

# Programming Languages and Techniques (CIS120)

## Lecture 1

Welcome  
Introduction to Program Design

# Introductions

- Steve Zdancewic\*
  - Levine Hall 511
  - [stevez@cis.upenn.edu](mailto:stevez@cis.upenn.edu)
  - <http://www.cis.upenn.edu/~stevez/>
  - Office hours: Mondays 3:30 – 5:00pm  
(& by appointment)
- Swapneel Sheth
  - Levine Hall 268
  - [swapneel@cis.upenn.edu](mailto:swapneel@cis.upenn.edu)
  - <http://www.cis.upenn.edu/~swapneel>
  - Office hours: Tues. & Weds. 4:00 – 5:00pm  
(& by appointment)



\*Pronounced phonetically as: “zuh dans wick”. I won’t get upset if you mispronounce my name (really!). I will answer to anything remotely close, or, you can call me just Professor, or Professor Z. Whatever you feel comfortable with.

# What is CIS 120?

- CIS 120 is a course in **program design**
- Practical skills:
  - ability to write larger (~1000 lines) programs
  - increased independence ("working without a recipe")
  - test-driven development, principled debugging
- Conceptual foundations:
  - common data structures and algorithms
  - several different programming idioms
  - focus on modularity and compositionality
  - derived from first principles throughout
- It will be fun!



# Prerequisites

- We assume you can already write small (10- to 100-line) programs in some imperative or OO language
  - Java experience is *strongly recommended*
  - CIS 110 or AP CS is typical
  - You should be familiar with using a compiler, editing code, and running programs you have created
- CIS 110 is an alternative to this course
  - If you have doubts, come talk to me or one of the TAs to figure out the right course for you

# CIS 120 Tools

- OCaml
  - Industrial-strength, statically-typed *functional* programming language
  - Lightweight, approachable setting for learning about program design
- Java
  - Industrial-strength, statically-typed *object-oriented* language
  - Many tools/libraries/resources available
- Eclipse
  - Widely used IDE



# Codio

- Codio codio.com
  - web-based development environment
  - see Piazza / class mailing list for setup info
  - remote access for on-line TA help
- Under the hood:
  - linux virtual machine (Ubuntu)
  - pre-configured per project with everything you need
- First time using it this semester!
  - may be snags
  - we are working with Codio to improve your experience



# Why two languages??

- Clean pedagogical progression
- Levels disparate backgrounds
- Practice in learning new tools
- Different perspectives on programming

"[The OCaml part of the class] was very essential to getting fundamental ideas of comp sci across. Without the second language it is easy to fall into routine and syntax lock where you don't really understand the bigger picture."

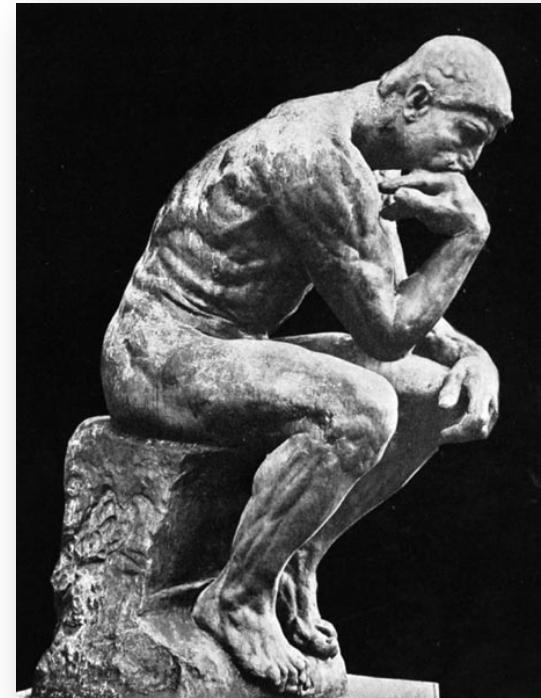
---Anonymous CIS 120 Student

"[OCaml] made me better understand features of Java that seemed innate to programming, which were merely abstractions and assumptions that Java made. It made me a better Java programmer."

--- Anonymous CIS 120 Student

# Philosophy

- Introductory computer science
  - Start with basic skills of “algorithmic thinking” (AP/110)
  - Develop systematic design and analysis skills in the context of larger and more challenging problems (120)
  - Practice with industrial-strength tools and design processes (120, 121, and beyond)
- Role of CIS120 and *program design*
  - Start with foundations of programming using the elegant design and precise semantics of the OCaml language
  - Transition (back) to Java *after* setting up the context needed to understand why Java and OO programming are useful tools
  - Give a taste of the breadth and depth of CS



# Administrivia

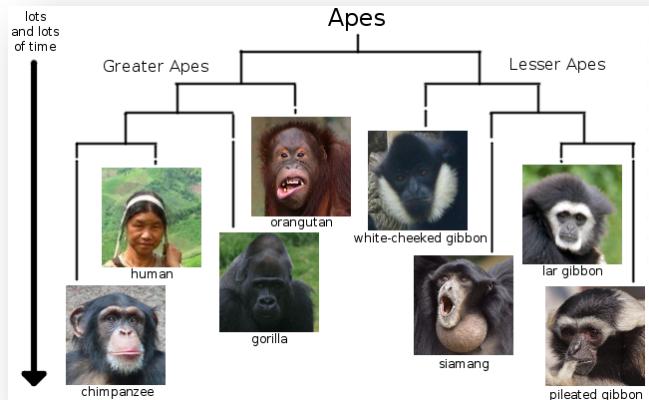
<http://www.seas.upenn.edu/~cis120/>

# Course Grade Breakdown

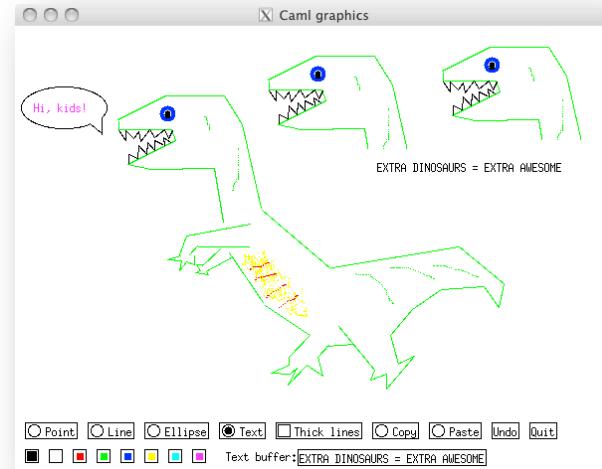
- Lectures
  - Presentation of ideas and concepts, interactive demos
  - Grade based on participation using “clickers”
  - Lecture notes & screencasts available on course website.
- Recitations / Labs (6% of final grade)
  - Practice and discussion in small group setting
  - Grade based on participation
- Homework (54% of final grade)
  - Practice, experience with tools
  - Exposure to broad ideas of computer science
  - Grade based on automated tests + style
  - **First assignment due on September 12th**
- Exams (40% of final grade)
  - **In class exams**, pencil and paper
  - Do you understand the terminology? Can you reason about programs? Can you synthesize solutions?

Warning: This is a  
*challenging* and  
*time consuming*  
(and rewarding :-) course!

# Some of the homework assignments...



Computing with DNA



Build a GUI Framework

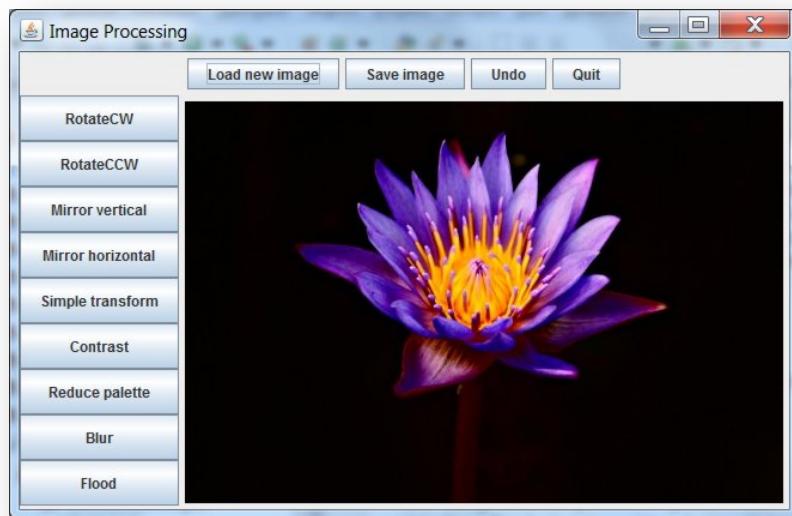
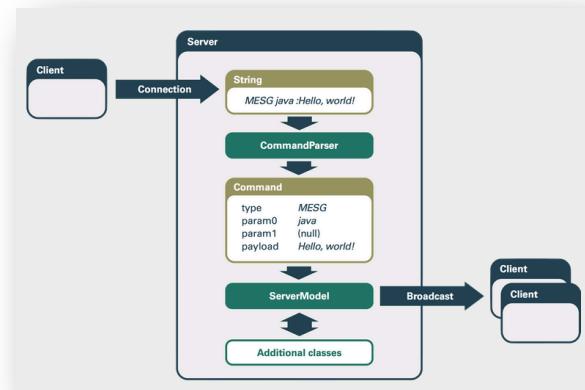


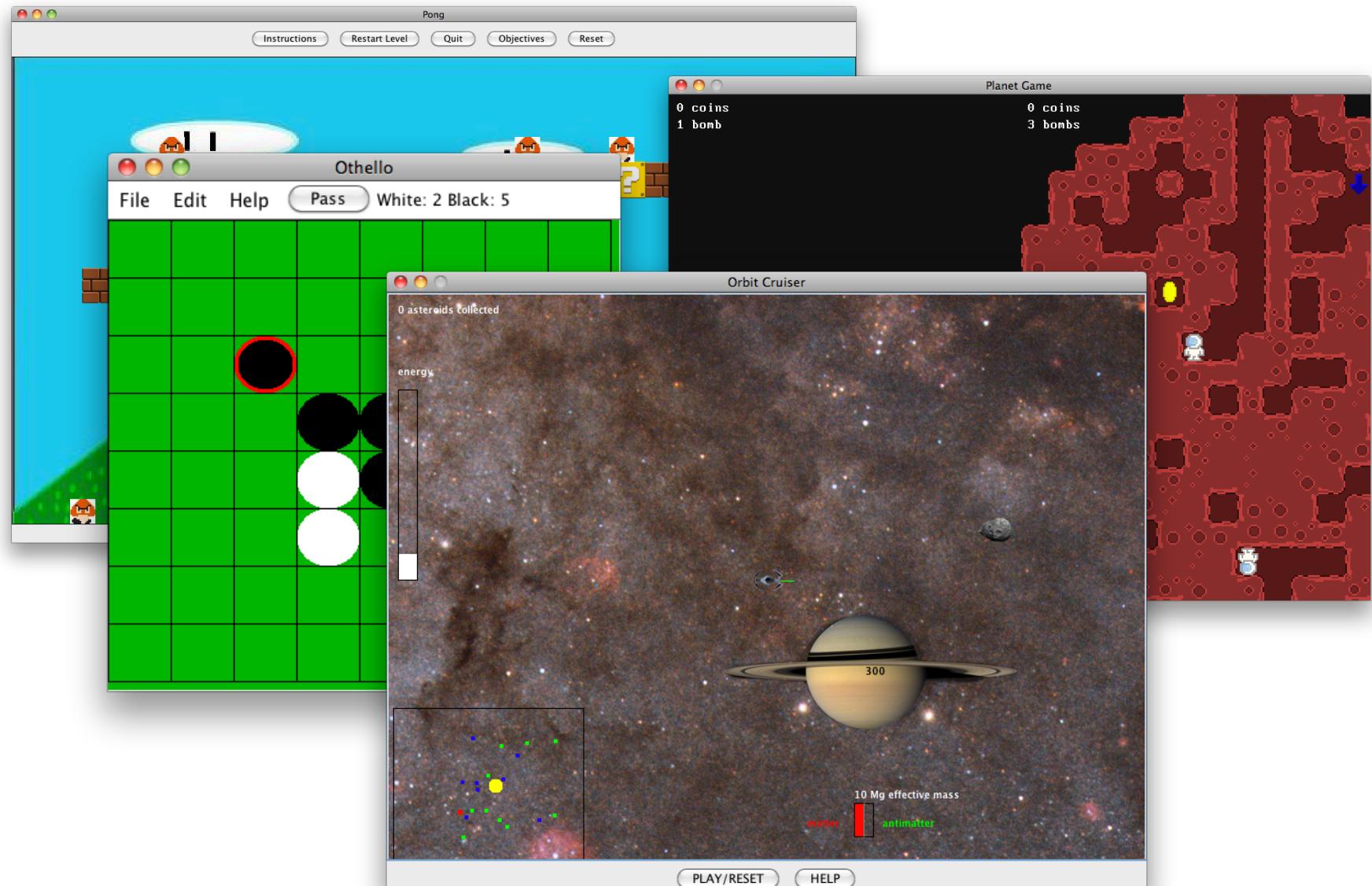
Image Processing

CIS120



Chat Client/Server

# Final project: Design a Game



# Lecture / Lab Balancing

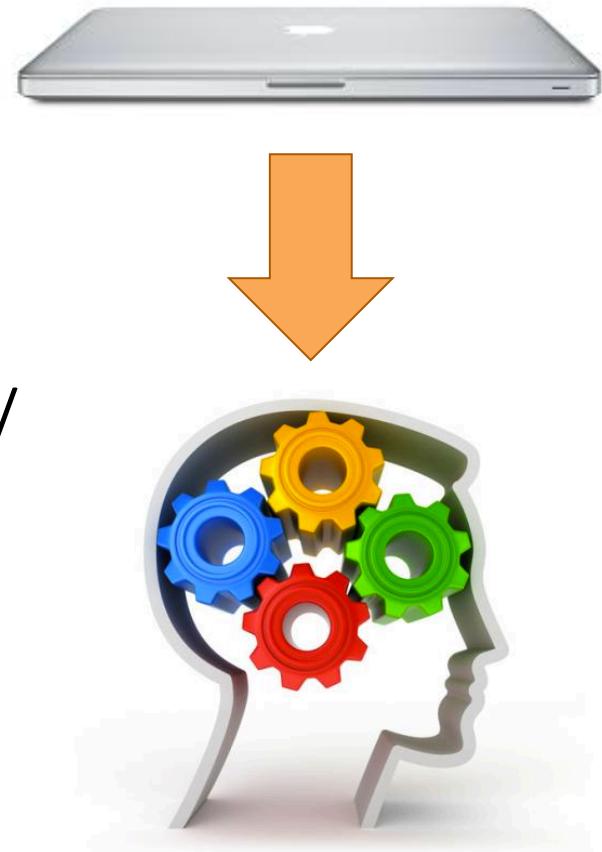
- The lecture material in the sections 001 and 002 will be identical
  - In fact, even the lecturer will be identical some of the time
  - 11AM lecture is overcrowded
  - NOON lecture is too empty
  - It would be good if some people can switch
- Lab sections are not balanced
  - Overcrowded: 202, 208, 212, or 214
  - Plenty of room: 205, 206, 218
- If you have the flexibility to switch, please fill out the change of recitation form

# Recitations / Lab Sections

- Recitations start next week
  - Bring your laptops
  - Try Codio before the first meeting
- Goals of first meeting:
  - Meet your TAs and classmates
  - Practice with OCaml before your first homework is due
- Office hours times on the web site calendar  
(under “Help” tab)

# No Devices

- Laptops *closed*... minds *open*
  - Although this is a computer science class, the use of electronic devices – laptops, cellphones, Kindles, iPads, etc., during lecture is *prohibited*.
- Why?
  - Laptop users tend to surf/chat/e-mail/game/text/tweet/etc.
  - They also distract those around them
  - Better to take notes *by hand*
  - You will get plenty of time in front of your computers while working on the course projects :-)



# Piazza

- We will use Piazza for most communications in this course
  - from us to you
  - from you to us
  - from you to each other
- If you are registered for the course, you will be signed up automatically
  - Look for a welcome email by tomorrow
- If not, please sign up at [piazza.com](https://piazza.com)

# Academic Integrity

*Penn's code of academic integrity:*

[http://www.upenn.edu/academicintegrity/ai\\_codeofacademicintegrity.html](http://www.upenn.edu/academicintegrity/ai_codeofacademicintegrity.html)

- Submitted homework must be *your individual work*
- Not OK:
  - Copying or otherwise looking at someone else's code
  - Sharing your code in any way  
(copy-paste, github, paper and pencil, ...)
  - Using code from a previous semester
- OK (and encouraged!):
  - Discussions of concepts
  - Discussion of debugging strategies
  - Verbally sharing experience

# Enforcement

- Course staff *will* check for copying
  - We use plagiarism detection tools on your code

*Violations will be treated seriously!*

- *Questions? See the course FAQ. If in doubt, ask.*

*Penn's code of academic integrity:*

[http://www.upenn.edu/academicintegrity/ai\\_codeofacademicintegrity.html](http://www.upenn.edu/academicintegrity/ai_codeofacademicintegrity.html)

# Program Design

# Fundamental Design Process

*Design* is the process of translating informal specifications (“word problems”) into running code.

