Structure and Interpretation of Computer Programs

Second Edition

Harold Abelson and
Gerald Jay Sussman
with Julie Sussman

This book is dedicated, in respect and admiration, to the spirit that lives in the computer.

I think that it's extraordinarily important that we in computer science keep fun in computing. When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every now and then, and after a while we began to take their complaints seriously. We began to feel as if we really were responsible for the successful, error-free perfect use of these machines. I don't think we are. I think we're responsible for stretching them, setting them off in new directions, and keeping fun in the house. I hope the field of computer science never loses its sense of fun. Above all, I hope we don't become missionaries. Don't feel as if you're Bible salesmen. The world has too many of those already. What you know about computing other people will learn. Don't feel as if the key to successful computing is only in your hands. What's in your hands, I think and hope, is intelligence: the ability to see the machine as more than when you were first led up to it, that you can make it more."

Alan J. Perlis (April 1, 1922-February 7, 1990)

Introduction

Questions

- What is computation?
- What is programming?
- What is design?

SICP in a Nutshell

- A classic CS textbook (highly regarded)
- It teaches you how to make a programming language
- Includes a lot of other CS along the way
- Focus is on thinking and problem solving


SIMPLISTIC SOLUTIONS
HOW TO THINK FOR YOURSELF

Our Plan

- Day 1 - Procedures and functions
- Day 2 - Data abstraction and data structures
- Day 3 - Objects, Concurrency, and Streams
- Day 4 - Metacircular evaluator
- Day 5 - Register machines (as much as possible)

Focus

- From the MIT syllabus

  Although the course incorporates a great deal of programming, its intellectual focus is not on particular programming-language constructs, data structures, or algorithms -- these are regarded as details. Instead, students are brought to appreciate a diversity of major programming paradigms: data abstraction, rule-based systems, object-oriented programming, functional programming, logic programming, and constructing embedded interpreters. Beyond that, there is a central concern with the technology of implementing languages and linguistic support for programming paradigms. **Students are encouraged to regard themselves as language designers and implementors rather than only language users.**

Some Thoughts

- SICP is challenging
- Many books introduce computers (bits, bytes, memory, etc.) and approach programming as a thing you do to manipulate a computer. Bottom up. Often emphasizing low-level details.
- SICP introduces a more abstract model of computation and derives everything from that. Top down. High level. Low-level details often "unimportant" or "left as an exercise" for later.

Some Thoughts

- DON’T put Lisp (or SICP) on a ivory-tower

  "Lisp has jokingly been called "the most intelligent way to misuse a computer". I think that description is a great compliment because it transmits the full flavor of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts." - Edsger Dijkstra

- The book is a guide, not a bible

- We're not doing everything (or in exactly the same way)

Takeaways

- More insight about the nature of computation and strategies for problem solving
- A deep dive into the internals of interpreters, programming languages, and machines
- You'll walk away seeing everything you already know through a different view.
- A springboard for further study of functional languages (Clojure, Haskell, etc.) and computer science.

Part 1

Computation and Procedures

Elements of Programming

- Primitives

```
42
3.7
```

- Combinations (operators, expressions)

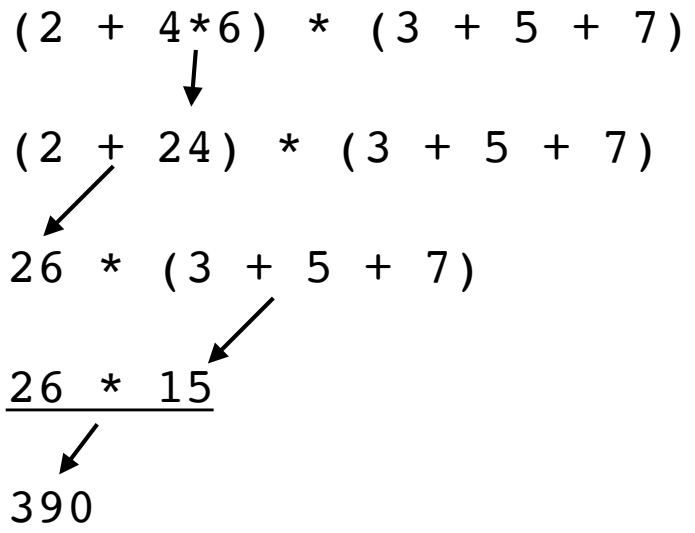
```
42 + 13
3.7 * 6.1
```

- Naming (abstraction)

```
radius = 5
pi = 3.14159
area = pi * radius * radius
```

Evaluation "Rules"

- Subexpressions are evaluated first (inside out)



- These are the rules from math class

Functions

- Functions (from math)

```
f(x) = 3*x*x + 2*x + 1
```

- Function application

```
f(2) -> 17
f(10) -> 321
```

- QUESTION: How do you evaluate a function?
- QUESTION: How would you teach it to a 4th grader?

Substitution Model

- Functions evaluate by substitution

```
f(x) = 3*x*x + 2*x + 1

f(2) -> 3*2*2 + 2*2 + 1 -> 17   # Substitute the “x”
```

- Twist: How do you evaluate this?

```
f(1+1)
```

Substitution Model

- Functions evaluate by substitution

```
f(x) = 3*x*x + 2*x + 1

f(2) -> 3*2*2 + 2*2 + 1 -> 17   # Substitute the “x”
```

- Twist: How do you evaluate this?

```
f(1+1)

f(1+1) -> f(2) -> 3*2*2 + 2*2 + 1
       -> 17
```

“Applicative” Order
(do it first)

Substitution Model

- Functions evaluate by substitution

```
f(x) = 3*x*x + 2*x + 1

f(2) -> 3*2*2 + 2*2 + 1 -> 17   # Substitute the “x”
```

- Twist: How do you evaluate this?

```
f(1+1)

f(1+1) -> f(2) -> 3*2*2 + 2*2 + 1
       -> 17

f(1+1) -> 3*(1+1)*(1+1) + 2*(1+1) + 1
       -> 3*2*2 + 2*2 + 1
       -> 17
```

“Applicative” Order
(do it first)

“Normal” Order
(do it later)

Substitution Model

- Functions evaluate by substitution

```
f(x) = 3*x*x + 2*x + 1

f(2) -> 3*2*2 + 2*2 + 1 -> 17   # Substitute the “x”
```

- Twist: How do you evaluate this?

```
f(1+1)

f(1+1) -> f(2) -> 3*2*2 + 2*2 + 1
       -> 17

f(1+1) -> 3*(1+1)*(1+1) + 2*(1+1) + 1
       -> 3*2*2 + 2*2 + 1
       -> 17
```

“Applicative” Order (do it first)

“Normal” Order (do it later)

- **Observe: There is more than one way to do it**

Deep Idea

- ALL of computation can be described by the evaluation of expressions and the application of functions

- The devil is in the details (mostly concerning “when”)

Scheme 101

- Primitives

```
42
3.7
```

- Combinations (prefix notation)

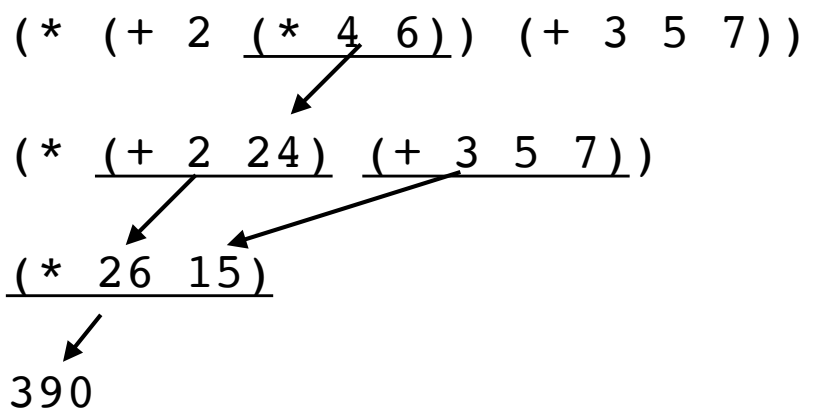
```
(+ 42 37)              -> 79
(- 1000 334)           -> 666
(* 5 99)               -> 495
(+ 2 3 4 5)            -> 14
```

- Naming (variables)

```
(define size 2)

size                    -> 2
(+ size 10)             -> 12
```

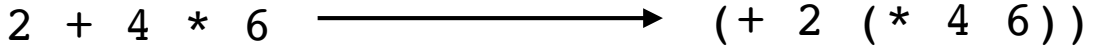
Expression Evaluation

- Subexpressions are evaluated first (inside out)



- Must get used to prefix-notation

- There are no implicit precedence rules (inner always first)



Scheme Naming

- Most things can be named (use define)

```
(define size 2)

(+ size 10)              -> 12
```

- This includes the operators themselves

```
(define op +)

(op size 10)            -> 12
```

- Most things are said to be “first class"

Procedures

- User-defined procedures

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

(square 10)   -> 100
```

- Alternative syntax

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

- Compare to Python

```
def square(x):
    return x * x

square = lambda x: x * x
```

Substitution Model

- How are procedures evaluated?

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

- Substitution concept: Names are substituted with values

```
(square 10)   ->  (* 10 10)    -> 100
```

- A concern.... what happens here?

```
(square (+ 2 3))
```

(Recall: There is more than one way to do it)

Evaluation Models

- Applicative Order - Arguments Evaluated First

```
(square (+ 2 3))

(square 5)

(* 5 5)

25
```

- Normal Order - Delay evaluation until only primitives remain

```
(square (+ 2 3))

(* (+ 2 3) (+ 2 3))

(* 5 5)

25
```

Note: Evaluation order is a recurring theme in SICP.

Scheme Evaluation

- Scheme uses applicative order (like Python)

```
(square (+ 2 3))

(square 5)

(* 5 5)

25
```

- This is how most programming languages work
- Arguments are fully evaluated before calling a procedure

Procedure Syntax

- Scheme syntax is extremely minimal

```
42            ; A primitive
(x y z)       ; A list (procedure application)
```

- There is no other syntax
- Question: Is every feature of Scheme a procedure?

```
(+ 2 3)         ; Is "+" a procedure?
(square 10)     ; Is "square" a procedure?
(define x 42)   ; Is "define" a procedure?
```

- Short answer: no

Special Forms

- Question: Is everything a procedure?
- Answer: No.

```
(define x 42)

(square 5)
```

Discuss the difference...

- "define" not a procedure. "x" is undefined (can't evaluate)
- An example of a "special form"
- Special forms don’t follow the usual evaluation rules

Python Special Forms

- and/or operators

```
>>> 2 + 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 2 or 1/0
2
>>>
```

- Something is "different" about the "or" operator
- It does NOT evaluate all of the arguments first
- There are NO magic/special methods for it either

Conditionals

- if-else (special form)

```
(if predicate consequent alternative)

(define (abs x)
    (if (< x 0) (- x) x))
```

- cond (special form)

```
(cond (pred1 expr1)
      (pred2 expr2)
       ...
      (predn exprn))

(define (abs x)
    (cond ((< x 0) (- x))
          ((= x 0) 0)
          ((> x 0) x)))
```

Conditionals

- if-else (special form)

```
(if predicate consequent alternative)

(define (abs x)
    (if (< x 0) (- x) x))
```

- cond (special form)

```
(cond (pred1 expr1)
      (pred2 expr2)
       ...
      (predn exprn))

(define (abs x)
    (cond ((< x 0) (- x))
          ((= x 0) 0)
          ((> x 0) x)))
```

Discussion:
Why are these special forms?

Conditionals

- if-else (special form)

```
(if predicate consequent alternative)

(define (abs x)
    (if (< x 0) (- x) x))
```

- cond (special form)

```
(cond (pred1 expr1)
      (pred2 expr2)
       ...
      (predn exprn))

(define (abs x)
    (cond ((< x 0) (- x))
          ((= x 0) 0)
          ((> x 0) x)))
```

Only one expression actually gets evaluated

(This is different than how procedures work. Procedures evaluate all of their arguments.)

Special Forms & Keywords

- Special forms usually map to"keywords" in other languages

```
if expr:
   ...
else:
   ...
```

- "if" and "else" are special. Not variables, not procedures.

```
a = if              # SyntaxError (python)
...
```

- Will find similar behavior in Scheme

```
(define a if)      ; bad syntax
```

Commentary

- SICP has many exercises that involve "strange" or "tricky" problems related to evaluation
- Example: Why can't you wrap "if" with a procedure?

```
def choose(test, consequent, alternative):
    if test:
        return consequent
    else:
        return alternative
```

- Doesn't work because all arguments are evaluated first
- This sort of thinking is setup for later chapters

Exercises

- Exercises 1.1, 1.2, 1.3, 1.4, 1.5 (page 20-21)
- Read ahead (section 1.1.7)

Case Study: Square Roots

- How do you compute sqrt(x)?
- One approach: Heron's Method (AD 60)

```
Step 1: Make a guess Y0 = 1

Step 2: Make a new guess Yn = avg(Yn-1, X/Yn-1)

Step 3: Iterate until guesses "converge"
```

- Example: sqrt(2)

```
Y0 = 1
Y1 = avg(1 + 2/1) = 1.5
Y2 = avg(1.5 + 2/1.5) = 1.4167
Y3 = avg(1.4167 + 2/1.4167) = 1.4142
Y4 = avg(1.4142 + 2/1.4142) = 1.4142
```

Case Study: Square Roots

- One approach: Heron's Method (AD 60)

```
Step 1: Make a guess Y0 = 1

Step 2: Make a new guess Yn = avg(Yn-1, X/Yn-1)

Step 3: Iterate until guesses "converge"
```

- Your challenge: Implement this process in Scheme

```
(define (sqrt x) (sqrt-iter 1 x))

; Helper function to drive repeated guessing
(define (sqrt-iter guess x)
     (if (good-enough? guess)
         guess
         (sqrt-iter (improve-guess guess x) x)))
```

- Reading: 1.1.7. Exercise 1.6, 1.7, 1.8

Commentary

- Why square roots? Why this example?
- SICP is about understanding the nature of computation
- Procedures: Computation via substitution
- Square roots: Computation via iterative approximation
- In practice, many things on computers are merely approximations (you can only get close to an answer)

Commentary

- This example involving square roots is also setting up a sub-plot concerning program design and the power of generalization
- Later sections expand upon this example making it more and more general purpose
- This is a common programming task---taking a specific case, recognizing patterns, and generalizing upon those patterns.

Encapsulation

- Idea: Procedures are a kind of "black box"

```
(sqrt 3) -> 1.7320508075688772
```

- Problem: Implementation might involve multiple procs

```
(define (sqrt x) ... )
(define (sqrt-iter guess x) ... )
(define (good-enough? guess x) ...)
(define (improve guess x) ...)
```

- Question: How do you put it together better?

Block Structure

- Nested procedures

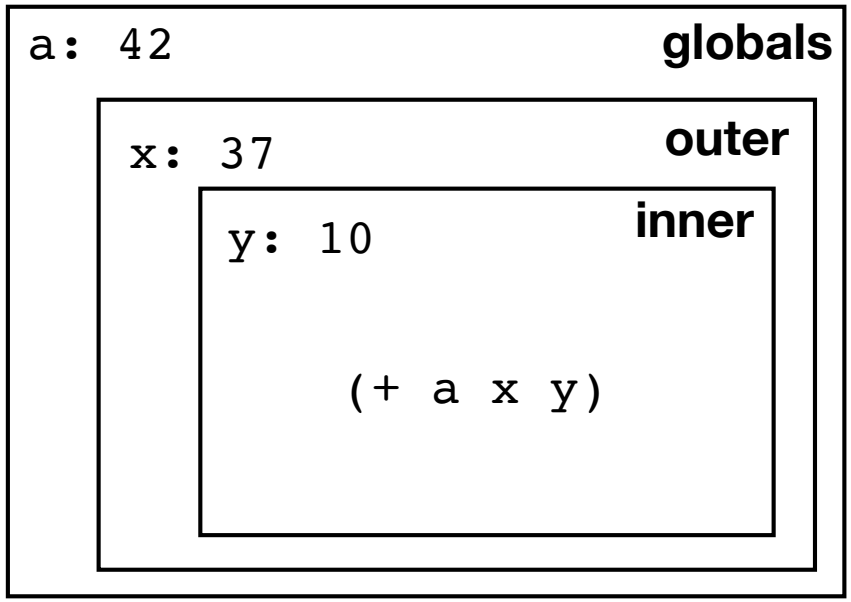
```
(define (sqrt x)
     (define (sqrt-iter guess) ...)
     (define (good-enough? guess) ...)
     (define (improve guess) ...)
     (sqrt-iter 1.0))
```

- There is a nesting of names (lexical scoping)
- Inner procedures see names defined in outer procedures

Name Binding

- Names are nested

