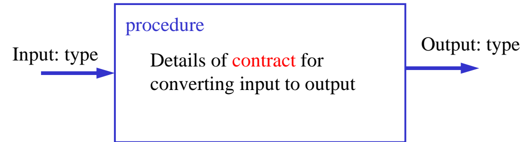
Today's topic: Abstraction

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

Procedural abstraction

- **Process of procedural abstraction**
  - Define formal parameters, capture pattern of computation as a process in body of procedure
  - Give procedure a name
  - Hide implementation details from user, who just invokes name to apply procedure

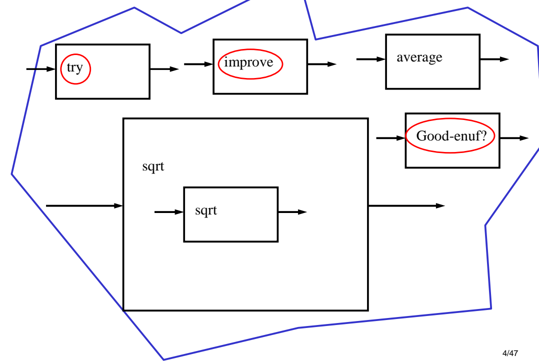


Procedural abstraction example: sqrt

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

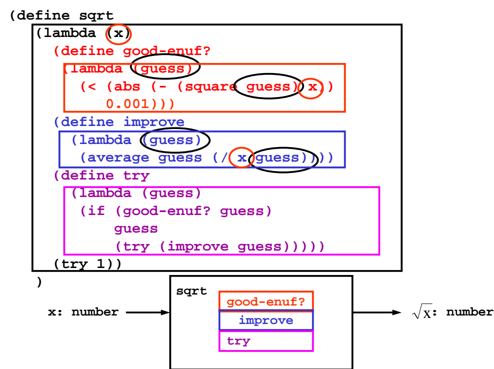
```
(define try (lambda (guess x)
              (if (good-enuf? guess x)
                  guess
                  (try (improve guess x) x))))
(define good-enuf? (lambda (guess x)
                     (< (abs (- (square guess) x)) 0.001)))
(define improve (lambda (guess x)
                  (average guess (/ x guess))))
(define average (lambda (a b) (/ (+ a b) 2)))
(define sqrt (lambda (x) (try 1 x)))
```


The universe of procedures for sqrt



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 try
      (lambda (guess)
        (if (good-enuf? guess)
            guess
            (try (improve guess)))))
    (try 1))
)
```



Typecasting

- We are going to find that it is convenient to reason about procedures (and data structures) in terms of the number and kinds of arguments, and the kind of output they produce
- We call this **typing** of a procedure or data structure

Types – a motivation

```
(+ 5 10) ==> 15

(+ "hi" 5)
 ;The object "hi", passed as the first argument to
 integer-add, is not the correct type
```

- Addition is not defined for strings

Types – simple data

- We want to collect a taxonomy of expression types:
  - Simple Data
    - Number
      - Integer
      - Real
      - Rational
    - String
    - Boolean
    - Names (symbols)
- We will use this for notational purposes, to reason about our code. Scheme checks types of arguments for built-in procedures, but *not for user-defined ones.*

Types – procedures

- Because procedures operate on objects and return values, we can define their types as well.
- We will denote a procedures type by indicating the types of each of its arguments, and the type of the returned value, plus the symbol → to indicate that the arguments are mapped to the return value
- E.g. **number → number** specifies a procedure that takes a number as input, and returns a number as value

Types

- `(+ 5 10) ==> 15`
  `(+ "hi" 5)`
  `;The object "hi", passed as the first argument to integer-add, is not the correct type`
- Addition is not defined for strings
- The type of the integer-add procedure is

`number, number → number`

two arguments, both numbers

result value of integer-add is a number

Why "integer-add"?

Type examples

expression:	evaluates to a value of type:
`15`	`number`
`"hi"`	`string`
`square`	`number → number`
`>`	`number,number → boolean`

`(> 5 4) ==> #t`

- The type of a procedure is a contract:
  - If the operands have the specified types, the procedure will result in a value of the specified type
  - otherwise, its behavior is undefined
    - maybe an error, maybe random behavior

Types, precisely

- A type describes a set of scheme values
  - `number → number` describes the set:

    all procedures, whose result is a number,
    which require one argument that must be a number

- Every scheme value has a type
  - Some values can be described by multiple types
  - If so, choose the type which describes the largest set
- Special form keywords like `define` do not name values
  - therefore special form keywords have no type

Your turn

- The following expressions evaluate to values of what type?