1. *Understand the problem*

What are the relevant concepts and how do they relate?

2. *Formalize the interface*

How should the program interact with its environment?

3. *Write test cases*

How does the program behave on typical inputs?

On unusual ones? On erroneous ones?

4. *Implement the required behavior*

Often by decomposing the problem into simpler ones  
and applying the same recipe to each

5. Revise / Refactor / Edit

## A design problem

Imagine that you own a movie theater. The more you charge, the fewer people can afford tickets. In a recent experiment, you determined a relationship between the price of a ticket and average attendance. At a price of \$5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime (\$.10) increases attendance by 15.

However, increased attendance also comes at increased cost; each attendee costs four cents (\$0.04). Every performance also has a base cost of \$180.

At what price do you make the highest profit?

# Step 1: Understand the problem

- What are the relevant concepts?
  - *(ticket) price*
  - *attendees*
  - *revenue*
  - *cost*
  - *profit*
- What are the relationships among them?
  - profit = revenue – cost
  - revenue = price \* attendees
  - cost = \$180 + attendees \* \$0.04
  - attendees = *some function of the ticket price*
- Goal is to determine profit, given the ticket price

So profit, revenue, and cost also depend on price.

# Step 2: Formalize the Interface

*Idea: we'll represent money in cents, using integers\**

comment documents  
the design decision

type annotations  
declare the input  
and output types\*\*

```
(* Money is represented in cents. *)
let profit (price : int) : int = ...
```

\* Floating point is generally a *bad* choice for representing money: bankers use different rounding conventions than the IEEE floating point standard, and floating point arithmetic isn't as exact as you might like. Try calculating  $0.1 + 0.1 + 0.1$  sometime in your favorite programming language...

\*\*OCaml will let you omit these type annotations, but including them is *mandatory* for CIS120. Using type annotations is good documentation; they also improve the error messages you get from the compiler. When you get a type error message from the compiler, the first thing you should do is check that your type annotations are there and that they are what you expect.

## Step 3: Write test cases

- By looking at the design problem, we can calculate specific test cases

```
let profit_500 : int =
    let price      = 500 in
    let attendees = 120 in
    let revenue   = price * attendees in
    let cost       = 18000 + 4 * attendees in
revenue - cost
```

# Writing the Test Cases in OCaml

- Record the test cases as assertions in the program:
  - the *command* `run_test` executes a test

a *test* is just a function that takes no input and returns true if the test succeeds

```
let test : bool =  
  (profit 500) = profit_500  
  
;; run_test "profit at $5.00" test
```

the string in quotes identifies  
the test in printed output  
(if it fails)

note the use of double semicolons  
before commands

## Step 4: Implement the Behavior

**profit** is easy to define:

```
let attendees (price : int) = ...  
  
let profit (price : int) =  
    let revenue = price * (attendees price) in  
    let cost = 18000 + 4 * (attendees price) in  
    revenue - cost
```

# Apply the Design Pattern Recursively

`attendees` requires a bit of thought:

```
let attendees (price : int) : int =  
    failwith "unimplemented"
```

“stub out”  
unimplemented  
functions

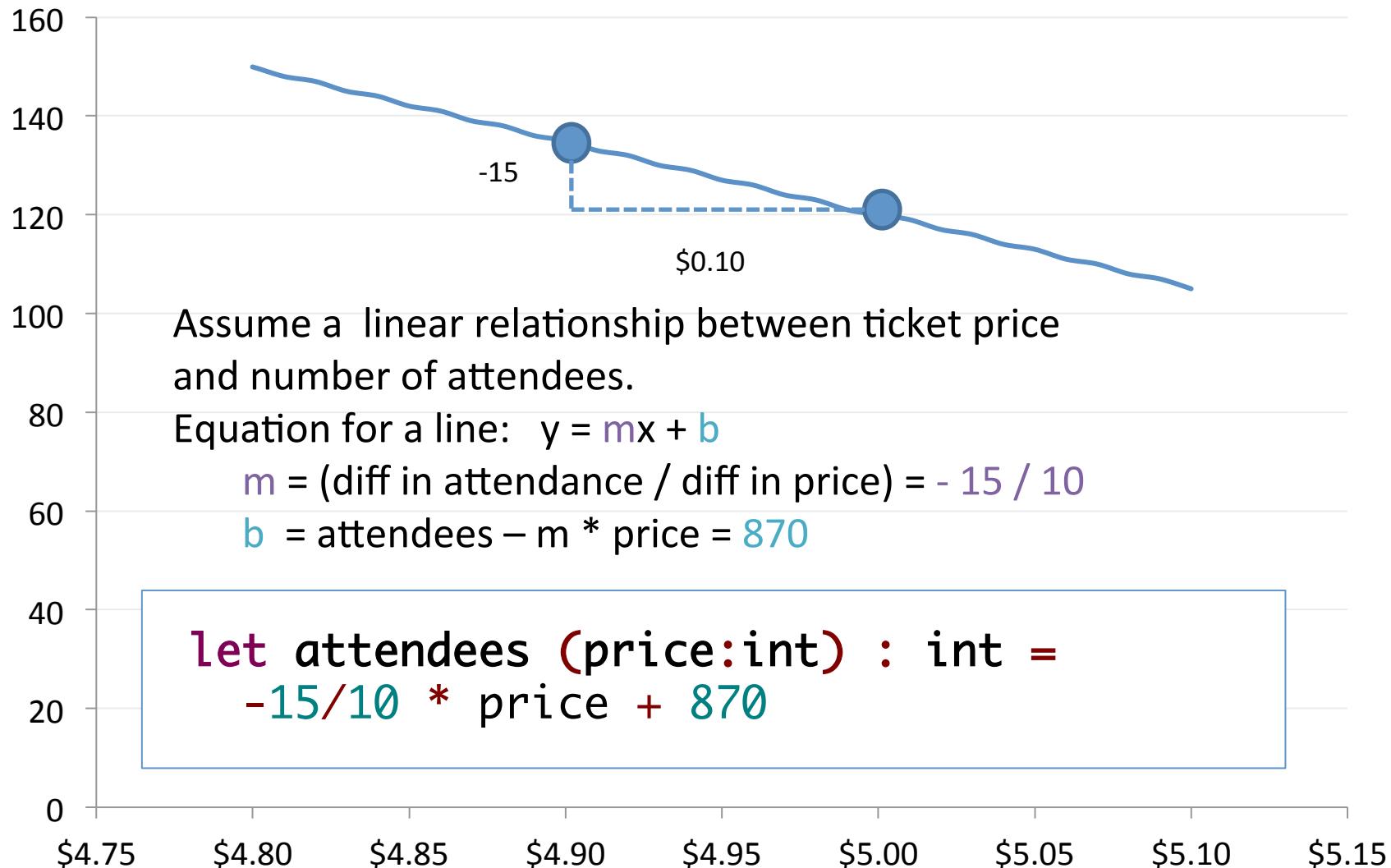
```
let test () : bool =  
    (attendees 500) = 120  
;; run_test "attendees at $5.00" test
```

```
let test () : bool =  
    (attendees 490) = 135  
;; run_test "attendees at $4.90" test
```

generate the tests  
from the problem  
statement *first*.

\*Note that the definition of `attendees` must go *before* the definition of `profit` because `profit` uses the `attendees` function.

# Attendees vs. Ticket Price



# Run!

# Run the program!

- One of our test cases for attendees failed...
- Debugging reveals that integer division is tricky\*
- Here is the fixed version:

```
let attendees (price:int) :int =  
    (-15 * price) / 10 + 870
```

\*Using integer arithmetic,  $-15 / 10$  evaluates to  $-1$ , since  $-1.5$  rounds to  $-1$ . Multiplying  $-15 * \text{price}$  before dividing by  $10$  increases the precision because rounding errors don't creep in.

# Using Tests

Modern approaches to software engineering advocate *test-driven development*, where tests are written very early in the programming process and used to drive the rest of the process.

We are big believers in this philosophy, and we'll be using it throughout the course.

In the homework template, we may provide one or more tests for each of the problems. They will often not be sufficient. You should *start* each problem by making up *more* tests.

# How *not* to Solve this Problem

```
let profit price =  
    price * (-15 * price / 10 + 870) -  
    (18000 + 4 * (-15 * price / 10 + 870))
```

This program is bad because it

- hides the structure and abstractions of the problem
- duplicates code that could be shared
- doesn't document the interface via types and comments

*Note that this program still passes all the tests!*

# Summary

- *To read:* Chapter 1 of the lecture notes and course syllabus. Both available on the course website
- *To do:* Sign up for Codio and try to log in.
  - TAs will hold office hours this week to help.
  - You can also use Piazza for discussions.
- *To do:* start HW01
  - due 9/12

# Programming Languages and Techniques (CIS120)

## Lecture 2 Value-Oriented Programming

# If you are joining us today...

- Read the course syllabus/lecture notes on the website
  - [www.cis.upenn.edu/~cis120](http://www.cis.upenn.edu/~cis120)
- Sign yourself up for Piazza
  - [piazza.com/upenn/fall2017/cis120](http://piazza.com/upenn/fall2017/cis120)
- Try out Codio
  - [www.cis.upenn.edu/~cis120/current/codio.shtml](http://www.cis.upenn.edu/~cis120/current/codio.shtml)
- *No laptops, tablets, smart phones, etc., during lecture*

# Announcements

- Please *read*:
  - Chapter 2 of the course notes
  - OCaml style guide on the course website  
([http://www.seas.upenn.edu/~cis120/current/programming\\_style.shtml](http://www.seas.upenn.edu/~cis120/current/programming_style.shtml))
- Homework 1: OCaml Finger Exercises
  - Available from Schedule page on course website
  - Practice using OCaml to write simple programs
  - Start with first 4 problems (lists next week!)
  - Due: *Tuesday, Sept. 12, at 11:59:59pm* (midnight)
  - Start early!

# Homework Policies

- Projects will be (mostly) automatically graded with immediate feedback
  - We'll give you some tests, as part of the assignment
  - You'll write your own tests to supplement these
  - Our grading script will apply *additional* tests
  - Your score is based on how many of these you pass
  - Your code must compile to get *any* credit
- Multiple submissions *are allowed*
  - First few submissions: no penalty
  - Each submission after the first few will be penalized
  - Your final grade is determined by the *best* raw score
- Late Policy
  - Submission up to 24 hours late costs 10 points
  - Submission 24-48 hours late costs 20 points
  - After 48 hours, no submissions allowed
- Style / Test cases:
  - manual grading of non-testable properties
  - feedback on style from your TAs

# Important Dates

- Homework:
  - Homework due dates will be listed on course calendar
  - Tuesdays at midnight: see schedule on web
- Exams:
  - 12% First midterm: **Friday, Oct. 13<sup>th</sup> in class**
  - 12% Second midterm: **Friday, Nov. 10<sup>th</sup> in class**
  - 16% Final exam: TBA
  - Contact instructor *well in advance* if you have a conflict
  - Make-up Exam Times will be announced beforehand

# Where to ask questions

- Course material
  - **Piazza Discussion Boards**
  - TA office hours, on webpage calendar
  - Tutoring
  - Prof office hours:
    - Zdancewic: Monday from 3:30 to 5:00 PM,
    - Sheth: Tues & Weds 4:00-5:00PM
    - or by appointment (changes will be announced on Piazza)
- HW/Exam Grading: see webpage
- About CIS majors & Registration
  - Ms. Desirae Cesar, Levine 309  
CIS Undergraduate coordinator

# Programming in OCaml

Read Chapter 2 of the CIS 120 lecture notes,  
available from the course web page

# What is an OCaml module?

```
;; open Assert ← module import  
  
let attendees (price:int) :int =  
  (-15 * price) / 10 + 870 ← function declarations  
  (use let keyword)  
  
let test () : bool =  
  attendees 500 = 120 ← identifier declarations  
  (also use let)  
  
;; run_test "attendees at 5.00" test ← commands  
  
let x : int = attendees 500 ←  
;; print_int x  
;; print_endline "end of demo"
```

# What does an OCaml program do?

```
;; open Assert
```

```
let attendees (price:int) :int =  
  (-15 * price) / 10 + 870
```

```
let test () : bool =  
  attendees 500 = 120
```

```
;; run_test "attendees at 5.00" test
```

```
let x = attendees 500
```

```
;; print_int x
```

To know if the test will pass, we need to know whether this expression is true or false

To know what will be printed we need to know the value of this expression

*To know what an OCaml program will do, we need to know what the value of each expression is*

# Value-Oriented Programming

pure, functional, strongly typed

# Course goal

*Strive for beautiful code.*

- Beautiful code
  - is *simple*
  - is easy to understand
  - is likely to be correct
  - is easy to maintain
  - takes skill to develop



# Value-Oriented Programming

- Java, C, C#, C++, Python, Perl, etc. are tuned for an **imperative** programming style
  - Programs are full of *commands*
    - “*Change x to 5!*”
    - “*Increment z!*”
    - “*Make this point to that!*”
- OCaml, on the other hand, promotes a **value-oriented** style
  - We’ve seen that there are a few *commands*...  
`print_endline, run_test`  
... but these are used rarely
  - Most of what we write is *expressions* denoting *values*

