

CSE1729: Introduction to Programming

Functional Programming in SCHEME: Substitution and Environment Semantics

Alexander Russell

Recall: SUBSTITUTION SEMANTICS for function application

- ✧ Consider the function definition `(define (f x) <body>)`.
- ✧ In the future, if the evaluator encounters `(f <arg>)`, it will:
 - ✧ **Evaluate** `<arg>` (as usual), resulting in a value `v`.
 - ✧ Apply `f` to the value `v`. This is accomplished in two steps:
 - ✧ **Substitute** occurrences of `x` in `<body>` with `v`, and
 - ✧ **Evaluate** `<body>` (after substitution) and return the result.

Eval-Appl diagrams: An example

- ✧ Consider the definitions

```
> (define (square x) (* x x))
```

```
> (define (fourth x) (square (square x)))
```

- ✧ Then...let's explore this via an “eval-apply” diagram.

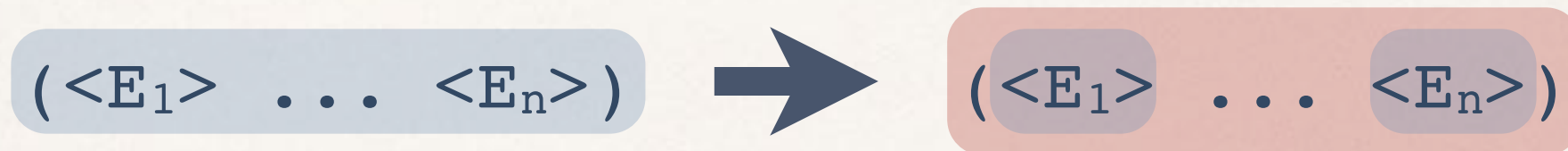
A red box indicates that an application/substitution is pending

A blue box indicates that an evaluation is pending

- ✧ In particular...

The rules for Eval/Apply, in diagrams

- ❖ Recall the standard evaluation rule. In our Eval/Apply diagram, it asserts that:



- ❖ Recall the substitution semantics rule for function application. It asserts that with the function:

$(\text{define } (f \ x) \ \langle \text{body} \rangle)$

we have:



Making eval and apply... tick and tock

(fourth 5)

Making eval and apply... tick and tock

(fourth 5)

(fourth 5)

Making eval and apply... tick and tock

(fourth 5)

(fourth 5)

Making eval and apply... tick and tock

(fourth 5) (define (fourth x) (square (square x)))

(fourth 5)

[x/5](square (square x))



Making eval and apply... tick and tock

```
(fourth 5)
```

```
(fourth 5)
```

```
[x/5](square (square x))
```

```
(square (square 5))
```

Making eval and apply... tick and tock

```
(fourth 5)
```

```
(fourth 5)
```

```
[x/5](square (square x))
```

```
(square (square 5))
```

Making eval and apply... tick and tock

```
(fourth 5)
```

```
(fourth 5)
```

```
[x/5](square (square x))
```

```
(square (square 5))
```

```
(square (square 5))
```


Making eval and apply... tick and tock

```
(fourth 5)
```

```
(fourth 5)
```

```
[x/5](square (square x))
```

```
(square (square 5))
```

```
(square (square 5))
```

```
(square (square 5))
```

Making eval and apply... tick and tock

```
(fourth 5)
```

```
(fourth 5)
```

```
[x/5](square (square x))
```

```
(square (square 5))
```

```
(square (square 5))
```

```
(square (square 5))
```

```
(square (square 5))
```

Making eval and apply... tick and tock

```
(fourth 5)
```

```
(fourth 5)
```

```
[x/5](square (square x))
```

```
(square (square 5))
```

```
(square (square 5))
```

```
(square (square 5))
```

```
(square (square 5))
```

```
(square ([x/5](* x x)))
```

```
(define (square x) (* x x))
```



Making eval and apply... tick and tock

```
(square ([x/5] (* x x)))
```

```
(square (* 5 5))
```

Making eval and apply... tick and tock

```
(square ([x/5] (* x x)))
```

```
(square (* 5 5))
```

```
(square (* 5 5))
```

Making eval and apply... tick and tock

```
(square ([x/5] (* x x)))
```

```
(square (* 5 5))
```

```
(square (* 5 5))
```

```
(square (* 5 5))
```


Making eval and apply... tick and tock

```
(square ([x/5] (* x x)))
```

```
(square (* 5 5))
```

```
(square (* 5 5))
```

```
(square (* 5 5))
```

```
(square (* 5 5))
```

Making eval and apply... tick and tock

```
(square ([x/5] (* x x)))
```

```
(square (* 5 5))
```

```
(square (* 5 5))
```

```
(square (* 5 5))
```

```
(square (* 5 5))
```

```
(square 25)
```

Making eval and apply... tick and tock

```
(square ([x/5] (* x x)))
```

```
(square (* 5 5))
```

```
(square (* 5 5))
```

```
(square (* 5 5))
```

```
(square (* 5 5))
```

```
(square 25)
```

```
[x/25] (* x x)
```

```
(define (square x) (* x x))
```

625

Making eval and apply... tick and tock

(* 25 25)

Making eval and apply... tick and tock

(* 25 25)

(* 25 25)

Making eval and apply... tick and tock

(* 25 25)

(* 25 25)

(* 25 25)

Making eval and apply... tick and tock

(* 25 25)

(* 25 25)

(* 25 25)

625

Why all the fuss about application semantics?

- ❖ *Recursion* will be our principal tool for program development; application semantics are critical for understanding how, precisely, this works.
- ❖ This reflects the fact that **recursion is the principle tool used to construct rich mathematical objects.**

The factorial function

Factorial is usually written in postfix notation (gasp!)

- ❖ Recall the factorial function:

$$n! = n \cdot (n-1) \cdot \dots \cdot 1$$

- ❖ Alternatively, we could write:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$$

- ❖ **This meaningfully defines a function, even though it is recursive:**

$$3! = 3 * 2! = 3 * 2 * 1! = 3 * 2 * 1 * 0! = 3 * 2 * 1 * 1 = 6$$

Recursion in SCHEME

- ✧ We can represent such a definition in SCHEME:

```
> (define (factorial n)
    (if (= n 0)
        1
        (* n (factorial (- n 1)))))
```

- ✧ Note, in particular, that `factorial` appears in the definition of `factorial`.
- ✧ This is a well-defined function; its meaning is determined by substitution semantics.

An Eval-Apply diagram for factorial

- ❖ We've introduced two colored boxes, which stand for "pending evaluation" and "pending application".
- ❖ Let us introduce a box for the **if** special form:

Behavior:

`(if #t <then> <else>)` → `<then>`

`(if #f <then> <else>)` → `<else>`

- ❖ With this in place, we can define:

`(if <pred> <then> <else>)` → `(if <pred> <then> <else>)`

Evaluation of factorial

```
(factorial 2)
```

```
(if (= 2 0) 1 (* 2 (factorial (- 2 1))))
```

```
(if (= 2 0) 1 (* 2 (factorial (- 2 1))))
```

```
(if #f 1 (* 2 (factorial (- 2 1))))
```

```
(* 2 (factorial (- 2 1)))
```

```
(* 2 (factorial (- 2 1)))
```

```
(* 2 (factorial (- 2 1)))
```

```
(* 2 (factorial 1))
```


and...hence...

```
(factorial 3)
(* 3 (factorial 2))
(* 3 (* 2 (factorial 1)))
(* 3 (* 2 (* 1 (factorial 0))))
```

Now what?

```
(factorial 0)
```

```
(if (= 0 0) 1 (* 0 (factorial (- 0 1))))
```

```
(if #t 1 (* 0 (factorial (- 0 1))))
```

(substitution)



(if special form)

1

! Green form does *not* evaluate all arguments !

Putting it all together...

```
(factorial 3)
(* 3 (factorial 2))
(* 3 (* 2 (factorial 1)))
(* 3 (* 2 (* 1 (factorial 0))))
(* 3 (* 2 (* 1 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

Play it again...substitution semantics for `factorial`

```
(factorial 2)
```

```
(factorial 2)
```

```
(factorial 2)
```

```
[x/2](if (= x 0) 1 (* 2 (factorial (- x 1))))
```

Which is...

```
(if (= 2 0) 1 (* 2 (factorial (- 2 1))))
```

...

Some conclusions about Scheme from substitution semantics

- ✧ The **name** of “local variables” does not matter. Why? They are just placeholders for substitution!

- ✧ As far as Scheme is concerned

`(define (double x) (* x 2))`

and

are identical!

`(define (double y) (* y 2))`

- ✧ Why? For any value v , $[x/v](* x 2)$

and

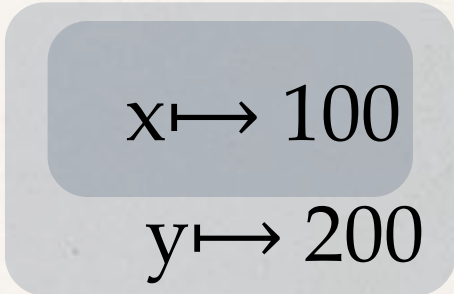
are identical!

$[y/v](* y 2)$

Variable shadows

- ❖ Substitution semantics explain what happens when a local variable has the same name as a variable in the enclosing environment.
- ❖ Question: How does the following code snippet behave?

```
> (define x 100)
> (define y 200)
> (define (add-to-y x) (+ x y))
> (add-to-y 2)
```



A diagram of an environment frame, represented as a rounded rectangle. Inside, there is a smaller rounded rectangle containing the text $x \mapsto 100$. Below this, the text $y \mapsto 200$ is written.

$x \mapsto 100$
 $y \mapsto 200$

$[x/2](+ x y) \rightarrow (+ 2 y)$

Note: x was never “looked up in this environment!”

Variable shadows

- ✧ With substitution semantics, variables are given values by two different processes:
 - ✧ Looking up in an environment, and
 - ✧ Substitution during function application.
- ✧ We can unify (and simplify) our understanding of Scheme by with *environment semantics*, which we will discuss again in more detail.

Recall the behavior of `factorial`

```
> (define (factorial n)
    (if (= n 0)
        1
        (* n (factorial (- n 1)))))
)
```

- ✦ Let's focus on the behavior of `factorial` when called with 0.

`if`...it's just *got* to be special!

- * The special evaluation rule for `if` is CRITICAL for this to work.
- * Suppose that `(if <pred> <exp1> <exp2>)` evaluated all of its arguments (as per usual evaluation). Then...

`(factorial 0)`

expands to

`(if (= 0 0) 1 (* 0 (factorial (- 0 1))))`

which would require evaluation of...

`(= 0 0)` and `1` and `(factorial -1)`

This will never terminate...

...which would “unwind” eternally