```
(lambda (a b c) (if (> a 0) (+ b c) (- b c)))
```

```
(lambda (p) (if p "hi" "bye"))
```

```
(* 3.14 (* 2 5))
```


Summary of types

- type: a set of values
- every value has a type
- procedure types (types which include →) indicate
  - number of arguments required
  - type of each argument
  - type of result of the procedure
- Types: a mathematical theory for reasoning efficiently about programs
  - useful for preventing certain common types of errors
  - basis for many analysis and optimization algorithms

Compound data

- Need a way of (procedure for) gluing data elements together into a unit that can be treated as a simple data element
- Need ways of (procedures for) 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”

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>`

Pairs: Nano Quiz

- `(define p1 (cons (+ 3 2) 4))`
- `(car P1) ==>`
- `(cdr P1) ==>`

Pairs Are A Data 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>
```

- Contract

```
; (car (cons <a> <b> )) ➔ <a>
; (cdr (cons <a> <b> )) ➔ <b>
```

- Operations

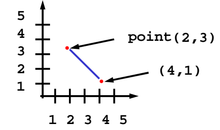
```
; pair? anytype -> boolean
  (pair? <z>)
    ==> #t if <z> evaluates to a pair, else #f
```

Pair Abstraction

- Once we build a pair, we can treat it as if it were a primitive (e.g. the same way we treat a number)
- Pairs have the property of closure, meaning we can use a pair anywhere we would expect to use a primitive data element :
  - *(cons (cons 1 2) 3)*

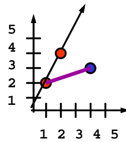
Building Additional Data Abstractions

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

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

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

(define P1 (make-point 2 3)
(define P2 (make-point 4 1)

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

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

(define S1 (make-seg P1 P2))
```



Using 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)))))
```

Constructor

Selector

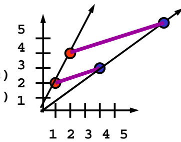
```
(stretch-point p1 2) ➔ (2 . 4)

p1 ➔ (1 . 2)
```

I now have a contract for stretch-point – given a point as input, it returns a point as output – and it doesn't care about how points are created!!

Using Data Abstractions

- 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))))))))
```

Selector for point

Selector for segment

Once again, I have a contract – given a segment as input, it returns a segment as output – and it doesn't care about how segments (or points) are created!!

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?

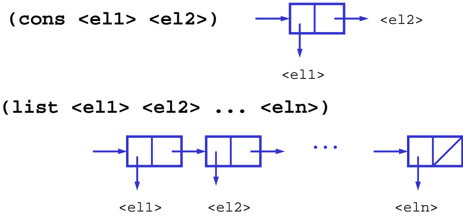
Conventional interfaces -- Lists

- 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
  - Terminates in an empty-list `'()` – signals no more pairs, or end of list
- Note that lists are closed under operations of `cons` and `cdr`.

Conventional Interfaces -- Lists

```
(cons <el1> <el2>)
```

```
(list <el1> <el2> ... <eln>)
```




```
(list 1 2 3 4)  ➔ (1 2 3 4)
```

Predicate
`(null? <z>)`
`==> #t` if `<z>` evaluates to empty list

Types – compound data

- Pair<A,B>
  - A compound data structure formed by a cons pair, in which the first element is of type A, and the second of type B: e.g. (cons 1 2) has type Pair<number, number>
- List<A>=Pair<A, List<A> or '()>
  - A compound data structure that is recursively defined as a pair, whose first element is of type A, and whose second element is either a list of type A or the empty list.
    - E.g. (list 1 2 3) has type List<number>; while (list 1 "string" 3) has type List<number | string>

Examples

25	; Number
3.45	; Number
"this is a string"	; String
(> a b)	; Boolean
(cons 1 3)	; Pair<Number, Number>
(list 1 2 3)	; List<Number>
(cons "foo" (cons "bar" '()))	; List<String>

… to be really careful

- For today we are going to create different constructors and selectors for a list, to distinguish from pairs …
  - `(define first car)`
  - `(define rest cdr)`
  - `(define adjoin cons)`
- These abstractions inherit closure from the underlying abstractions

Common patterns of data manipulation

- Have seen common patterns of procedures
- When applied to data structures, often see common patterns of procedures as well
  - Procedure pattern reflects recursive nature of data structure
  - Both procedure and data structure rely on
    - Closure of data structure
    - Induction to ensure correct kind of result returned

Common pattern #1: cons'ing up a list

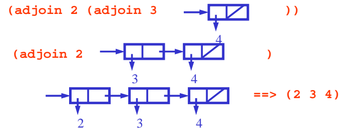
- Motivation:

```
(define 1thru4 (lambda() (list 1 2 3 4)))

(define (2thru7) (list 2 3 4 5 6 7))

...
```

Common pattern #1: cons'ing up a list

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

```
(e-i 2 4)
(if (> 2 4) '() (adjoin 2 (e-i (+ 1 2) 4)))
(if #f '() (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 '())))
```



Common pattern #2: cdr'ing down a list

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

Note how induction ensures that code is correct – relies on closure property of data structure

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


Cdr'ing and Cons'ing Examples

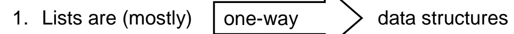
```
(define (copy lst)
  (if (null? lst)             ; test
      '()                     ; base case
      (adjoin (first lst)     ; recursion
              (copy (rest lst)))))
```

```
(append (list 1 2) (list 3 4))
==> (1 2 3 4)
```

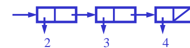
Strategy: "copy" list1 onto front of list2.

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


Some facts of lists




```
(define x (list 2 3 4))
(car x) =>
(car (cdr x)) =>
```




```
2. (cdr(cdr(cdr(cdr x)))) =>
```


Common Pattern #3: Transforming a List

```
(define group (list p1 p2 … p9))

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

`stretch-group` separates operations on points from operations on the group

Walks (cdr's) down the list, creates a new point, cons'es up a new list of points.

Common Pattern #3: Transforming a List

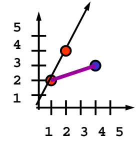
```
(define add-x (lambda (gp)
                (if (null? gp)
                    0
                    (+ (point-x (first gp))
                       (add-x (rest gp))))))
(define add-y (lambda (gp)
                (if (null? gp)
                    0
                    (+ (point-y (first gp))
                       (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)))))
```

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.

Using 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)))))
```

Selector

car?

```
(stretch-point p1 2) → (2 . 4)

p1 → (1 . 2)
```


Abstractions Have Two Communities

- *Builders*

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

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

- *Users*

```
(* scale (point-x pt))
```

- *Frequently the same person/people*
- *Schizophrenia can be a solid foundation for good programming style*

Elements of a Data Abstraction

-- Pair Abstaction --

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

IGNORANCE NEED TO KNOW

6. Concrete Representation & Implementation

Could have alternative implementations!

Rational Number Abstraction

- A rational number is a ratio n/d
- a/b + c/d = (ad + bc)/bd
  - 2/3 + 1/4 = (2*4 + 3*1)/12 = 11/12
- a/b * c/d = (ac)/(bd)
  - 2/3 * 1/3 = 2/9

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. 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!

Rational Number Abstraction

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

---

6. Concrete Representation & Implementation

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


Rational Number Abstraction

1. Constructor
2. Accessors
3. Contract
4. 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))
```


Rational Number Abstraction

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

---

6. Concrete Representation & Implementation

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


Rational Number Abstraction

1. Constructor
2. Accessors
3. Contract
4. 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))
```


Rational Number Abstraction

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

---

6. Concrete Representation & Implementation

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


Rational Number Abstraction

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

---

6. Concrete Representation & Implementation

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

Additional 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))))
```

53/47

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!!

54/47

"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)
    (/ (car r) (gcd (car r) (cdr r))))

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

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

55/47

Alternative "Rationalizing" Implementation

- Strategy: remove common factors when create a rational number

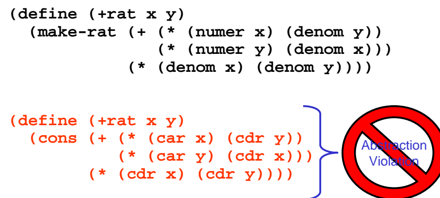
```
(define (numer r) (car r))

(define (denom r) (cdr r))

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

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

Either implementation is fine – most importantly no other code has to change if I switch from one to the other!!

56/47

Alternative +rat Operations

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



57/47

Lessons learned

- Valuable to build strong abstractions
  - Hide details behind names of accessors and constructors
  - Rely on closure of underlying implementation
- Enables user to change implementation without having to change procedures that use abstraction
- Data abstractions tend to have procedures whose structure mimics their inherent structure

58/47