Metaphorically, we might say that  
imperative programming is about *doing*  
while  
value-oriented programming is about *being*



# Programming with Values

- Programming in *value-oriented* (a.k.a. *pure* or *functional*) style can be a bit challenging at first



- But it often leads to code that is much simpler to understand

# Values and Expressions

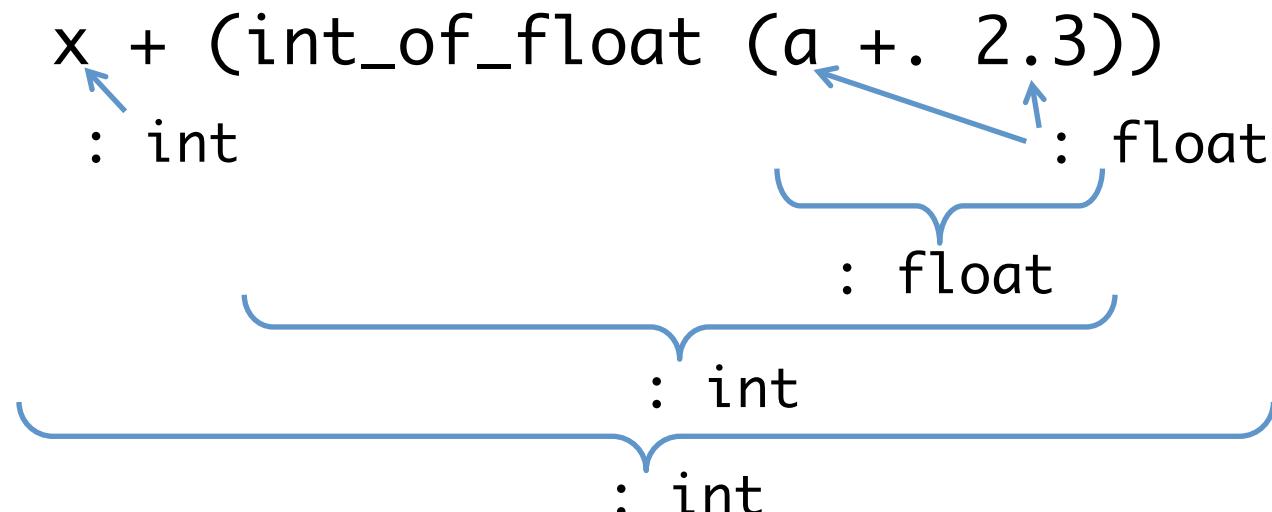
| Types  | Values           | Operations*                      | Expressions       |
|--------|------------------|----------------------------------|-------------------|
| int    | -1 0 1 2         | + * - /                          | 3 + (4 * x)       |
| float  | 0.12 3.1415      | +. *. -. /.                      | 3.0 *. (4.0 *. a) |
| string | “hello” “CIS120” | ^ <small>(concatenation)</small> | “Hello, ” ^ s     |
| bool   | true false       | &&    not                        | (not b1)    b2    |

- Each type corresponds to a set of well-typed values.

\*Note that there is no automatic conversion from float to int, etc., so you must use explicit conversion operations like `string_of_int` or `float_of_int`

# Types

- Every *identifier* has a unique associated type.
- "Colon" notation associates an identifier with its type:  
 $x : \text{int}$        $a : \text{float}$   
 $s : \text{string}$        $b1 : \text{bool}$
- Every OCaml *expression* has a unique type determined by its constituent *subexpressions*



# Type Errors

- OCaml will use *type inference* to check that your program to ensure that it uses types consistently.
  - It will give you an error if not

```
x + (string_of_float (a +. 2.3))  
: int  
: float  
: float  
: string
```

ERROR: expected int but found string

NOTE: Every time OCaml points out a type error, it is indicating a likely bug! Well-typed Ocaml programs often "just work".

# Sneak Preview

- Every expression has a type, and OCaml has a rich type structure:

`(+) : int -> int -> int` *function types*  
`string_of_int : int -> string`

`() : unit`  
`(1, 3.0) : int * float` *tuple types*

`[1;2;3] : int list` *list types*

- We will see all of these, and how to define our own types, in the upcoming lectures...

# Calculating Expression Values

OCaml's model of computation

# Simplification vs. Execution

- We can think of an OCaml expression as just a way of writing down a *value*
- We can visualize running an OCaml program as a sequence of *calculation* or *simplification* steps that eventually lead to this value
- By contrast, a running Java program is best thought of as performing a sequence of *actions* or *commands*
  - ... *a variable named x gets created*
  - ... *then we put the value 3 in x*
  - ... *then we test whether y is greater than z*
  - ... *the answer is true, so we put the value 4 in x*
- Each command modifies the *implicit, pervasive state* of the machine

# Calculating with Expressions

OCaml programs mostly consist of *expressions*.

Expressions *simplify* to values:

3  $\Rightarrow$  3

(values compute to themselves)

3 + 4  $\Rightarrow$  7

2 \* (4 + 5)  $\Rightarrow$  18

attendees 500  $\Rightarrow$  120

The notation  $\langle \text{exp} \rangle \Rightarrow \langle \text{val} \rangle$  means that the expression  $\langle \text{exp} \rangle$  computes to the final value  $\langle \text{val} \rangle$ .

Note that the symbol ' $\Rightarrow$ ' is *not* OCaml syntax. We're using it to talk about the way OCaml programs behave.

# Step-wise Calculation

- We can understand  $\Rightarrow$  in terms of single step calculations, written  $\mapsto$
- For example:

$$(2+3) * (5-2)$$

$$\mapsto 5 * (5-2)$$

because  $2+3 \mapsto 5$

$$\mapsto 5 * 3$$

because  $5-2 \mapsto 3$

$$\mapsto 15$$

because  $5*3 \mapsto 15$

# Conditional Expressions

```
if s = "positive" then 1 else -1
```

```
if day >= 6 && day <= 7  
then "weekend" else "weekday"
```

- OCaml conditionals are also *expressions*: they can be used inside of other expressions:

```
(if 3 > 0 then 2 else -1) * 100
```

```
if x > y then "x is bigger"  
else if x < y then "y is bigger"  
else "same"
```

# Simplifying Conditional Expressions

- A conditional expression yields the value of either its ‘then’-branch or its ‘else’-branch, depending on whether the test is ‘true’ or ‘false’.
- For example:

```
(if 3 > 0 then 2 else -1) * 100  
→ (if true then 2 else -1) * 100  
→ 2 * 100  
→ 200
```

- The type of a conditional expression is the (single!) type shared by *both* of its branches.
- It doesn’t make sense to leave out the ‘else’ branch in an ‘if’. (What would be the result if the test was ‘false’?)

# Top-level Let Declarations

- A let declaration gives a *name* (a.k.a. an *identifier*) to the value denoted by some expression

```
let pi : float = 3.14159
let seconds_per_day : int = 60 * 60 * 24
```

- The *scope* of a top-level identifier is the rest of the file after the declaration.

“scope” of a name = “the region of the program in which it can be used”

# Immutability

- Once defined by `let`, the binding between an identifier and a value cannot be changed!

```
int x = 3;  
x = 4;
```

**Java / C / C++ / ...**  
*imperative update*

'`x = 4`' is a command  
that means 'update the  
contents of location  
`x` to be 3'

The state associated with '`x`'  
*changes as the program runs.*

```
let x : int = 3 in  
x = 4
```

**Ocaml**  
*named expressions*

'`let x : int = 3`' simply gives  
the value 3 the name '`x`'

'`x = 4`' asks does '`x` equal 4'?  
(a boolean value, false)

Once defined, the value  
bound to '`x`' never changes

# Local Let Expressions

- Let declarations can appear both at top-level and *nested* within other expressions.

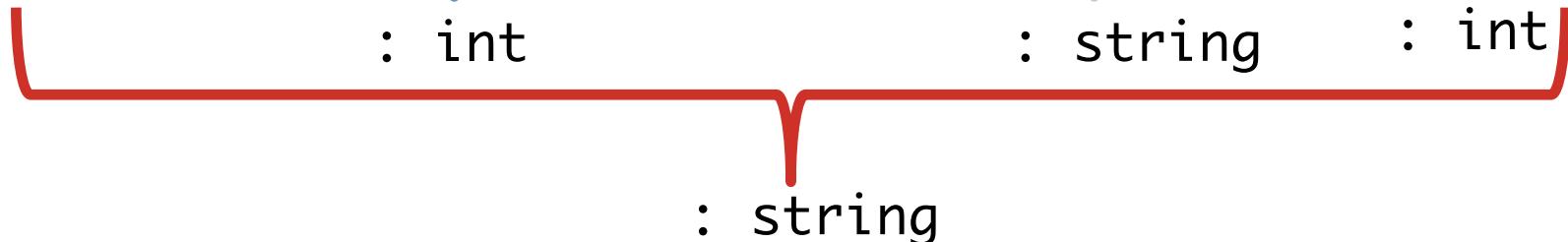
```
let profit_500 : int =
    let attendees = 120 in
        let revenue = attendees * 500 in
            let cost = 18000 + 4 * attendees in
                revenue - cost
```

The scope of attendees is the expression after the 'in'

- Local (nested) let declarations are followed by 'in'
  - e.g. attendees, revenue, and cost
- Top-level let declarations do not use 'in'
  - e.g. profit\_500
- The scope of a local identifier is just the expression after the 'in'

# Typing Let Expressions

```
let x = 3 + 5 in string_of_int (x * x)
```

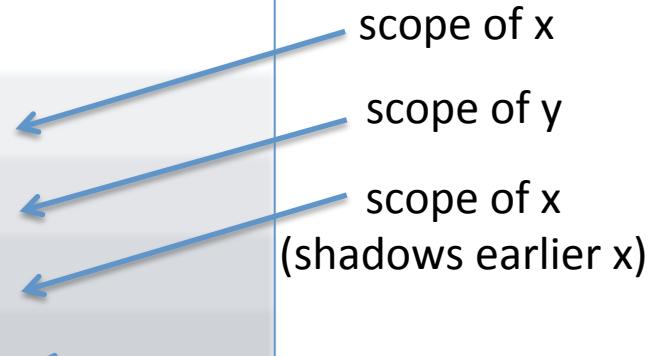


- A let-bound identifier has the type of the expression it is bound to.
- The type of the whole local let expression is the type of the expression after the ‘in’
- Recall: type annotations are written using colon:  
`let x : int = ... ((x + 3) : int) ...`

# Scope

Multiple declarations of the same variable or function name are allowed. The later declaration *shadows* the earlier one for the rest of the program.

```
let total : int =
  let x = 1 in
  let y = x + 1 in
  let x = 1000 in
  let z = x + 2 in
  x + y + z
```



scope of total is the rest of the file

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘`let...in`’ part
  - simplify what's left

```
let total : int =
  let x = 1 in
  let y = x + 1 in
  let x = 1000 in
  let z = x + 2 in
    x + y + z
```

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
let x = 1 in
let y = x + 1 in
let x = 1000 in
let z = x + 2 in
x + y + z
```

First, we simplify the right-hand side of the declaration for identifier total.

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
  let x = 1 in
  let y = x + 1 in
  let x = 1000 in
  let z = x + 2 in
    x + y + z
```

This r.h.s. is  
already a  
value.

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what’s left

```
let total : int =
  let x = 1 in
  let y = 1 + 1 in
  let x = 1000 in
  let z = x + 2 in
  x + y + z
```

Substitute 1  
for x here.

But not  
here because  
the second x  
shadows the first.

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
  let x = 1 in ←
  let y = 1 + 1 in
  let x = 1000 in
  let z = x + 2 in
    x + y + z
```

Discard the local let since it's been substituted away.

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =  
  
  let y = 1 + 1 in  
  let x = 1000 in  
  let z = x + 2 in  
    x + y + z
```

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =  
let y = 1 + 1 in  
let x = 1000 in  
let z = x + 2 in  
x + y + z
```

Simplify the expression remaining in scope.

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
```

```
let y = 1 + 1 in  
let x = 1000 in  
let z = x + 2 in  
x + y + z
```

Repeat!

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘`let...in`’ part
  - simplify what's left

```
let total : int =  
  
  let y = 2 in  
  let x = 1000 in  
  let z = x + 2 in  
    x + y + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘`let...in`’ part
  - simplify what's left

```
let total : int =  
  
  let y = 2 in  
  let x = 1000 in  
  let z = x + 2 in  
    x + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘`let...in`’ part
  - simplify what's left

```
let total : int =  
  
  let y = 2 in  
  let x = 1000 in  
  let z = x + 2 in  
    x + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a `let` expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘`let...in`’ part
  - simplify what's left

```
let total : int =
```

```
  let x = 1000 in  
  let z = x + 2 in  
    x + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
```

```
let x = 1000 in  
let z = x + 2 in  
x + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
```

```
let x = 1000 in  
let z = x + 2 in  
x + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
```

```
let x = 1000 in  
let z = 1000 + 2 in  
1000 + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
```

```
let x = 1000 in  
let z = 1000 + 2 in  
1000 + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
```

```
let z = 1000 + 2 in  
1000 + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
```

```
let z = 1000 + 2 in  
1000 + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
```

```
let z = 1000 + 2 in  
1000 + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
```

```
let z = 1002 in  
1000 + 2 + z
```

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
```

```
let z = 1002 in  
1000 + 2 + 1002
```

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
```

```
let z = 1002 in  
1000 + 2 + 1002
```

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
```