```
(if #t
  1
  (* 0 (if #f
    1
    (* -1 (if #f
      1
      (* -2 (if #f
        1
        (* -3 (if #f
          1
          (* -4 (if ...
```

Remark: A similarly nonterminating computation would ensue if we called `(factorial -1)` or `(factorial (/ 1 2))`; why?

“Special” treatment of other primitive functions

- ❖ Thus, special “incomplete” evaluation is essential for meaningful recursive programming. For this reason, other primitive functions whose values can be determined by “incomplete” evaluation are also treated as special forms:
- ❖ $(\text{and } \langle x_1 \rangle \langle x_2 \rangle \dots \langle x_n \rangle)$ uses “short-circuited” evaluation. The expressions $\langle x_1 \rangle, \dots$ are evaluated one at a time, left to right. If any evaluate to $\#f$, evaluation stops (and $\#f$ is returned). Otherwise, $\#t$ is returned.
- ❖ $(\text{or } \langle x_1 \rangle \langle x_2 \rangle \dots \langle x_n \rangle)$ uses “short-circuited” evaluation. The expressions $\langle x_1 \rangle, \dots$ are evaluated one at a time, left to right. If any evaluate to $\#t$, evaluation stops (and $\#t$ is returned).

Another example: The Fibonacci numbers

- ✧ The *Fibonacci numbers* are defined by the rule:

$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

- ✧ Note, then, that the sequence F_0, F_1, F_2, \dots is

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

each is the sum of the previous two.

The Fibonacci numbers in SCHEME

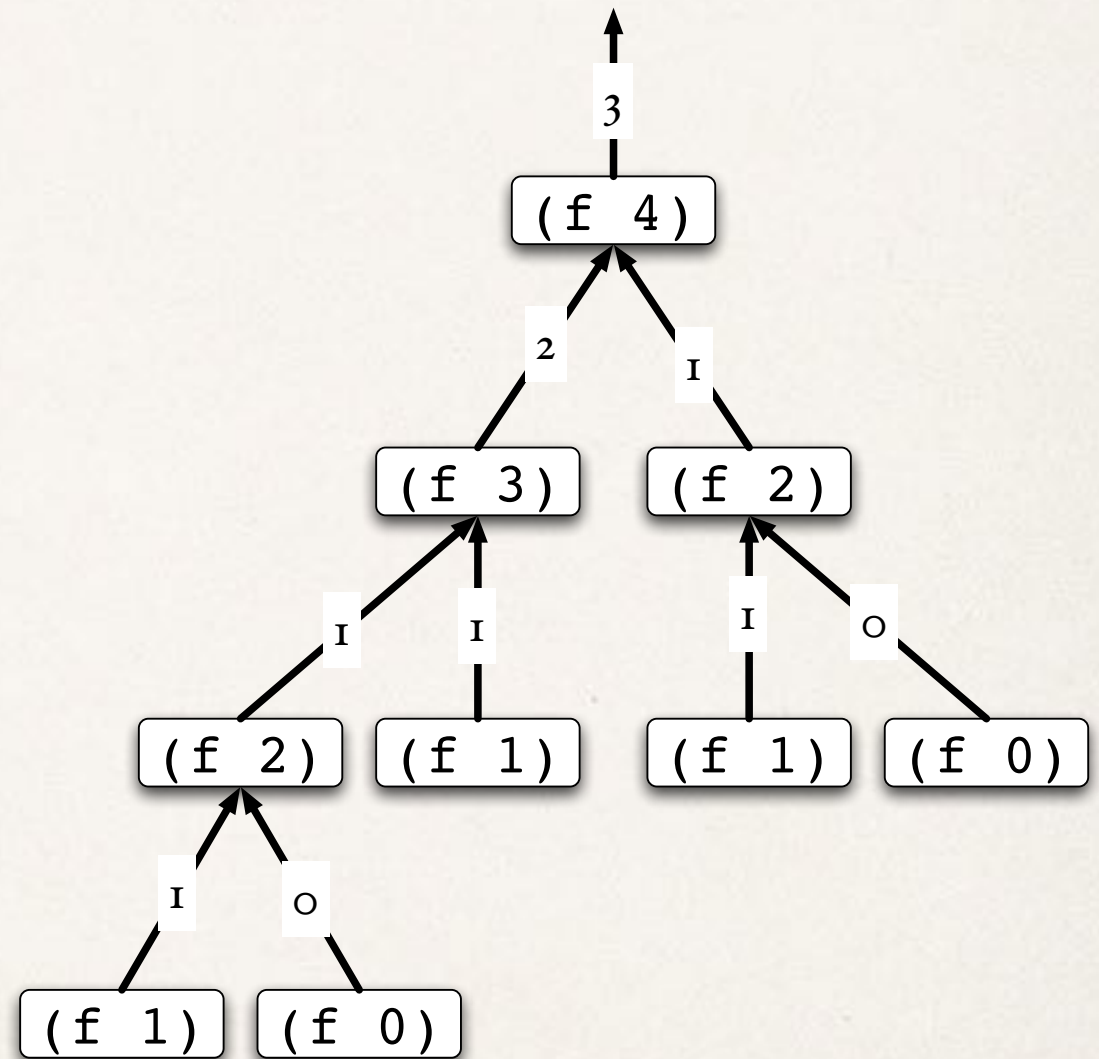
- ✧ As with the factorial function, we can naturally capture this definition in SCHEME.

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        ((> n 1) (+ (fib (- n 1))
                      (fib (- n 2)))))
  )
)
```

- ✧ Notice, as with `factorial`, how closely the SCHEME definition can mirror the mathematical definition.

The Fibonacci *evaluation tree*

- The Fibonacci function gives rise to an “**evaluation tree**” as shown. Here each node returns the sum of the value of its children.
- Note that some “sub”-problems are evaluated many times.
- Question: How many times is $(f\ 1)$ evaluated, in total?



Recursion is delicate business

```
> (define (recurse x) (recurse x))  
> (recurse 1)  
>  
> (if #t 1 (recurse 1))  
> 1  
> (if #f 1 (recurse 1))  
>
```



“Iterative” constructs in SCHEME

- ✧ Consider computing the sum of the first n numbers in SCHEME.

- ✧ Note that
$$\underbrace{(1 + \dots + n)}_{\sum_{i=1}^n i} = n + \underbrace{(1 + \dots + (n - 1))}_{\sum_{i=1}^{n-1} i}$$

- ✧ And thus:

```
> (define (number-sum n)
    (if (= n 0)
        0
        (+ n (number-sum (- n 1)))))
> (number-sum 10)
55
```


The evaluation tree for number-sum

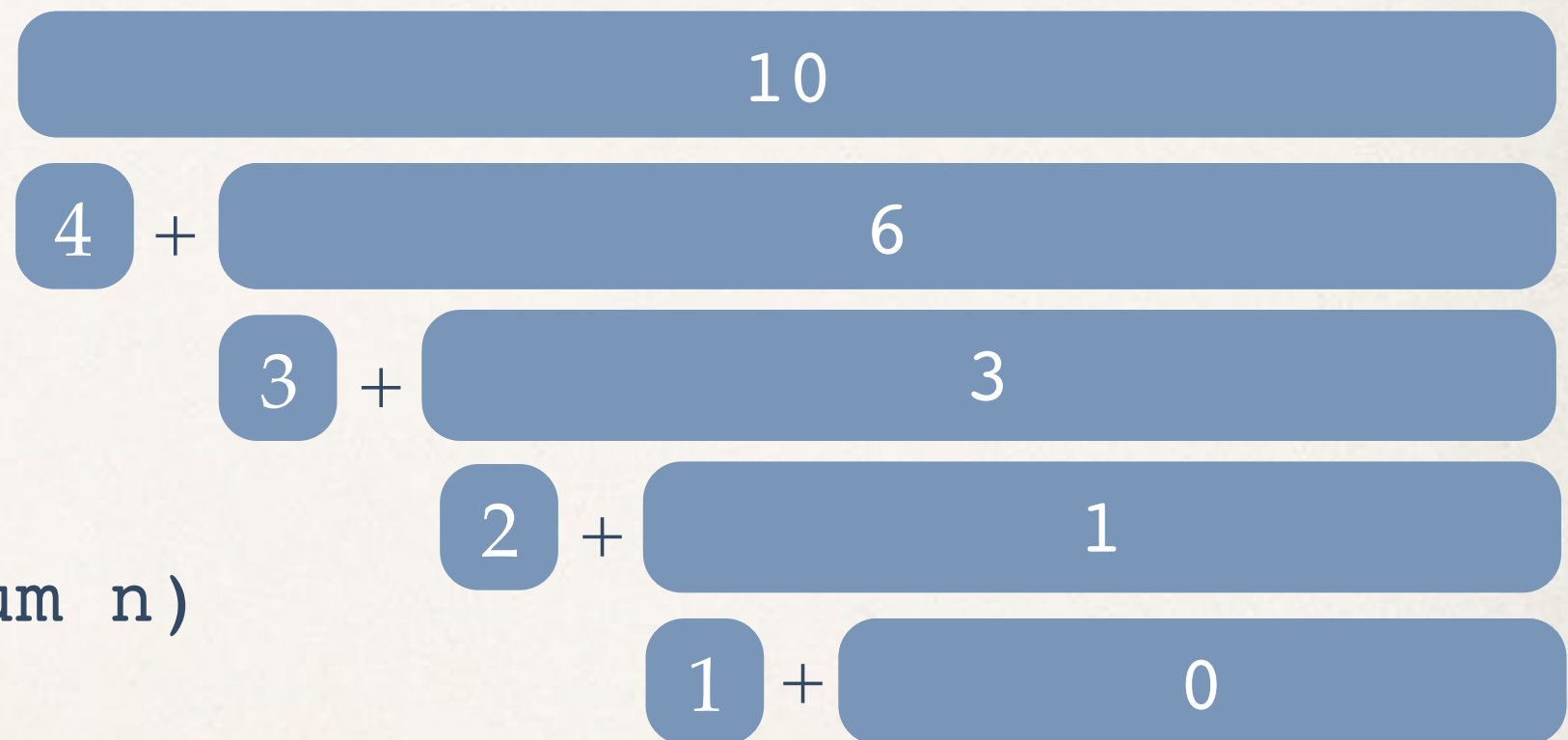
* (number-sum 4)
generates a call to
(number-sum 3); it
will add 4 to the result
and return the value.

* (number-sum 3)
generates a call to

> (define (number-sum n)
 (if (= n 0)
 0
 (+ n (number-sum (- n 1)))))
)

> * (number-sum 10) is
55 called: returning 0.

* (number-sum 4)



Recursive decomposition requires love and understanding...

- ✧ Not all recursive decompositions of a problem are the same...
- ✧ There can be major conceptual and computational differences...

Example: Multiplication in terms of addition

- ✧ Consider the definition of multiplication as repeated addition:

$$a \times b = \underbrace{b + \cdots + b}_{a \text{ times}}$$

- ✧ We can express this in SCHEME:

```
(define (mult a b)
  (if (= a 0)
      0
      (+ b (mult (- a 1) b))))
```


Efficiency considerations

- ❖ How many recursive calls are generated by

`(mult 200 2) ← (mult 199 2) ← (mult 198 2) ←`

- ❖ How about

`(mult 2 200) ← (mult 1 200) ← (mult 0 200)`

- ❖ We could write a more *efficient* program by “recursing on the smaller of a and b.” Thus

A more efficient multiply...

We could write a new program to exploit this...

```
(define (fmult a b)
  (cond ((= a 0) 0)
        ((= b 0) 0)
        ((<= a b) (+ b (mult (- a 1) b)))
        ((> a b) (+ a (mult a (- b 1))))))
```

Now it will only recurse $\min(a,b)$ times. Alternatively,

```
(define (fmult a b)
  (if (> a b) (mult b a) (mult a b)))
```

To be really fancy, we could reduce both a and b at the same time...

Remember that $ab = (a-1)(b-1) + a + b - 1$. Thus we could also express multiply as...

```
(define (fmult a b)
  (cond ((= a 0) 0)
        ((= b 0) 0)
        (else (+ -1
                  a
                  b
                  (fmult (- a 1) (- b 1))))))
```

This will also recurse $\min(a,b)$ times.

Actually, all three of these algorithms are terrible...why?

- ❖ With paper and pencil, how long would it take you to multiply two 16 digit numbers? Perhaps a few hours?
- ❖ With the program above, the computation

```
(fmult 10000000000000000 10000000000000000)
```

will never complete, even on a *very* fast computer. (Try it.) Why?

What does the evaluation tree look like?

- ✧ Well, 1000000000000000000 will generate a call to
 - ✧ 9999999999999999999, and hence to
 - ✧ 9999999999999999998, and hence to
 - ✧ 9999999999999999997, and hence
- ✧ In total 10000000000000000000 calls must be completed. If the computer could carry out a million calls per second, this would take 10000000000 seconds, a little over 30 years.

We can fix this by using more information about multiplication...

