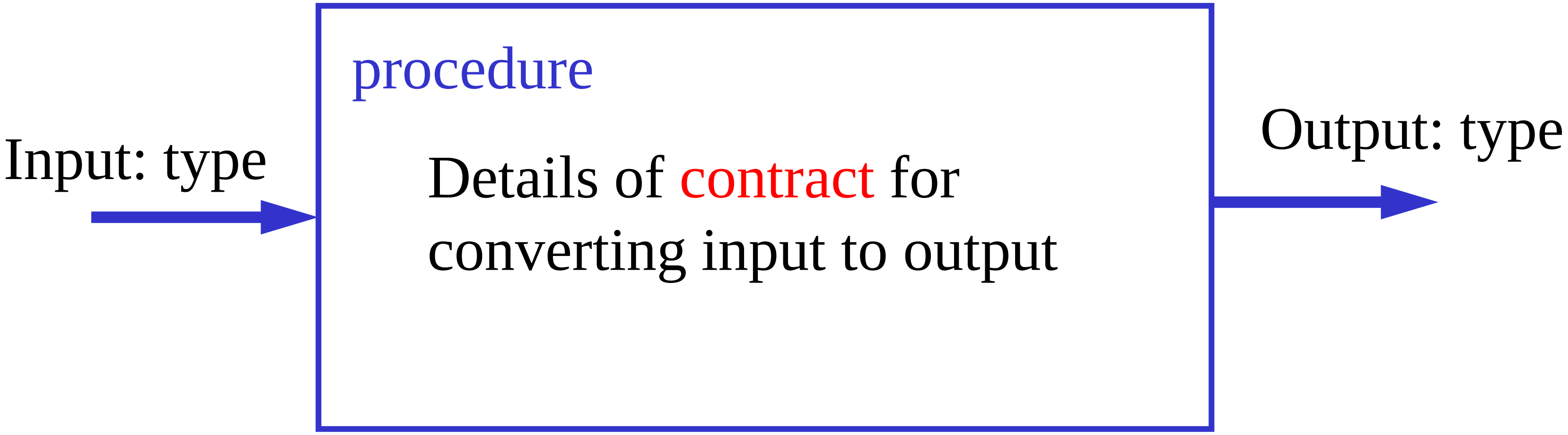
Higher-Order Procedures

- Today's topics
  - Procedural abstractions
  - Capturing patterns across procedures – Higher Order Procedures