```
(define a 42)

(define (outer x)
     (define (inner y)
          (+ a x y))
     (inner 10))

(outer 37)
```



- Terminology:
- "Bound variable" - Bound to a procedure argument
- "Free variable" - Defined outside a procedure

Exercise

- Rewrite sqrt procedure to use internal procedures
- Read pg. 26-30
- Modify your code to match page 30

Computational Processes

- Compute a factorial: 5!
- One approach: 5 * 4 * 3 * 2 * 1

```
(define (fact n)
    (if (= n 1)
        n
        (* n (fact (- n 1)))))
```

- This is Linear Recursion

```
(fact 5)
(* 5 (fact 4))
(* 5 (* 4 (fact 3)))
(* 5 (* 4 (* 3 (fact 2))))
(* 5 (* 4 (* 3 (2 * (fact 1)))))
...
```

- Requires storage to grow linearly (to hold partial results)

Computational Processes

- A different approach: 5 * 4 * 3 * 2 * 1

```
(define (fact n)
    (define (fact-iter n result)
        (if (= n 1) result
            (fact-iter (- n 1) (* n result))))
    (fact-iter n 1)
)
```

- This is Linear Iteration (result is carried along each step)

```
(fact-iter 5 1)
(fact-iter 4 5)
(fact-iter 3 20)
(fact-iter 2 60)
(fact-iter 1 120)
...
```

- Note: Moving to next step requires no retention of information from current step. (Tail-call optimization)

Commentary

- Linear iteration via recursion can be highly efficient

```
(fact-iter 5 1)
(fact-iter 4 5)
(fact-iter 3 20)
(fact-iter 2 60)
(fact-iter 1 120)
...
```

- Series of state changes involving only procedure args
- Not a familiar style to most programmers because it involves recursion and it requires certain optimizations
- Would NOT work well in Python for instance.

Computational Processes

- How to remember: Recursion vs. Iteration
- "Left behind" vs. "Carry forward"
- Iteration carries everything needed to get the final answer forward to the next procedure call
- Recursion leaves part of the calculation behind. Returns later to pick it up and get final result

Computational Processes

- Sometimes it's complicated: Fibonacci Numbers

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

- This is Tree Recursion

```
(fib 5)
(+ (fib 4) (fib 3))
(+ (+ (fib 3) (fib 2)) (+ (fib 2) (fib 1)))
...
```

- Exponential growth of required steps

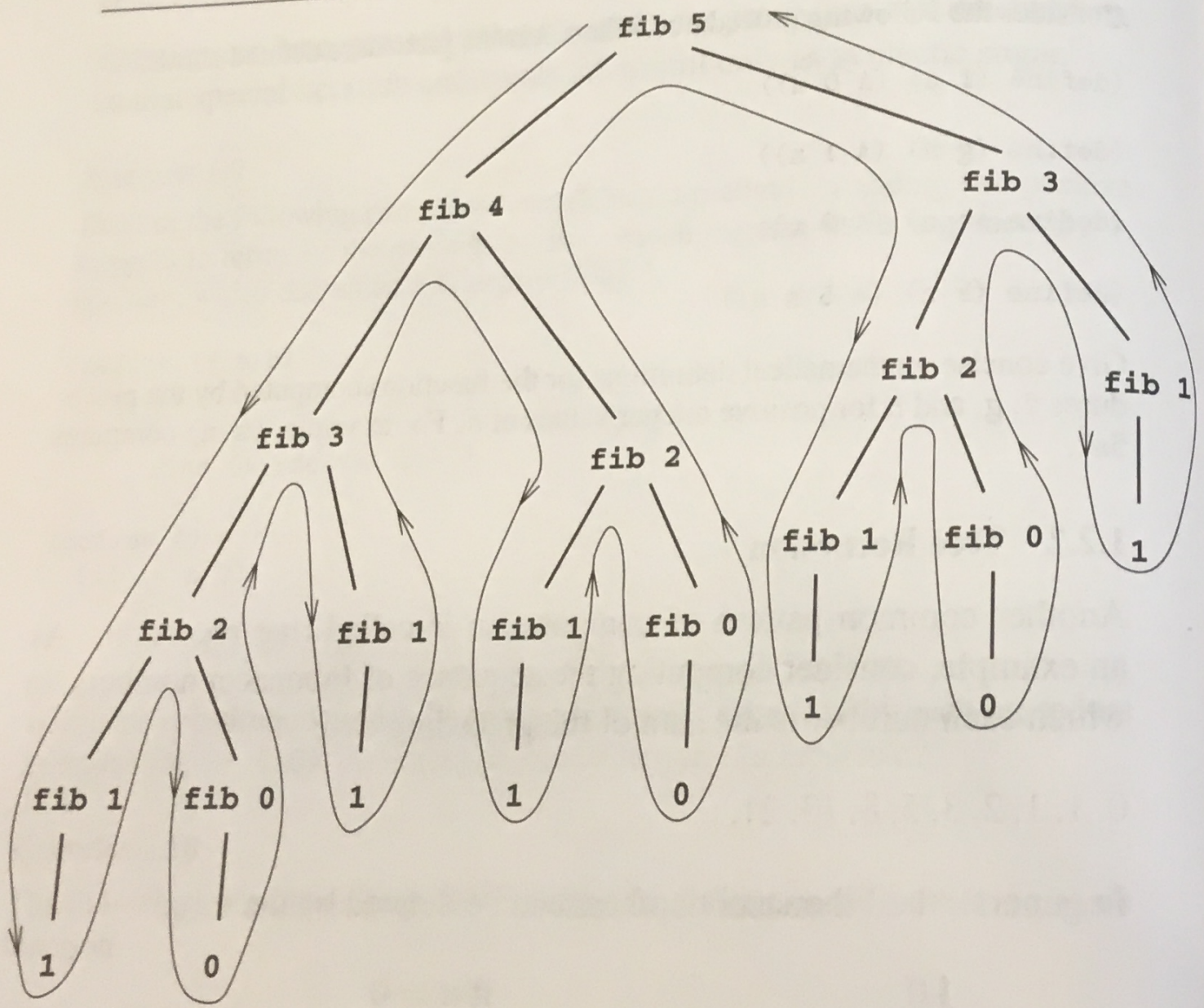


Figure 1.5 The tree-recursive process generated in computing `(fib 5)`.

Commentary

- SICP has many exercises that explore the difference between recursive and iterative processes
- Example: Converting from one to the other
- Partly it's to better understand computation process
- Comes into play later when talking about machines

Exercise

- Reading: Section 1.2-1.2.1
- Exercise 1.9, 1.11
- Read section 1.2.4
- Exercises 1.16, 1.17, 1.18

Observation

- One goal in programming is to generalize and not write so much code
- DRY (Don't Repeat Yourself)
- Abstraction

Problem

- Three sums...

$$\sum_{n=1}^{5} n = 1 + 2 + 3 + 4 + 5$$

$$\sum_{n=1}^{5} n^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3$$

$$\sum_{n=1}^{5} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2}$$

- Do you write three procedures?

Problem

- Three sums...

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

(define (sum-cubes a b)
    (if (> a b)
        0
        (+ (* a a a) (sum-cubes (+ a 1) b))))

(define (sum-inv-squares a b)
    (if (> a b)
        0
        (+ (/ 1.0 (* a a)) (sum-inv-squares (+ a 1) b))))
```

- Most of the code is the same. How to generalize?

Problem

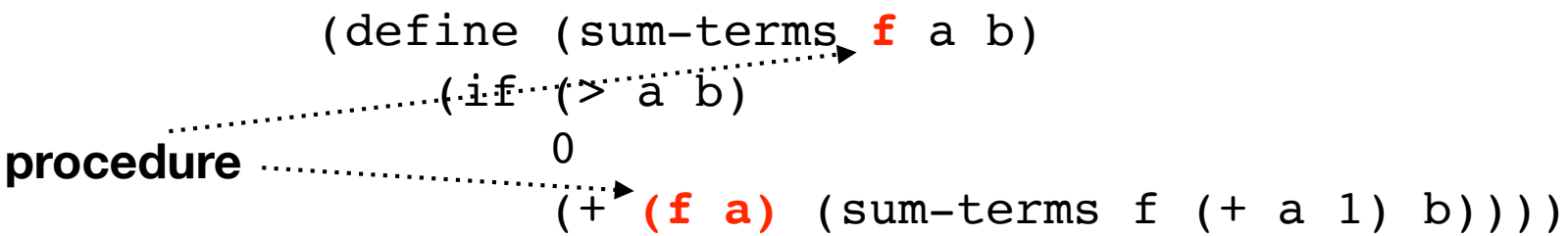
- Three sums...

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

(define (sum-cubes a b)
    (if (> a b)
        0
        (+ (* a a a) (sum-cubes (+ a 1) b))))

(define (sum-inv-squares a b)
    (if (> a b)
        0
        (+ (/ 1.0 (* a a)) (sum-inv-squares (+ a 1) b))))
```

- There is a basic template that the code follows

Higher Order Procedures

- Big Idea: Procedures can be passed as arguments




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

(define (identity n) n)
(define (cube n) (* n n n))
(define (invsquare n) (/ 1.0 (* n n)))

(sum-terms identity 1 5)
(sum-terms cube 1 5)
(sum-terms invsquare 1 5)
```

- Notice: sum-terms is a more general procedure

Anonymous Functions

- lambda special form

```
(lambda (args) expr)
```

- Example

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

(sum-terms (lambda (n) n) 1 5)
(sum-terms (lambda (n) (* n n n)) 1 5)
(sum-terms (lambda (n) (/ 1.0 (* n n))) 1 5)
```

Exercise

- Read/work 1.3.1
- Exercises 1.30-1.32

Problem

- Complicated evaluation (Python)

```
r = x * (1 + x*y)**2 + y * (1 - y) + (1 + x*y) * (1 - y)
```

- There are a few reused subexpressions

```
1 + x*y
1 - y
```

- You might simplify with some temporary variables

```
a = 1 + x*y
b = 1 - y

r = x * a**2 + y * b + a * b
```

- But, you could make it weirder...

The Procedure Hack

- Complicated evaluation

```
r = x * (1 + x*y)**2 + y * (1 - y) + (1 + x*y) * (1 - y)
```

- You could re-cast this using a helper procedure instead

```
def helper(a, b):
   return x * a**2 + y * b + a * b

r = helper(1 + x*y, 1 - y)
```

- But, you could also do this…

```
r = (lambda a, b: x * a**2 + y*b + a*b)(1+x*y, 1-y)
```

- What?!?! Why?

The Procedure Hack

- Complicated function

```
f(x,y) = x * (1 + x*y)**2 + y * (1 - y) + (1 + x*y) * (1 - y)
```

- This is a "standard" technique in Scheme

```
(define (f x y)
    (define (f-helper a b)
        (+ (* x (square a)) (* y b) (* a b)))
    (f-helper (+ 1 (* x y)) (- 1 y)))
```

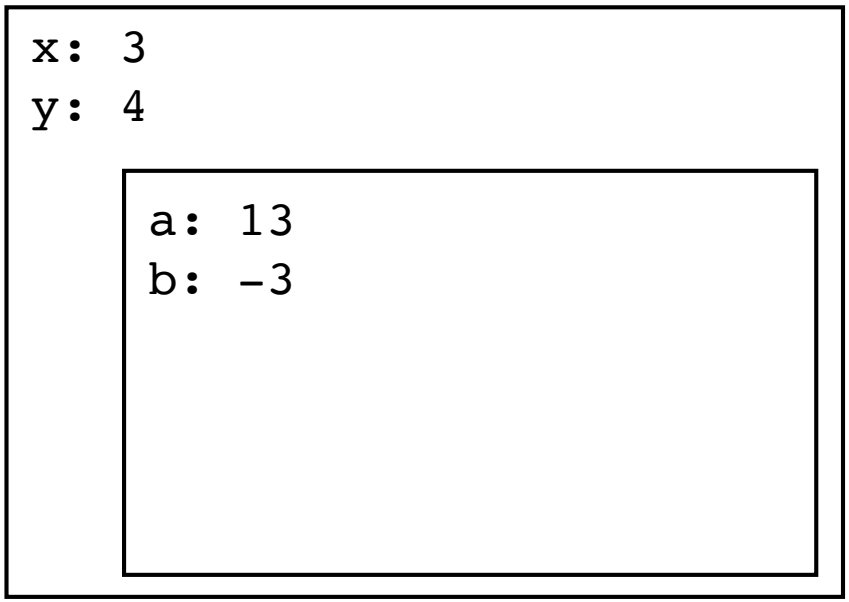
- Recast using a lambda trick

```
(define (f x y)
    ((lambda (a b)
        (+ (* x (square a)) (* y b) (* a b)))
     (+ 1 (* x y)) (- 1 y)))
```

The Procedure Hack

- A picture view

```
(define (f x y)
    ((lambda (a b)
        (+ (* x (square a)) (* y b) (* a b)))
     (+ 1 (* x y)) (- 1 y)))

(f 3 4)
```



- It's setting up a nested name binding

Local Variables

- The "lambda" hack

```
(define (f x y)
    ((lambda (a b)
       (+ (* x (square a)) (* y b) (* a b)))
     (+ 1 (* x y)) (- 1 y)))
```

- It's more easily accomplished using "let"

```
(define (f x y)
    (let ((a (+ 1 (* x y)))
          (b (- 1 y)))
     (+ (* x (square a)) (* y b) (* a b))))
```

- It's a bit brain-bending

Let Special Form

- Let

```
(let ((var1 exp1)
      (var2 exp2)
       ...
      (varn expn))
      body)
```

- It is a syntax translation to the following

```
((lambda (var1 var2 ... varn) body)
 exp1 exp2 ... expn)
```

- "Let" is not a new thing, but a pure transformation of syntax. Change to a lambda and evaluate that.

"Syntactic Sugar"

- A term that sometimes gets thrown about
- Simply stated: syntax translation to existing features
- It exists in Python (e.g., decorators)

```
@decorator
def func():
    ...
```

```
def func():
    ...
func = decorator(func)
```

- Not really necessary, but provided for convenience

Digression

- Programming language designers think about "syntactic sugar" A LOT
- Thought: What is the absolute minimum set of features required to implement a programming language?
- Can higher-level abstractions be implemented purely as syntax translations to lower-level abstractions?
- Some examples: preprocessors, macros, etc.

Exercise

- Read/work examples in section 1.3.2
- Carefully contemplate pg. 65.
- Exercise 1.34

Returning Procedures

- Procedures can be created and returned

```
(define (f x)
    (lambda (y) (+ x y))

(define g (f 10))

(g 20)    --> 30
```

- A critical detail: procedures remember the environment in which they were defined (sometimes called a "closure").

- Note: SICP avoids the "closure" terminology

Python Closures

- Compare:

```
define f(x):
    define add(y):
        return x + y
    return add

>>> g = f(10)
>>> g(20)
30
>>> f(2)(3)
5
>>>
```

- The function returned remembers the variables in the environment in which it was defined.

Returning Procedures

- Creating procedures from other procedures can be useful
- In some sense, it's a form of "code generation."
- Example: Calculus (derivatives, integrals, etc...)
- Case study: Fixed Points (pg. 68)

Calculator Game

```
1234567890
35136.41828644462
187.44710797034085
13.691132457555907
3.700153031640165 4
1.923578184436537 4
1.386931211140818 3
1.177680436765771 4
...
1
1
1
1
```

Keep pressing sqrt key over and over again

Calculator Game

```
1234567890
35136.41828644462
187.44710797034085
13.691132457555907
3.700153031640165 4
1.923578184436537 4
1.386931211140818 3
1.177680436765771 4
...
1
1
1
1
```

You found a "fixed-point"

Keep pressing sqrt key over and over again