- ❖ On a computer dividing by 2 and multiplying by 2 can be done very quickly--we can improve our program:
- ❖ **Observation:** Suppose we wish to multiply x and y .
 - ❖ If we're lucky, x is even, and we have

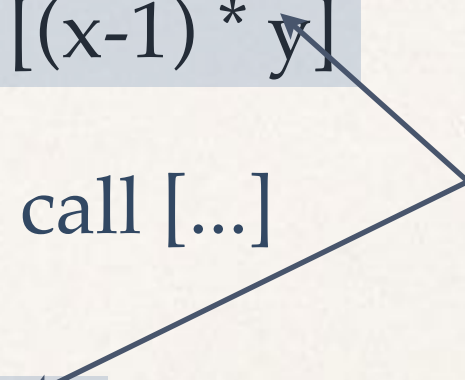
$$x \times y = 2 \times \left[\frac{x}{2} \times y \right]$$

The diagram illustrates the recursive multiplication formula $x \times y = 2 \times \left[\frac{x}{2} \times y \right]$. The expression $\left[\frac{x}{2} \times y \right]$ is enclosed in a light blue rounded rectangle. Two arrows originate from the text 'These operations can be done quickly' and point to the $\frac{x}{2}$ and the $\times y$ parts of the expression. A third arrow points from the text 'x has been reduced by half in this recursive call' to the $\frac{x}{2}$ part of the expression.

These operations can be done quickly

x has been reduced by **half** in this recursive call

Fast multiplication with division & multiplication by 2

- ❖ On a computer dividing by 2 and multiplying by 2 can be done very quickly--we can improve our program:
 - ❖ **Idea:** To multiply x and y (positive whole numbers):
 - ❖ If x is odd, fix it! The answer is: $y + [(x-1) * y]$
Now, $x-1$ is even in the recursive call [...]
 - ❖ If x is even: the answer is: $2 * [x/2 * y]$
- 
- The diagram consists of two arrows originating from the right side of the text blocks. One arrow points from the expression $[(x-1) * y]$ in the odd case to the recursive call $[x/2 * y]$ in the even case. The other arrow points from the text 'Recursive calls' to the same $[x/2 * y]$ expression.

Now, one of the numbers in the recursive call [...] has been significantly reduced--it's only half the previous size!

Capturing this idea in a Scheme program

- * On a computer dividing by 2 and multiplying by 2 can be done very quickly--we can improve our program:

```
(define (even x) (= (modulo x 2) 0))
(define (twice x) (* x 2))
(define (half x) (/ x 2))

(define (rfmult a b)
  (cond ((= 0 a) 0)
        ((= 0 b) 0)
        ((even a) (twice (rfmult (half a) b)))
        (else (+ b (twice (rfmult (half (- a 1))
                                   b)))))
  )
)
```

How has the evaluation tree changed?

- ❖ Well, `(rfmult 2k x)` will generate a call to
 - ❖ `(rfmult 2k-1 x)`, and hence to
 - ❖ `(rfmult 2k-2 x)`, and hence to
 - ❖ `(rfmult 2k-3 x)`, ...
- ❖ In total, if called on `1000000000000000000000000 < 254`, only 54 calls must be completed. Your computer can do this in a fraction of a second. (Try it.)

Computing square roots by averaging

- ❖ One simple way to compute an approximation to the square root of a number x is to

- ❖ Start with two guesses, a and b , with the property that

$$a < \sqrt{x} < b$$

(For example, if $x > 1$, we could start with $a = 1$, $b = x$.) Thus we know that the actual square root is between a and b .

- ❖ If $(a + b)/2$ is larger than the square root (which we can check by comparing $[(a + b)/2]^2$ with x) we know the real square root lies between a and $(a + b)/2$.
- ❖ Otherwise, the real square root lies between $(a + b)/2$ and b .

Square roots by “binary search”

Suppose $x > 1$. Then $\text{sqrt}(x)$ is certainly between 1 and x .

1 x



$x/2$

Is $(x/2)^2 > x$ or not?



$x/4$

Is $(x/4)^2 > x$ or not?



...

For example...

- ❖ To compute the square root of 10:
 - ❖ start with the window: $[1, 10]$ (we know the square root lies in this range).
 - ❖ Consider $(1 + 10)/2 = 5.5$. Since $5.5^2 > 10$, this is larger than $\text{sqrt}(10)$.
 - ❖ Now we know the square root lies in $[1, 5.5]$.
- ❖ Repeating this process, we find that it lies in $[1, 3.25]$.
- ❖ Repeating again, we find that it lies in $[2.125, 3.25]$

In SCHEME

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

```
(define (square a) (* a a))
```

```
(define (sqrt-converge x a b)
  (if (< (abs (- a b)) .000001)
```

a

```
    (if (> (square (average a b)) x)
        (sqrt-converge x a (average a b))
        (sqrt-converge x (average a b) b))))
```

N.b.: (average a b)
is referred to three
times here.

Now, we might like to define a more attractive square root function that does not require choosing a and b:

```
(define (new-sqrt x) (sqrt-converge x 1 x))
```

Local variables

- ❖ `(average a b)` is referred to several times in `sqrt-converge`. Wouldn't it be nice if we could temporarily bind a "local" variable to this value?
- ❖ The `let` construct does exactly this:

```
(let ((x1 <expr1>)
      (x2 <expr2>)
      ...
      (xk <exprk>))
  <body-expr>)
```
- ❖ **Semantics:** Evaluate each `<expri>`, yielding a value `vi`. Create a new environment by starting with the current one and binding each `xi` to `vi`. Then return the value of `<body-expr>` in this environment.

sqrt-converge reloaded

```
(define (sqrt-converge x a b)
  (let ((avg (/ (+ a b) 2)))
    (if (< (abs (- a b)) .000001)
        a
        (if (> (square avg) x)
            (sqrt-converge x a avg)
            (sqrt-converge x avg b))))))
```

The `let` statement
binds `avg` to $(a+b)/2$
for the shaded block
of code

We say that the `let` statement constructs a new environment, just like the enclosing environment, but in which `abs` has been bound to the value of $(a+b)/2$.

Lets go crazy!

```
> (define a 3)
```

```
> a
```

```
3
```

```
> (let ((a 10)
        (b (+ a 1))))
```

```
  b)
```

```
4
```

```
> a
```

```
3
```

```
> (let ((a 10)
        (b (+ a 1))))
```

```
  a)
```

```
10
```

```
> a
```

```
3
```

“Ambient” environment

a:3

Let environment

a:3

a:10

b:4

a is unchanged in the enclosing environment!

The original binding of
a is shadowed

a:3

a:10

b:4

“Ambient” environment

a:3

Lets go even crazier!

```
> (define a 3)
> (let ((a (+ a 1)))
  (let ((b (+ a 1)))
    b))
5
```

“Ambient” environment

a:3

Let environment

a:3

a:4

Let environment

a:3

a:4

b:5

Defining local *functions*...like local variables..but even more awesome!

- * A number is **perfect** if it is the sum of its (proper) divisors; $6 = 3 + 2 + 1$ is perfect; 8 is not $4 + 2 + 1$ so it is not perfect.

So ~~A version of~~ ~~we may define our tools so if~~ inside...

```
(define (divides? a b) (= 0 (modulo b a)))
```

```
(define (divisor-sum n k)
  (cond ((= k 1) 1)
        ((divides? k n) (+ k (divisor-sum n (- k 1))))
        (else (divisor-sum n (- k 1)))))
```

← Divides only used here...it's local

```
(define (perfect? n) (= n (divisor-sum n (- n 1))))
```


Local defines affect the local environment...

```
(define (f x)
  (define (average a b) (/ (+ a b) 2))
  (average 1 x))
```

- ❖ When `f` is called...
 - ❖ An environment is created (in which `x` is bound to the actual parameter)
 - ❖ The function `average` is defined, and added to the environment.
- ❖ ...and...finally, the body `(average 1 x)` is evaluated.

Function application constructs new environments...

- ❖ Recall that `let` produces a (new) environment with new variable bindings. (Incidentally, you could also make this precise by means of substitution.)
- ❖ Recall, also, our variant of substitution semantics which we called “environment semantics” ...
- ❖ This will alleviate the need to understand functions in terms of substitution: everything will be captured with environments.
- ❖ Also handles a difficulty: Function bodies that refer to variables that are not the formal parameter...

Variables in function bodies: What if you set them free?

- ✧ Consider the declaration `(define (f x) (+ a x))`.
- ✧ Question: To what does `a` refer?
- ✧ Remark: It's not so obvious what the answer should be!

```
> (define a 10)
> (define (f x) (+ x a))
> (f 100)
110
> (let ((a 20))
    (f 100))
```

110

Oh no!

Substitution semantics is *WRONG*

```
> (define y 10)
> (define a 10)
> (define (f x) (+ x a))
> (f 100)
110
> (let ((a 20))
    (f 100))
```

Substitution here would
have given...120

- ❖ Substitution semantics is fine for simple function bodies.
- ❖ Function bodies with free variables require environment semantics... let's look in more detail...

Recall Environment semantics:

A more sophisticated model of function application

- ✧ Consider `(define (f x) <body>)`.
- ✧ In the future, if the interpreter is called upon to evaluate `f` on the value `v` it will:
 - ✧ create a *new environment*, identical to the environment in which `f` was **defined**, but in which `x` has been bound to `v`. (This shadows any existing binding of `x` in def'n environment.)
 - ✧ evaluate the expression `<body>` in this new environment; the resulting value of `<body>` is the value this function returns.



Body evaluated in this environment

Lexical scope and variable clashes

- ❖ Scheme uses a precise set of rules to determine the binding of a variable. These conditions are known as *scoping rules* for the binding.
- ❖ SCHEME uses *lexical scope*.
- ❖ The other natural choice is *dynamic scope*.
- ❖ Example:
 - Two potentially relevant environments:
 - The environment at the *time of definition* (in which $x = 10$).
 - The environment at time of *invocation* (in which $x = 100$).
 - Lexical scoping rules (which SCHEME uses) always rely on the environment at *definition time*.

```
> (let ((x 10))  
  (define (g y) (+ x y))  
  (let ((x 100))  
    (g 1000)))
```

1010

>

Another example, pictorially

```
(define (f x)
  (define (g y)
    (+ x y))
  (let ((x 5))
    (g 11)))
```

> (f 6) ← Call to f

17

Another example, pictorially

`(define (f x)` ← Binding for x

`(define (g y)`

`(+ x y))`

`(let ((x 5))`

`(g 11)))`

`> (f 6)`

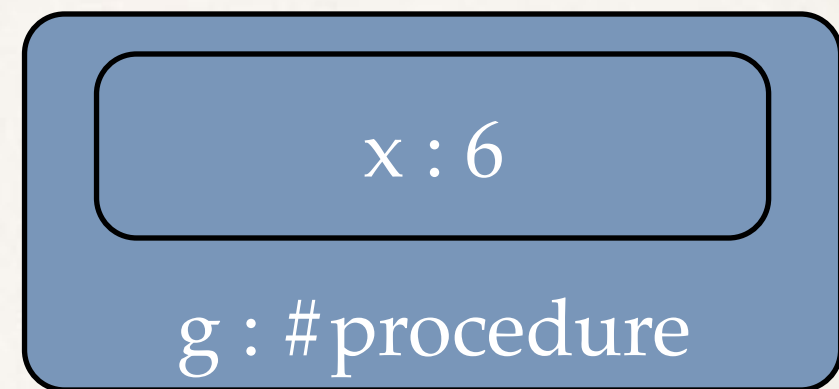
17



x:6

Another example, pictorially

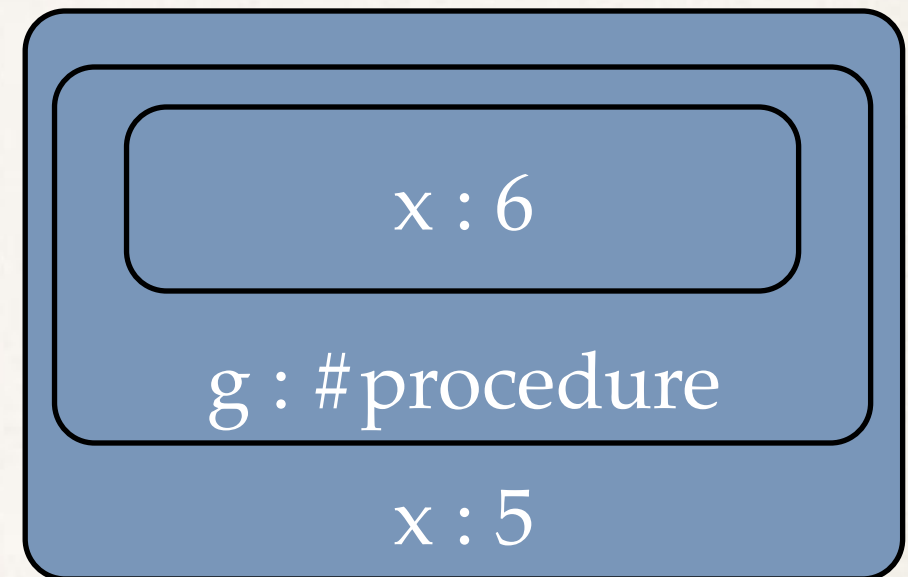
```
(define (f x)
  (define (g y) ← define g
    (+ x y))
  (let ((x 5))
    (g 11)))
> (f 6)
17
```



The definition of `g` grabs the current environment and its binding of `x`.
Any evaluation inside `g` that needs `x` will use this copy!

