append (list 1 2) (list 3 4))
⇒ (1 2 3 4)
```

Strategy: "copy" list1 onto front of list2.

```
(define (append list1 list2)
  (cond ((null? list1) list2) ; base
        (else
         (adjoin (first list1) ; recursion
                  (append (rest list1)
                          list2))))))
```

6.001 SICP

23/38

Common Pattern #3: Transforming a List

```
(define group (list p1 p2 ... p9))

(define stretch-group
  (lambda (gp sc)
    (if (null? gp)
        nil
        (adjoin (stretch-point (first gp) sc)
                  (stretch-group (rest gp) sc)))))
```

Note how application of **stretch-group** to a list of points nicely separates the operations on points from the operations on the group, without worrying about details. Note also how this procedure walks down a list, creates a new point, and cons'es up a new list of points.

6.001 SICP

24/38

Common Pattern #3: Transforming a List

```
(define add-x (lambda (gp)
  (if (null? gp)
      0
      (+ (point-x (first gp))
          (add-x (rest gp)))))
(define add-y (lambda (gp)
  (if (null? gp)
      0
      (+ (point-y (first gp))
          (add-y (rest gp)))))
(define centroid (lambda (gp)
  (let ((x-sum (add-x gp))
        (y-sum (add-y gp))
        (how-many (length gp)))
    (make-point (/ x-sum how-many)
                 (/ y-sum how-many)))))
```

6.001 SICP

25/38

Lessons learned

- There are conventional ways of grouping elements together into compound data structures.
- The procedures that manipulate these data structures tend to have a form that mimics the actual data structure.
- Compound data structures rely on an inductive format in much the same way recursive procedures do. We can often deduce properties of compound data structures in analogy to our analysis of recursive procedures by using induction.

6.001 SICP

26/38

Pairs, Lists, & Trees

Trees

- user *knows* trees are also lists

Lists

- user *knows* lists are also pairs

Pairs

- contract for `cons`, `car`, `cdr`

- Conventions that enable us to think about lists and trees.

- Specified the implementations for lists and trees:
weak *data abstraction*

- How build *stronger abstractions*?

6.001 SICP

27/38

Elements of a Data Abstraction

-- Pair Abstraction --

1. Constructor

```
; cons: A, B -> Pair<A,B>; A & B = anytype
(cons <x> <y>) ==> <p>
```

2. Accessors

```
(car <p>) ; car: Pair<A,B> -> A
(cdr <p>) ; cdr: Pair<A,B> -> B
```

3. Contract

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

4. Operations

```
; pair?: anytype -> boolean
(pair? <p>)
```

5. Abstraction Barrier

Say nothing about implementation!

6. Concrete Representation & Implementation

Could have alternative implementations!

28/38

Rational numbers as an example

- A rational number is a ratio n/d
- $a/b + c/d = (ad + bc)/bd$
- $a/b * c/d = (ac)/(bd)$

6.001 SICP

29/38

Rational Number Abstraction

1. Constructor

```
; make-rat: integer, integer -> Rat
(make-rat <n> <d>) -> <r>
```
2. Accessors

```
; numer, denom: Rat -> integer
(numer <r>)
(denom <r>)
```
3. Contract

```
(numer (make-rat <n> <d>)) ==> <n>
(denom (make-rat <n> <d>)) ==> <d>
```
4. Layered Operations

```
(print-rat <r>) prints rat
(+rat x y) ; +rat: Rat, Rat -> Rat
(*rat x y) ; *rat: Rat, Rat -> Rat
```
5. Abstraction Barrier

Say nothing about implementation!

6.001 SICP

30/38

Rational Number Abstraction

1. Constructor
2. Accessors
3. Contract
4. Layered Operations
5. Abstraction Barrier

6. Concrete Representation & Implementation

```
; Rat = Pair<integer, integer>
(define (make-rat n d) (cons n d))
(define (numer r) (car r))
(define (denom r) (cdr r))
```

6.001 SICP

31/38

Alternative Rational Number Abstraction

1. Constructor
2. Accessors
3. Contract
4. Layered Operations
5. Abstraction Barrier

6. Concrete Representation & Implementation

```
; Rat = List
(define (make-rat n d) (list n d))
(define (numer r) (car r))
(define (denom r) (cadr r))
```

6.001 SICP

32/38

print-rat Layered Operation

```
; print-rat: Rat -> undef
(define (print-rat rat)
  (display (numer rat))
  (display "/")
  (display (denom rat)))
```

6.001 SICP

33/38

Layered Rational Number Operations

```
; +rat: Rat, Rat -> Rat
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

; *rat: Rat, Rat -> Rat
(define (*rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
```

6.001 SICP

34/38

Using our system

```
* (define one-half (make-rat 1 2))
* (define three-fourths (make-rat 3 4))
* (define new (+rat one-half three-fourths))

(numer new) → 10
(denom new) → 8
Oops – should be 5/4 not 10/8!!
```

6.001 SICP

35/38

“Rationalizing” Implementation

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

Strategy: remove common factors when **access** numer and denom

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

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

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

6.001 SICP

36/38

Alternative “Rationalizing” Implementation

- Strategy: remove common factors when **create** a rational number

```
(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))))
```

6.001 SICP

37/38

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)))))
```

6.001 SICP

38/38