1000 + 2 + 1002

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int =
```

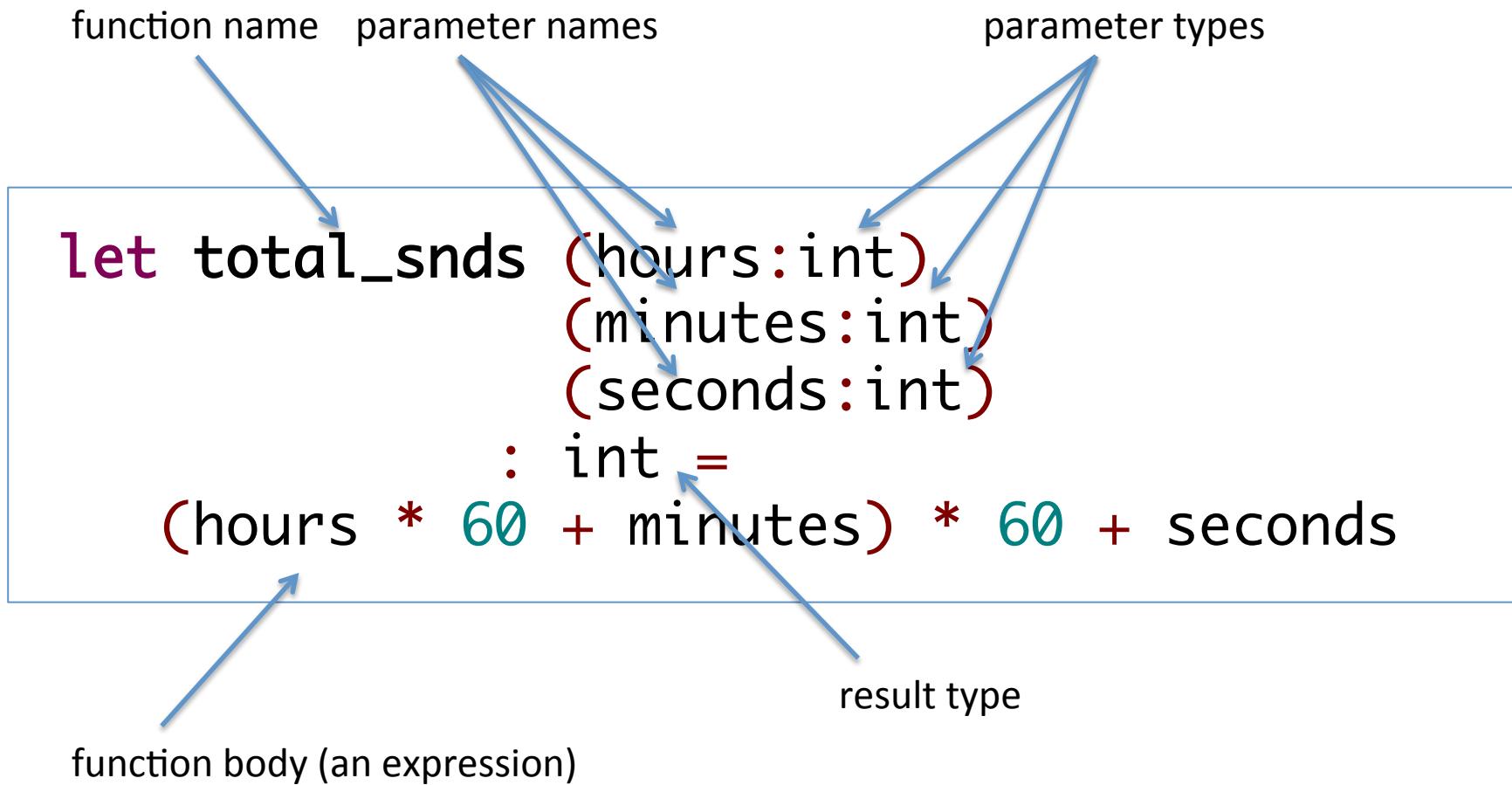
$$\underline{1000 + 2 + 1002} \quad \Rightarrow \quad 2004$$

# Simplifying Let Expressions

- To calculate the value of a let expression:
  - first calculate the value of the right hand side
  - then *substitute* the resulting value for the identifier in its scope
  - drop the ‘let...in’ part
  - simplify what's left

```
let total : int = 2004
```

# (Top-level) Function Declarations



# Function Calls

Once a function has been declared, it can be invoked by writing the function name followed by a list of arguments. This is a *function application* expression.

```
total_secs 5 30 22
```

(Note that the list of arguments is *not* parenthesized.)

# Calculating With Functions

- To calculate the value of a function application, first calculate values for its arguments and then *substitute* them for the parameters in the body of the functions.

```
total_snvs (2 + 3) 12 17
→ total_snvs 5 12 17
→ (5 * 60 + 12) * 60 + 17      subst. the args in the body
→ (300 + 12) * 60 + 17
→ 312 * 60 + 17
→ 18720 + 17
→ 18737
```

```
let total_secs (hours:int)
              (minutes:int)
              (seconds:int)
: int =
  (hours * 60 + minutes) * 60 + seconds
```

# Things (for you) to do...

- Sign up for Piazza if necessary
- Sign up for Codio
  - Link posted on Piazza
- Homework 1: OCaml Finger Exercises
  - Practice using OCaml to write simple programs
  - Start with first 4 problems
    - (needed background on lists coming next week!)
  - Start early!

# Programming Languages and Techniques (CIS120)

Lecture 3

September 6, 2017

Value-Oriented Programming (continued)  
Lists and Recursion

# CIS 120 Announcements

- Homework 1: OCaml Finger Exercises
  - Due: Tuesday 9/12 at midnight
  - Safari automatically unzips submit.zip: see Piazza for how to disable that
- Reading: Please read *up to* Chapter 3
- Questions?
  - Post to Piazza (privately if you need to include code!)
  - Look at HW1 FAQ
- TA office hours: on course Calendar webpage
- Recitations start today!

Have you started working on HW 1?

# Review: Value-Oriented Programming

- Ocaml promotes a **value-oriented** style
  - We've seen that there are a few *commands*...  
`print_endline, run_test`  
... but these are used rarely
  - Most of what we write is *expressions* denoting *values*
- We can think of an OCaml expression as just a way of writing down a *value*
- We can visualize running an OCaml program as a sequence of *calculation* or *simplification* steps that eventually lead to this value

# Purity

- The meaning (“denotation”) of a pure expression doesn’t change over time:

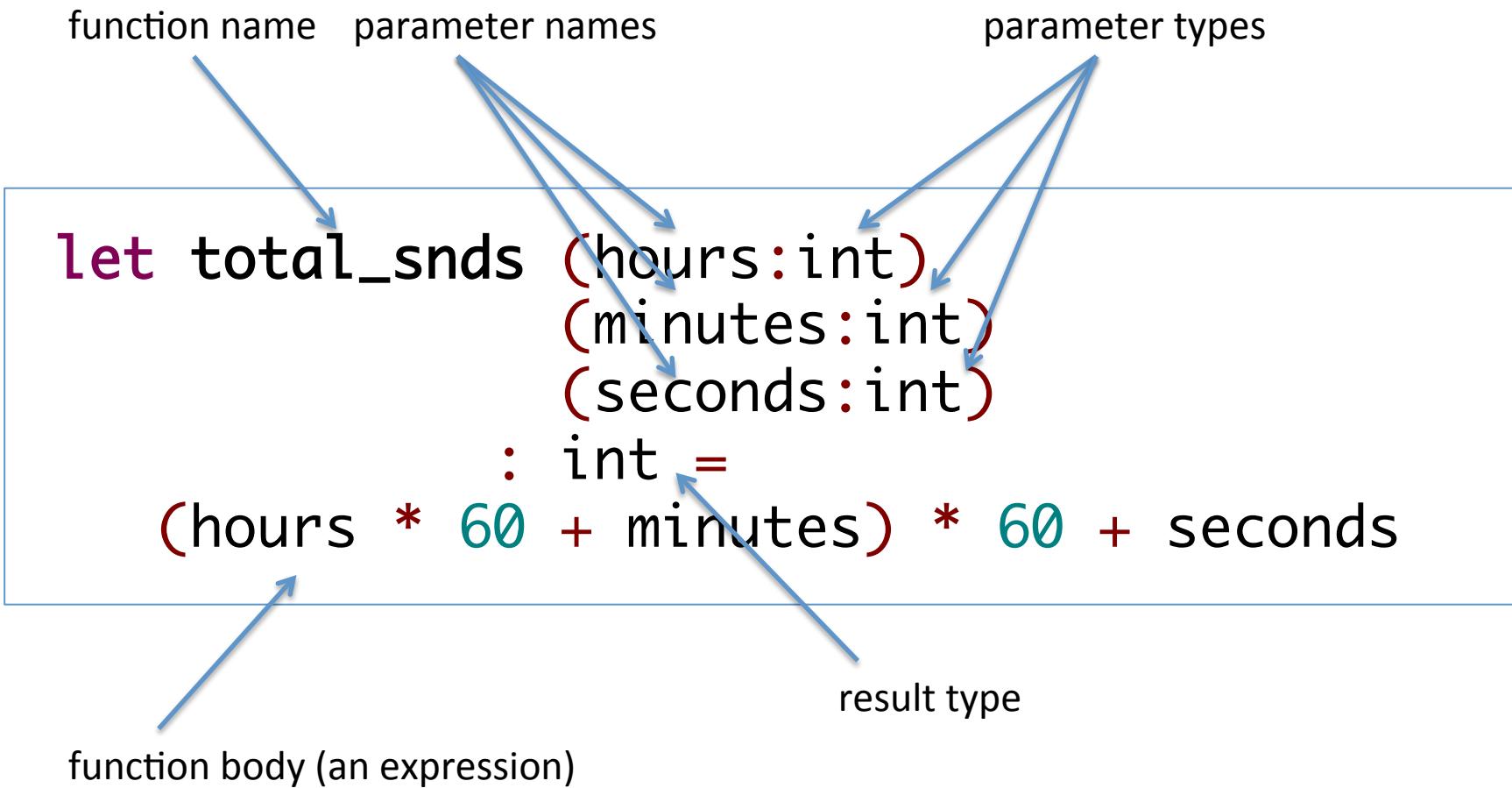
```
let atts : int = (attendees 500)
```

```
let atts2 : int = atts + atts
```

```
let atts2 : int =
  (attendees 500)
+ (attendees 500)
```

Both ways of computing atts2 yield the same value (in contrast with an imperative language, which runs attendees’ *effects* twice.)

# (Top-level) Function Declarations



# Function Calls

Once a function has been declared, it can be invoked by writing the function name followed by a list of arguments. This is a *function application* expression.

```
total_snd 5 30 22
```

(Note that the list of arguments is *not* parenthesized.)

# Calculating With Functions

- To calculate the value of a function application, first calculate values for its arguments and then *substitute* them for the parameters in the body of the functions.

```
total_snvs (2 + 3) 12 17
→ total_snvs 5 12 17
→ (5 * 60 + 12) * 60 + 17      subst. the args in the body
→ (300 + 12) * 60 + 17
→ 312 * 60 + 17
→ 18720 + 17
→ 18737
```

```
let total_secs (hours:int)
              (minutes:int)
              (seconds:int)
: int =
  (hours * 60 + minutes) * 60 + seconds
```

# Working with Lists

# A Design Problem / Situation

Suppose we are asked by Penn to design a new email system for notifying instructors and students of emergencies or unusual events.

*What should we be able to do with this system?*

Subscribe students to the list, query the size of the list, check if a particular email is enrolled, compose messages for all the list, filter the list to just students, etc.



**Slate** YOUR NEWS COMPANION JAN. 19 2016 12:43 PM

This “Blizzard for the Ages” Headed for the East Coast Is Very Much the Real Deal

By Eric Holthaus

**Snow certain; how much is question**

1 to 2 feet of snow possible this weekend  
Milk, bread, eggs stocked ahead of snowstorm

# Design Pattern

## 1. Understand the problem

What are the relevant concepts and how do they relate?

## 2. Formalize the interface

How should the program interact with its environment?

## 3. Write test cases

How does the program behave on typical inputs? On unusual ones? On erroneous ones?

## 4. Implement the behavior

Often by decomposing the problem into simpler ones and applying the same recipe to each

# 1. Understand the problem

*How do we store and query information about email addresses?*

Important concepts are:

1. An mailing list (collection of email addresses)
2. A fixed collection of *instructor\_emails*
3. Being able to *subscribe* students & instructors to the list
4. Counting the *number\_of\_emails* in a list
5. Determining whether a list *contains* a particular email
6. Given a message to send, *compose* messages for all the email addresses in the list
7. *remove\_instructors*, leaving an email list just containing the list of enrolled students

## 2. Formalize the interface

- Represent an email by a *string* (the email address itself)
- Represent an email list using an *immutable list of strings*
- Represent the collection of instructor emails using a *toplevel definition*

```
let instructor_emails : string list = ...
```

- Define the interface to the functions:

```
let subscribe (email : string)  
            (lst : string list) : string list = ...
```

```
let length (lst : string list) : int = ...
```

```
let contains (lst : string list) (email : string) : bool =  
...  
...
```

### 3. Write test cases

```
let l1 : string list = [ "stevez@cis.upenn.edu";  
                        "mattch@seas.upenn.edu";  
                        "davisbet@seas.upenn.edu" ]  
let l2 : string list = [ "mattch@seas.upenn.edu" ]  
let l3 : string list = []  
  
let test () : bool =  
  (length l1) = 3  
;; run_test "length l1" test  
  
let test () : bool =  
  (length l2) = 1  
;; run_test "length l2" test  
  
let test () : bool =  
  (length l3) = 0  
;; run_test "length l3" test
```

Define email lists for testing.  
Include a variety of lists of different sizes and incl. some instructor and non-instructor emails as well.

# Lists

A Value-Oriented Approach

# What is a list?

A list value is either:

[ ]            the *empty* list, sometimes called *nil*

or

v :: tail    a *head* value v, followed by a list of the remaining elements, the *tail*

- Here, the ‘::’ operator *constructs* a new list from a head element and a shorter list.
  - This operator is pronounced “cons” (for “construct”)
- Importantly, *there are no other kinds of lists.*
- Lists are an example of an *inductive datatype*.

# Example Lists

To build a list, cons together elements, ending with the empty list:

1 :: 2 :: 3 :: 4 :: [ ]

a list of (four) ints

"abc" :: "xyz" :: [ ]

a list of (two) strings

(false :: [ ]) :: (true :: [ ]) :: [ ]

a list of lists that each contain booleans

[ ]

the empty list

# Explicitly parenthesized

':::' is an ordinary operator like + or ^, except it takes an element and a *list* of elements as inputs:

```
1 ::: ( 2 ::: ( 3 ::: ( 4 ::: [ ] ) ) )
```

a list of four numbers

```
"abc" ::: ("xyz" ::: [ ] )
```

a list of two strings

```
true ::: [ ]
```

a list of one boolean

```
[ ]
```

the empty list

# Convenient Syntax

Much simpler notation: enclose a list of elements in [ and ] separated by ;

```
[ 1;2;3;4 ]
```

a list of (four) ints

```
[ "abc";"xyz" ]
```

a list of (two) strings

```
[ [ false ];[ true ] ]
```

a list of lists that each contain booleans

```
[ ]
```

the empty list

# NOT Lists

These are *not* lists:

[ 1 ; true ; 3 ; 4 ]

different element types

1 :: 2

2 is not a list

3 :: [ ] :: [ ]

different element types

# Calculating With Lists

- Calculating with lists is just as easy as calculating with arithmetic expressions:

$(2+3)::(12 / 5)::[]$

$\mapsto 5::(12 / 5)::[]$  because  $2+3 \Rightarrow 5$

$\mapsto 5::2::[]$  because  $12/5 \Rightarrow 2$

A list is a value whenever all of its elements are values.

# List Types\*

The type of lists of integers is written

`int list`

The type of lists of strings is written

`string list`

The type of lists of booleans is written

`bool list`

The type of lists of lists of strings is written

`(string list) list`

etc.

\*Note that lists in OCaml are *homogeneous* – all of the list elements must be of the same type. If you try to create a list like [1; "hello"; 3; true] you will get a type error.

# Interactive Interlude

email.ml

Which of the following expressions has the type  
**int list** ?

- 1) [3; true]
- 2) [1;2;3]:::[1;2]
- 3) []:::[1;2]:::[]
- 4) (1::2):::(3:::4):::[]
- 5) [1;2;3;4]

Answer: 5

Which of the following expressions has the type  
`(int list) list` ?

- 1) `[3; true]`
- 2) `[1;2;3]:::[1;2]`
- 3) `[]:::[1;2]:::[ ]`
- 4) `(1:::2):::(3:::4):::[ ]`
- 5) `[1;2;3;4]`

Answer: 3

# What can we do with lists?

*What operations can we do on lists?*

1. Access the elements
2. Create new lists by adding an element
3. Calculate its length
4. Search the list
5. Transform the list
6. Filter the list
7. ...

Value oriented programming:

We can *name* the sub-components of a list.

We can construct new values using those names.

# Pattern Matching

OCaml provides a single expression called *pattern matching* for inspecting a list and naming its subcomponents.