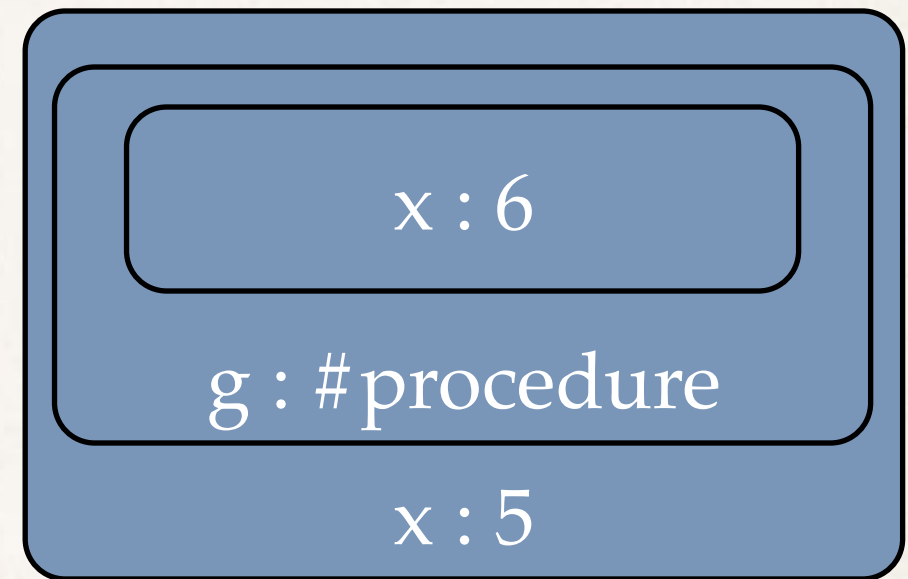
Another example, pictorially

```
(define (f x)
  (define (g y)
    (+ x y))
  (let ((x 5)) ← Bind x
    (g 11)))
> (f 6)
17
```



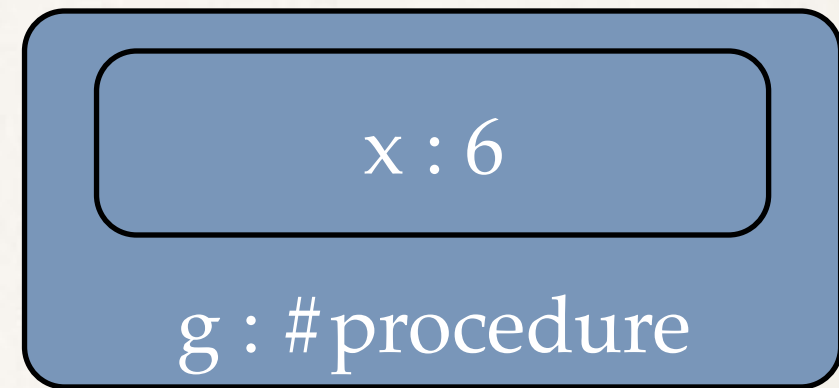
Another example, pictorially

```
(define (f x)
  (define (g y)
    (+ x y))
  (let ((x 5))
    (g 11))) ← Call g
> (f 6)
17
```



Another example, pictorially

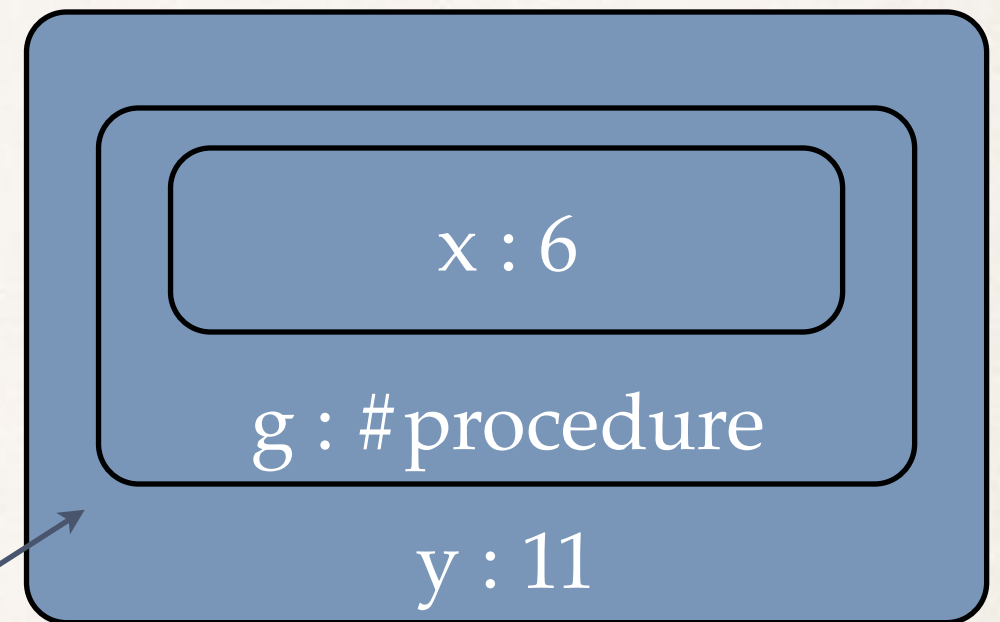
```
(define (f x)
  (define (g y)
    (+ x y))
  (let ((x 5))
    (g 11)))
> (f 6)
17
```



Starts with the definition environment!

Another example, pictorially

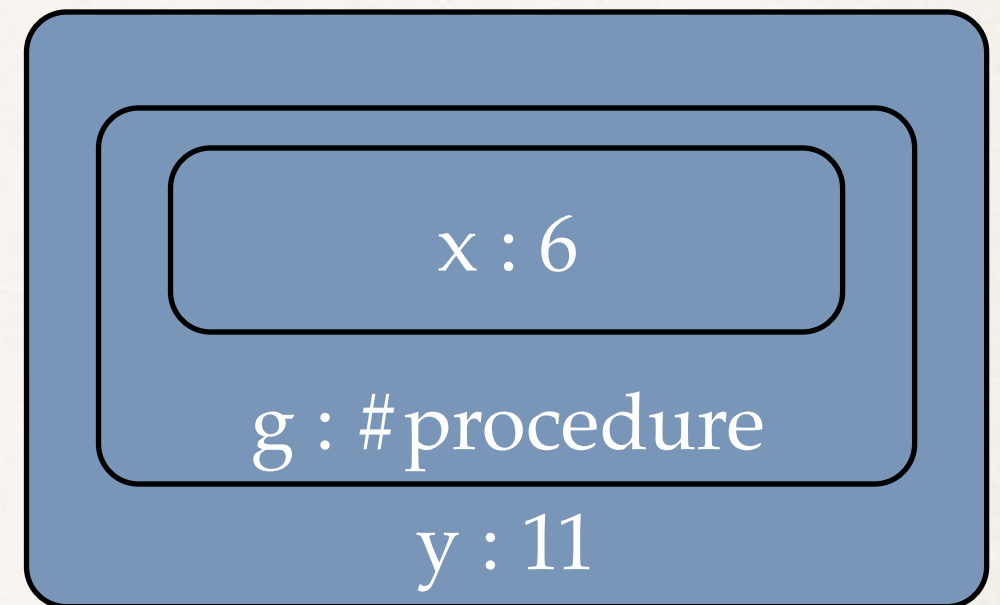
```
(define (f x)
  (define (g y) ← bind y
    (+ x y))
  (let ((x 5))
    (g 11)))
> (f 6)
17
```



Starts with the definition environment!

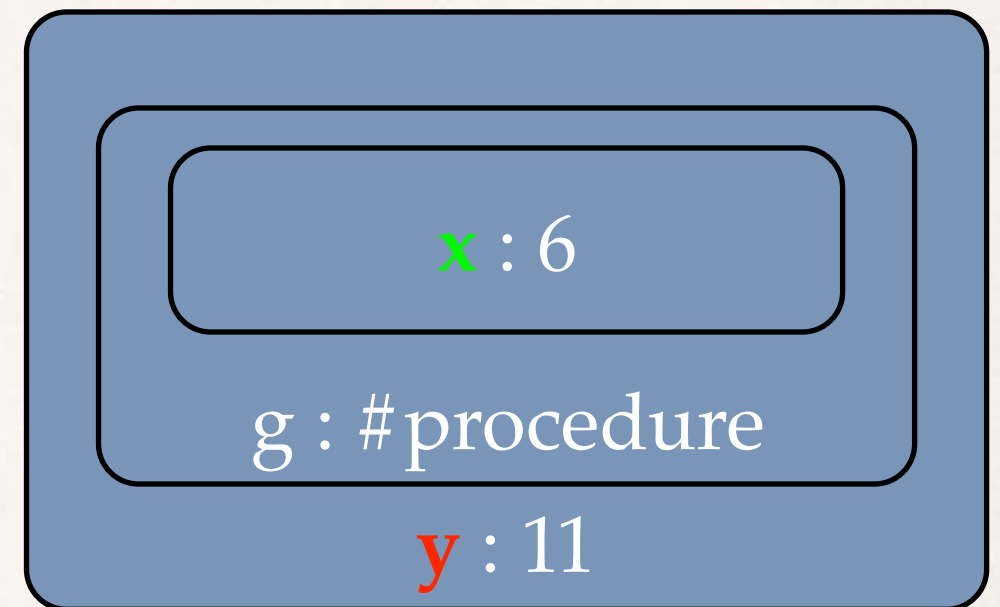
Another example, pictorially

```
(define (f x)
  (define (g y)
    (+ x y)) ← eval
  (let ((x 5))
    (g 11)))
> (f 6)
17
```



Another example, pictorially

```
(define (f x)
  (define (g y)
    (+ x y)) ← eval
  (let ((x 5))
    (g 11)))
> (f 6)
17
```



Returns the value 17

Lexical scope and the life of a Scheme function

- * “Free” variables in the body of a scheme function are assigned values from the environment in which the function was *defined*. This makes reasoning about their values easy, they are always drawn from the same environment!

f is defined

a:10

```
(define (f x)
  (+ x a))
```

f is applied

a:100

(f 0) = 10

a:1000

(f 0) = 10

...