```
f = sqrt
```

```
1 = f(1) = f(f(1)) = f(f(f(1)))
```

Pondering: Are there other such functions and values?

Fixed Points

- A general procedure for finding numeric fixed points

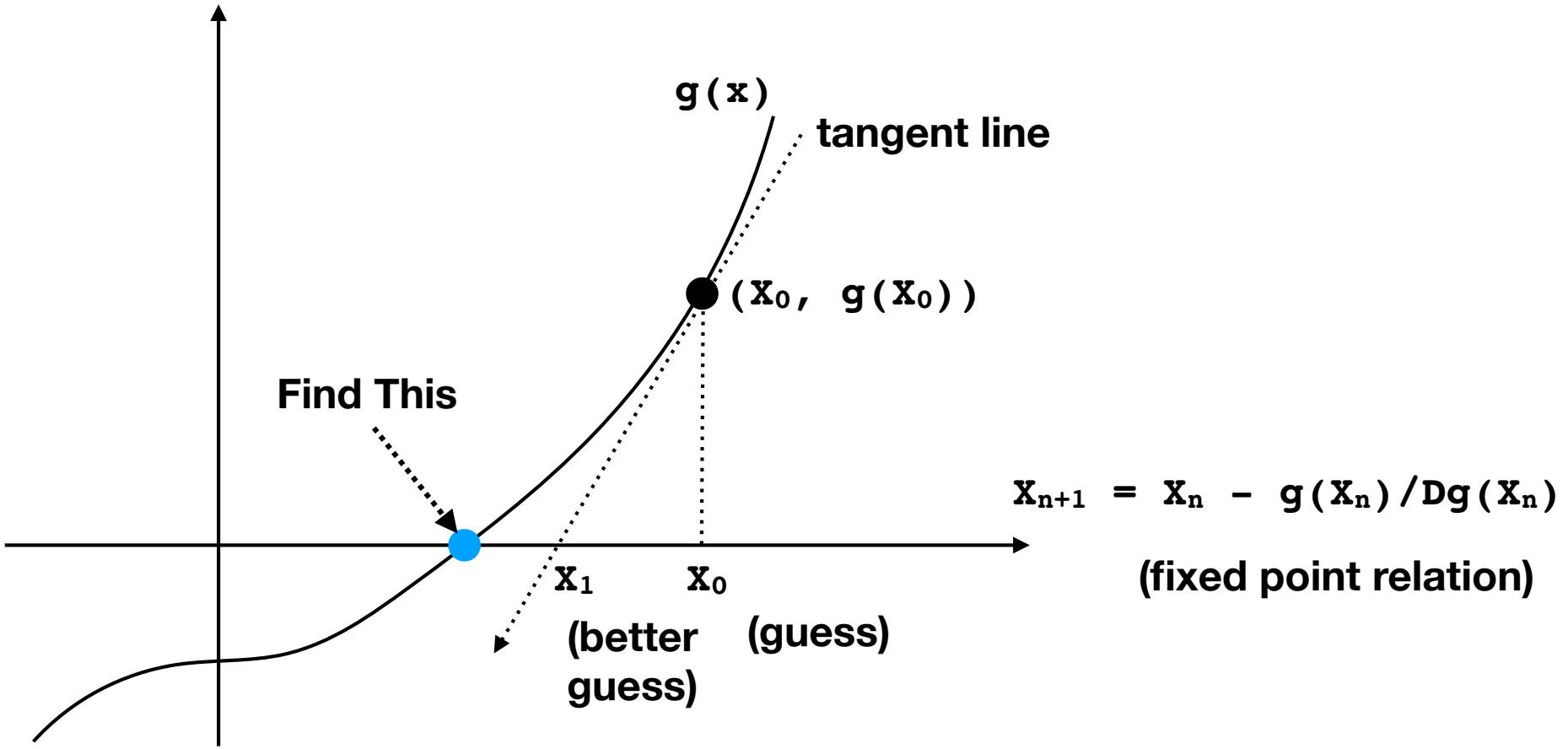
```
(define (fixed-point f first-guess tolerance)

    ; Check if two guesses were close enough
    (define (close-enough? v1 v2)
       (< (abs (- v1 v2)) tolerance))

    ; Try the next value
    (define (try guess)
         (let ((next (f guess)))
            (if (close-enough? guess next)
                next
                (try next))))
    ; Run it
    (try first-guess))
```

- Examples: Enter the above code and try it

```
(fixed-point sqrt 4 0.00001)      ; -> 1.0000006610368821
(fixed-point cos 1.0 0.00001)     ; -> 0.739085292401439
```

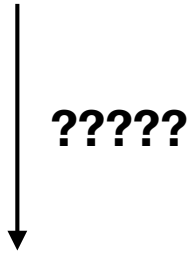
Fixed Points

- Fixed-points are the basis of useful math calculations
- Example: Newton's Method (root-finding)



Problem

- User input: A function g(x)



- Algorithm: find fixed-point of f(x) = x - g(x)/Dg(x)

- To do this, you need to transform the input function g(x) into a different function to run a fixed-point finder on it

- Deep concept: functions can be transformed into other functions

Solution

- Computing derivatives: Dg(x)

```
(define dx 0.0000001)
(define (deriv g)
    (lambda (x)
        (/ (- (g (+ x dx)) (g x)) dx)))
```

- Newton-transform

```
(define (newton-transform g)
    (let ((dg (deriv g)))
      (lambda (x) (- x (/ (g x) (dg x))))))
```

- Applications: Root-finding

```
(define (sqrt y)
   (fixed-point
       (newton-transform (lambda (x) (- (* x x) y)))
        y
        0.00001
    )
)
```

Commentary

- Fixed-points are a weird curiosity
- There is a whole area of mathematics: fixed-point theory
- Overlap with computation: certain calculations involve repeated steps with convergence to a fixed point
- Transformation of functions -> functions is common even outside of "math" (e.g., decorators, metaprogramming)
- Style in SICP may feel weird (more math focused)

Exercise

- Exercises 1.41, 1.42, 1.43, 1.46

Chapter 2

Data Abstraction

Data Abstraction Intro

- You need to work with more than primitive types
- Records, lists, trees, etc.

Pairs

- Scheme provides pairs. Created using cons

```
(define p (cons 3 4))
```

- Accessing components

```
(car p)      -> 3
(cdr p)      -> 4
```

- Compare to Python tuple (values packed together)

Historical Digression

- LISP originally implemented on an IBM 704
- 36-bit word size. One instruction format was

- There were "operations" for accessing the parts

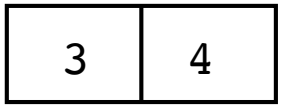
```
CAR    - Contents of Address
CDR    - Contents of Decrement
```

- The names have persisted...

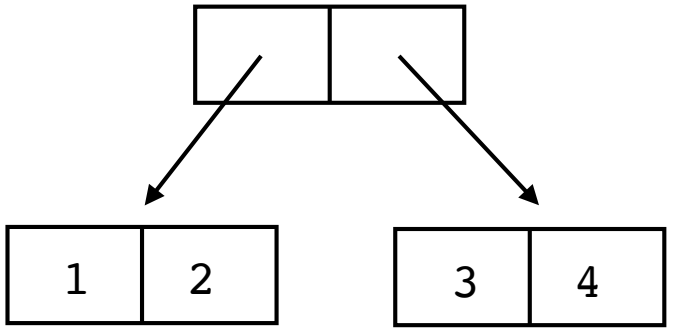
Pairs

- You can make pairs out of anything, including other pairs

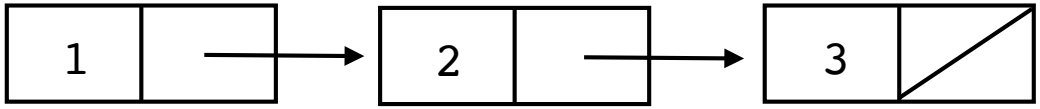
```
(cons 3 4)
```




```
(cons (cons 1 2) (cons 3 4))
```




```
(cons 1 (cons 2 (cons 3 nil))
```



- You can also make anything out of pairs...

Commentary

- Pairs are kind of like the "machine language" of data
- They are an implementation detail
- You're not supposed to worry about the implementation
- Big idea: Abstraction

Abstraction

- Problem: Represent fractions (rational numbers)

```
3/4    ->  (define a (cons 3 4))
2/3    ->  (define b (cons 2 3))

a + b  -> (cons (+ (* (car a) (cdr b)) (* (cdr a) (car b)))
                (* (cdr a) (cdr b))
```

- Would you code it like this? NO!

- You will abstract details to functions

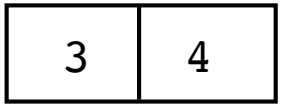
```
(define (make-rat numer denom) ...)
(define (numer rat) ...)
(define (denom rat) ...)
(define (add-rat a b) ...)
```

- Low-level details "unimportant" (encapsulation)

What is Data?

- A pair might be a small record (internal to Scheme)

```
(cons 3 4)
```



- But maybe it's this devious function hack...

```
(define (cons a b)
    (define (dispatch m)
       (cond ((= m 0) a)
             ((= m 1) b))
    )
    dispatch)
(define (car p) (p 0))
(define (cdr p) (p 1))
```

- Again, you don't care about specifics. (Care about API)

Understanding SICP

- The book is written in a "top-down" style
- You take the problem you're solving and you figure out the high-level parts that you need. You THEN worry about how to make those parts later (details)
- This is quite different than a lot of "modern" software development where you download packages and you then you figure out how to solve the problem using the parts that have been provided.

Exercise

- Reading 2.1.1, Enter code, Exercise 2.1
- Reading 2.1.2, Exercise 2.2
- Reading 2.1.3, Exercise 2.4, 2.5, 2.6 (Deep Thinking)

Digression

Lambda Calculus

- Developed in 1930s by Alonzo Church
- Idea: Smallest universal language for expressing "effective calculability"
- Was exploring ideas from foundations of mathematics
- Gist: Functions are the only thing needed

λ-Calculus

- There are only three valid expressions:

  Names: `x, y, etc.`

  Function: λ<*name*>.<*expression*>

  Application: <*expression*><*expression*>

- There is NOTHING else
  - No numbers or other primitives
  - No math operators (+,-, etc.)
  - No data structures (pairs, etc.)

Example:

- A simple function (identity)

(scheme)

```
λx.x
```

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

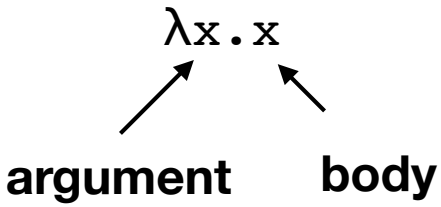
- Application of the function

(scheme

```
(λx.x)y
    ↓
    x
    ↓ Substitute [y/x]
    y
```

```
((lambda (x) x) y)
                 ↓
                 y
```

- Reading it is a bit tricky at first



Exercise

- What is the result?

  `(λx.(λy.xy))ab`

- What is the result?

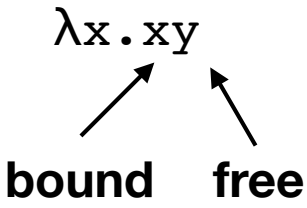
  `(λx.(λy.xy))y`

- And this?

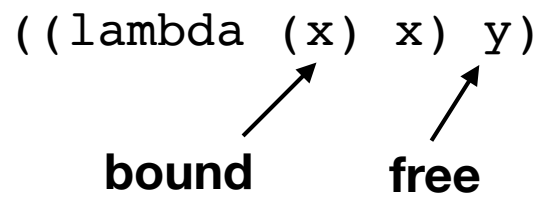
  `(λx.(λy.(x(λx.xy))))y`

Variable Binding

- Variables are "bound" or "free"

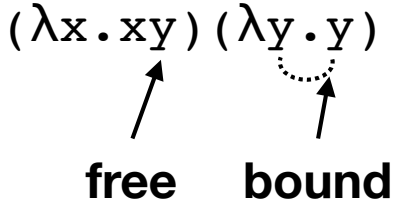


(scheme)



- Bound variables are attached to a function argument

- But, it gets tricky when multiple functions are involved



(These are two different "y" variables)

Variable Renaming

- A BAD substitution

```
(λx.xy)(λy.y)
     ↓
    xy
     ↓   Substitute [λy.y/x]
  λy.yy
```

- You need to rename bound variables (alpha-conversion)

```
(λx.xy)(λz.z)
     ↓
    xy
     ↓   Substitute [λz.z/x]
  λz.zy
```

- Honestly, this is a "code style" problem.

The Rules (in Scheme)

- Rule 1: Single argument functions are the only thing

```
(define (f x) ...)      ; YES!
(define (f x y) ...)    ; NO!
```

- Rule 2: Functions can only return functions

```
(define (f x) (lambda (y) ...))     ; YES!
(define (f x) x)                    ; YES! (See rule 1)
(define (f x) (+ x 1))              ; NO! (No numbers)
```

- Rule 3: Function application is the only operation

```
(define (f x) (lambda (y) (x y)))  ; YES!
(define (f x) (lambda (y) (y x)))  ; YES!
```

Multiple Arguments

- Multiple argument functions can be decomposed into single argument functions (currying)

```
(define (f x y) (+ x y))

(define (f x)
    (lambda (y) (+ x y))

> ((f 2) 3)
5
>
```

- Looks weird, but is essential for later discussion (remember, we're only allowed single-argument functions)

Moment of Reflection

- Can you actually do anything useful with this?
- If so, what?

Pairs

- Recall: SICP (p. 91-92). Exercise 2.4.

```
(define (cons x y)
    (lambda (m) (m x y)))

(define (car z)
   (z (lambda (p q) p)))

(define (cdr z)
   (z (lambda (p q) q)))
```

- Maybe you could do something like that!

Exercise

- Define the following 1-argument functions

```
(define (CONS a)
   (lambda (b) (lambda (z) ((z a) b))))

(define (CAR p) ???)    ; pick a above

(define (CDR p) ???)    ; pick b above
```

- Verify that it works:

```
> (define p ((CONS 2) 3))
> (CAR p)
2
> (CDR p)
3
>
```

Solution

- Define the following 1-argument functions

```
(define (CONS a)
   (lambda (b) (lambda (z) ((z a) b))))

(define (CAR p) (p (lambda (a) (lambda (b) a))))

(define (CDR p) (p (lambda (a) (lambda (b) b))))
```

- Verify that it works:

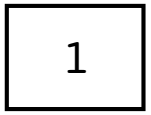
```
> (define p ((CONS 2) 3))
> (CAR p)
2
> (CDR p)
3
>
```

For More Information

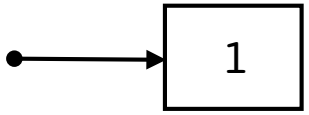
- Beazley, "Lambda Calculus from the Ground Up", presented at PyCon 2019
- That tutorial is highly polished and better than what I would present here.
- Not absolutely essential for rest of SICP
- But, you're getting a taste of it in Exercises 2.4-2.6

Box/Pointer Diagram

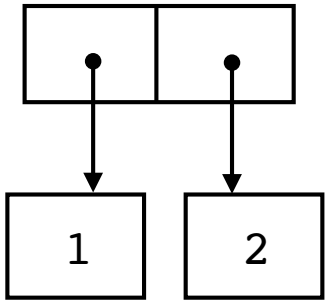
- Primitive objects are shown in a box



- Pointers to objects shown as an arrow



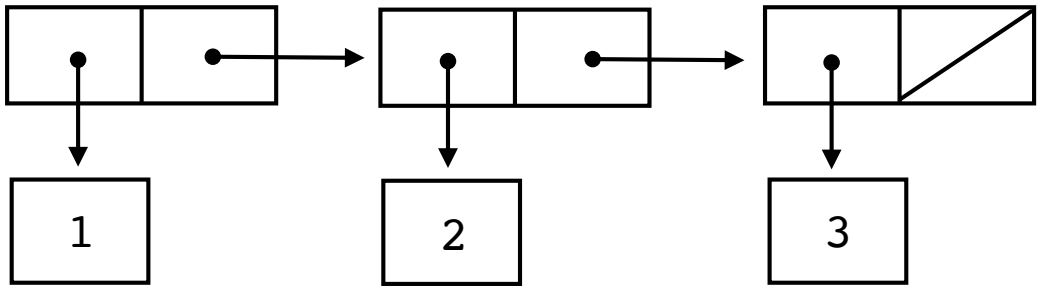
- Pair: (cons 1 2)



Lists/Sequences

- Scheme lists are linked cons cells.

```
(list 1 2 3)   -> (cons 1 (cons 2 (cons 3 nil))
```



- (list a b c) is a special form that creates linked cons cells
- nil represents empty list. Sometimes written '().
- (null? s) tests for empty list

car/cdr shortcuts

- Combinations of car/cdr are frequently used

```
(define s (list 1 2 3))

(car s)              -> 1
(cdr s)              -> (list 2 3)
(car (cdr s))        -> 2
(car (cdr (cdr s))) -> 3

; Shortcuts

(car s)    -> 1
(cadr s)   -> 2
(caddr s)  -> 3
```

Traversal

- Traversing over elements of a list

```
(define s (list 1 2 3))

(define (something s <arg>)
    (if (null? s) nil
        (<op> (something (cdr s) <arg>))))
```

- It's recursive.