```
let mylist : int list = [1; 2; 3; 5]  
  
let y : int =  
  begin match mylist with  
  | [] -> 42  
  | first::rest -> first+10  
  end
```

match expression  
syntax is:

begin match ... with  
| ... -> ...  
| ... -> ...  
end

case  
branches

Case analysis is justified because there are only *two* shapes a list can have.

Note that `first` and `rest` are identifiers that are bound in the body of the branch

- `first` names the head of the list; its type is the element type.
- `rest` names the tail of the list; its type is the list type

The type of the match expression is the (one) type shared by its branches.

# Calculating with Matches

- Consider how to evaluate a match expression:

```
begin match [1;2;3] with
  | [] -> 42
  | first::rest -> first + 10
end
```

→  
1 + 10

→  
11

Note: [1;2;3] equals 1::(2::(3::[]))

It doesn't match the pattern [] so the first branch is skipped, but it *does* match the pattern `first::rest` when `first` is 1 and `rest` is (2::(3::[])).  
So, substitute 1 for `first` in the second branch.

# The Inductive Nature of Lists

A list value is either:

[ ]                    the *empty* list, sometimes called *nil*

or

v :: tail    a *head* value v, followed by a *list* of the remaining elements, the *tail*

- Why is this well-defined? The definition of list mentions ‘list’!
- Solution: ‘list’ is *inductive*:
  - The empty list [] is the (only) list of 0 elements
  - To construct a list of (1+n) elements, add a head element to an *existing* list of n elements
  - The set of list values contains all and only values constructed this way
- Corresponding computation principle: *recursion*

# Recursion

## *Recursion principle:*

Compute a function value for a given input by combining the results for strictly smaller subcomponents of the input.

- The structure of the computation follows the inductive structure of the input.

- Example:

$$\text{length } 1::2::3::[] = 1 + (\text{length } 2::3::[])$$

$$\text{length } 2::3::[] = 1 + (\text{length } 3::[])$$

$$\text{length } 3::[] = 1 + (\text{length } [])$$

$$\text{length } [] = 0$$

# Interactive Interlude

email.ml

# Recursion Over Lists in Code

The function calls itself *recursively* so the function declaration must be marked with rec.

Lists are either empty or nonempty.  
*Pattern matching* determines which.

```
let rec length (l : string list) : int =
  begin match l with
    | [] -> 0
    | (x :: rest) -> 1 + length rest
  end
```

If the list is non-empty, then “x” is the first string in the list and “rest” is the remainder of the list.

Patterns specify the **structure** of the value and (optionally) give **names** to parts of it.

# Calculating with Recursion

`length ["a"; "b"]`

→ (*substitute the list for l in the function body*)

```
begin match "a"::"b"::[] with
| [] -> 0
| (x :: rest) -> 1 + length rest
end
```

→ (*second case matches with rest = "b"::[]*)

`1 + (length "b"::[])`

→ (*substitute the list for l in the function body*)

```
1 + (begin match "b"::[] with
| [] -> 0
| (x :: rest) -> 1 + length rest
end )
```

→ (*second case matches again, with rest = []*)

`1 + (1 + length [])`

→ (*substitute [] for l in the function body*)

...

→ `1 + 1 + 0 ⇒ 2`

```
let rec length (l:string list) : int=
begin match l with
| [] -> 0
| (x :: rest) -> 1 + length rest
end
```

# Recursive function patterns

Recursive functions over lists follow a general pattern:

```
let rec length (l : string list) : int =
begin match l with
| [] -> 0
| (x :: rest) -> 1 + length rest
end
```

```
let rec contains (l:string list) (s:string) : bool =
begin match l with
| [] -> false
| (x :: rest) -> s = x || contains rest s
end
```

# Structural Recursion Over Lists

Structural recursion builds an answer from smaller components:

```
let rec f (l : ... list) ... : ... =
begin match l with
| [] -> ...
| (hd :: rest) -> ... f rest ...
end
```

The branch for [ ] calculates the value ( $f [ ]$ ) directly.

- this is the *base case* of the recursion

The branch for `hd::rest` calculates

$(f(hd::rest))$  given `hd` and  $(f \text{ rest})$ .

- this is the *inductive case* of the recursion

# Design Pattern for Recursion

1. Understand the problem  
What are the relevant concepts and how do they relate?
2. Formalize the interface  
How should the program interact with its environment?
3. Write test cases
  - If the main input to the program is an immutable list, make sure the tests cover both empty and non-empty cases
4. Implement the required behavior
  - If the main input to the program is an immutable list, look for a recursive solution...
    - Is there a direct solution for the empty list?
    - Suppose someone has given us a partial solution that works for lists up to a certain size. Can we use it to build a better solution that works for lists that are one element larger?

# Programming Languages and Techniques (CIS120)

Lecture 4  
September 8, 2017

Lists and Tuples

# Announcements

- Read Chapters 3 and 4 of the lecture notes
- HW01 due Tuesday at midnight

# Review: What is a list?

A list value is either:

[ ]                  the *empty* list, sometimes called *nil*

or

v :: tail    a *head* value v, followed by a list of the remaining elements, the *tail*

- The ‘::’ operator, pronounced “cons”, *constructs* a new list from a head element and a shorter list.
- *There are no other kinds of lists.*
- Lists are an example of an *inductive datatype*.

Which of the following expressions has the type  
**int list** ?

- 1) [3; true]
- 2) [1;2;3]:::[1;2]
- 3) []:::[1;2]:::[]
- 4) (1::2):::(3:::4):::[]
- 5) [1;2;3;4]

# Pattern Matching

OCaml provides a single expression called *pattern matching* for inspecting a list and naming its subcomponents.

```
let mylist : int list = [1; 2; 3; 5]  
  
let y : int =  
  begin match mylist with  
  | [] -> 42  
  | first::rest -> first+10  
  end
```

match expression  
syntax is:

begin match ... with  
| ... -> ...  
| ... -> ...  
end

Case analysis is justified because there are only *two* shapes a list can have.

Note that `first` and `rest` are identifiers that are bound in the body of the branch

- `first` names the head of the list; its type is the element type.
- `rest` names the tail of the list; its type is the list type

The type of the match expression is the (one) type shared by its branches.

# Calculating with Matches

- Consider how to evaluate a match expression:

```
begin match [1;2;3] with
  | [] -> 42
  | first::rest -> first + 10
end
```

→  
1 + 10

→  
11

Note: [1;2;3] equals 1::(2::(3::[]))

It doesn't match the pattern [] so the first branch is skipped, but it *does* match the pattern `first::rest` when `first` is 1 and `rest` is (2::(3::[])).  
So, substitute 1 for `first` in the second branch.

# The Inductive Nature of Lists

A list value is either:

[ ]                    the *empty* list, sometimes called *nil*

or

v :: tail        a *head* value v, followed by a *list* of the remaining elements, the *tail*

- Why is this well-defined? The definition of list mentions ‘list’!
- Solution: ‘list’ is *inductive*:
  - The empty list [] is the (only) list of 0 elements
  - To construct a list of (1+n) elements, add a head element to an *existing* list of n elements
  - The set of list values contains all and only values constructed this way
- Corresponding computation principle: *recursion*

# Recursion

## *Recursion principle:*

Compute a function value for a given input by combining the results for strictly smaller subcomponents of the input.

- The structure of the computation follows the inductive structure of the input.

- Example:

$$\begin{aligned}\text{length } 1::2::3::[] &= 1 + (\text{length } 2::3::[]) \\ \text{length } 2::3::[] &= 1 + (\text{length } 3::[]) \\ \text{length } 3::[] &= 1 + (\text{length } []) \\ \text{length } [] &= 0\end{aligned}$$

# Recursion Over Lists in Code

The function calls itself *recursively* so the function declaration must be marked with rec.

Lists are either empty or nonempty.  
*Pattern matching* determines which.

```
let rec length (l : string list) : int =
  begin match l with
    | [] -> 0
    | (x :: rest) -> 1 + length rest
  end
```

If the list is non-empty, then “x” is the first string in the list and “rest” is the remainder of the list.

Patterns specify the **structure** of the value and (optionally) give **names** to parts of it.

# Calculating with Recursion

`length ["a"; "b"]`

→ (*substitute the list for l in the function body*)

```
begin match "a"::"b"::[] with
| [] -> 0
| (x :: rest) -> 1 + length rest
end
```

→ (*second case matches with rest = "b"::[]*)

`1 + (length "b"::[])`

→ (*substitute the list for l in the function body*)

```
1 + (begin match "b"::[] with
| [] -> 0
| (x :: rest) -> 1 + length rest
end )
```

→ (*second case matches again, with rest = []*)

`1 + (1 + length [])`

→ (*substitute [] for l in the function body*)

...

→ `1 + 1 + 0 ⇒ 2`

```
let rec length (l:string list) : int=
begin match l with
| [] -> 0
| (x :: rest) -> 1 + length rest
end
```

# Interactive Interlude

email.ml

# Tuples and Tuple Patterns

# Forms of Structured Data

OCaml provides two ways of packaging multiple values together into a single compound value:

- **Lists:**
  - *arbitrary-length* sequence of values of a single, *fixed type*
  - example: a list of email addresses
- **Tuples:**
  - *fixed-length* sequence of values of *arbitrary types*
  - example: tuple of name, phone #, and email

# Tuples

- In OCaml, tuples are created by writing the values, separated by commas, in parentheses:

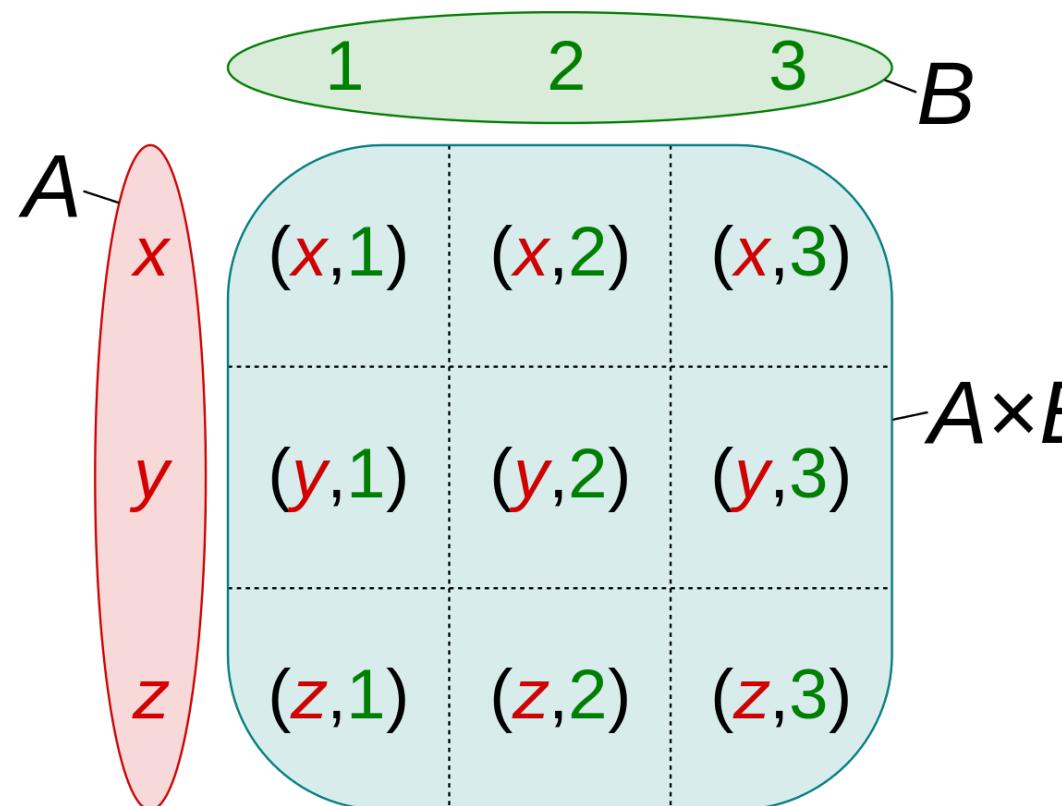
```
let my_pair = (3, true)
let my_triple = ("Hello", 5, false)
let my_quaduple = (1,2,"three",false)
```

- Tuple types are written using '\*'
  - e.g. `my_triple` has type:

```
string * int * bool
```

# Cartesian Products

- The values of a tuple (a.k.a. product) type are tuples of elements from each component type.



Ocaml  
notation:  
 $A * B$

# Pattern Matching Tuples

- Tuples can be inspected by pattern matching:

```
let first (x: string * int) : string =
  begin match x with
    | (left, right) -> left
  end
```

```
first ("b", 10)
⇒
"b"
```

- As with lists, the pattern follows the syntax of the values, naming the subcomponents

# Mixing Tuples and Lists

- Tuples and lists can mix freely:

```
[ (1,"a"); (2,"b"); (3,"c") ]  
  : (int * string) list
```

```
( [1;2;3], ["a"; "b"; "c"] )  
  : (int list) * (string list)
```

# Nested Patterns

- So far, we've seen simple patterns:

[ ]

*matches empty list*

x :: t1

*matches nonempty list*

(a, b)

*matches pairs (tuples with 2 elts)*

(a, b, c)

*matches triples (tuples with 3 elts)*

- Like expressions, patterns can *nest*:

x :: [ ]

*matches lists with 1 element*

[x]

*matches lists with 1 element*

x :: (y :: t1)

*matches lists of length at least 2*

(x :: xs, y :: ys)

*matches pairs of non-empty lists*

# Wildcard Pattern

- Another handy pattern is the wildcard pattern: `_`
  - `_ :: tl`      *matches a non-empty list, but only names tail*
  - `(_, x)`      *matches a pair, but only names the 2<sup>nd</sup> part*
- A wildcard pattern indicates that the value of the corresponding subcomponent is irrelevant.
  - And hence needs no name.

# Unused Branches

- The branches in a match expression are considered in order from top to bottom.
- If you have “redundant” matches, then some later branches might not be reachable.
  - OCaml will give you a warning

```
let bad_cases (l : int list) : int =
  begin match l with
    | [] -> 0
    | x::_ -> x
    | x::y::tl -> x + y (* unreachable *)
  end
```

This case matches more lists than that one does.

# What is the value of this expression?

```
let l = [1; 2] in  
  
begin match l with  
| x :: y :: t    -> 1  
| x :: []         -> 2  
| x :: t          -> 3  
| []              -> 4  
end
```

```
let l = [1; 2] in
begin match l with
| x :: y :: t -> 1
| x :: [] -> 2
| x :: t -> 3
| [] -> 4
end
```

```
let l = 1 :: 2 :: [] in
begin match l with
| x :: y :: t -> 1
| x :: [] -> 2
| x :: t -> 3
| [] -> 4
end
```

1

What is the value of this expression?