With lexical scope...

Functions behave like functions!

- ❖ “Free” variables in the body of a scheme function are assigned values from the environment in which the function was *defined*. This makes reasoning about their values easy, they are always drawn from the same environment!
- ❖ When definition environments never change, we are using **functional programming**.
- ❖ There are some cool things you can do by fiddling with definitional environments *after a function has been defined*. Part III of the course.

An example of environment semantics...

- * Consider the following definition for computing the volume of a cylindrical solid of height h and radius r .

```
(define (volume h r) (* 3.1415 r r h))
```

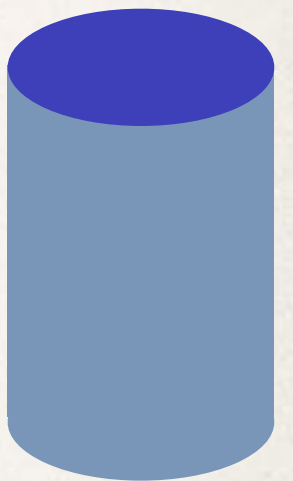
- * Evaluation can be understood in terms of an environment:

```
(volume 8 2)
```

```
(* 3.1415 r r h)
```

in the environment

```
h:8  
r:2
```



A more nuanced definition of `define`

- ❖ The `define` command forcibly adds a new binding in the current environment. This is one of the few ways that an environment can change in SCHEME.
- ❖ Thus

```
> (define a 6)
> (define (f x)
  (define a 5)
  (+ x a))
> (f 0)
5
```

a:6

a:6

a:5

Environment clutter and local functions

- ✧ Consider the definition

```
(define (square a)      (* a a))
(define (sqrt-converge x a b)
  (let ((avg (/ (+ a b) 2)))
    (if (< (abs (- a b)) .000001)
        a
        (if (> (square avg) x)
            (sqrt-converge x a avg)
            (sqrt-converge x avg b)))))
(define (new-sqrt x) (sqrt-converge x 1 x))
```

- ✧ We wished to define new-sqrt, but introduced many other functions into the environment. What if someone clobbers them or, in general, they clash with other functions?

Environmental protection...leave behind only what you intended

- ✧ Making internal structure (e.g., `sqrt-converge`) available to the user is dirty, provides opportunities for error.
- ✧ To avoid this, we can place the definitions inside `new-sqrt`:

```
(define (new-sqrt-i x)
  (define (square z) (* z z))
  (define (sqrt-converge t a b)
    (let ((avg (/ (+ a b) 2)))
      (if (< (abs (- a b)) .000001)
          a
          (if (> (square avg) t)
              (sqrt-converge t a avg)
              (sqrt-converge t avg b))))))
  (sqrt-converge x 1 x))
```

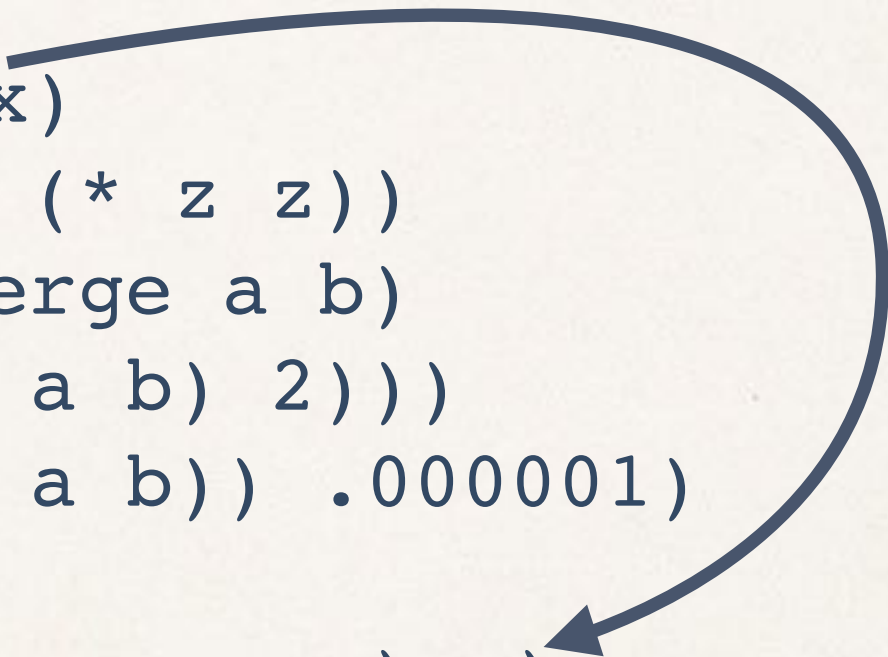
Environment

square: fn
sqrt-converge: fn
x: value

Scope

- ❖ Note, also, that `sqrt-converge` is called with `x`. If it is defined inside the environment of `new-sqrt`, `x` already appears in the environment! Thus we can further simplify the definition:

```
(define (new-sqrt-i x)
  (define (square z) (* z z))
  (define (sqrt-converge a b)
    (let ((avg (/ (+ a b) 2)))
      (if (< (abs (- a b)) .000001)
          a
          (if (> (square avg) x)
              (sqrt-converge a avg)
              (sqrt-converge avg b))))))
  (sqrt-converge 1 x))
```



How can we understand this? In terms of environment semantics

```
(define (new-sqrt-i x)
  ...
  (define (sqrt-converge a b)
    ...
    (if (> (square avg) x)
        ...))
  (sqrt-converge 1 x))
```

(new-sqrt-i 6)

Body:

```
(define (sqrt-converge a b)
  ...
  (if (> (square avg) x)
      ...))
(sqrt-converge 1 x))
```

- ❖ Consider the definition.
- ❖ Consider a call to `new-sqrt-i`.
- ❖ Creates an environment
`x:6`
- ❖ The define is evaluated in this environment,
`x:6, sqrt-converge:fn`

Example: Testing Primality

- ❖ Recall that a (whole, positive) number n is composite if it has a divisor other than 1 and n . Otherwise, it is *prime*.*
- ❖ What is a natural way to determine if a number p is composite? **Test to see if it has a divisor d , $1 < d < n$.**
- ❖ Note: It's not obvious how to define (`composite p`) in terms of (`composite k`) for smaller k : we will need to introduce some other functions to help structure

Thus, n is composite if:
Some number between 2 and $n-1$ divides it evenly.

Idea: Let's say that a number is *k-smooth* if it has a divisor d so that $1 < d \leq k$.

Thus, n is composite if:
It is $(n-1)$ -smooth.

A recursive expansion of...

being smooth

- * Note that n is k -smooth if k divides n or it is $(k-1)$ -smooth. Why?
- * Thus:

```
(define (divides a b) (= (modulo b a) 0))  
(define (smooth n k)  
  (and (>= k 2)  
        (or (divides k n)  
              (smooth n (- k 1)))))  
  
(define (composite n) (smooth n (- n 1)))
```

Note: This uses **short-circuited evaluation** of AND and OR. How?

Without short-circuit ?


```
(define (divides a b) (= (modulo b a) 0))
(define (smooth n k)
  (if (< k 2)
      #f
      (if (divides k n)
          #t
          (smooth n (- k 1))))))

(define (composite n) (smooth n (- n 1)))
```

You can optimize this...

- ❖ With one possible exception, divisors come in pairs. If n has a divisor d for which $1 < d < n$, then it has one
 - ❖ that is no more than $n/2$.
 - ❖ that is no more than \sqrt{n} .
- ❖ Why? Suppose that $d * d' = n$. If both d and d' were larger than \sqrt{n} , their product would be larger than n !
- ❖ Thus:

`(define (composite n) (smooth n (floor (sqrt n))))`

rounds down


Example: Testing primality

- ✧ As a number is prime when it is not composite,

```
(define (divides a b) (= (modulo b a) 0))  
(define (smooth n k)  
  (and (>= k 2)  
        (or (divides k n)  
              (smooth n (- k 1)))))  
(define (isprime p)  
  (not (smooth p (floor (sqrt p)))))
```

- ✧ Note: (smooth k) returns #t if there is a divisor of n between 2 and k.
- ✧ We can hide the definitions of divides and smooth...

Nesting the environments...

- ✦ As `divides` and `smooth` are only used inside `prime`:

```
(define (prime n)
  (define (divides a b) (= (modulo b a) 0))
  (define (smooth k)
    (and (>= k 2)
         (or (divides k n)
              (smooth (- k 1)))))
  (not (smooth (floor (sqrt n)))))
```

- ✦ INTERESTING SIMPLIFICATION: `(smooth k)` no longer needs to be passed `n`, it exists in the defining environment!
- ✦ Let's trace the environments...

The inner environment during a call to `(prime 6)`

If we make the invocation:

`(prime 6)`

the body is evaluated in an environment where `n: 6`.

```
(define (divides a b) (= (modulo b a) 0))  
(define (smooth k)  
  (and (>= k 2)  
        (or (divides k n)  
              (smooth (- k 1)))))  
(not (smooth (floor (sqrt n)))))
```

Thus, `smooth` is defined in an environment where `n = 6`.

Notice that `divides` is always called with `b = n`. Further simplification...

One further simplification of the primality tester

```
(define (prime n)
  (define (divisor a) (= (modulo n a) 0))
  (define (smooth k)
    (and (>= k 2)
         (or (divisor k)
              (smooth (- k 1)))))
  (not (smooth (floor (sqrt n)))))
```



divides was always called with $b = n$.

Reading

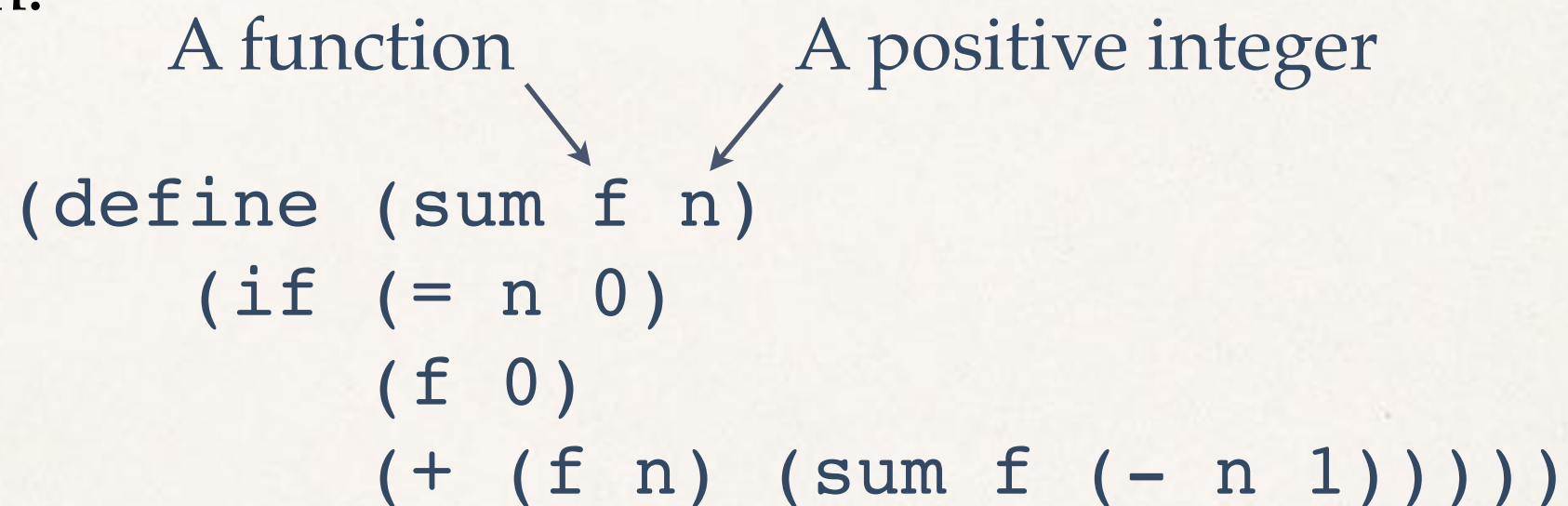
- ❖ With some adaptations and omissions, the previous slides cover material from Section 1.2 of SICP.
- ❖ You might find it interesting to look over the Revised⁵ Report on Scheme, a definition of the language. (Posted on the website.)