```
(define (length s)
    (if (null? s) 0
        (+ 1 (length (cdr s)))))
```

Common Operations

- Picking out the nth element

```
(define (list-ref s n)
   (if (= n 0) (car s)
       (list-ref s (- n 1))))
```

- Mapping a procedure onto items

```
(define (list-map proc s)
   (if (null? s) nil
       (cons (proc (car s))
             (list-map proc (cdr s)))))

(define a (list 1 2 3))
(define squares (list-map (lambda (x) (* x x)) a)
```

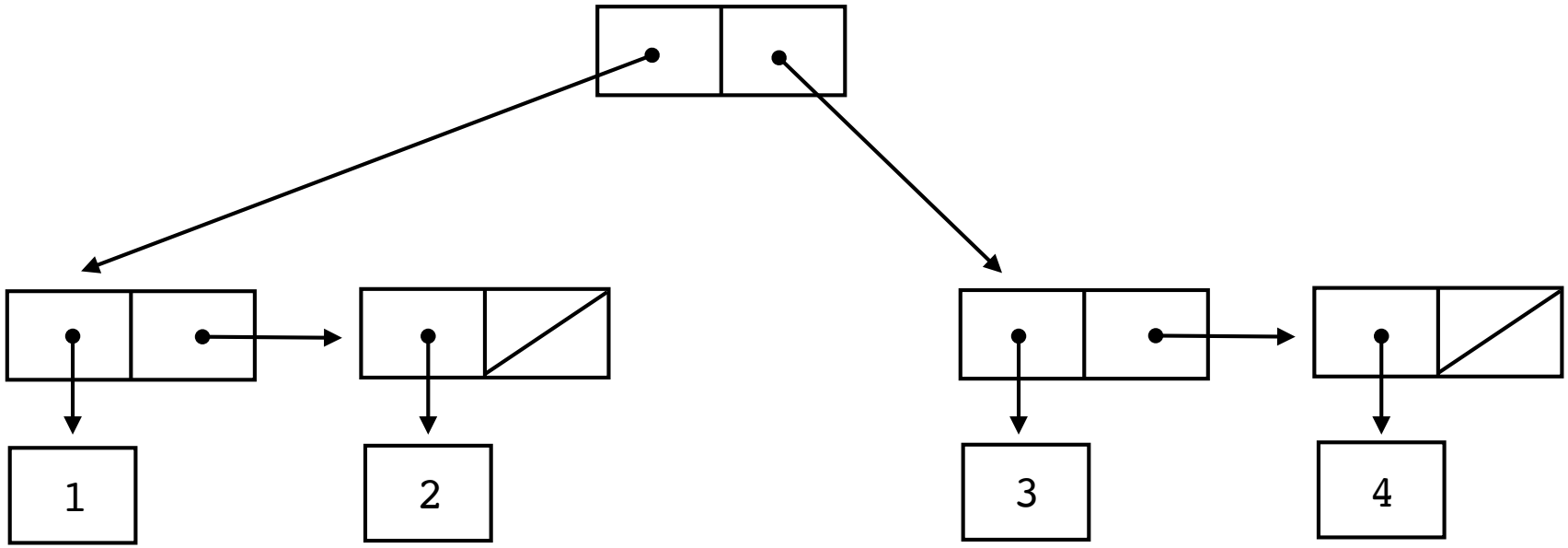
Exercise

- Reading 2.2.1
- Exercises 2.17 - 2.18, 2.20, 2.21, 2.22, 2.23

Trees

- Can build tree structures from cons-cells

```
(cons (list 1 2) (list 3 4))
```



- Overall idea is similar to lists (just more brain bending)

Exercise

- Reading 2.2.2
- Exercises 2.25, 2.26
- Bonus: Exercise 2.27, 2.28

Conventional Interfaces

- Big idea: Lists are a foundational data structure
- There are conventional ways of manipulating lists
- Common operations

```
enumerate s                 # Produces items in s
filter proc s               # Filter out items
map proc s                  # Transform items
accumulate op s             # Accumulation (reduce)
```

- Useful things can be done by combining these concepts

Conventional Interfaces

- You already do this in Python
- Example:

```
for item in s:                    # "Enumerates" items
    ...

[expr for item in s]              # Mapping (list comp)

[item for item in s if expr]      # Filter

reduce(op, s)                     # Reduction
```

- You don't make "custom operations". You use a standard set of list operations

Exercise

- Reading 2.2.3
- Exercise 2.33, 2.36, 2.38, 2.39

Symbols/Quotation

- Symbolic data

```
(quote 42)    -> 42
(quote a)     -> a
(quote (a b c)) -> (list (quote a) (quote b) (quote c))
```

- Sometimes written

```
'42
'a
'(a b c)
```

- Note: Quoting is NOT the same as a text-string

- It leaves the next item unevaluated

Symbols/Quotation

- Evaluation

```
(define a 3)
(define b 4)

(define c (+ a b))          -> 7

(define d '(+ a b))         -> (list '+ 'a 'b)
```

- All scheme expressions are lists. So, quoting gets you an unevaluated list. Symbols are left intact as symbols.

Symbols/Quotation

- What's the point of symbols? (reminder: NOT strings)

```
(define a 'blah)          -> blah
(define d '(+ a b))       -> (list '+ 'a 'b)
```

- Quotation allows arbitrary expressions to be captured
- Useful in metaprogramming (e.g., macros, etc.)
- Also serve a purpose in data abstraction

Exercise

- Reading 2.3
- Exercises 2.53, 2.54, 2.55
- Work through Example 2.3.2 (symbolic differentiation)
- Exercise 2.56, 2.57

Aside: Quasiquotes

- A quote where selected parts can be unquoted

```
(quasiquote (a b (unquote (+ 3 4)) c)) -> (list 'a 'b 7 'c)
```

- Alternative syntax:

```
`(a b ,(+ 3 4) c)
```

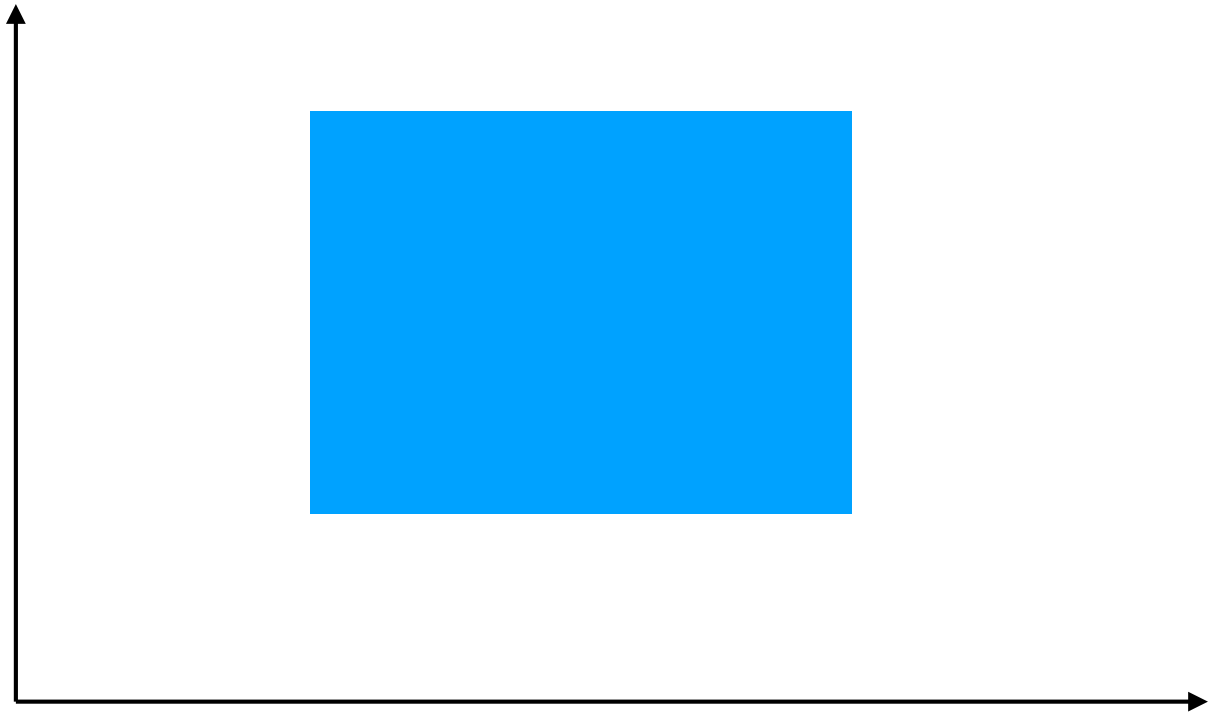
unquote

- Not used much in SICP, but you might find that quote (') and backtick (`) seem "the same". Not the same. Backticks are a quasiquote.

Commentary

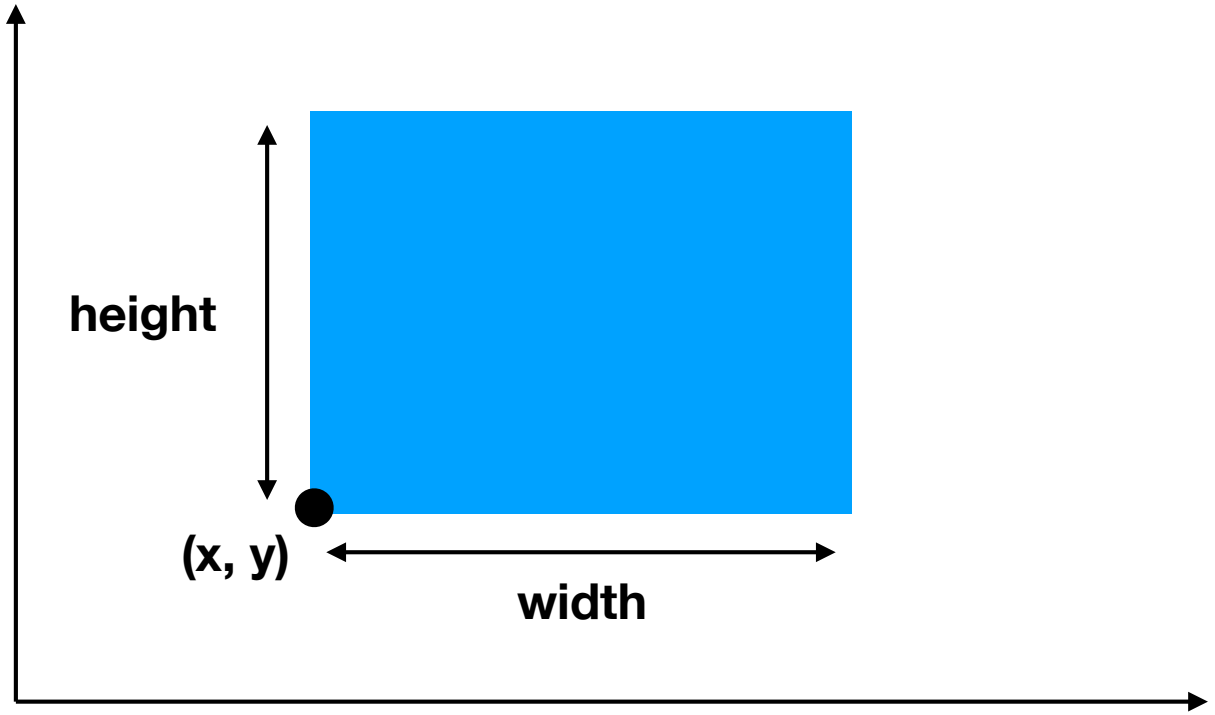
- Going to deviate from SICP for the next bit
- TLDR; It develops concepts related to "objects"
- Will introduce same problems, but use a simpler example and series of guided exercises

Problem: THE BOX



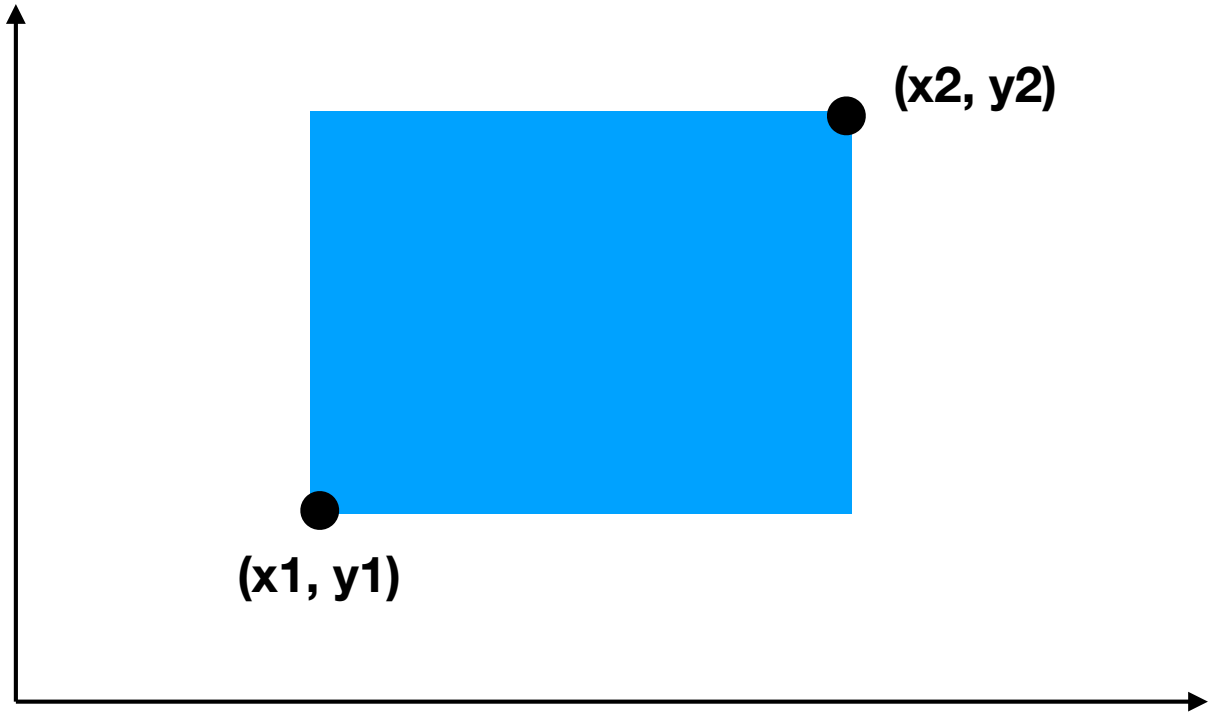
- "Oh such a beautiful box! How do I represent thee?"

The Lengths




```
(cons (cons x y) (cons height width))
```

The Points




```
(cons (cons x1 y1) (cons x2 y2))
```

Bob

```
(define (make-box x y w h)
  (cons (cons x y) (cons w h)))

(define (width box)
   (car (cdr box))

(define (height box)
   (cdr (cdr box))

(define (area box)
    (* (width box) (height box))
```

Alice

```
(define (make-box x1 y1 x2 y2)
 (cons (cons x1 y1) (cons x2 y2)))

(define (width box)
   (abs (- (car (cdr box))
           (car (car box)))))

(define (height box)
   (abs (- (cdr (cdr box))
           (cdr (car box)))))

(define (area box)
    (* (width box) (height box))
```

Two different boxes. Similar functionality...

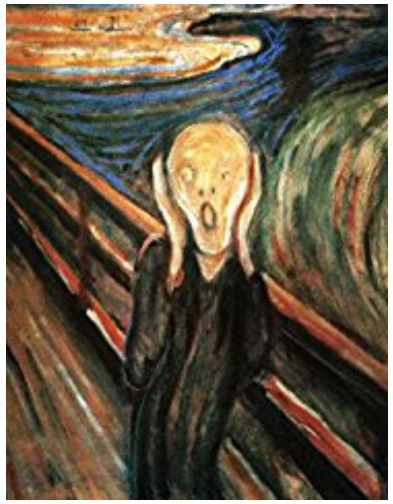
Bob

```
(define (make-box x y w h)
  (cons (cons x y) (cons w h)))

(define (width box)
   (car (cdr box))

(define (height box)
   (cdr (cdr box))

(define (area box)
    (* (width box) (height box))
```

Alice

```
(define (make-box x1 y1 x2 y2)
 (cons (cons x1 y1) (cons x2 y2)))

(define (width box)
   (abs (- (car (cdr box))
           (car (car box)))))

(define (height box)
   (abs (- (cdr (cdr box))
           (cdr (car box)))))