```
let l = [(2,true); (3,false)] in  
  
begin match l with  
| (x,false) :: tl    -> 1  
| w :: (x,y) :: z   -> x  
| x                  -> 4  
end
```

Answer: 3

What is the value of this expression?

```
let l = [(2,true); (3,false)] in  
  
begin match l with  
| (_,false) :: _          -> 1  
| _ :: (x,_) :: _         -> x  
| _                         -> 4  
end
```

Answer: 3

# Exhaustive Matches

- Pattern matching is *exhaustive* if there is a pattern for every possible value
- Example of a *non-exhaustive* match:

```
let sum_two (l : int list) : int =
  begin match l with
    | x::y::_ -> x+y
  end
```

- OCaml will give you a warning and show an example of what isn't covered by your cases

# Exhaustive Matches

- Pattern matching is *exhaustive* if there is a pattern for every possible value
- Example of an *exhaustive* match:

```
let sum_two (l : int list) : int =
  begin match l with
    | x::y::_ -> x+y
    | _ -> failwith "not a length 2 list"
  end
```

- The wildcard pattern and failwith are useful tools for ensuring match coverage

## More List & Tuple Programming

see [patterns.ml](http://patterns.ml)

# Recursive function patterns

Recursive functions over lists follow a general pattern:

```
let rec length (l : string list) : int =
begin match l with
| [] -> 0
| (x :: rest) -> 1 + length rest
end
```

```
let rec contains (l:string list) (s:string) : bool =
begin match l with
| [] -> false
| (x :: rest) -> s = x || contains rest s
end
```

# Structural Recursion Over Lists

Structural recursion builds an answer from smaller components:

```
let rec f (l : ... list) ... : ... =
begin match l with
| [] -> ...
| (hd :: rest) -> ... f rest ...
end
```

The branch for [ ] calculates the value ( $f [ ]$ ) directly.

- this is the *base case* of the recursion

The branch for `hd::rest` calculates

$(f(hd::rest))$  given `hd` and  $(f \text{ rest})$ .

- this is the *inductive case* of the recursion

# Design Pattern for Recursion

1. Understand the problem  
What are the relevant concepts and how do they relate?
2. Formalize the interface  
How should the program interact with its environment?
3. Write test cases
  - If the main input to the program is an immutable list, make sure the tests cover both empty and non-empty cases
4. Implement the required behavior
  - If the main input to the program is an immutable list, look for a recursive solution...
    - Is there a direct solution for the empty list?
    - Suppose someone has given us a partial solution that works for lists up to a certain size. Can we use it to build a better solution that works for lists that are one element larger?

## Example: zip

- zip takes two lists of the same length and returns a single list of pairs:

```
zip [1; 2; 3] ["a"; "b"; "c"] =>  
[(1,"a"); (2,"b"); (3,"c")]
```

```
let rec zip (l1: int list)  
          (l2: string list) : (int * string) list =  
begin match (l1, l2) with  
| ([] , []) -> []  
| (x :: xs, y :: ys) -> (x, y) :: (zip xs ys)  
| _ -> failwith "zip: unequal length lists"  
end
```

# Programming Languages and Techniques (CIS120)

Lecture 5

September 11, 2017

Tuples, Datatypes and Trees

# Announcements

- Homework 1 due tomorrow at 11:59pm!
- Homework 2 available soon
  - due Tuesday, September 19<sup>th</sup>
- Read Chapters 5 – 7

# Recap: Lists, Recursion, & Tuples

# Recursive function patterns

Recursive functions over lists follow a general pattern:

```
let rec length (l : string list) : int =
begin match l with
| [] -> 0
| (x :: rest) -> 1 + length rest
end
```

```
let rec contains (l:string list) (s:string) : bool =
begin match l with
| [] -> false
| (x :: rest) -> s = x || contains rest s
end
```

# Structural Recursion Over Lists

Structural recursion builds an answer from smaller components:

```
let rec f (l : ... list) ... : ... =
begin match l with
| [] -> ...
| (hd :: rest) -> ... f rest ...
end
```

The branch for [ ] calculates the value ( $f [ ]$ ) directly.

- this is the *base case* of the recursion

The branch for `hd::rest` calculates

$(f(hd::rest))$  given `hd` and  $(f \text{ rest})$ .

- this is the *inductive case* of the recursion

Given the definition below, which of the following is correct?

```
let rec f (l1:int list) (l2:int list) : int list =
begin match l1 with
| [] -> l2
| x::xs -> x :: (f xs l2)
end
```

1.  $(f [1;2] [3;4]) = [3;4;1;2]$
2.  $(f [1;2] [3;4]) = [1;2;3;4]$
3.  $(f [1;2] [3;4]) = [4;3;2;1]$
4.  $(f [1;2] [3;4]) = [1;3;2;4]$
5. none of the above

Answer: 2

CIS120

```
let rec f (l1:int list) (l2:int list) : int list =
begin match l1 with
| [] -> l2
| x::xs -> x :: f xs l2
end
```

```
f [1; 2] [3;4]
⇒ 1 :: (f [2] [3;4])
⇒ 1 :: 2 :: (f [] [3;4])
⇒ 1 :: 2 :: [3;4]
= [1;2;3;4]
```

# Design Pattern for Recursion

1. Understand the problem  
What are the relevant concepts and how do they relate?
2. Formalize the interface  
How should the program interact with its environment?
3. Write test cases
  - If the main input to the program is an immutable list, make sure the tests cover both empty and non-empty cases
4. Implement the required behavior
  - If the main input to the program is an immutable list, look for a recursive solution...
    - Is there a direct solution for the empty list?
    - Suppose someone has given us a partial solution that works for lists up to a certain size. Can we use it to build a better solution that works for lists that are one element larger?

# Forms of Structured Data

OCaml provides two ways of packaging multiple values together into a single compound value:

- **Lists:**
  - *arbitrary-length* sequence of values of a single, *fixed type*
  - example: a list of email addresses
- **Tuples:**
  - *fixed-length* sequence of values of *arbitrary types*
  - example: tuple of name, phone #, and email

What is the type of this expression?

(1, [1], [[1]])

1. int
2. int list
3. int list list
4. (int \* int list) list
5. int \* (int list) \* (int list list)
6. (int \* int \* int) list
7. *none (expression is ill typed)*

Answer: 5

CIS120

What is the type of this expression?

```
[ (1,true); (0, false) ]
```

1. int \* bool
2. int list \* bool list
3. (int \* bool) list
4. (int \* bool) list list
5. *none (expression is ill typed)*

Answer: 3

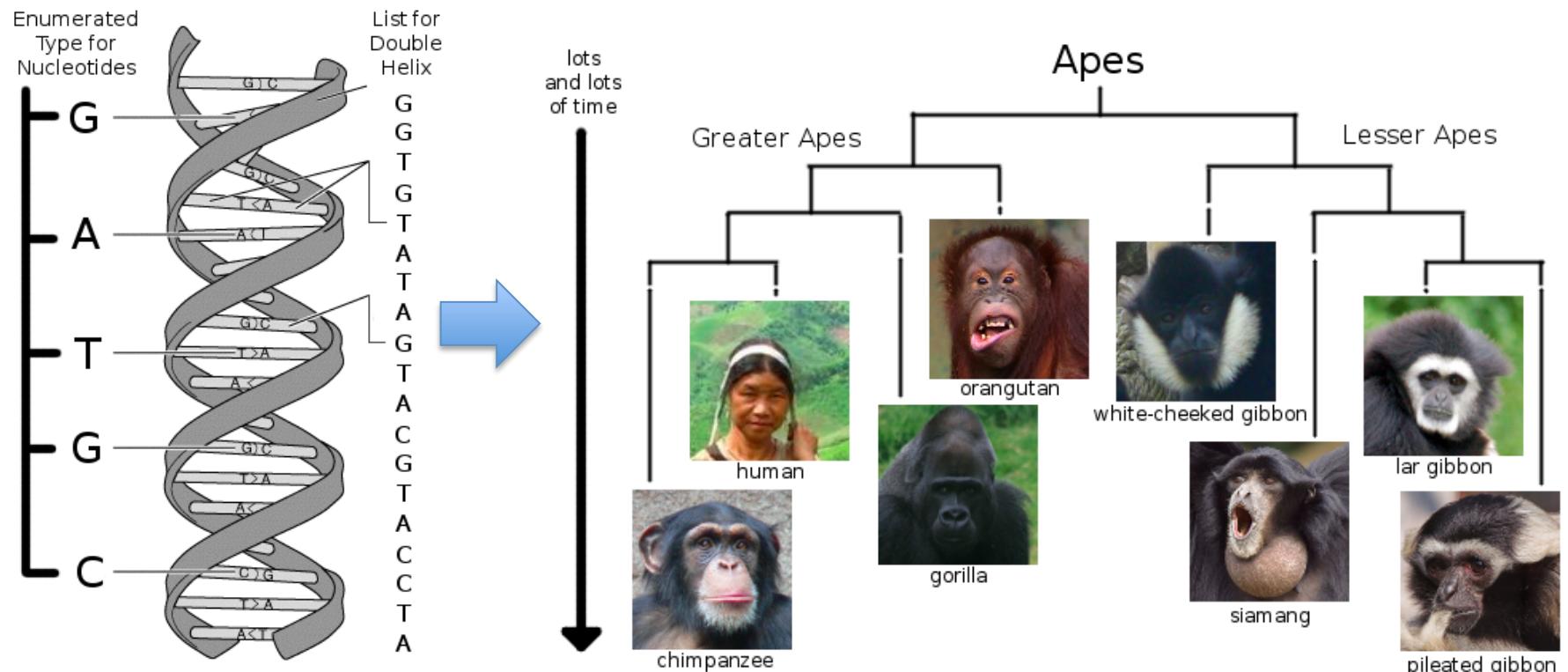
# Datatypes and Trees

# Building Datatypes

- Programming languages provide a variety of ways of creating and manipulating structured data
- We have already seen:
  - *primitive datatypes* (int, string, bool, ... )
  - *lists* (int list, string list, string list list, ... )
  - *tuples* (int \* int, int \* string, ...)
- Rest of Today:
  - user-defined datatypes
  - type abbreviations

# HW 2 Case Study: Evolutionary Trees

- Problem: reconstruct evolutionary trees\* from DNA data.
  - What are the relevant abstractions?
  - How can we use the language features to define them?
  - How do the abstractions help shape the program?



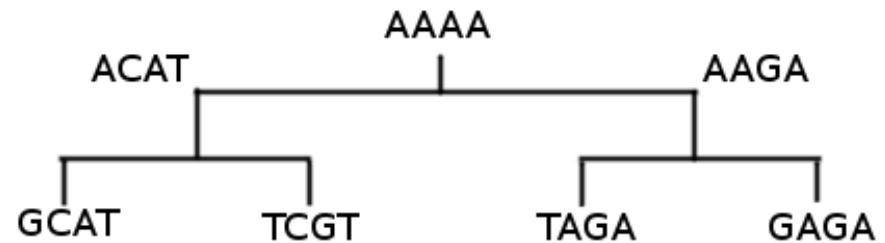
\*Interested? Check out this suggested reading:

Dawkins: *The Ancestor's Tale: A Pilgrimage to the Dawn of Evolution*

CIS120

# DNA Computing Abstractions

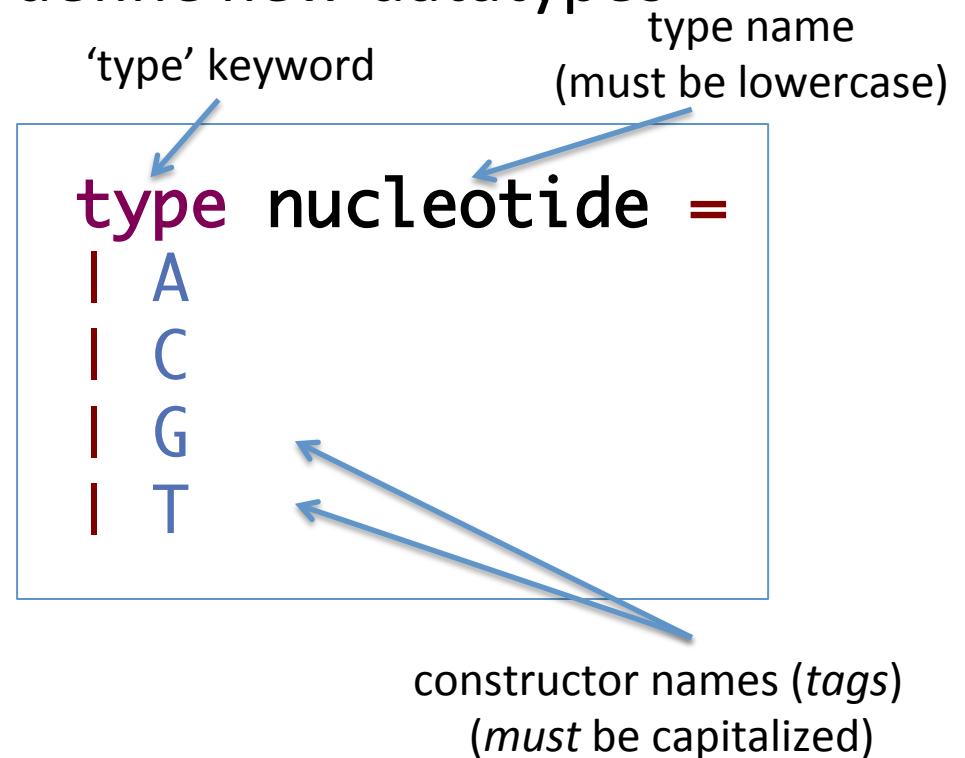
- Nucleotide
  - Adenine (A), Guanine (G), Thymine (T), or Cytosine (C)
- Helix
  - a sequence of nucleotides: e.g. AGTCCGATTACAGAGA...
  - genetic code for a particular species (human, gorilla, etc)
- Phylogenetic tree
  - Binary tree with helices (species) at the nodes and leaves



# Simple User-Defined Datatypes

- OCaml lets programmers define *new* datatypes

```
type day =
| Sunday
| Monday
| Tuesday
| Wednesday
| Thursday
| Friday
| Saturday
```



- The constructors *are* the values of the datatype
  - e.g. A *is* a nucleotide and [A; G; C] *is* a nucleotide list

# Pattern Matching Simple Datatypes

- Datatype values can be analyzed by pattern matching:

```
let string_of_n (n:nucleotide) : string =
begin match n with
| A -> "adenine"
| C -> "cytosine"
| G -> "guanine"
| T -> "thymine"
end
```

- There is one case per constructor
  - you will get a warning if you leave out a case or list one twice
- As with lists, the pattern syntax follows that of the datatype values (i.e. the constructors)

# A Point About Abstraction

- We *could* represent data like this by using integers:
  - Sunday = 0, Monday = 1, Tuesday = 2, etc.
- But:
  - Integers support different operations than days do:  
Wednesday - Monday = Tuesday (?!)
  - There are *more* integers than days (What day is 17?)
- Confusing integers with days can lead to bugs
  - Many *scripting* languages (PHP, Javascript, Perl, Python,...) violate such abstractions (`true == 1 == "1"`), leading to pain and misery...

Most modern languages (Java, C#, C++, OCaml,...) provide user-defined types for this reason.

# Type Abbreviations

- OCaml also lets us *name* types **without** making new abstractions:

```
type helix = nucleotide list
type codon = nucleotide * nucleotide
              * nucleotide
```

The diagram shows two type definitions. The first definition, `type helix = nucleotide list`, has a blue arrow pointing from the word "type" to the start of the line, labeled "type keyword". The second definition, `type codon = nucleotide * nucleotide * nucleotide`, has a blue arrow pointing from the word "type" to the start of the line, labeled "type name". A large blue bracket is positioned under the three `nucleotide` terms in the `codon` definition. Below this bracket, a blue arrow points upwards to the text "definition in terms of existing types no constructors!".

type keyword

type name

definition in terms of existing types  
no constructors!

- i.e. a codon is the same thing a triple of nucleotides  
`let x : codon = (A,C,C)`
- Makes code easier to read & write

# Data-Carrying Constructors

- Datatype constructors can also carry values

```
type measurement =  
| Missing  
| NucCount of nucleotide * int  
| CodonCount of codon * int
```

keyword 'of'

Constructors may take a tuple of arguments

- Values of type ‘measurement’ include:

Missing

NucCount(A, 3)

CodonCount((A,G,T), 17)

# Pattern Matching Datatypes

- Pattern matching notation combines syntax of tuples and simple datatype constructors:

```
let get_count (m:measurement) : int =
  begin match m with
    | Missing          -> 0
    | NucCount(_, n)  -> n
    | CodonCount(_, n) -> n
  end
```

- Datatype patterns *bind* variables (e.g. ‘n’) just like lists and tuples

```
type nucleotide = | A | C | G | T  
type helix = nucleotide list
```

What is the type of this expression?

(A, "A")

1. nucleotide
2. nucleotide list
3. helix
4. nucleotide \* string
5. string \* string
6. *none (expression is ill typed)*

Answer: 4

# Recursive User-defined Datatypes

- Datatypes can mention themselves!