In SCHEME, functions are *first class objects*.

- ❖ Functions can be passed as arguments: Consider the following definition:

A function A positive integer

```
(define (sum f n)
  (if (= n 0)
      (f 0)
      (+ (f n) (sum f (- n 1)))))
```



This computes the sum: $f(0) + f(1) + \dots + f(n)$

Both f and n are passed as arguments.

Then...

- ✦ To compute the sum of the first n squares: $0^2 + 1^2 + 2^2 + 3^2 + 4^2 + 5^2$

```
> (define (square x) (* x x))  
> (sum square 5)  
55
```

- ✦ To compute the sum of the first n cubes: $0^3 + 1^3 + 2^3 + 3^3 + 4^3 + 5^3$

```
> (define (third x) (* x x x))  
> (sum third 5)  
225
```

Another example. A tool for partial power series

- * A power-series expander. `term` is a function that should return the coefficient of x^k .

```
(define (power-series x term k)
  (if (< k 0)
      0
      (+ (* (term k) (expt x k))
          (power-series x term (- k 1)))))
```

- * Then `(power-series x term k)` should return:

$\text{term}(0) + \text{term}(1)x + \text{term}(2)x^2 + \dots + \text{term}(k)x^k$.

Generating partial power series

- * Sin and Cos: Setting the stage

```
(define (fact n) (if (= n 0)
                     1
                     (* n (fact (- n 1)))))
(define (odd t) (= (modulo t 2) 1))
```

- * The term definitions:

```
(define (sin-term t)
  (if (odd t) (/ (expt -1 (/ (- t 1) 2)) (fact t))
      0))
(define (cos-term t)
  (if (odd t) 0 (/ (expt -1 (/ t 2)) (fact t))))
```


Understanding sin-term

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

- ✧ The *coefficients* of the even terms are always 0

$$\sin(x) \approx 1 \cdot x + 0 \cdot x^2 - \frac{1}{3!} \cdot x^3 + 0 \cdot x^4 + \frac{1}{5!} \cdot x^5 + 0 \cdot x^6 - \frac{1}{7!} \cdot x^7 + \dots$$

- ✧ term(0) = 0 even
- ✧ term(1) = 1
- ✧ term(2) = 0 even
- ✧ term(3) = - 1 / 3!
- ✧ term(4) = 0 even
- ✧ term(5) = + 1 / 5!
- ✧ term(6) = 0 even
- ✧ term(7) = - 1 / 7!

How to get the sign?

It alternates!

$$(-1)^{\frac{t-1}{2}}$$

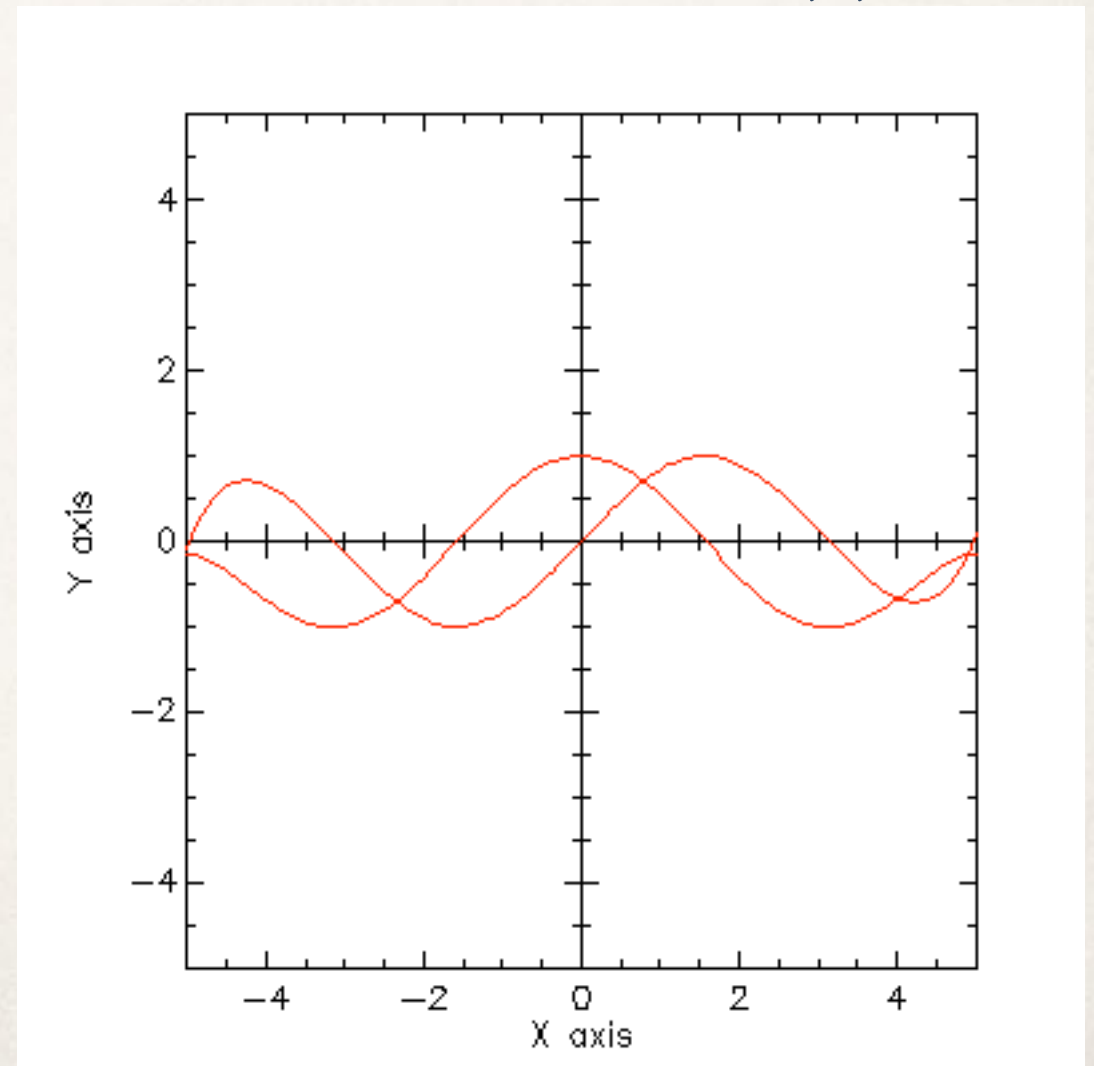
Now the power series are easy to generate

Now we can define functions from the first 10 terms of each power series:

```
(define (sin10 x) (power-series x sin-term 10))  
(define (cos10 x) (power-series x cos-term 10))
```

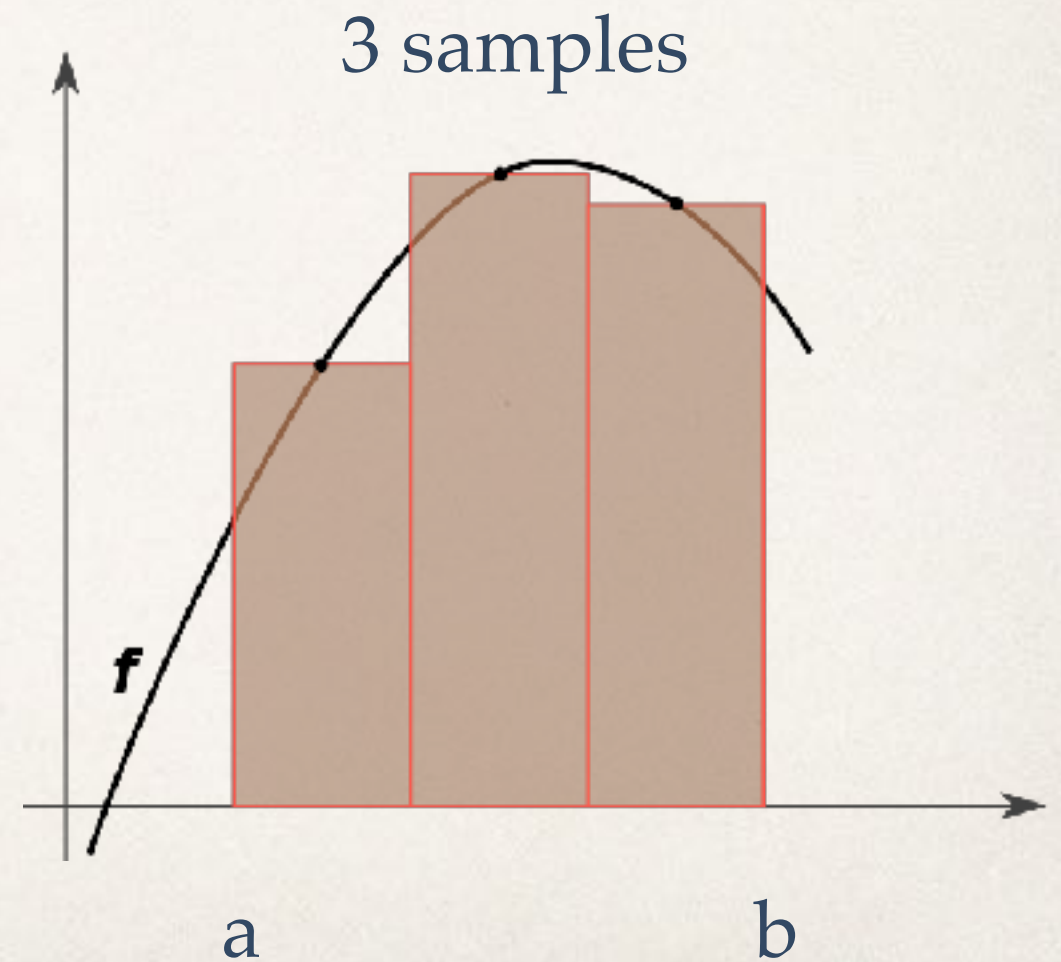
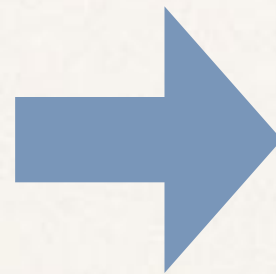
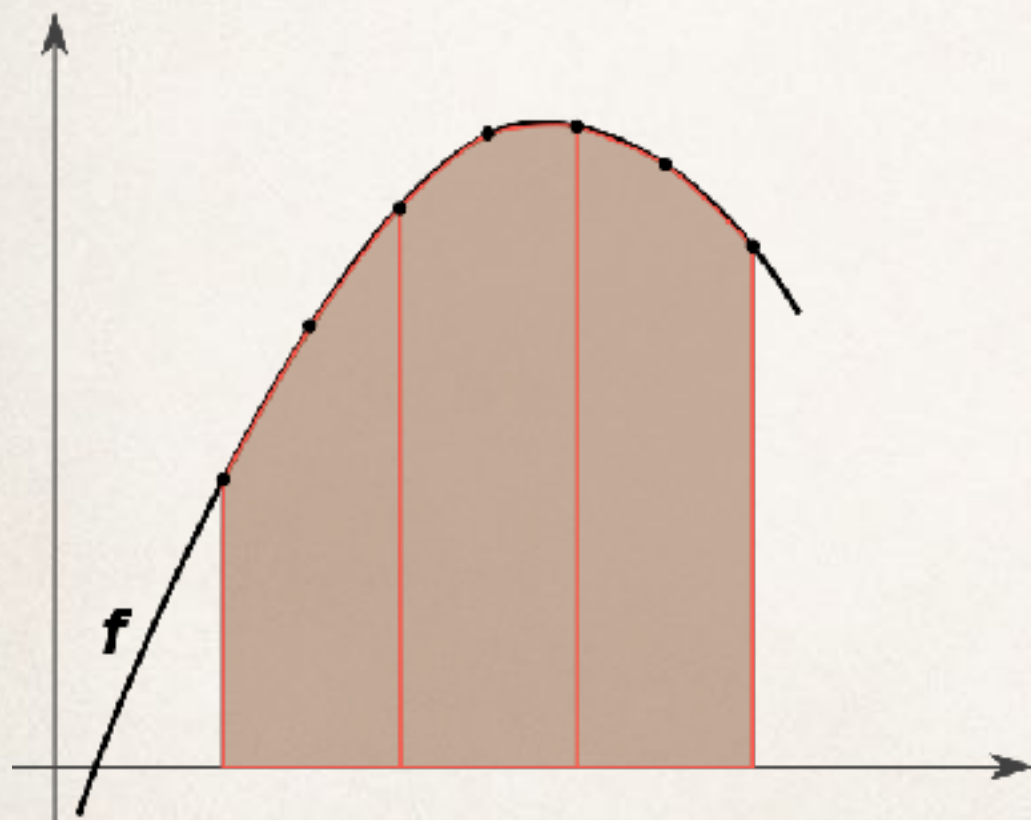
Then, for example,

```
(require plot)  
(plot (mix (line sin10)  
           (line cos10)))
```