(define (area box)
    (* (width box) (height box))
```

Now, as often happens, suppose that both boxes must coexist in the same program....



Problem: Naming

- Function names must be unique (namespace clash)

```
(define (make-bob-box x y w h)
   (cons (cons x y) (cons w h)))

(define (bob-width box)
   (car (cdr box))

(define (bob-height box)
   (cdr (cdr box))

(define (bob-area box)
    (* (bob-width box)
       (bob-height box))
```

```
(define (make-alice-box x1 y1 x2 y2)
 (cons (cons x1 y1) (cons x2 y2)))

(define (alice-width box)
   (abs (- (car (cdr box))
           (car (car box)))))

(define (alice-height box)
   (abs (- (cdr (cdr box))
           (cdr (car box)))))

(define (alice-area box)
    (* (alice-width box)
       (alice-height box))
```

Exercise

- Enter the shown code for "bob" and "alice"
- Verify that it works

```
> (define a (make-alice-box 1 2 3 4))
> (define b (make-bob-box 1 2 3 4))
> (alice-area a)
4
> (bob-area b)
12
```

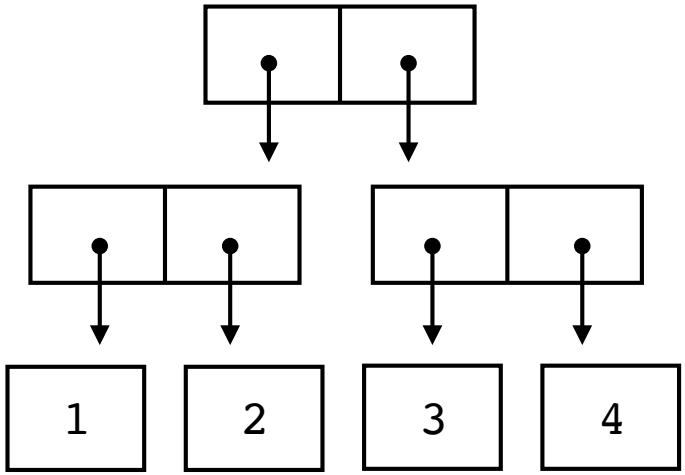
- Look at the resulting data structures

```
> a
'((1 . 2) 3 . 4)
> b
'((1 . 2) 3 . 4)
```

Problem: Identification

```
a = (make-alice-box 1 2 3 4)
b = (make-bob-box 1 2 3 4)
```

- They have the exact same structure!



- What box is it? Is it even a box? Is it a tree?

Problem: Generic Procs

- There are different implementations of a proc

```
(define (bob-width box) ...)
(define (alice-width box) ...)
```

- Can you make a generic procedure?

```
a = (make-alice-box 2 3 4 5)
b = (make-bob-box 2 3 4 5)

(width a)       # -> 2
(width b)       # -> 4
```

- You can't do this unless you can identify the box

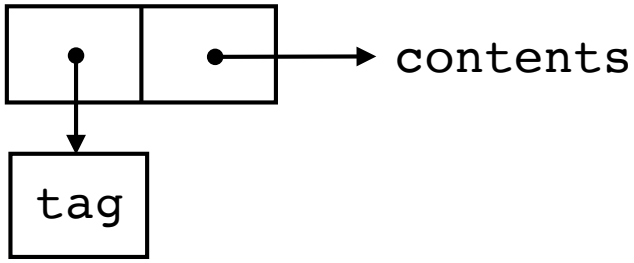
Solution: Types

- To address these problems, you have to introduce the concept of “types”
- Comment: This is a huge topic that expands far beyond the coverage provided in SICP

Data Tagging

- One approach: Attach type tags to data

```
(define (attach-tag tag contents) (cons tag contents))
(define (type-tag datum) (car datum))
(define (contents datum) (cdr datum))
```



- The tag becomes an extra item of data carried around

Data Tagging

- Modified box code with tags

```
; constructor function
(define (make-bob-box x y w h)
      (attach-tag 'bob-box
          (cons (cons x y) (cons w h))))

; type-check function
(define (bob-box? b) (eq? (type-tag b) 'bob-box))

; Various methods
(define (bob-width b) (car (cdr (contents b))))
(define (bob-height b) (cdr (cdr (contents b))))
```

Exercise

- Modify the "bob" and "alice" code to use tags
- Verify that this still works:

```
> (define a (make-bob-box 1 2 3 4))
> (define b (make-alice-box 1 2 3 4))
> (bob-area a)
12
> (alice-area b)
4
```

- Look at the resulting data structures

```
> a
'(bob-box (1 . 2) 3 . 4)
> b
'(alice-box (1 . 2) 3 . 4)
>
```

Generic Procedures

- Type-tags can be used to dispatch based on type

```
(define (width b)
   (cond ((bob-box? b) (bob-width b))
         ((alice-box? b) (alice-width b))
    )
)
```

- Example:

```
> (define a (make-bob-box 1 2 3 4))
> (define b (make-alice-box 1 2 3 4))
> (width a)
3
> (width b)
2
>
```

Exercise

- Make generic procedures for width, height, and area
- Verify operation:

```
> (define a (make-bob-box 1 2 3 4))
> (define b (make-alice-box 1 2 3 4))
> (area a)
12
> (area b)
4
```

- Look at the resulting data structures

```
> a
'(bob-box (1 . 2) 3 . 4)
> b
'(alice-box (1 . 2) 3 . 4)
>
```

Problem: Additivity

- Yes, you can write general procedures using the tags

```
(define (width b)
   (cond ((bob-box? b) (bob-width b))
         ((alice-box? b) (alice-width b))
    )
)
```

- However, it's hard-coded to look for specific tags. If new objects are added, then code has to be changed. It's brittle. In the words of SICP, it's not "additive" (meaning you can't add new types)

Data Directed Dispatch

- Dispatch functions through a table

```
registry = {                        # PSEUDOCODE
   '(width bob-box) -> bob_width,
   '(width alice-box) -> alice-width
   ...
}

(define (width box)
    (registry['width (type_tag box)] box))
```

- Problem: We have not talked about tables/mappings at all. SICP waves hands and assumes that you can do it (via unspecified magic)

- We need a dictionary or hash

Racket Hash Table

- Core functions

```
(make-hash)              ;  Creates a mutable hash table
(hash-set! h key val)    ;  Add an entry
(hash-ref h key)         ;  Look up a value
```

- Example:

```
> (define h (make-hash))
> (hash-set! h 'foo 42)
> (hash-set! h 'bar 37)
> h
'#hash((foo . 42) (bar . 37))
> (hash-ref h 'foo)
42
> (hash-ref h 'bar)
37
>
```

Data Directed Dispatch

- Registry management

```
(define registry (make-hash))
(define (register name tag func)
  (hash-set! registry (list name tag) func))

(define (lookup name tag)
  (hash-ref registry (list name tag)))
```

- How it gets used

```
; Registration of functions
(register 'width 'bob-box bob-width)
(register 'width 'alice-box alice-width)

; Generic procedure
(define (width box)
   ((lookup 'width (type-tag box)) box))
```

Exercise

- Add registry lookup table to your code
- Register various width/height functions with registry
- Rewrite generic width/height functions to use registry
- Make sure everything still works

Code Organization

- Complaint: The code is a mess of names

```
width
height
bob-width
bob-height
alice-width
alice-height
```

- Everything lives in a unified namespace

- "*Namespaces are one honking great idea -- let's do more of those!*"

Namespaces

- Put the functions inside an outer function

```
(define (install-bob-box-funcs)

  (define (width box)
    (car (cdr (contents box))))

  (define (height box)
    (cdr (cdr (contents box))))

  (register 'width 'bob-box width)
  (register 'height 'bob-box height)
)

(install-bob-box-funcs)
```

- Have it register the internal functions

Exercise

- Put the generic "bob" and "alice" functions into a namespace
- Make sure the generic functions still work

Complaints

- This feels very complicated
    - Global function registry
    - Various constructor functions
    - Type-tagging/unboxing
    - Installation functions
- Let's look at a different approach

Message Passing

- Create a function that receives messages

```
(define (make-bob-box x y width height)
  (define (dispatch message)
    (cond ((equal? message 'width) width)
          ((equal? message 'height) height)
          ((equal? message 'type) 'bob-box)
          )
    )
  dispatch
)
```

- Example:

```
> (define a (make-bob-box 1 2 3 4))
> (a 'width)
3
> (a 'height)
4
> (a 'type)
'bob-box
>
```

Message Passing

- Generic procedures use the messages

```
(define (width box)
   (box 'width))

(define (height box)
   (box 'height))
```

- Example: It works the same way as before

```
> (define a (make-bob-box 1 2 3 4))
> (width a)
3
> (height b)
4
>
```

Exercise

- Reimplement "alice" and "bob" code as message passing
- Make a third kind of box consisting of the following data
  - Lower left corner (x, y)
  - Width
  - Aspect ratio (height = width*aspect)

Commentary: Objects

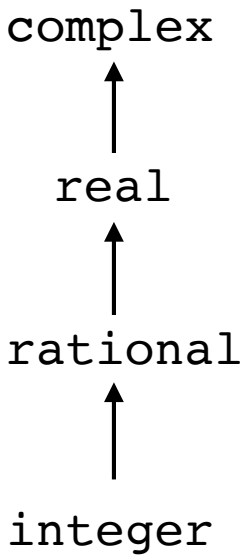
- OO programming is based on message passing
- Terminology is a bit unusual, but it's the basis of (.)

```
obj.attr   # Send "attr" message to "obj"
```

- A class serves as a namespace for functions
- Dispatching often table-driven (e.g., dicts in Python)

Type Hierarchies

- May have a hierarchy of related types



- For example, the numeric type tower (above)
- Inheritance. Subtyping.

Mixed Type Operations

- Example: Python

```
>>> a = 1              # int
>>> b = 2.5            # float
>>> a + b
3.5
>>>

>>> from fractions import Fraction
>>> f = Fraction(1, 4)
>>> g = 3 + 4j
>>> f + g
(3.25+4j)
>>>
```

- Comment: It works, but tricky complexity is involved

Mixed Type Operations

- You might only have functions for compatible types

```
add-complex
add-real
add-rational
add-integer
```

- How to make mixed-type operations work?

Mixed Type Operations

- One solution: Implement all possible combinations

```
add-integer-integer
add-integer-real
add-integer-rational
add-integer-complex
add-real-integer
add-real-rational
add-real-complex
...
```

- Uh. No.

Type Coercion

- Implement special conversion functions

```
integer-to-rational
integer-to-real
integer-to-complex
rational-to-real
rational-to-complex
real-to-complex
```

- Incorporate a conversion process into the implementation

Type Coercion

- Implementation with conversion

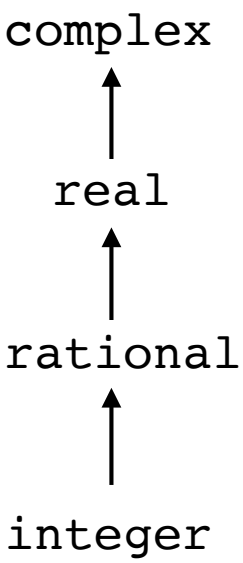
```
(define (add-real x y)
   (cond
      ((integer? y) (add-real x (integer-to-real y))
      ((real? y) (+ x y))
    )
)
```

- But, it gets messy. What if arguments are reversed?

```
(add-real 2.5 1)
(add-real 1 2.5)   ; ????
```

Type Lifting

- Also: Do types need to know the whole hierarchy?



- Do you need this function?

```
integer-to-complex
```

Type Lifting

- Solution: Give each type a function that "raises" the type to the next highest level (it's "supertype")

```
integer-raise       ; integer->rational
rational-raise      ; rational->real
real-raise          ; real->complex
```

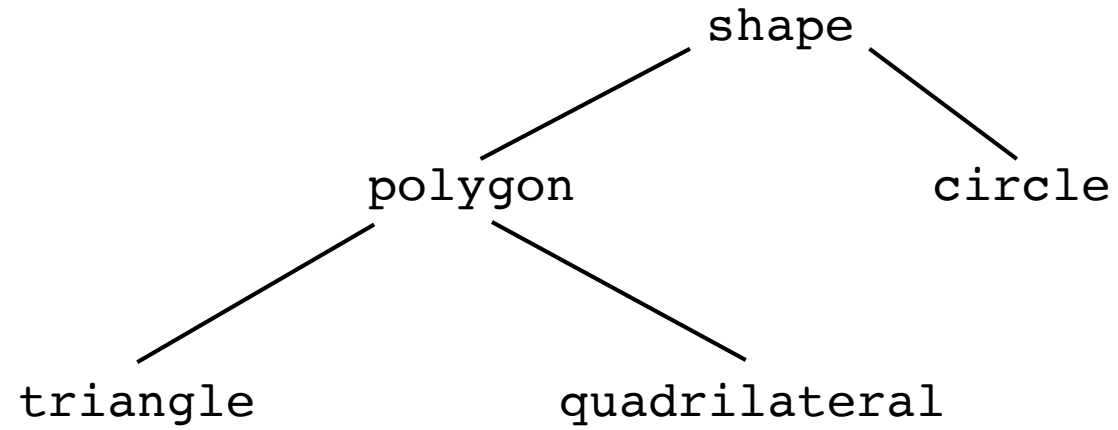
- You then modify dispatching to raise arguments

```
(add 'integer 'real)
          |  integer-raise
          ↓
(add 'rational 'real)
          |  rational-raise
          ↓
(add 'real 'real)          Match!
```

- In a nutshell: inheritance

Type Hierarchies

- May have a hierarchy of related types



- Subtyping. Inheritance.

- A subtype might be "raised" to a higher level type

- SICP doesn't explore this angle much except to say that type hierarchies are "very difficult".

Case Study

- Demo: Handling of mixed-type arithmetic in Python
- Demo: Inheritance implementation in Python
- Discussion: Type linearization

Scheme Project

- Implement a mini Scheme interpreter in Python
- See the file scheme.py

Chapter 3

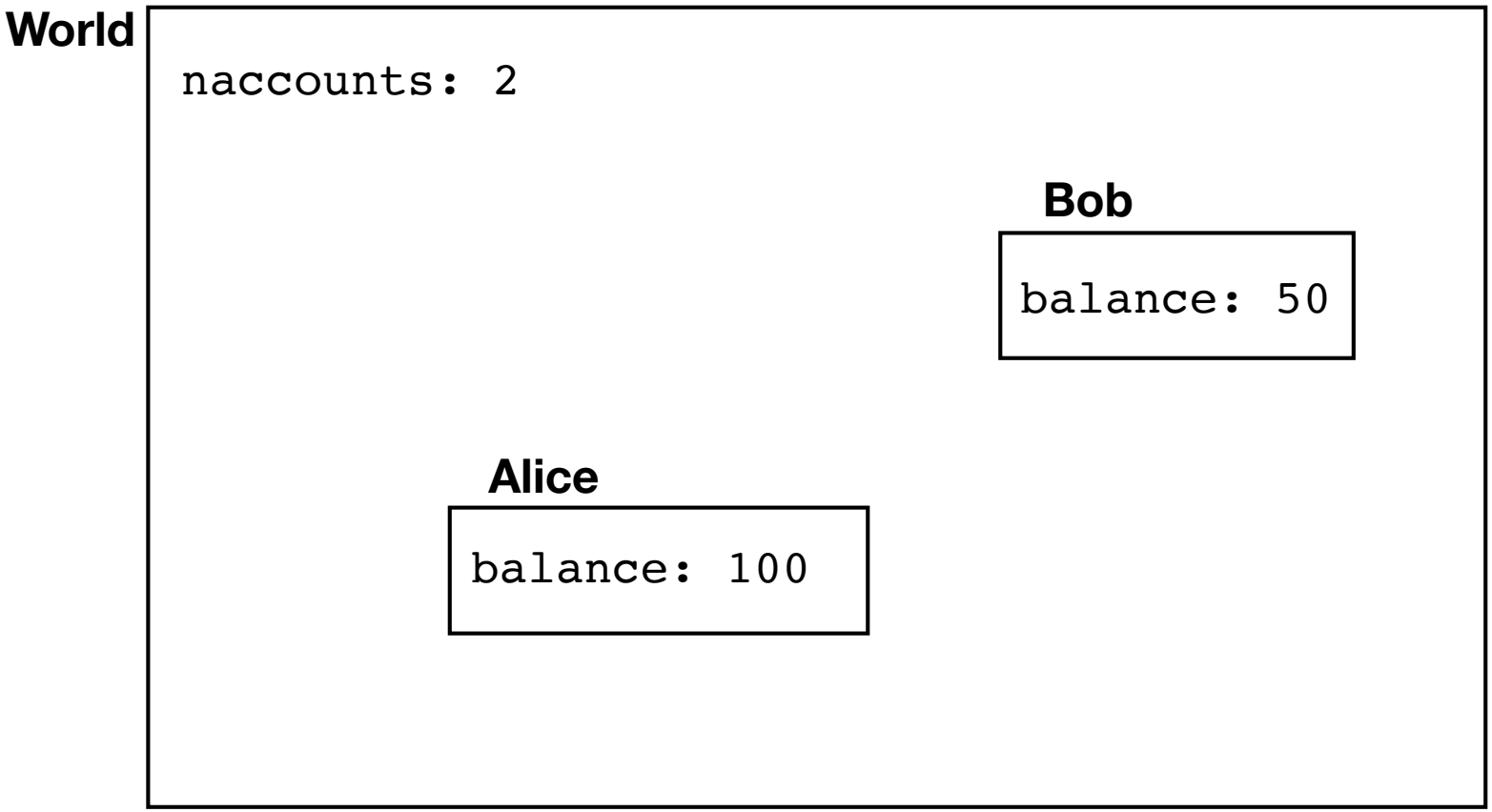
Modularity, Objects, and State

Overview

- How do you manage large systems?
- How do you model the physical world?
- You need some kind of strategy/conceptual model
- This section explores these matters
    - Objects, concurrency, streams

Objects

- Break a system into distinct parts and focus on their behavior (which changes over time)



- Objects carry state. The state changes.

Assignment

- set! - Changes the value of an existing variable