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



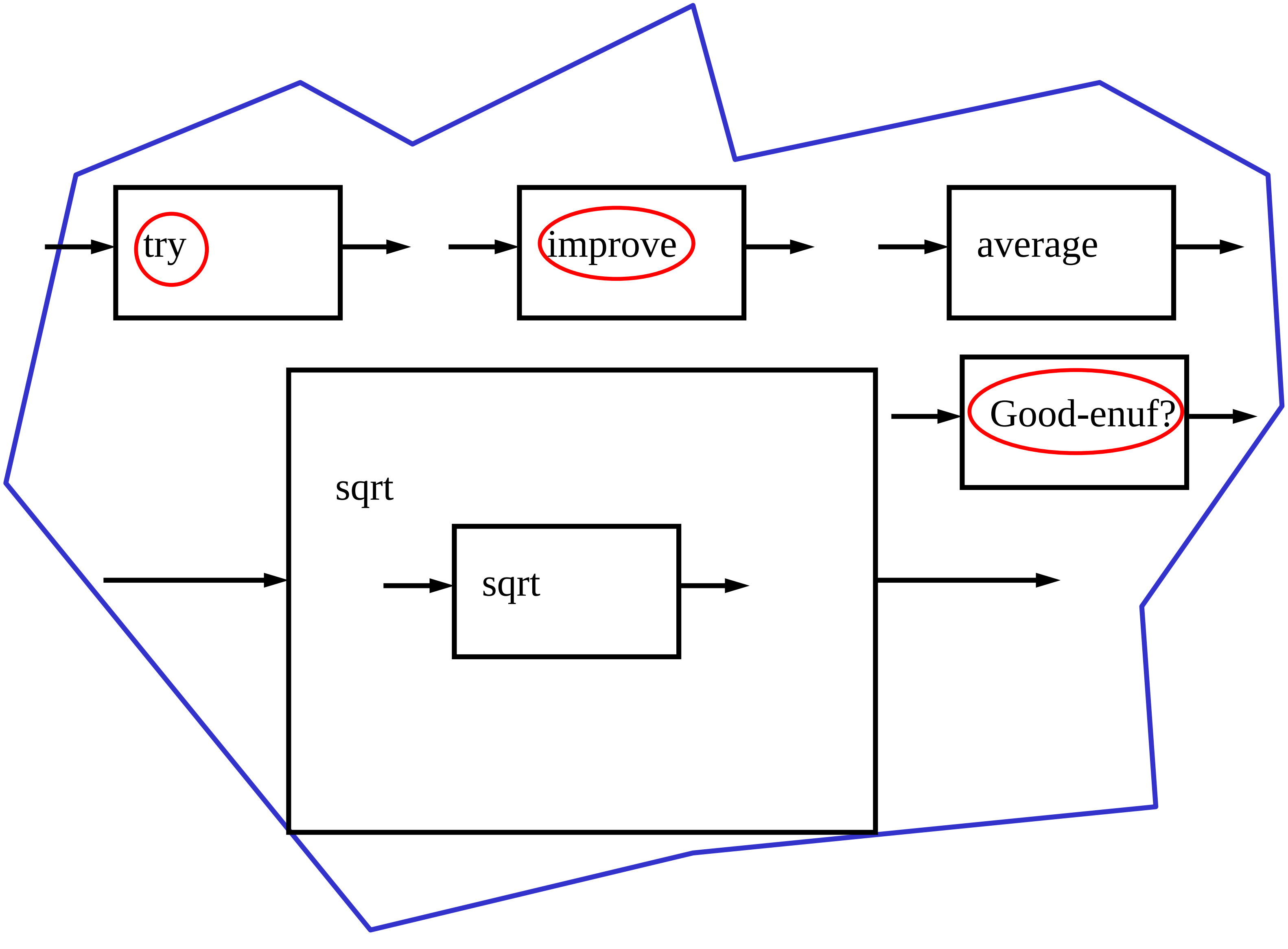
Procedural abstraction example: sqrt

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

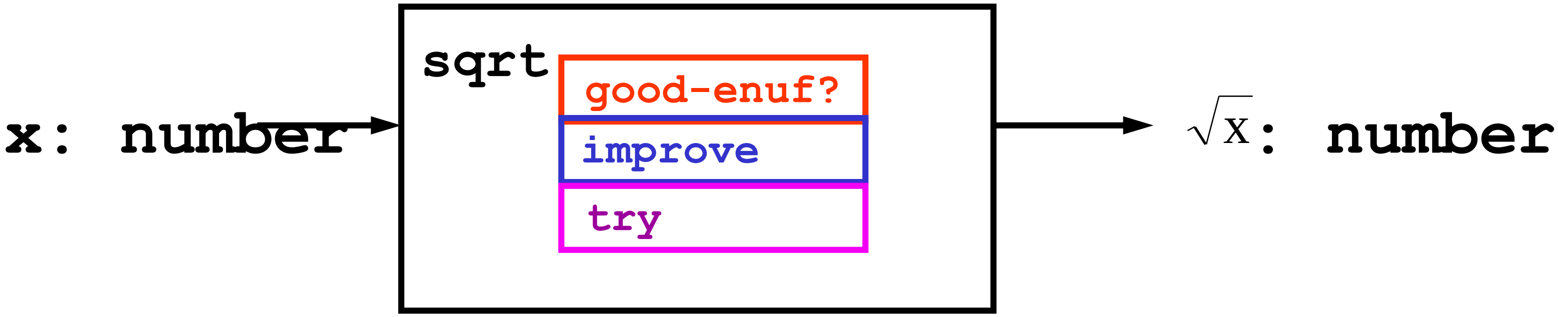
```
(define try (lambda (guess x)
                 (if (good-enuf? guess x)
                     guess
                     (try (improve guess x) x))))
(define good-enuf? (lambda (guess x)
                      (< (abs (- (square guess) x)) 0.001)))
(define improve (lambda (guess x)
                    (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
  ```

Type examples

- expression: evaluates to a value of type:

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)))
```

number, number, number $\longrightarrow$ number

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

Boolean $\longrightarrow$ string

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

number

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

What is procedure abstraction?

Capture a common pattern

```
(* 2 2)
(* 57 57)
(* k k)
```

```
(lambda (x) (* x x))
```

Actual pattern

Formal parameter for pattern

Give it a name (define square (lambda (x) (* x x)))

Note the type: number → number

Other common patterns

- 1 + 2 + … + 100
- 1 + 4 + 9 + … + 100^2
- 1 + $1/3^2$ + $1/5^2$ + … + $1/101^2$ (= $\pi^2/8$)

$$\sum_{k=1}^{100} k$$

$$\sum_{k=1}^{100} k^2$$

$$\sum_{k=1,odd}^{101} k^{-2}$$

```
(define (sum-integers a b)
   (if (> a b)
       0
       (+ a (sum-integers (+ 1 a) b))))

(define (sum-squares a b)
   (if (> a b)
       0
       (+ (square a)
          (sum-squares (+ 1 a) b))))

(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1 (square a))
         (pi-sum (+ a 2) b))))
```

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

Let’s examine this new procedure

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

What is the type of this procedure?

$(num \rightarrow num,\ num,\ num \rightarrow num,\ num) \rightarrow num$

1. What type is the output?
2. How many arguments?
3. What type is each argument?

Higher order procedures

- A higher order procedure:
  takes a procedure as an *argument* or returns one as a *value*

```
(define (sum-integers a b)
      (if (> a b)
          0
          (+ a (sum-integers (+ 1 a) b))))

(define (sum term a next b)
   (if (> a b)
       0
       (+ (term a)(sum term (next a) next b))))

(define (sum-integers1 a b)
 (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))
```

Higher order procedures

```
(define (sum-squares a b)
      (if (> a b)
          0
          (+ (square a)
              (sum-squares (+ 1 a) b))))

(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)(sum term (next a) next b))))

(define (sum-squares1 a b)
       (sum square a (lambda (x) (+ x 1)) b))
```

Higher order procedures

```
(define (pi-sum a b)
     (if (> a b)
         0
         (+ (/ 1 (square a))
             (pi-sum (+ a 2) b))))

(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)(sum term (next a) next b))))

(define (pi-sum1 a b)
     (sum (lambda (x) (/ 1 (square x))) a
          (lambda (x) (+ x 2)) b))
```

Higher order procedures

- Takes a procedure as an argument or returns one as a value

```
(define (sum-integers1 a b)
       (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))

(define (sum-squares1 a b)
       (sum square a (lambda (x) (+ x 1)) b))

(define (add1 x) (+ x 1))

(define (sum-squares1 a b)(sum square a add1 b))

(define (pi-sum1 a b)
     (sum (lambda (x) (/ 1 (square x))) a
          (lambda (x) (+ x 2)) b))

(define (add2 x) (+ x 2))

(define (pi-sum1 a b)
    (sum (lambda (x) (/ 1 (square x))) a add2 b))
```

Returning A Procedure As A Value

```
(define (add1 x) (+ x 1))
(define (add2 x) (+ x 2))

(define incrementby (lambda (n) . . . ))

(define add1 (incrementby 1))
(define add2 (incrementby 2))
. . .
(define add37.5 (incrementby 37.5))
```

incrementby: # → (# → #)

Returning A Procedure As A Value

```
(define incrementby
      (lambda(n)                        ))

  (incrementby                           2) →
( (lambda(n)(lambda (x) (+ x n))) 2)

      (lambda (x) (+ x 2))
```

```
(incrementby 2) → a procedure of one var (x) that
                  increments x by 2

((incrementby 3) 4) → ?

( (lambda(x)(+ x 3))  4) →
```

Nano-Quiz/Lecture Problem

```
(define incrementby
      (lambda(n)(lambda (x) (+ x n))))

(define f1 (incrementby 6)) → ?

(f1 4) →



(define f2 (lambda (x)(incrementby 6))) → ?