```
type tree =
  | Leaf of helix
  | Node of tree * helix * tree
```

base constructor  
(nonrecursive)

Node carries a  
tuple of values

recursive  
definition

- Recursive datatypes can be taken apart by pattern matching (and recursive functions).

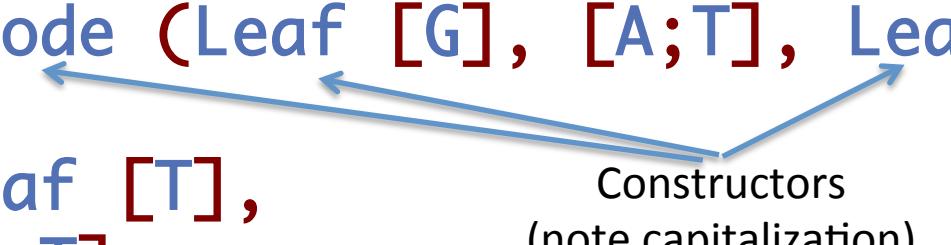
# Syntax for User-defined Types

```
type tree =
| Leaf of helix
| Node of tree * helix * tree
```

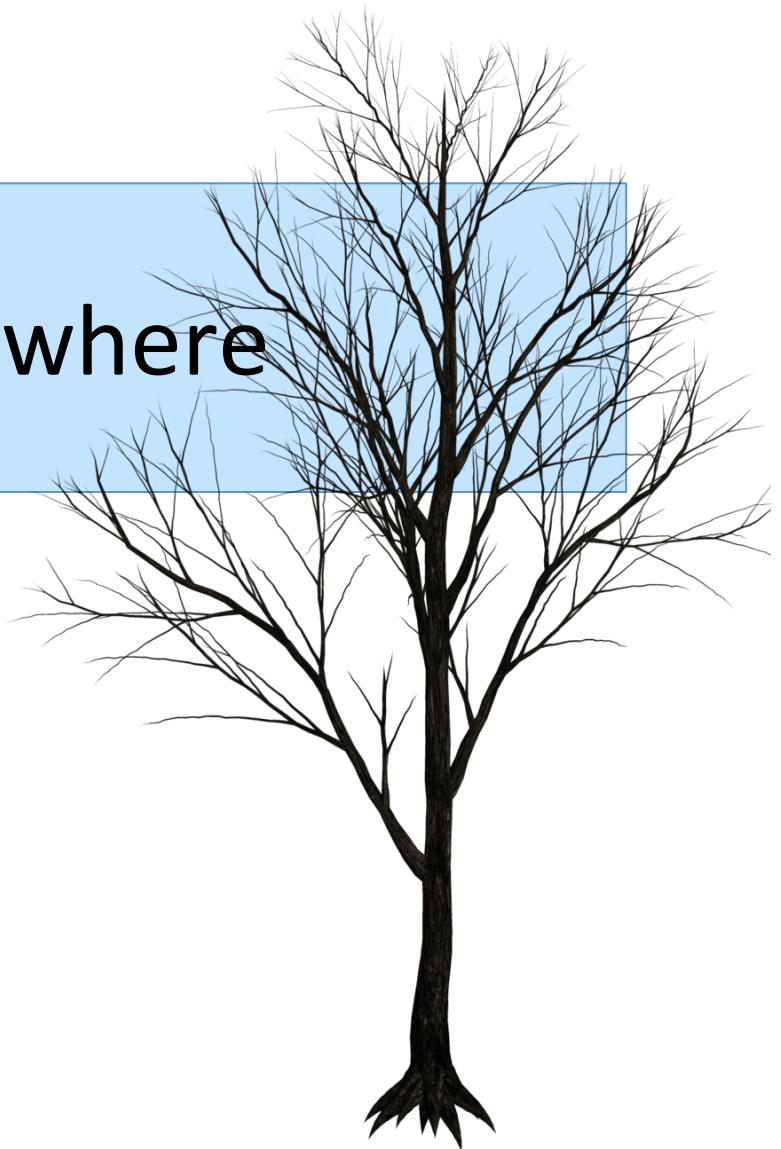
- Example values of type `tree`