```
(define balance 100)      ; Creates a variable

(define (withdraw amt)
    (begin
       (set! balance (- balance amt))  ; Update the variable
       balance
     ))

(withdraw 10)
90
(withdraw 50)
40
```

- There is a side-effect. A state change.

begin

- (begin expr1 expr2 ... exprn)

```
(define balance 100)

(begin
    (display balance)
    (newline)
    (set! balance 75)
    (display balance)
    (newline)
    balance
)
```

- Allows for a sequence of steps. Only result from last expression is returned.

- Imperative programming. Step-by-step, changing values

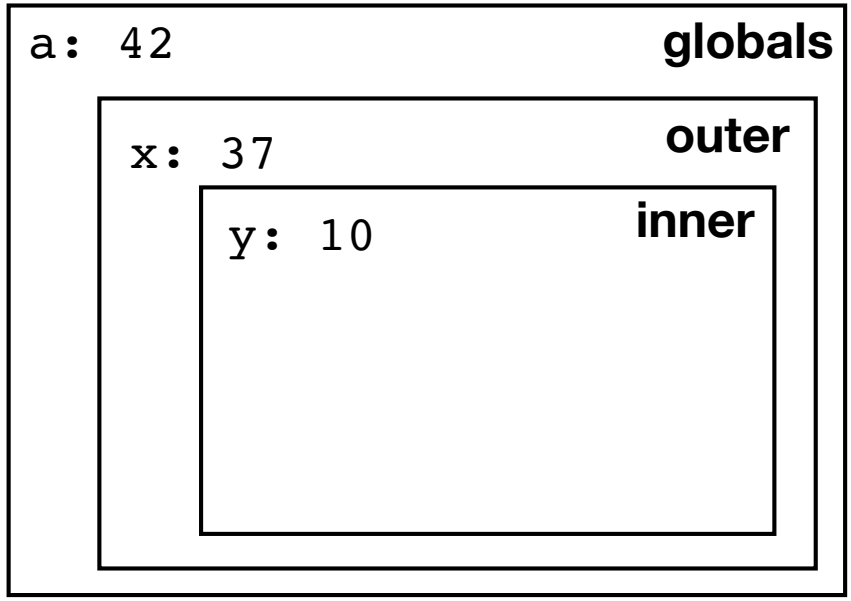
Digression

- We have not used assignment once so far!
- This is amazing. We're on pg. 217.
- Functional programming: Programming without assignments. No mutation.
- Alas... Assignment changes everything
- Especially the evaluation model

Environments

- Previously, environments are about name binding

```
(define a 42)

(define (outer x)
     (define (inner y)
          (+ a x y))
     (inner 10))

(outer 37)
```



- With assignment, environments become a place to store and update state (environments represent objects)

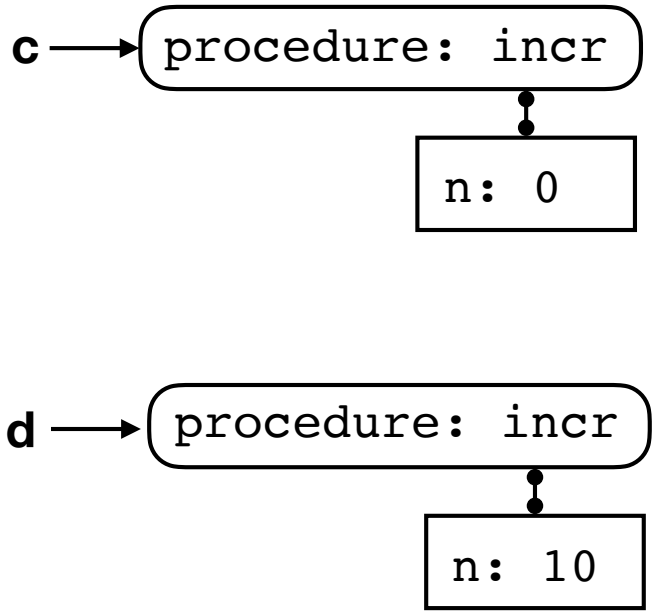
Environments as Objects

- Example: A counter

```
(define (make-counter n)
    (define (incr)
        (begin
            (set! n (+ n 1))
            n
        ))
    incr)
```

- Example:

```
(define c (make-counter 0))
(define d (make-counter 10))

(c)   -> 1
(c)   -> 2
(d)   -> 11
(d)   -> 12
```



Commentary

- You can do this in Python too (although a bit strange)

```
def make_counter(n):
    def incr():
        nonlocal n
        n += 1
        return n
    return incr
```

- Example:

```
>>> c = make_counter(0)
>>> d = make_counter(10)
>>> c()
1
>>> c()
2
>>> d()
11
>>>
```

Environments as Objects

```
(define (make-account balance)
    (define (withdraw amount)
        (begin (set! balance (- balance amount))
               balance)

    (define (deposit amount)
        (begin (set! balance (+ balance amount))
               balance)

    (define (dispatch m)
        (cond ((eq? m 'withdraw) withdraw)
              ((eq? m 'deposit) deposit)))
    dispatch)
```

- Example:

```
(define acc (make-account 100))
((acc 'withdraw) 30)
((acc 'deposit) 75)
```

Exercise

- Exercise 3.1, 3.2
- Reading 3.1.3, Exercise 3.8
- Group Reading: Section 3.2

Evaluation Model

- Assignment breaks the whole model of evaluation
- Recall: Substitution model (from math)

```
f(x, y) = 2*x*y - 3*x

f(4, 5) = 2*4*5 - 3*4
```

- It's a simple symbol replacement--argument names are replaced by argument values
- Alas, it doesn't work with mutation...

Evaluation Model

- Example:

```
(define (make-accum n)
   (lambda (x)
       (set! n (+ n x))
       n
   ))
```

```
(define a (make-accum 0))
(a 10)
10
(a 5)
15
(a 3)
18
```

- Substitution Model

```
(make-accum 0)
```

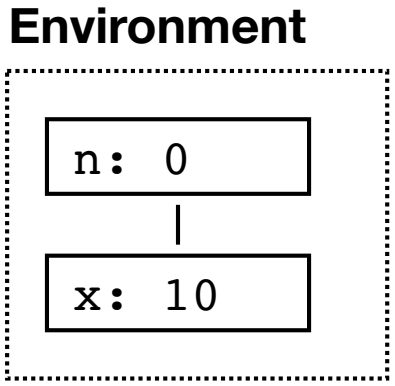
```
(lambda (x)
     (set! n (+ 0 x))
     0
)
```

This is wrong. The result is supposed to be the updated value of n. Instead, this is the initial value.

Environment Model

- All names represent environment get/set operations

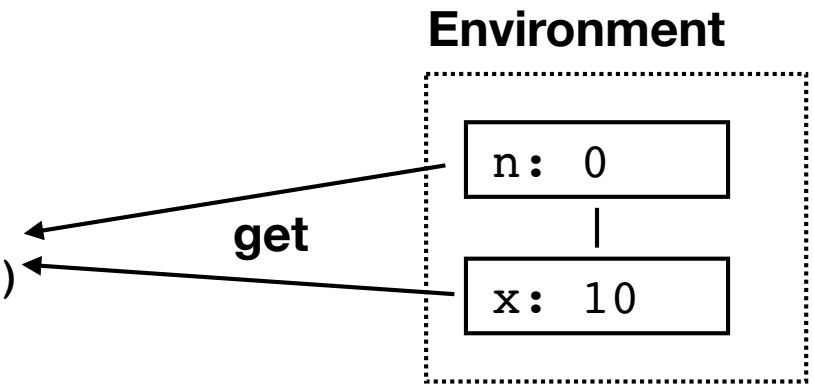
```
(define (make-accum n)
   (lambda (x)
       (set! n (+ n x))
       n
   ))

(define a (make-accum 0))

(a 10) -> (set! n (+ n x))
          n
```



Environment Model

- All names represent environment get/set operations

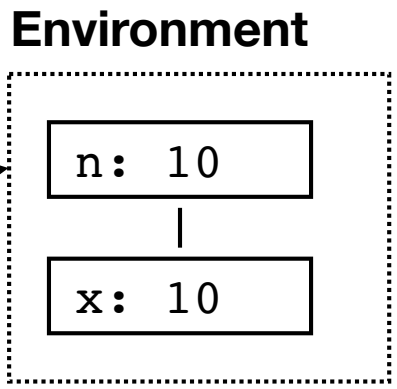
```
(define (make-accum n)
   (lambda (x)
       (set! n (+ n x))
       n
   ))

(define a (make-accum 0))

(a 10) -> (set! n (+ n x))
          n

          (set! n (+ 0 10))
          n
```



Environment Model

- All names represent environment get/set operations

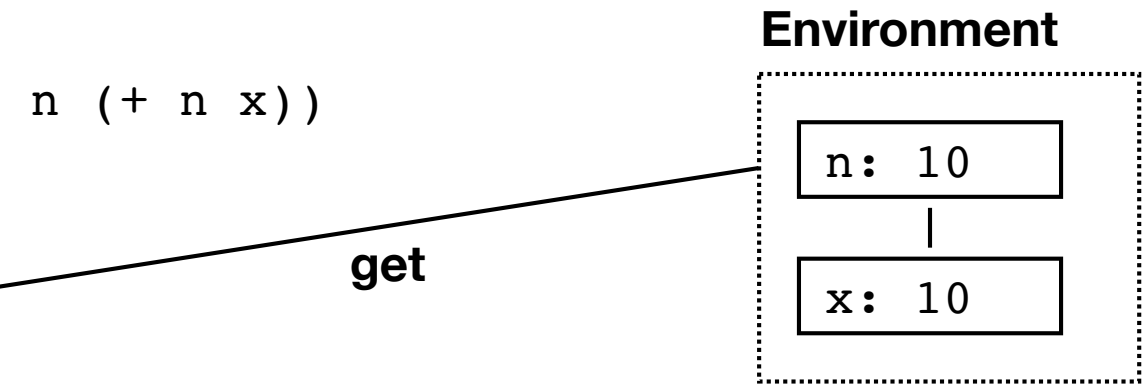
```
(define (make-accum n)
   (lambda (x)
       (set! n (+ n x))
       n
   ))

(define a (make-accum 0))

(a 10) -> (set! n (+ n x))
          n

          (set! n 10)
          n
```



Environment Model

- All names represent environment get/set operations

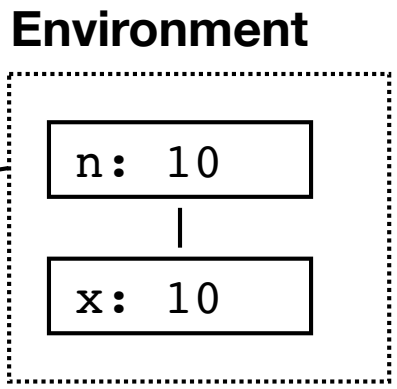
```
(define (make-accum n)
   (lambda (x)
       (set! n (+ n x))
       n
   ))

(define a (make-accum 0))

(a 10) -> (set! n (+ n x))
          n

          n
```



Environment Model

- All names represent environment get/set operations

```
(define (make-accum n)
   (lambda (x)
       (set! n (+ n x))
       n
   ))

(define a (make-accum 0))

(a 10) -> (set! n (+ n x))
          n

          10
```



Classes

- In OO languages, the main purpose of a class is to manage the environment

```
class Account:
    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        self.balance -= amount

    def deposit(self, amount):
        self.balance += amount
```

- In Python, the environment is a dict (__dict__ attribute)
- Methods operate on this environment.

Commentary

- Understanding the nesting/linking of environments is tricky
- Key ideas:
    - All procedures are attached to their definition environment (sometimes called a "closure")
    - Calling a procedure creates a new execution environment (sometimes called an "activation frame")
- Group Exercise: 3.9, 3.10, 3.11

Mutable Data

- You can also change the contents of a cons cell

```
(define c (cons 3 4))

(set-car! c 10)
(set-cdr! c 20)
```

- Allows for mutable lists, queues, trees, etc.

- Warning: In Racket, must use mcons, mcar, mcdr

```
(define c (mcons 3 4))

(mcar c)            ; -> 3
(mcdr c)            ; -> 4
(set-mcar! c 10)
(set-mcdr! c 20)
```

Exercise

- Reading 3.3 - 3.3.1
- Exercise 3.12, 3.14, 3.16
- Reading 3.3.2, Exercise 3.22, 3.23 (Queues)
- Making a table (p. 266-268). Read/Implement.

Extended Exercise

- Section 3.3.5 : Propagation of constraints
- Exercise 3.33

Mutability Issues

- Introducing mutability introduces new semantic problems
- All about time and sequencing
- Results now depend on the sequence of operations
- This is very different from pure functional programming. Pure functional programming is "timeless." It doesn't matter when or how you evaluate things because it's all substitution and it all eventually works itself out. Not true with assignment!

Argument Evaluation

- Evaluation order now matters in function arguments
- Arguments evaluated left-to-right? Right-to-left?
- Evaluation may result in "side-effects"

Assignment Sequencing

- Consider this bit of code:

```
(define (factorial n)
   (let ((product 1)
         (counter 1))
   (define (iter)
        (if (> count n)
            product
            (begin (set! product (* counter product))
                   (set! counter (+ counter 1))
                   (iter))))
    (iter)))
```

- What if the assignments change order?

```
(set! counter (+ counter 1))
(set! product (* counter product))
```

- Order matters (a source of programming errors)

Object "Sameness"

- Suppose you make two pairs

```
(define a (cons 2 3))
(define b (cons 2 3))
```

- Are they the same? Discuss.

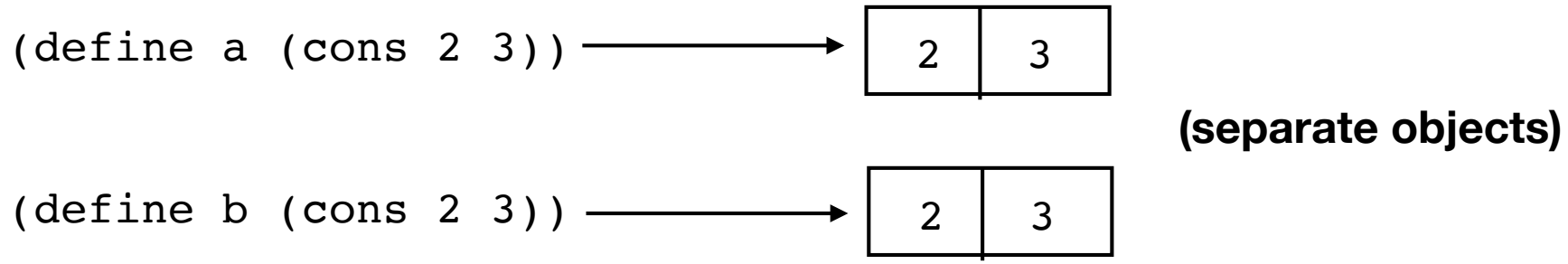
- But what about this?