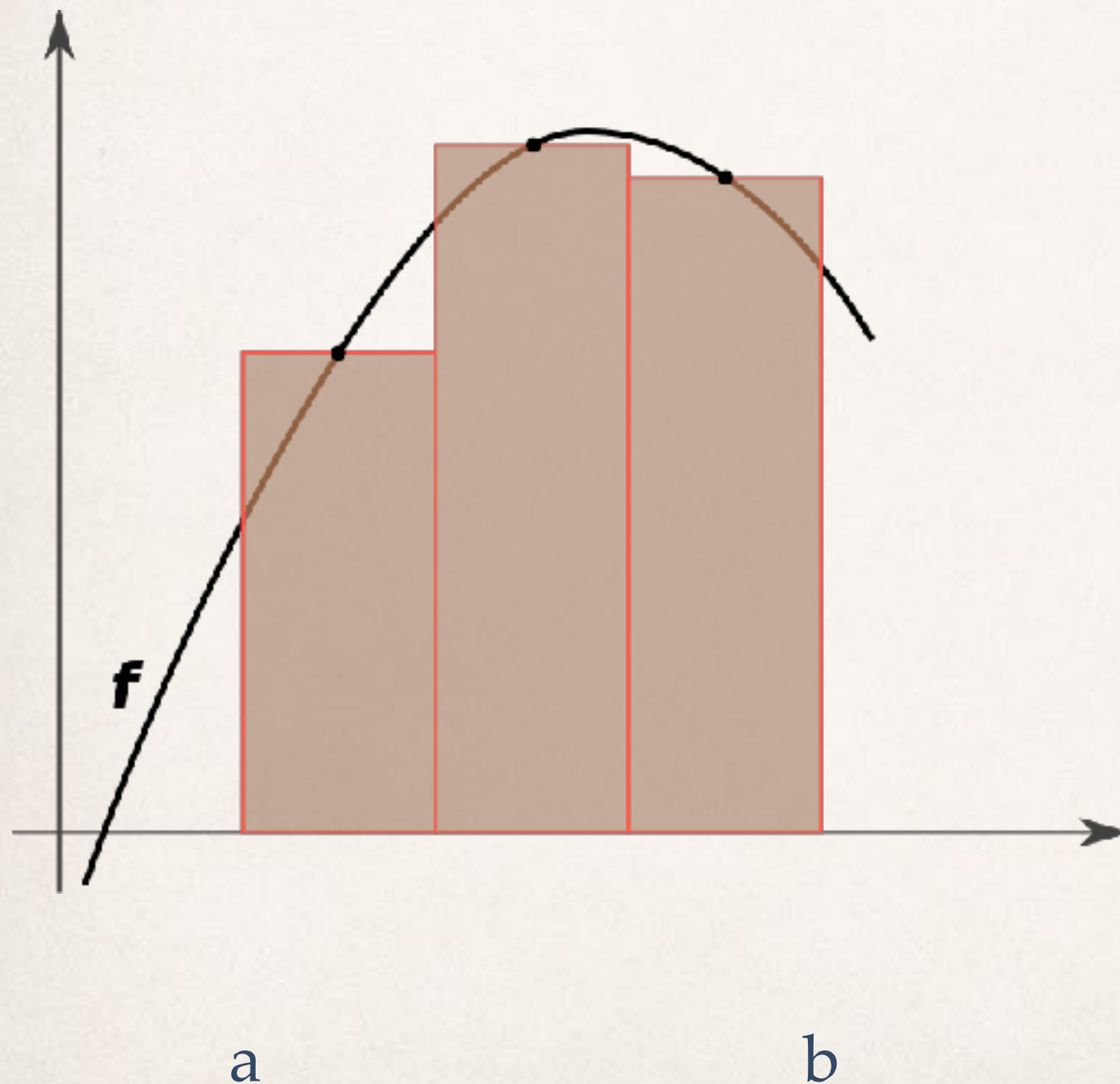


Numeric integration

- ❖ To approximate the area under a function, we approximate the area by a sequence of rectangles:



The technical details



- 3 samples, so...
- rectangle width = $(b-a)/3$,
- each sample position is $(b-a)/6$ from the left side of its rectangle,
- area of each rectangle is $f(\text{sample}) * (b-a)/3$.

Putting our sum function to work...a generic integrator

- * We wish to approximate the area under f over the interval $[a,b]$ by summing the areas of n equal width rectangles.
- * `(sample k)` determines the location of the k^{th} sample. Since our samples are the “midpoints” of the rectangle, you can check that they are:

$$a + \frac{b-a}{2n} + \frac{k}{n}(b-a) \quad k = 1, \dots, n-1$$

```
(define (integrate f a b n)
  (define (sample k)
    (f (+ a (/ (- b a) (* 2 n))
          (* k (/ (- b a) n)))))
  (define (rectangle-area k)
    (* (sample k) (/ (- b a) n)))
  (sum rectangle-area (- n 1)))
```

 This function is passed to sum

Unnamed functions

- ❖ SCHEME has a mechanism for defining functions without names:

argument body
 ↘ ↘
(lambda (x) (* x x))

is the function that returns the square of its argument.

- ❖ If you wish to sum the values of the first n squares, instead of defining `square` first, you can directly pass the function:

```
> (sum (lambda (x) (* x x)) 10)  
385
```


Define revisited

- ✧ If we enlarge our notion of value to include function values, we can simplify the definition of `define` as an operator that always binds a name to a value.

```
(define (square x) (* x x))
```

is the same as...

```
(define square (lambda (x) (* x x)))
```

Let revisited

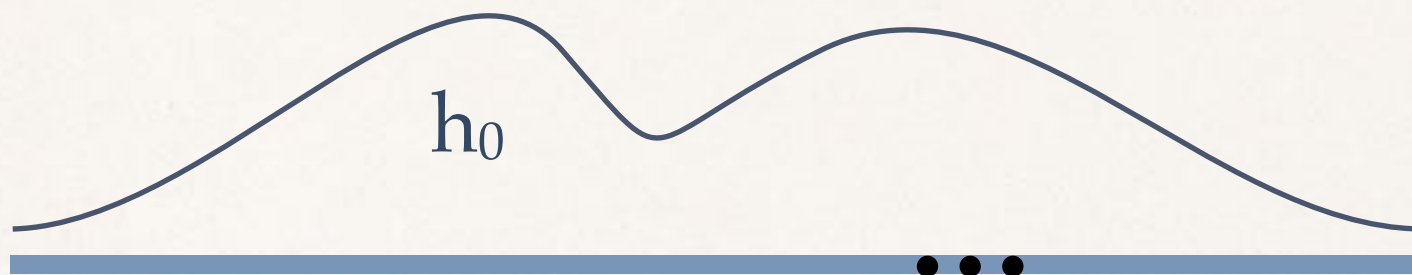
- * We can express `let` using `lambda` and the standard application rule!

```
(let ((x1 <expr1>)
      ...
      (xk <exprk>))
  <let-expr>)
```

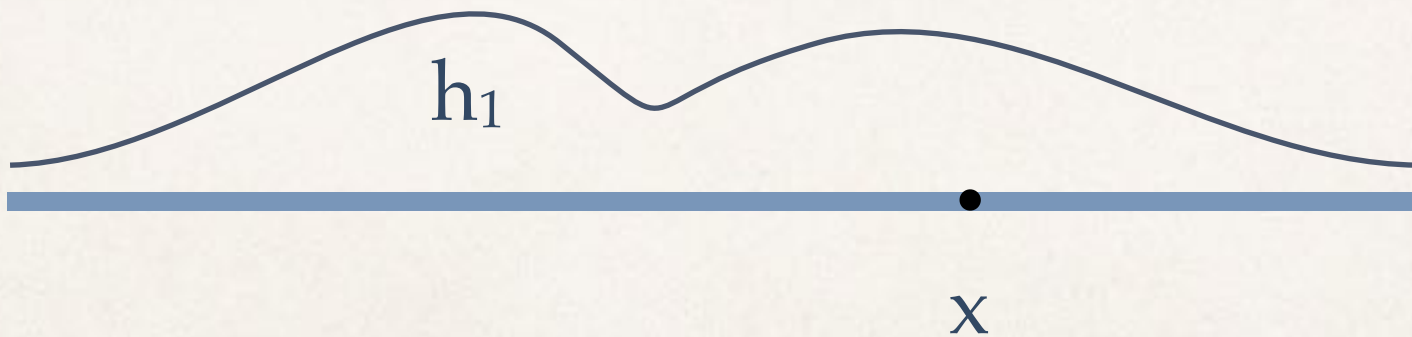
...is the same as...

```
((lambda (x1 ... xk) <let-expr>)
  <expr1> ... <exprk>)
```

The heat flow equation



$$h_1(x) = \frac{h_0(x - dx) + h_0(x + dx)}{2} \quad \text{The average of two close points}$$



Returning functions as “values”

- * The `lambda` form provides an easy way to return a function as a value.