(f2 4)     → ?
((f2 4) 6) → ?
```

Procedures as values: Derivatives

$$f: x \to x^2 \qquad\qquad f: x \to x^3$$

$$f': x \to 2x \qquad\qquad f': x \to 3x^2$$

- Taking the derivative is a function: $D(f) = f'$
- What is its *type?*

D: (# → #) → (# → #)

Computing derivatives

- A good approximation:

$$Df(x) \approx \frac{\underline{f(x+\varepsilon)} - \underline{f(x)}}{\underline{\varepsilon}}$$

```
(define deriv
  (lambda (f)
    (lambda (x) (/ (- (f (+ x epsilon)) (f x))
                                epsilon)) ))
```

(number → number) → (number→ number)

Using “deriv”

```
(define square (lambda (y) (* y y)) )
(define epsilon 0.001)

((deriv square) 5)    (define deriv
                       (lambda (f)
                        (lambda (x) (/ (- (f (+ x epsilon))
                                          (f x))
                                       epsilon)) ))
```

Finding fixed points of functions

Square root of x is defined by $\sqrt{x} = x / \sqrt{x}$

Think of as a transformation $f : y \to \dfrac{x}{y}$ then if we can find a $y = \sqrt{x}$, then $f(y) = y$, and such a y is called a fixed point of f.

- Here's a common way of finding fixed points
  - Given a guess x_1, let new guess by $f(x_1)$
  - Keep computing f of last guess, till close enough

```
(define (close? u v)   (< (abs (- u v)) 0.0001))

(define (fixed-point f i-guess)

     (define (try g)

        (if (close? (f g) g)

            (f g)

            (try (f g))))

    (try i-guess))
```

Using fixed points

(fixed-point (lambda (x) (+ 1 (/ 1 x))) 1)
→ 1.6180

or $x = 1 + 1/x$ when $x = (1 + \sqrt{5})/2$

```
(define (sqrt x)
  (fixed-point
     (lambda (y) (/ x y))
     1))
```

$$y = x/y$$

$$y^2 = x$$

$$y = \sqrt{x}$$

Unfortunately if we try (sqrt 2), this oscillates between 1, 2, 1, 2,

```
(define (fixed-point f i-guess)
   (define (try g)
     (if (close? (f g) g)
        (f g)
        (try (f g))))
  (try i-guess))
```

So damp out the oscillation

```
(define (average-damp f)
  (lambda (x)
     (average x (f x))))
```

Check out the type:

(number → number) → (number → number)

that is, this takes a procedure as input, and returns a **NEW** procedure as output!!!

- ((average-damp square) 10)
- ((lambda (x) (average x (square x))) 10)
- (average 10 (square 10))
- 55

… which gives us a clean version of sqrt

```
(define (sqrt x)
  (fixed-point
    (average-damp
       (lambda (y) (/ x y)))
    1))
```

Compare this to Heron's algorithm (the one we saw earlier)
– same process, but ideas intertwined with code

```
(define (cbrt x)
  (fixed-point
    (average-damp
       (lambda (y) (/ x (square y))))
    1))
```

Procedures as arguments: a more complex example

- `(define compose (lambda (f g x) (f (g x))))`

```
(compose square double 3)
(square (double 3))
(square (* 3 2))
(square 6)
(* 6 6)
36
```

What is the type of compose? Is it:

`(number → number), (number → number), number → number`

No! Nothing in compose requires a number

Compose works on other types too

```
(define compose (lambda (f g x) (f (g x))))
```

```
(compose
 (lambda (p) (if p "hi" "bye"))
 (lambda (x) (> x 0))
 -5
 )  ==> "bye"
```

```
boolean → string
number → boolean
number
result: a string
```

Will any call to compose work?

```
(compose < square 5)
```

wrong *number of args* to <

```
<: number, number → boolean
```

```
(compose square double "hi")
```

wrong *type of arg* to double

```
double: number → number
```

Type of compose

```
(define compose (lambda (f g x) (f (g x))))
```

- Use type variables.

compose: $(B \to C), (A \to B), A \to C$

- Meaning of type variables:

  All places where a given type variable appears must match when you fill in the actual operand types

- The constraints are:
  - F and G must be functions of one argument
  - the argument type of G matches the type of X
  - the argument type of F matches the result type of G
  - the result type of compose is the result type of F

Higher order procedures

- Procedures may be passed in as arguments
- Procedures may be returned as values
- Procedures may be used as parts of data structures

- Procedures are first class objects in Scheme!!