```
(define a (cons 2 3))
(define b (cons 2 4))      ; Not the same

(set-cdr! b 3)             ; Is b now the same as a?
```

- Object value vs. object identity (e.g., "== " vs "is")

Referential Transparency

- If two expressions have the same value, can they be substituted for each other in any context?



- Mutability makes everything more complicated

```
(define a (cons 2 3))
(define b a)
(set-car! a 4)
(car b)          ; --> 4
```

Operation Sequencing

- Basic operations often involve a sequence of steps

```
(define (withdraw x)
    (set! balance (- balance x))
```

- Under the hood....

```
(define (withdraw x)
    (set! balance (- (get balance) x))
```

```
Step 1: (get balance)
Step 2: (- (get balance) x)
Step 3: (set! balance (- (get balance) x))
```

- There is time sequencing involved.

Concurrency

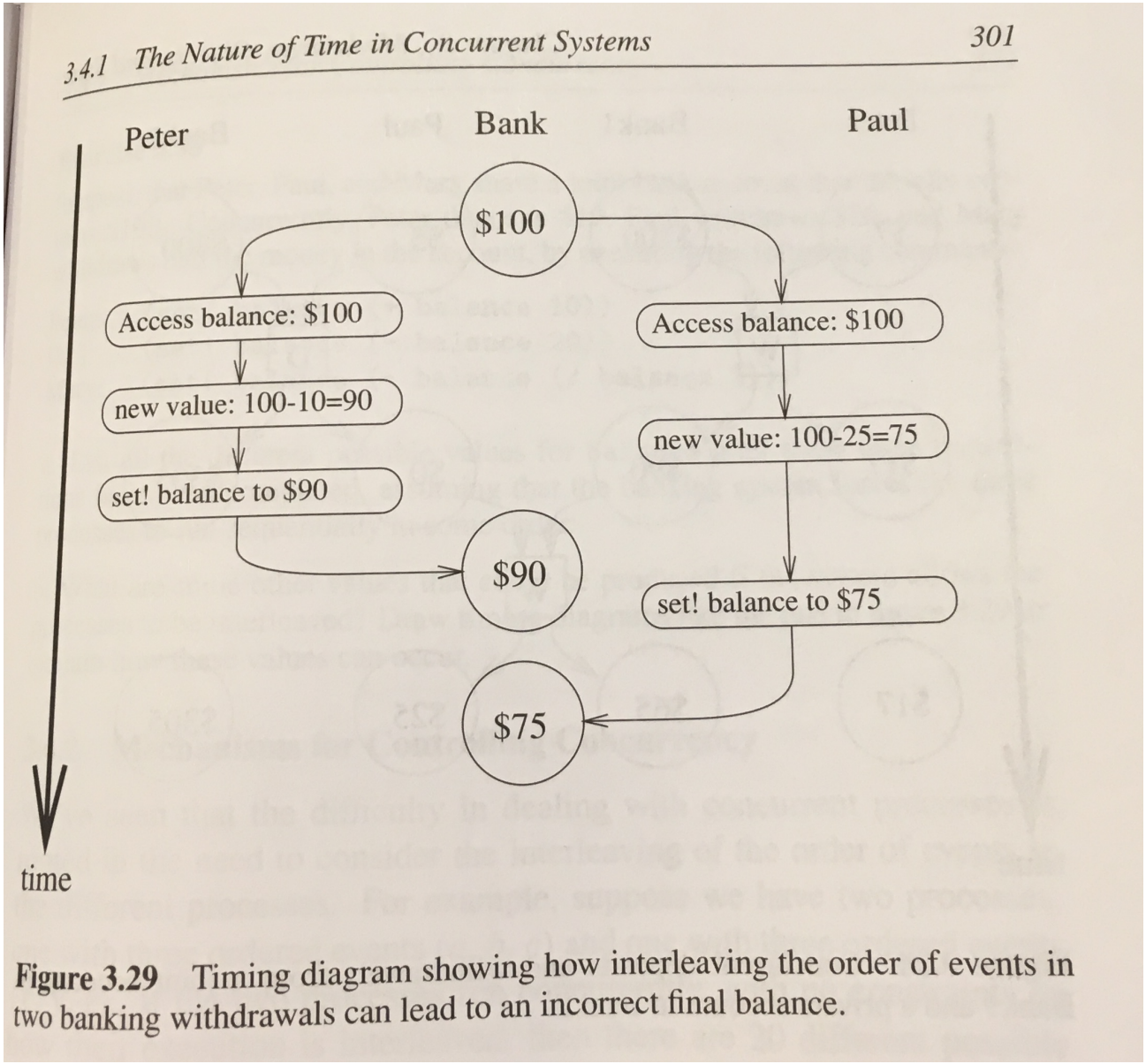


Figure 3.29 Timing diagram showing how interleaving the order of events in two banking withdrawals can lead to an incorrect final balance.

- Known as a race condition

Commentary

- Section 3.4 discusses concurrency in some detail
- Topics
    - Threads
    - Data races
    - Synchronization
    - Mutex locks
- Not going to discuss further. Same issues in Python.

Streams

- Break a system into a "workflow" and focus on data flow

- Real world example: Unix pipes

```
tail -f logfile | grep foo | awk '{print $1}'
```

- Another example: Python generators

```
def count_forever(n):
    while True:
        yield n
        n += 1
```

Digression

- Python Generators

```
def count_forever(n):
    while True:
        yield n
        n += 1
```

- What is special about it?

- It does nothing until a value is demanded

```
>>> c = count_forever(0)
>>> next(c)
0
>>> next(c)
1
>>>
```

- Delayed execution

Delayed Evaluation

- (delay expr)

```
(define p (delay (+ 3 4))
```

- Creates a "promise". Does NOT evaluate the expression
- (force promise) - Forces evaluation

```
(force p)      -> 7
```

- Sort of similar to a "Future" in other languages

Delay Implementation

- It's a syntactic/environment hack

```
(define p (delay (+ 3 4))

(define p (lambda () (+ 3 4)))      ; AKA "Thunk"
```

- Force, evaluates the procedure

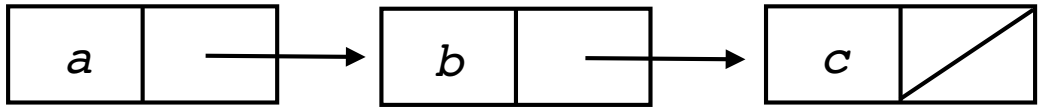
```
(define (force p) (p))
```

- Big thought: When do things happen? When does evaluation occur?

Stream Concept

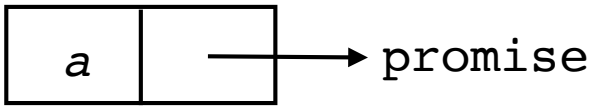
- A Scheme list (sequence)

```
(cons a (cons b (cons c nil))
```



- A Stream - Remainder of stream is "delayed"

```
(cons a (delay b))
```




```
(define (stream-car stream) (car stream))
(define (stream-cdr stream) (force (cdr stream)))
```

Streams Illustrated

```
(define (enumerate-interval start stop)
     (if (> start stop)
         '()
         (cons start
            (delay
               (enumerate-interval (+ start 1) stop)))))

(define (display-stream s)
     (if (null? s) nil
         (begin (display (stream-car s))
                (newline)
                (display-stream (stream-cdr s)))))

(display-stream (enumerate-interval 0 3))
0
1
2
3
```

stream-cons

- To simplify streams, there is a special form

```
(stream-cons first rest) ⟶ (cons first (delay rest))
```

- Puzzler: Why can't you make this a procedure?

```
(define (stream-cons first rest) (const first (delay rest)))
```

- Hint: Think about how procedure arguments are evaluated.

Stream Calculations

- Problem: Compute

$$\sum_{n=1}^{5} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2}$$

- "Streams" in Python:

```
nums = range(1, 6)
squares = (x*x for x in nums)
invsquares = (1/x for x in squares)
sum(invsquares)
```

- Streams in Scheme will be similar concept

Stream Calculations

- In Scheme

```
(define (enumerate-interval start stop)
    (if (> start stop) '()
        (stream-cons start
                     (enumerate-interval (+ start 1) stop))))

(define (stream-map proc s)
    (if (eq? s '()) '()
        (stream-cons (proc (stream-car s))
                     (stream-map proc (stream-cdr s)))))

(define nums (enumerate-interval 1 5))
(define squares (stream-map (lambda (x) (* x x)) nums))
(define invsquares (stream-map (lambda (x) (/ 1.0 x)) squares))
(sum invsquares)
```

- Again, a pipelining idea...

Exercise

- Reading 3.5.1
- Exercises: 3.50, 3.51, 3.52
- Reading 3.5.2
- Exercises: 3.53, 3.54

Chapter 4

Metalinguistic Abstraction

In a nutshell...

- What do you do if the complexity of your problem transcend the capabilities of your programming language?
- You make your own programming language
- Of course!
- Domain-specific languages, etc.

Eval

- Basically, this section is about making your own "eval"
- In Python:

```
>>> eval('3 + 4 * 5 + 10')
33
>>> exec('for i in range(5): print(i)')
0
1
2
3
4
>>>
```

- But there's a twist with Scheme...

Programs as Data

- Scheme programs are already scheme lists
- Remember quotation?

```
(define prog
   '(define (fact n) (if (= n 1) 1 (* n (fact (- n 1))))))
```

- Try it...

```
> prog
'(define (fact n) (if (= n 1) 1 (* n (fact (- n 1)))))
> (car prog)
'define
> (cadr prog)
'(fact n)
> (caddr prog)
'(if (= n 1) 1 (* n (fact (- n 1))))
>
```

Implications

- There are deep consequences of programs as data
- Can manipulate programs using same tools
- There is a uniformity to it
- Contrast: Having to use a lexer/parser to construct abstract syntax trees, manipulating syntax trees, etc.

Eval as a Process

- Eval is a computational process of evaluation and application

```
(+ (* 3 4) (* 6 7))
```

- You evaluate the subexpressions

```
(+ 12 42)
```

- You apply the operator (+)

```
54
```

Big Idea

- Programming languages are designed as a kind of dance involving "eval" and "apply"
- When are expressions evaluated?
- How are expressions evaluated?
- When are procedures applied?
- There are many design choices

Metacircular Evaluator

- We're going to build the "metacircular evaluator"
- Definition: An evaluator that's written in the same language as what's being evaluated.
- In a nutshell: A Lisp program that evaluates Lisp programs

Challenge

- We're going to build a Lisp interpreter as described in section 4.1 as a group. I'm going to guide you through the process with live-coding and discussion.

- Reading this section of the book is challenging--there are many moving parts. As written, it's very hard to make incremental progress.

- Will use some features of Racket to simplify. Also, may short-cut certain parts

- Stretch goal: implement "Lazy Scheme" (section 4.2).

Chapter 5

Computing with Register Machines

How does it all work?

- To this point, SICP has covered a lot about computation, abstraction, programming, etc.
- However, it still feels very magical and unsettling
- For example: implementing Lisp in Lisp (????)
- At some point, it has to work in physical reality
- Big question: How?

Major Topics

- Register machines
- Machine code/assembly language
- Garbage collection/memory management
- How to make a compiler

Our Goals

- Will focus primarily on the design of register machines and how computational processes map to register machines
- Will attempt to make a VERY simple machine emulator
- This is a large chapter. I am going to try an navigate through the highlights and the "big ideas"

Registers

- It is a place to hold a value (a box)

- There are just two operations (get/set)
- Can be built with actual hardware (memory)

Exercise

- Write Scheme code to model a register

```
(define (make-register)
    ...
)

(define r (make-register))
(r 'get)          ; Get a value
(r 'set! 42)      ; Set a value
```

- Hint: It's mutable state.

- Use "message passing" to make an object.

An Iterative Process

- Consider this procedure

```
(define (fact n)
    (define (fact-iter n result)
         (if (= n 1) result
             (fact-iter (- n 1) (* n result))))
    (fact-iter n 1)
)
```

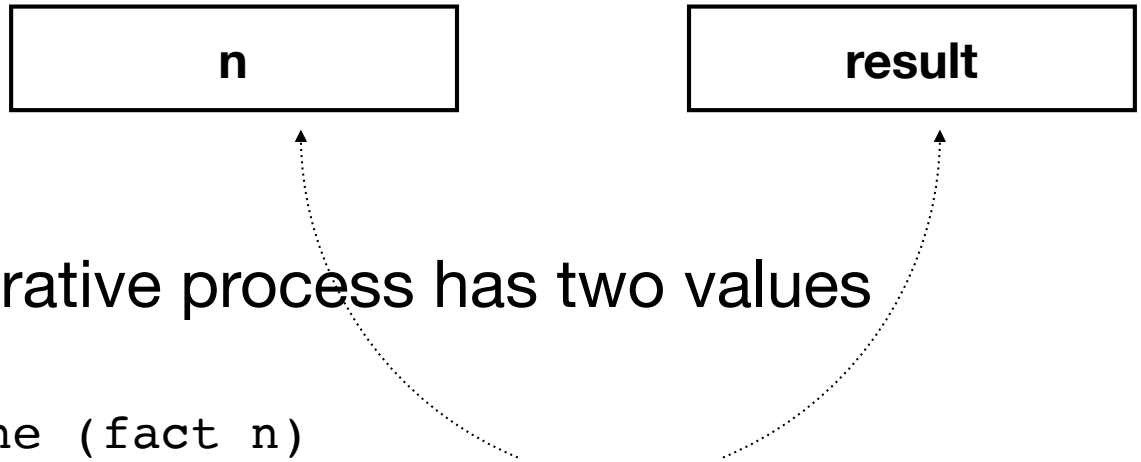
- It produces this sequence of operations

```
(fact-iter 5 1)
(fact-iter 4 5)
(fact-iter 3 20)
(fact-iter 2 60)
(fact-iter 1 120)
...
```

- Study it closely for a moment....

Mapping to Registers

- All values are held in registers

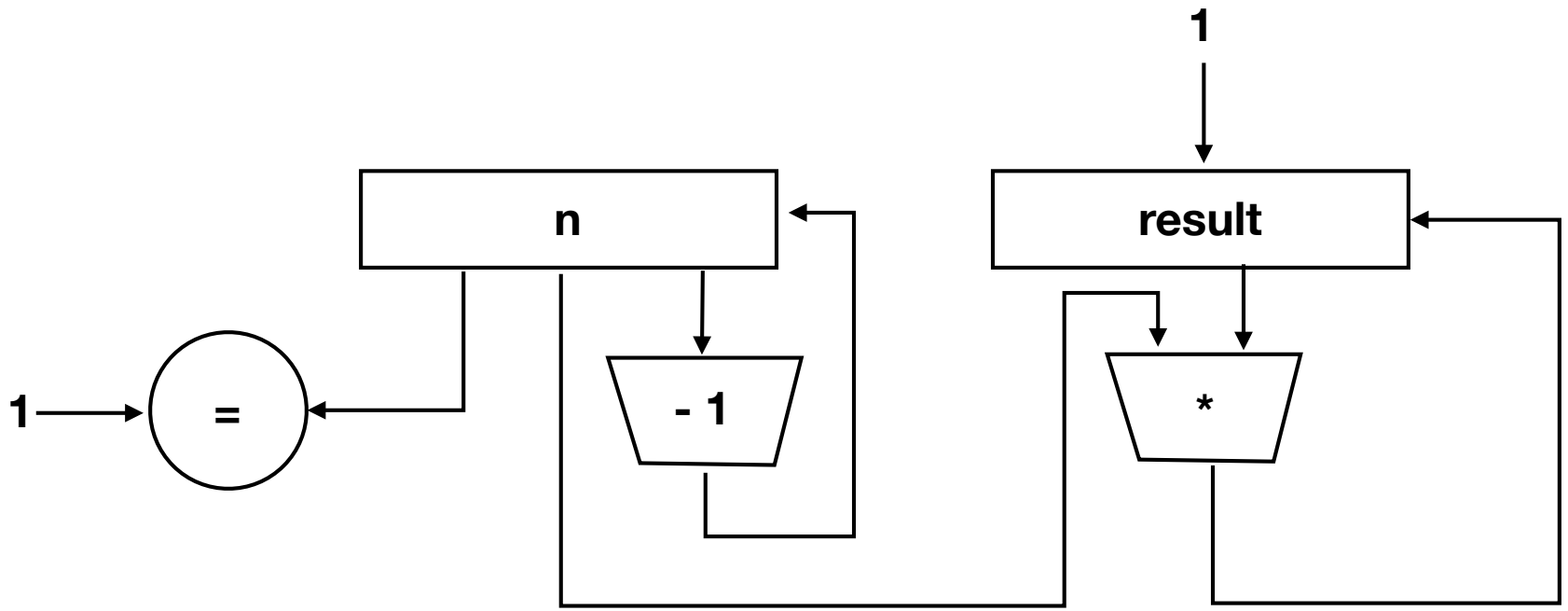


- Our iterative process has two values

```
(define (fact n)
    (define (fact-iter n result)
        (if (= n 1) result
            (fact-iter (- n 1) (* n result))))
    (fact-iter n 1)
)
```

Data Flow

- How does data flow in/out of registers?