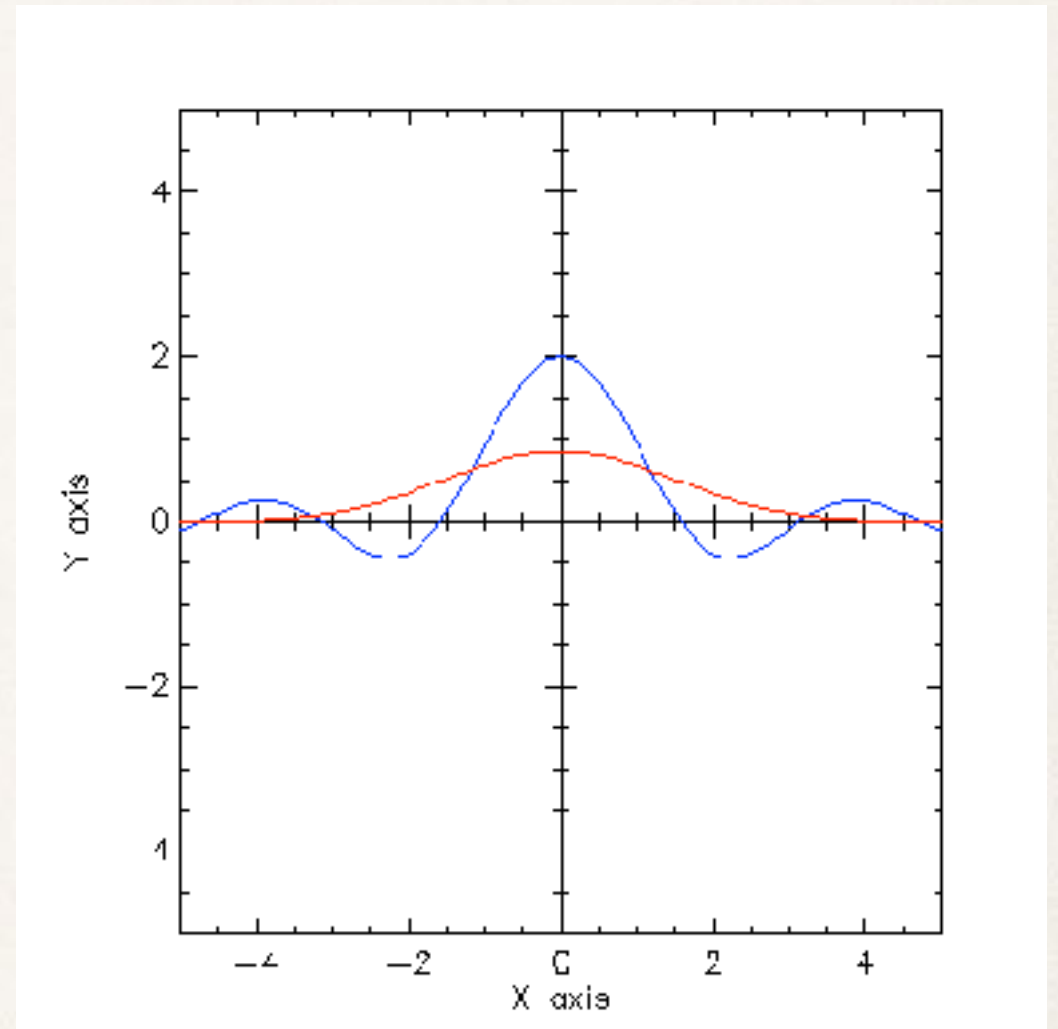
```
(define (heat-flow f dx)
  (lambda (x) (/ (+ (f (+ x dx))
                    (f (- x dx)))
                2)))
```

```
(define (heat-flow-evolve f dx t)
  (if (= t 0)
      f
      (heat-flow (heat-flow-evolve f dx (- t 1)) dx)))
```

Heat flow evolution

```
(define (curve x)
  (/ (sin (* 2 x)) x))

(plot (mix (line curve #:color 'blue)
  (line (heat-flow-evolve
    curve
    .5
    8)
    #:color 'red)))
```



Reminder about the life and times of an environment...

- ❖ Environments contain bindings of variables to values.
- ❖ The `define` command destructively adds a binding to an environment.
- ❖ There are two ways that new environments are created:
 - ❖ During function evaluation.
 - ❖ During `let` evaluation.
- ❖ These new environments **always** inherit all bindings from the environment from which they were created: however, the bindings of arguments shadow existing bindings.

(Actually, these are the same internal process)

An example

```
(define (f x)
  (define (g y) (+ x y))
  (define (h x) (+ x (g x))))
(h (+ x 10))
```

```
(f 100)
```

Call to (g 110)
y -> 110

Call to (h 110)
x -> 110

Call to (f 100)
x -> 100
g -> fn
h -> fn

Ambient
f -> fn

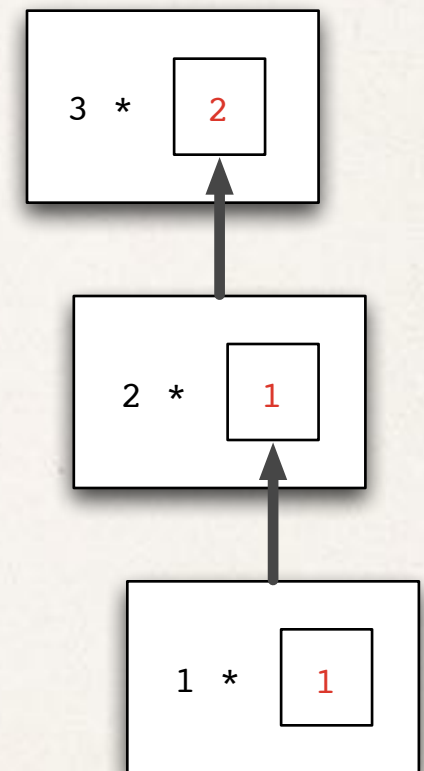
Recursion vs. iteration:

“Recursion”

- ✧ Consider the familiar factorial function:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

- ✧ Let's trace the evaluation of `(fact 3)`.
Note how the multiplications
 $(* 3 \square), (* 2 \square), \dots$
are *pending* while the recursive calls
complete.

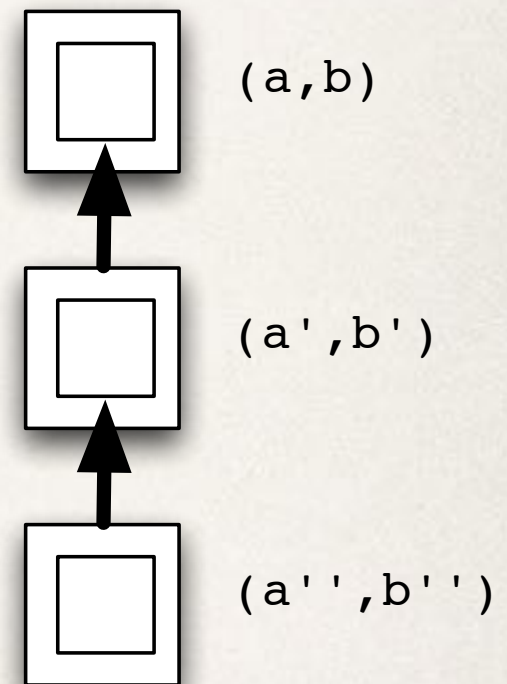


Recursion vs. iteration: “Iteration”

- Consider the `sqrt-converge` function we defined for extracting square roots:

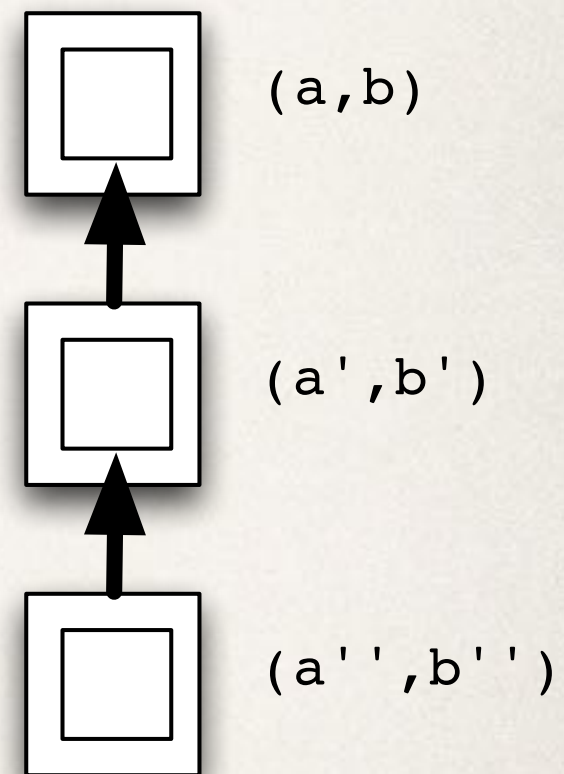
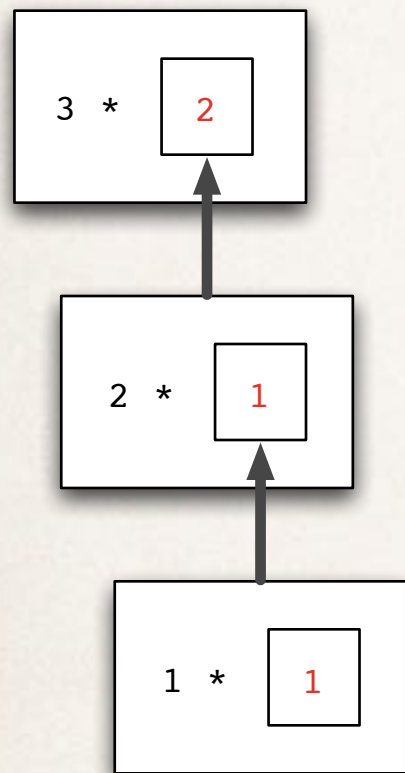
```
(define (sqrt-converge a b)
  (let ((avg (/ (+ a b) 2)))
    (if (< (abs (- a b)) .000001) a
        (if (> (square avg) x)
            (sqrt-converge a avg)
            (sqrt-converge avg b)))))
```

- Note that a call to `(sqrt-converge a b)` typically generates a call to `(sqrt-converge a' b')`. In fact, the *result* of `(sqrt-converge a b)` is simply the the *result* of `(sqrt-converge a' b')` *without further processing or pending operations*. This is called tail recursion.



Tail recursion requires no memory of pending operations

- ❖ In general, *recursion requires memory of the local state* of the calling procedure (including local variables and pending operations) in order to compute a final value.
- ❖ Tail recursion (or iteration) *does not require any such memory*. The value of the calling process is simply the value of the subprocess. The caller's environment can be discarded.



Conversion to tail recursion typically requires passing state

- ❖ Converting a function definition to a tail recursive call can significantly speed-up computation.
- ❖ Recall the original version of factorial:

```
(define (fact n)
  (if (= n 0) 1
      Pending → (* n
                    (fact (- n 1)))))
```

- ❖ Idea: *Let's send the pending operation along to the subprocess.* The subprocess is responsible for: computing (n-1) factorial *and* multiplying the result by n.

This results in...


- * New definition: function that computes a factorial *and* multiplies by a second “accumulator” argument.

```
(define (fact-accumulate n a)
  (if (= n 0) a
      (fact-accumulate (- n 1)
                        (* n a))))
```

Returns: (factorial of n) x (a)

Wrapping this to conceal the internal machinery

```
(define (fact-tr n)
  (define (fact-accumulate m a)
    (if (= m 0) a
```

Nothing pending  (fact-accumulate (- m 1) (* m a))))

```
(fact-accumulate n 1))
```

- Now this is tail recursive.
- Why is **accumulate** an appropriate name for the second argument?

Continuation passing: a principled method to induce tail recursion

- ✧ Instead of: call f on $\langle \text{arg} \rangle$, then apply g to the result:

`(g (f <arg>))` e.g. `(* n (fact (- n 1))`

- ✧ We call f^c with $\langle \text{arg} \rangle$ and ask it to apply g to the value when it is complete. In this case g is the *continuation*: what would have been done when f returned.

`(fc <arg> g)`

e.g.

`(fact-c (- n 1) (lambda (k) (* k n)))`

The continuation



Factorial, continuation passing style

- ❖ Basic ingredient: function `fact-c` that computes the factorial of its first argument and then...*applies its second argument to the result.*

Compute factorial of `m`, then apply `c`

```
(define (fact-c m c)
  (if (= m 0) (c 1)
      (fact-c (- m 1)
              (lambda (x) (c (* m x))))))
```

Note! In order to be tail recursive, `fact-c` asks the recursively called `fact-c` to finish the computation, multiplying by `m`. (and, then, applying the continuation `c` `fact-c` was called with.).

The final product: `factorial` in continuation passing style

```
(define (fact-cps n)
  (define (fact-c m c)
    (if (= m 0) (c 1)
        (fact-c (- m 1)
                  (lambda (x) (c (* m x))))))
  (fact-c n (lambda (x) x)))
```

- ❖ Note: It's tail recursive. Observe how the continuations “pile-up” in the recursive call. This second argument holds all of the pending operations.
- ❖ Note: initially, we simply call `fact-c` with the identity continuation.