```
(define (fact n)
    (define (fact-iter n result)
        (if (= n 1) result
            (fact-iter (- n 1) (* n result))))
    (fact-iter n 1)
)
```

Instructions

- An "instruction" performs some kind of operation and stores the result in a register

- Example: Decrement N

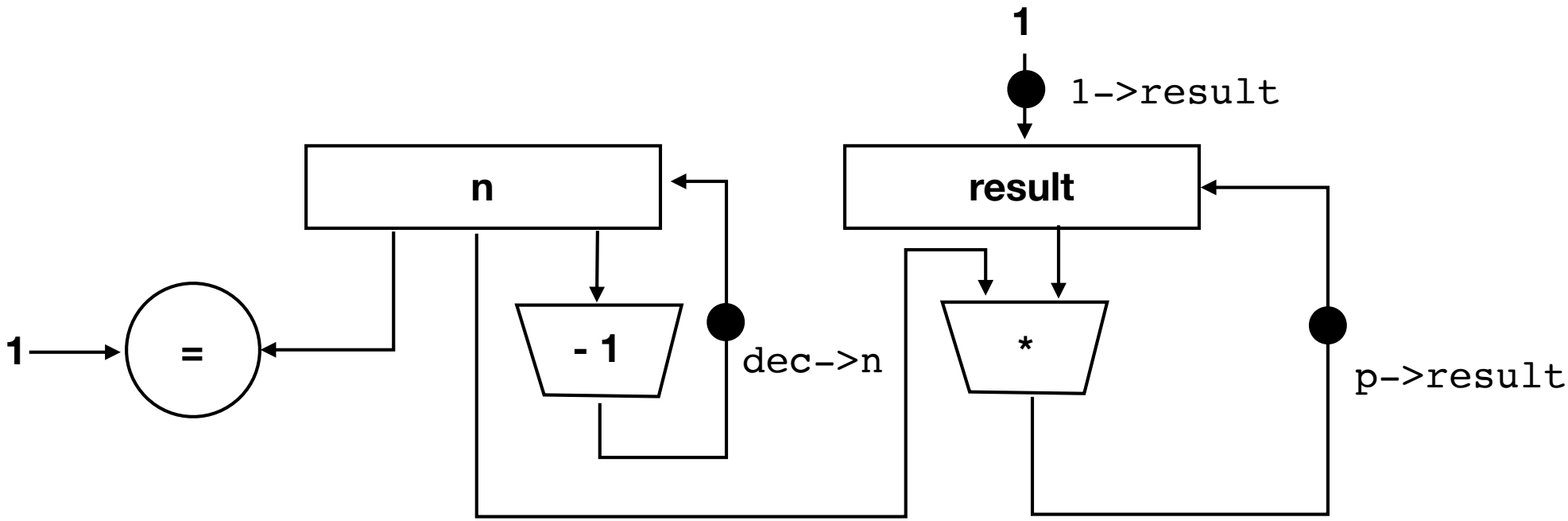
```
(define n (make-register))

(define (dec->n)
    (n 'set! (- (n 'get) 1)))
```

- Example use:

```
> (n 'set! 5)
> (dec->n)
> (n 'get)
4
>
```

Exercise

- Write instruction functions for the black dots



- All functions should have zero-arguments
- Executing the function performs the operation shown

Instruction Ordering

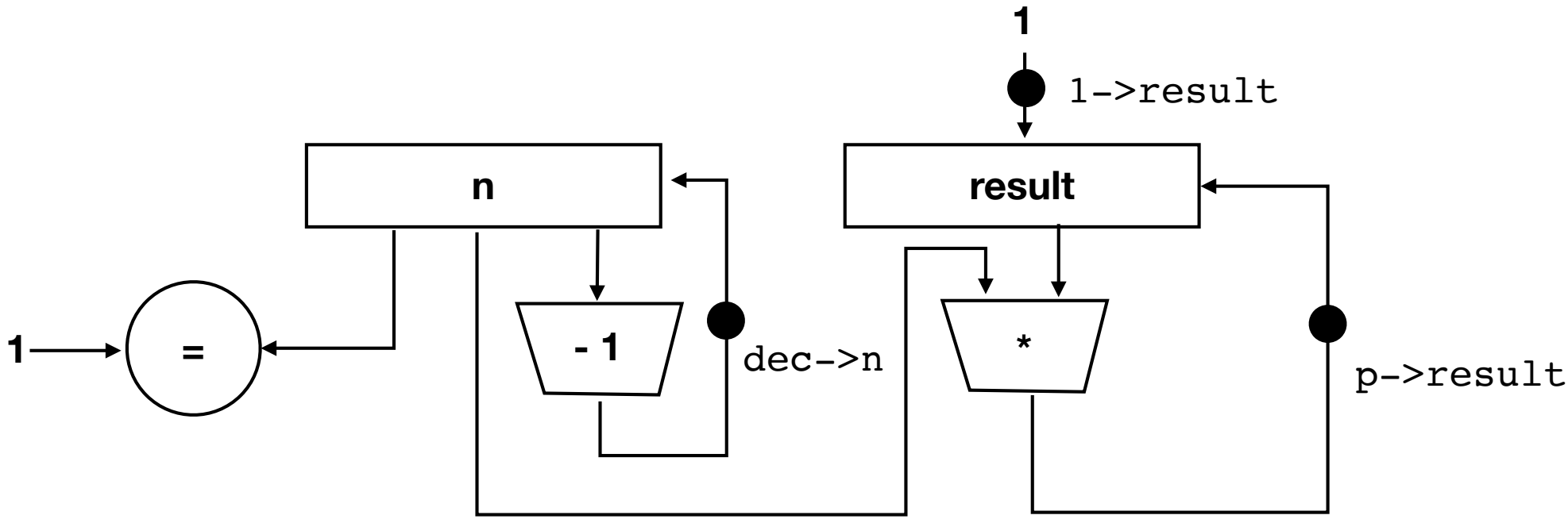
- To carry out the procedure, instructions must execute in a precise order

```
(define (fact n)
    (define (fact-iter n result)
         (if (= n 1) result
             (fact-iter (- n 1) (* n result))))
    (fact-iter n 1)
)
```

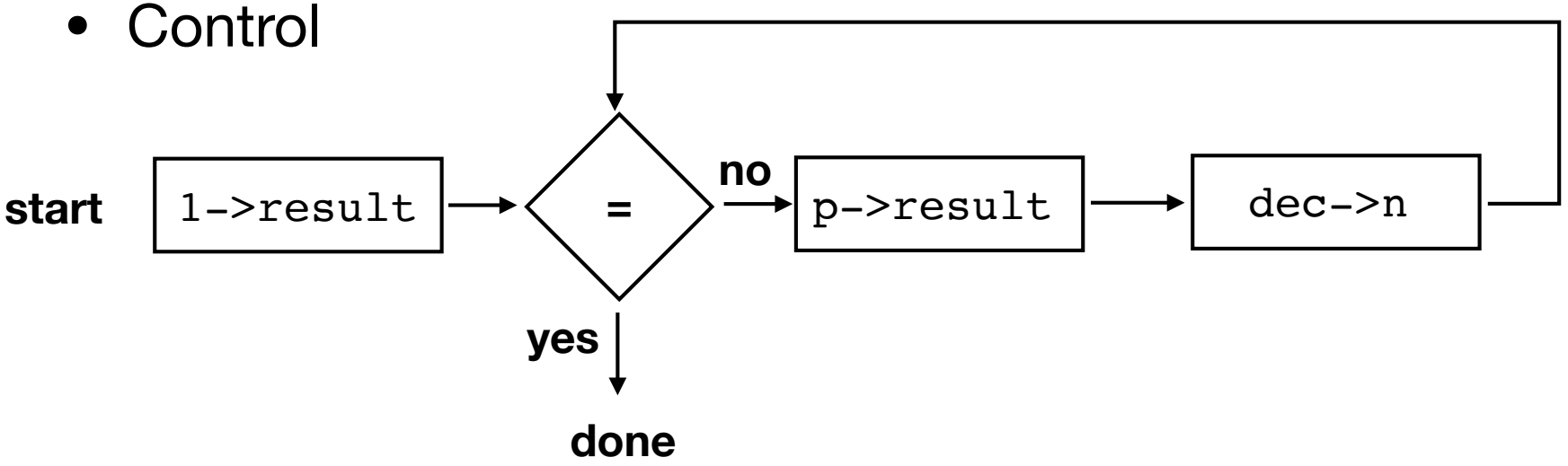
- Example: What is the ordering of computing the product and decrementing n above?

- Hint: It's not left-to-right!

Control Sequencing



- Control



Exercise

- Make some registers

```
(define n (make-register))
(define result (make-register))
```

- Make the core operations

```
(define (n-is-one?) (= (n 'get) 1))
(define (1->result) (result 'set! 1))
(define (dec->n) (n 'set! (- (n 'get) 1)))
(define (p->result) (result 'set! (* (n 'get) (result 'get))
```

- Run the calculation of fact 4 by hand

- You're only allowed the use of these functions

Solution

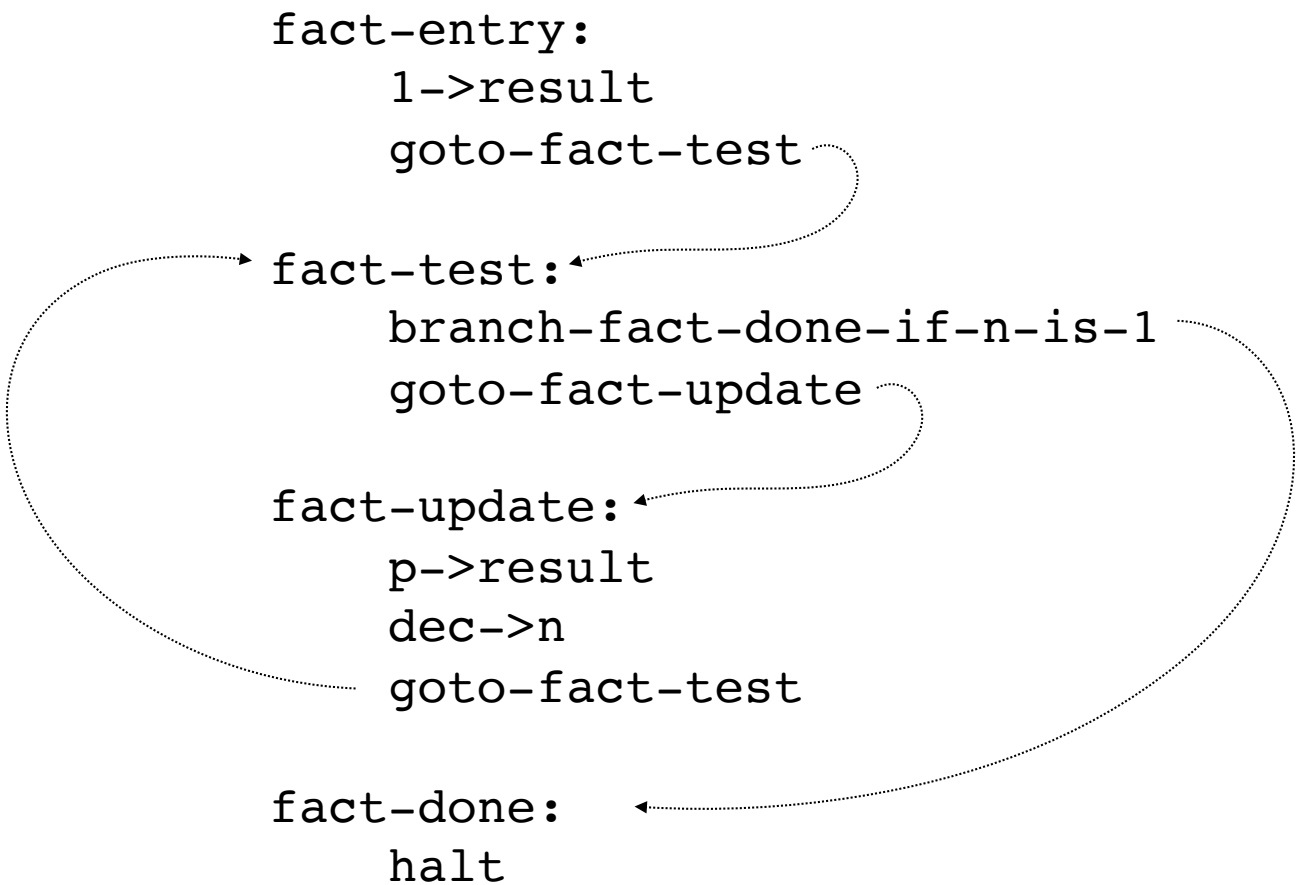
```
> (n 'set! 4)
> (1->result)
> (n-is-one?)
#f
> (p->result)
> (dec->n)
> (n-is-one?)
#f
> (p->result)
> (dec->n)
> (n-is-one?)
#f
> (p->result)
> (dec->n)
> (n-is-one?)
#t
> (result 'get)
24
>
```

Challenge

- How to turn the control into instructions?
- Can you automate the entire calculation?
- What is control flow?
- Basically, it's "gotos"

Expressing Control Flow

- Introduce blocks, labels, and branching



Program Counter

- Introduce a register "PC" that holds current instruction

```
(define pc (make-register))
```

- Define control-flow instructions to set the pc

```
(define (goto-fact-test)
    (pc 'set! fact-test))

(define (goto-fact-update)
    (pc 'set! fact-update)

(define (branch-fact-done-if-n-is-1)
    (if (= (n 'get) 1)
        (pc 'set! fact-done)
        '()
    )
)
```

Defining Blocks

- Define blocks as lists of instructions

```
(define fact-entry (list
    1->result
    goto-fact-test
))

(define fact-test (list
    branch-fact-done-if-n-is-1
    goto-fact-update
))

(define fact-update (list
    p->result
    dec->n
    goto-fact-test
))

(define fact-done (list))
```

Execution

- Write a procedure that executes instructions as long as the PC points to something

```
(define (execute)
   (let ((instructions (pc 'get)))
       (cond ((null? instructions) 'done)
             (else
                (pc 'set! (cdr instructions))
                ((car instructions))
                (execute)
             )
        )
    )
)
```

- To run

```
(n 'set! 4)
(pc 'set! fact-entry)
(execute)
(result 'get)
```

Challenge: Recursion

- How do you encode a recursive procedure into registers?

```
(define (fact n)
    (if (= n 1)
        1
        (* n (fact (- n 1)))))
```

- Challenge: Concept of a subroutine
- Challenge: Reentrancy
- Challenge: Keeping track of intermediate results

Generalizing

- Can you generalize into an abstract machine language

- Four basic instructions

```
(assign regname value)   ; Assign to a register
(test value)             ; Test if value is true
(branch label)           ; Branch if last test true
(goto label)             ; Unconditional branch
```

- Values can come from a few different places

```
(reg name)               ; Register
(op name args)           ; Result of an operation
(const val)              ; Constant
(label name)             ; Instructions mapped to label
```

- Example instructions

```
(assign result (const 1))
(assign n (op -) (reg n) (const 1))
(test (op =) (reg n) (const 1))
```

Example Code

- Iterative factorial

```
(define (fact n)
    (define (fact-iter n result)
         (if (= n 1) result
             (fact-iter (- n 1) (* n result))))
    (fact-iter n 1)
)
```

- "Machine code" - Note: Not Scheme/Lisp

```
(
fact-entry
  (assign result (const 1))
test-n
  (test (op = (reg n) (const 1)))
  (branch (label fact-done))
  (assign result (op * (reg n) (reg result)))
  (assign n (op - (reg n) (const 1)))
  (goto (label test-n))
fact-done
)
```

Challenge

- Create a "machine" that can run this machine code

```
(
fact-entry
  (assign result (const 1))
test-n
  (test (op = (reg n) (const 1)))
  (branch (label fact-done))
  (assign result (op * (reg n) (reg result)))
  (assign n (op - (reg n) (const 1)))
  (goto (label test-n))
fact-done
)
```

- Will work as a group...

Further Topics

- If we make it any further...
- Memory management/garbage collection
- Implementing a lisp interpreter in "machine code"
- Implementing a compiler