```
let t1 = Leaf [A;G]
let t2 = Node (Leaf [G], [A;T], Leaf [A])
let t3 =
  Node (Leaf [T],
        [T;T],
        Node (Leaf [G;C], [G], Leaf []))
```

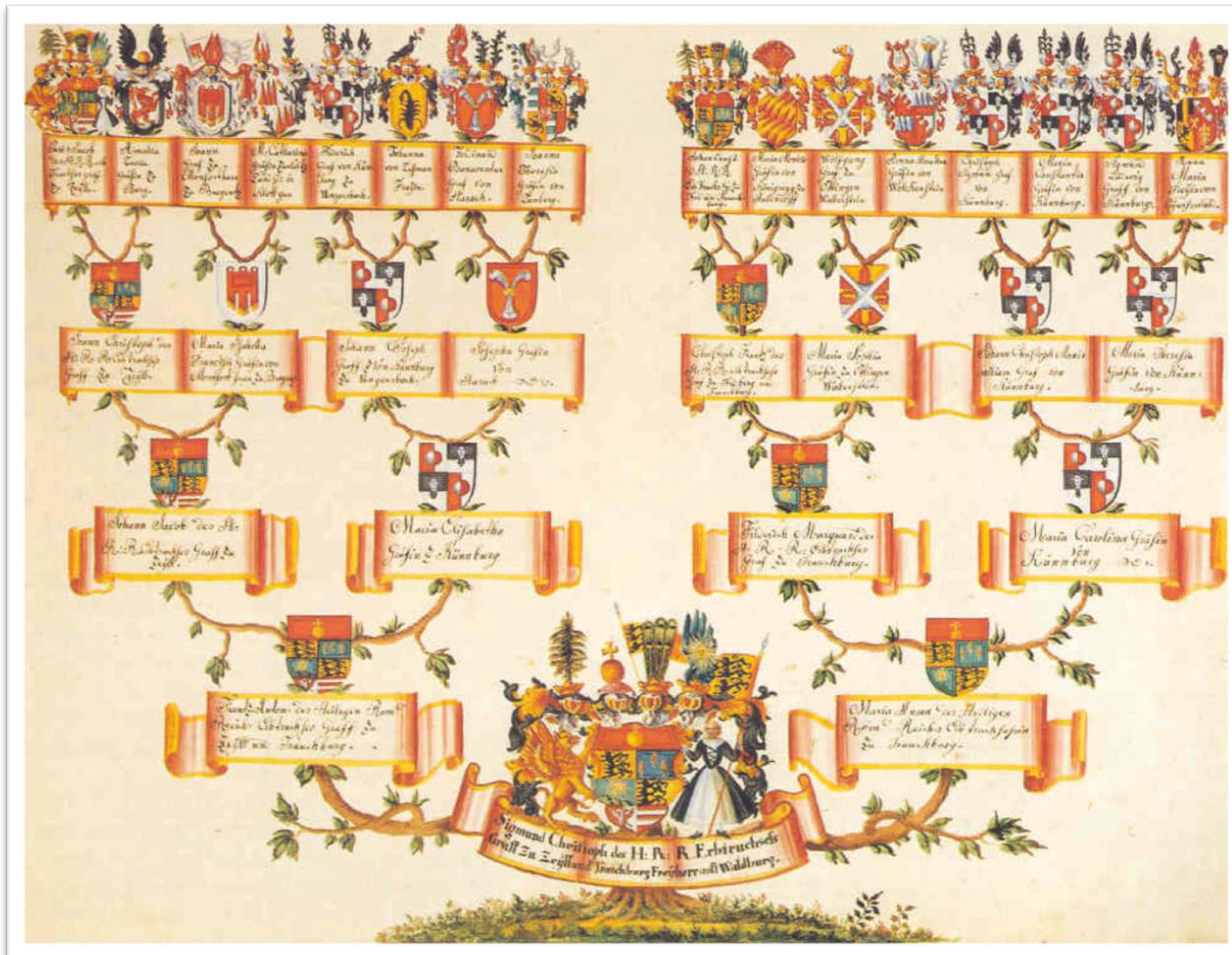
Constructors  
(note capitalization)



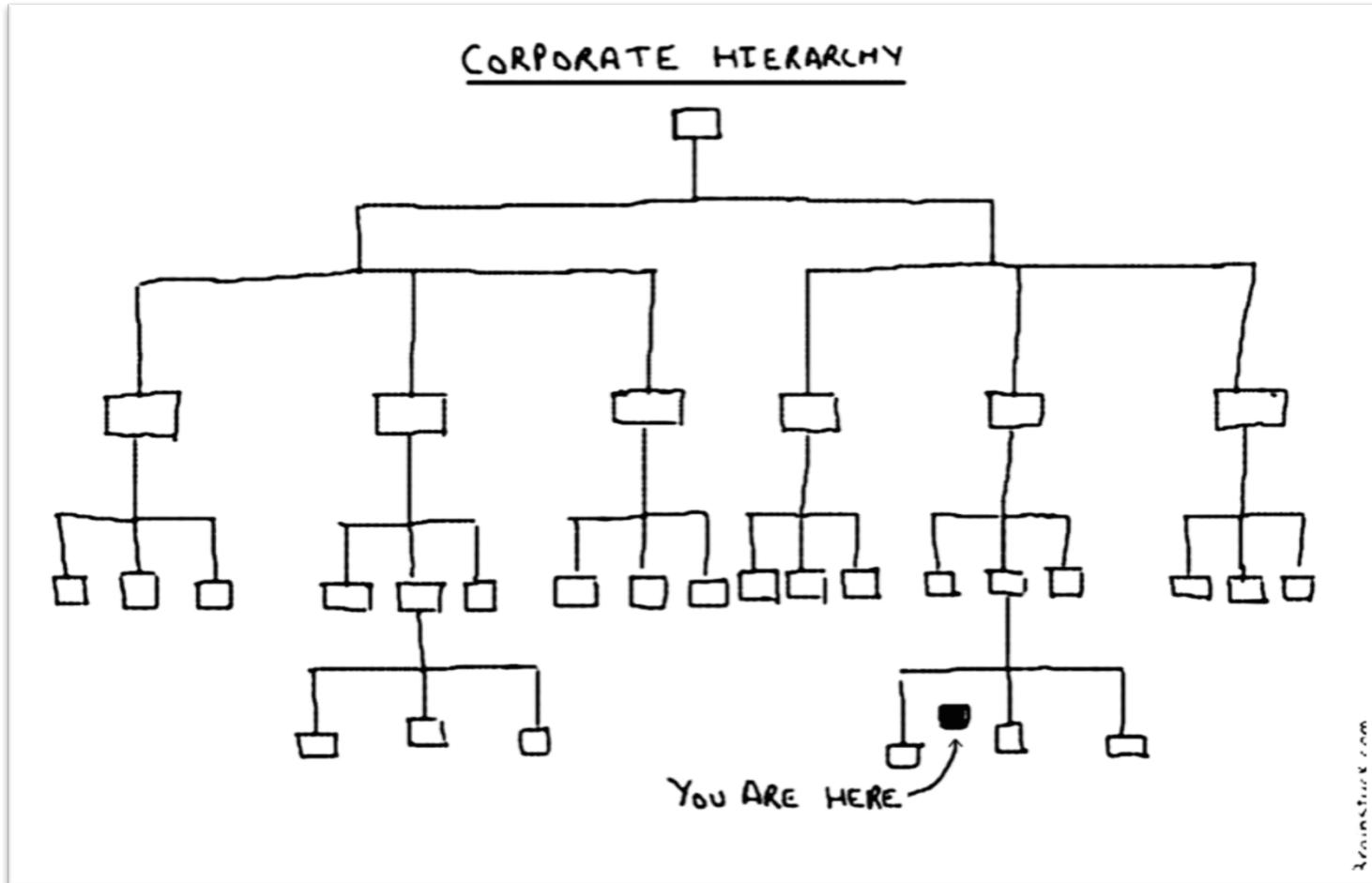
Trees are everywhere



# Family trees

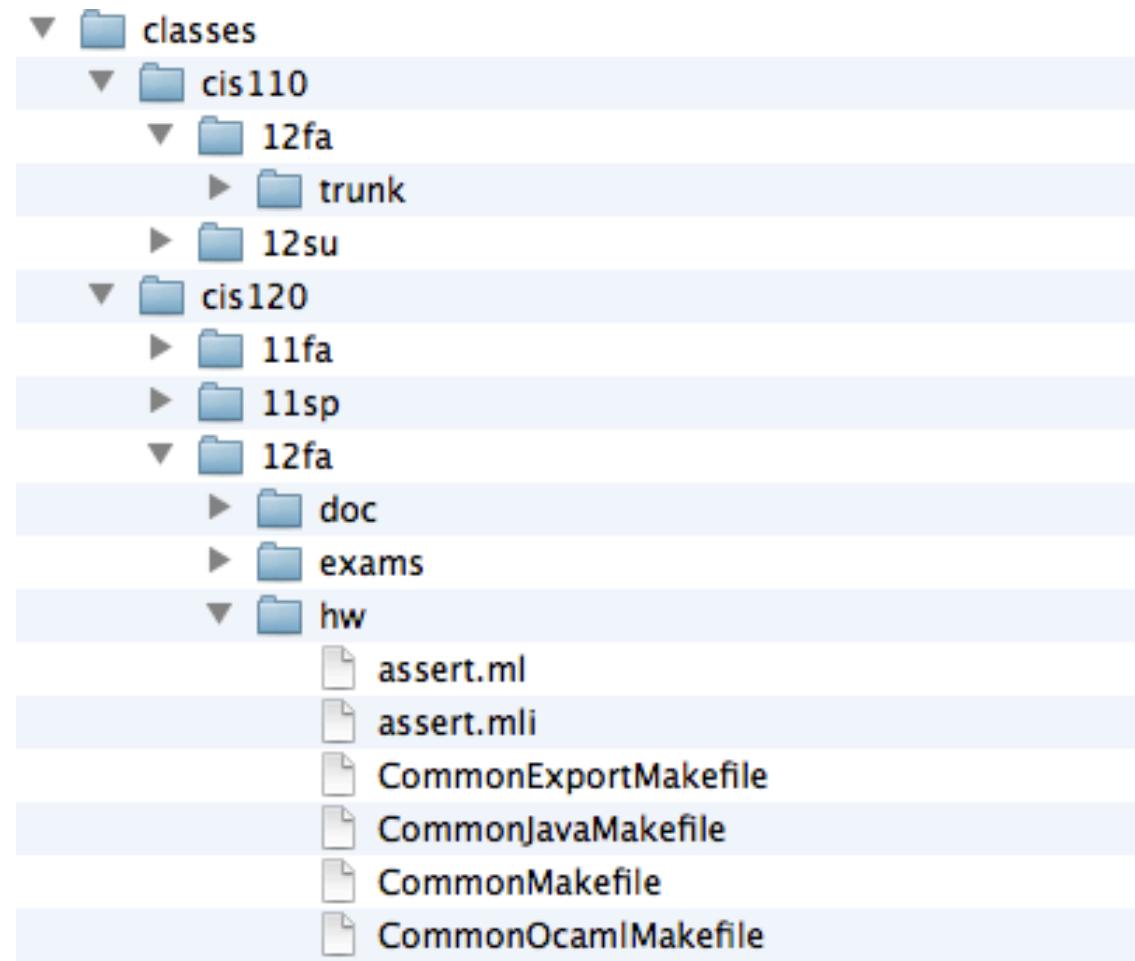


# Organizational charts

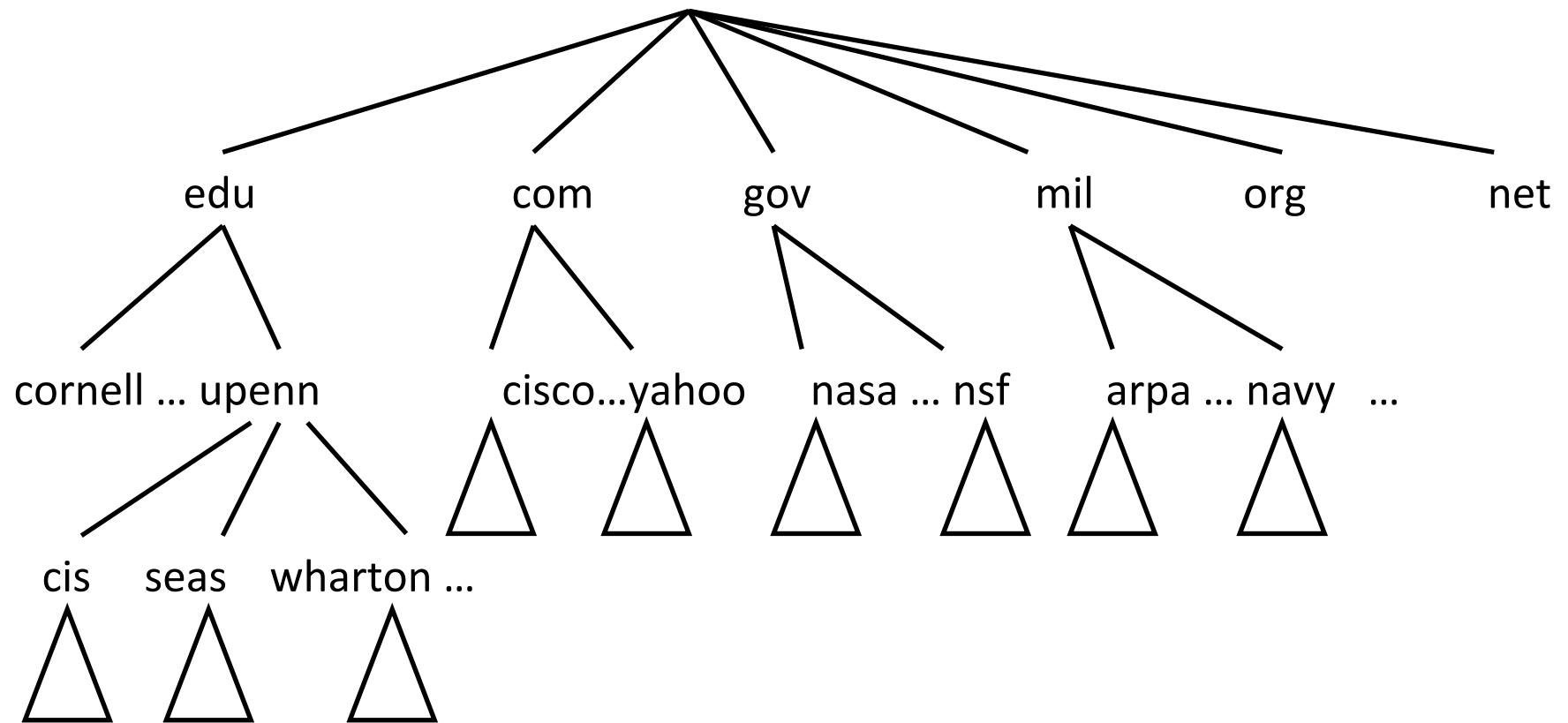


2023-2024 © CIS120

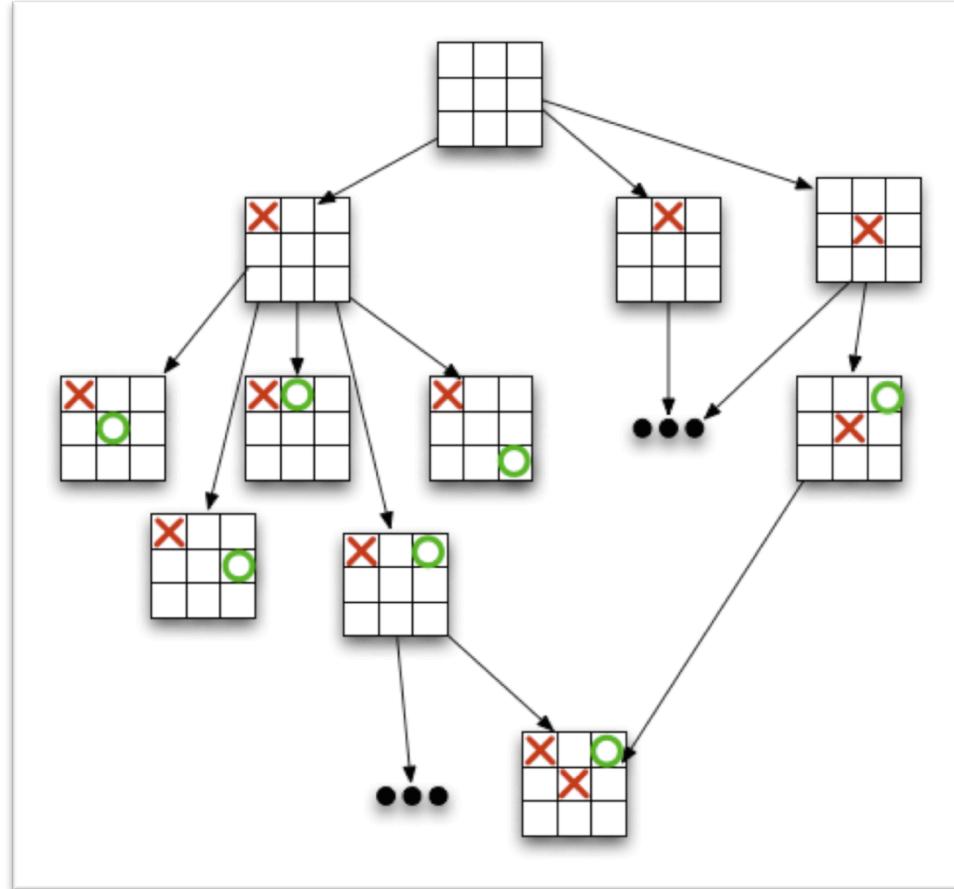
# Filesystem Directory Structure



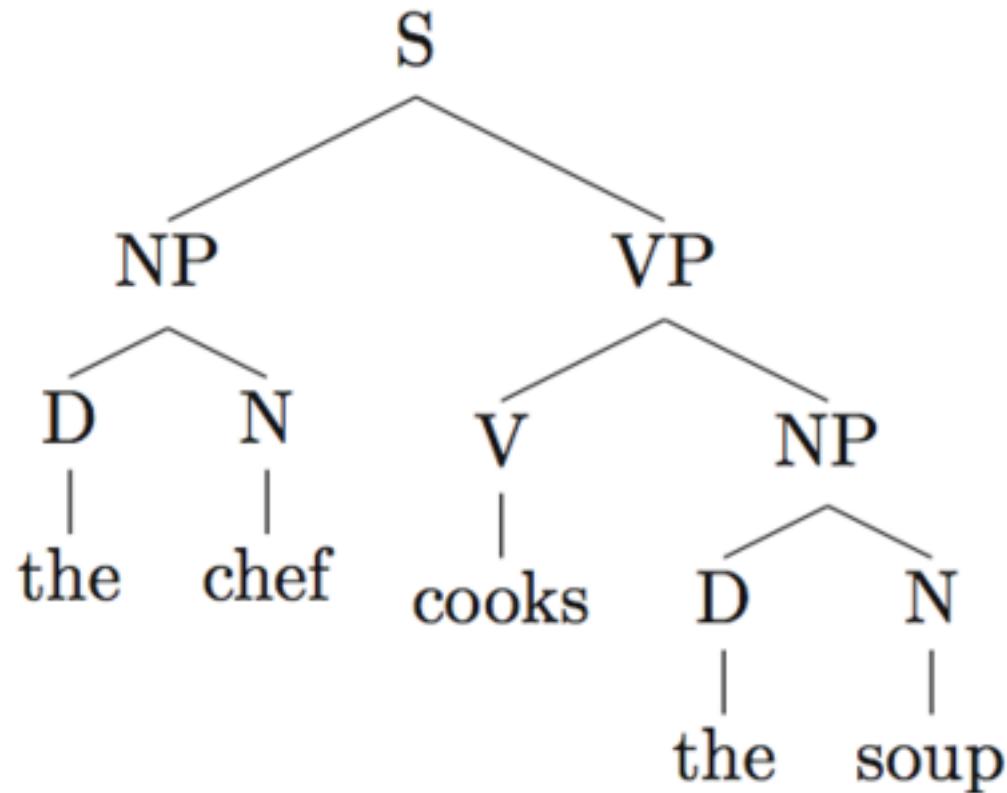
# Domain Name Hierarchy



# Game trees



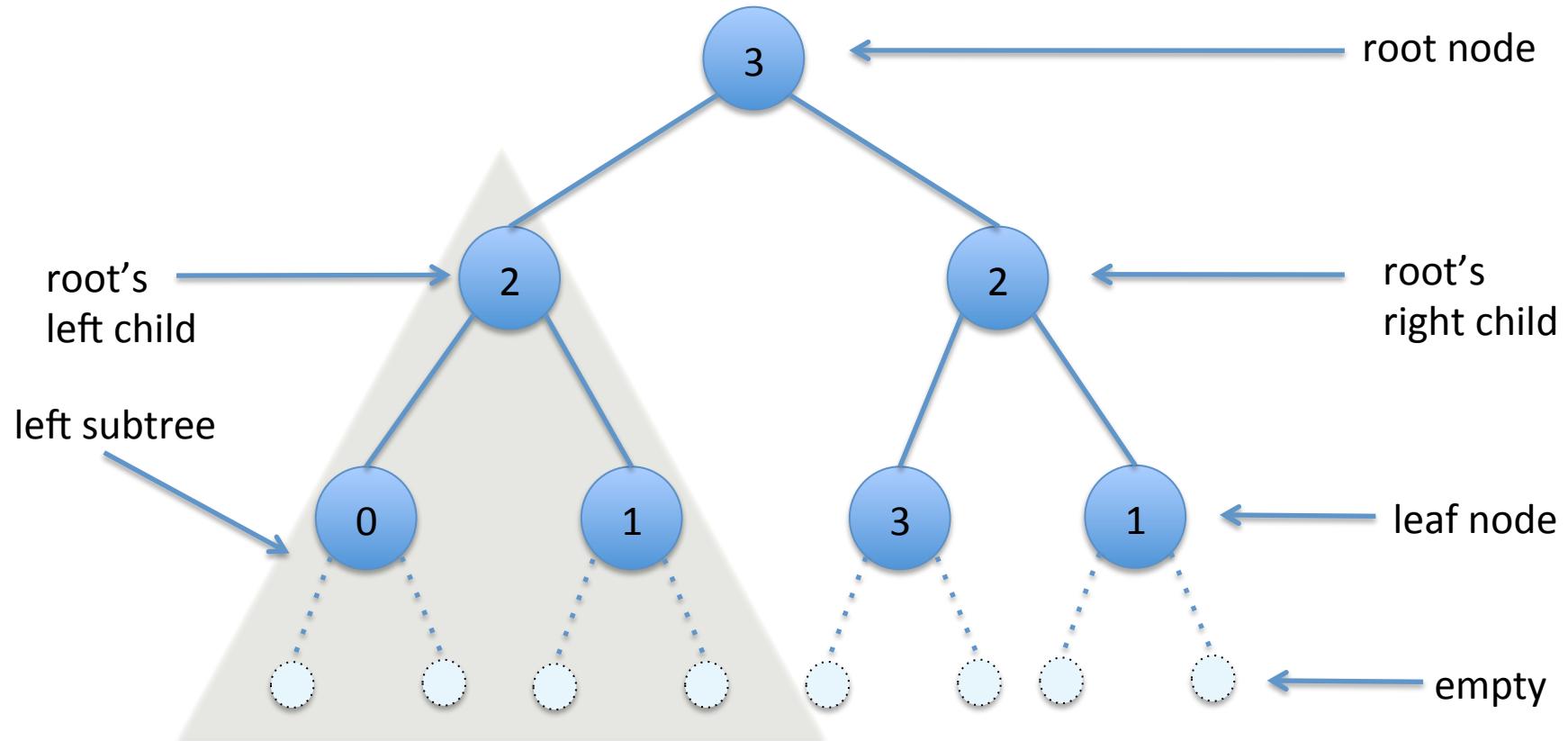
# Natural-Language Parse Trees



# Binary Trees

A particular form of tree-structured data

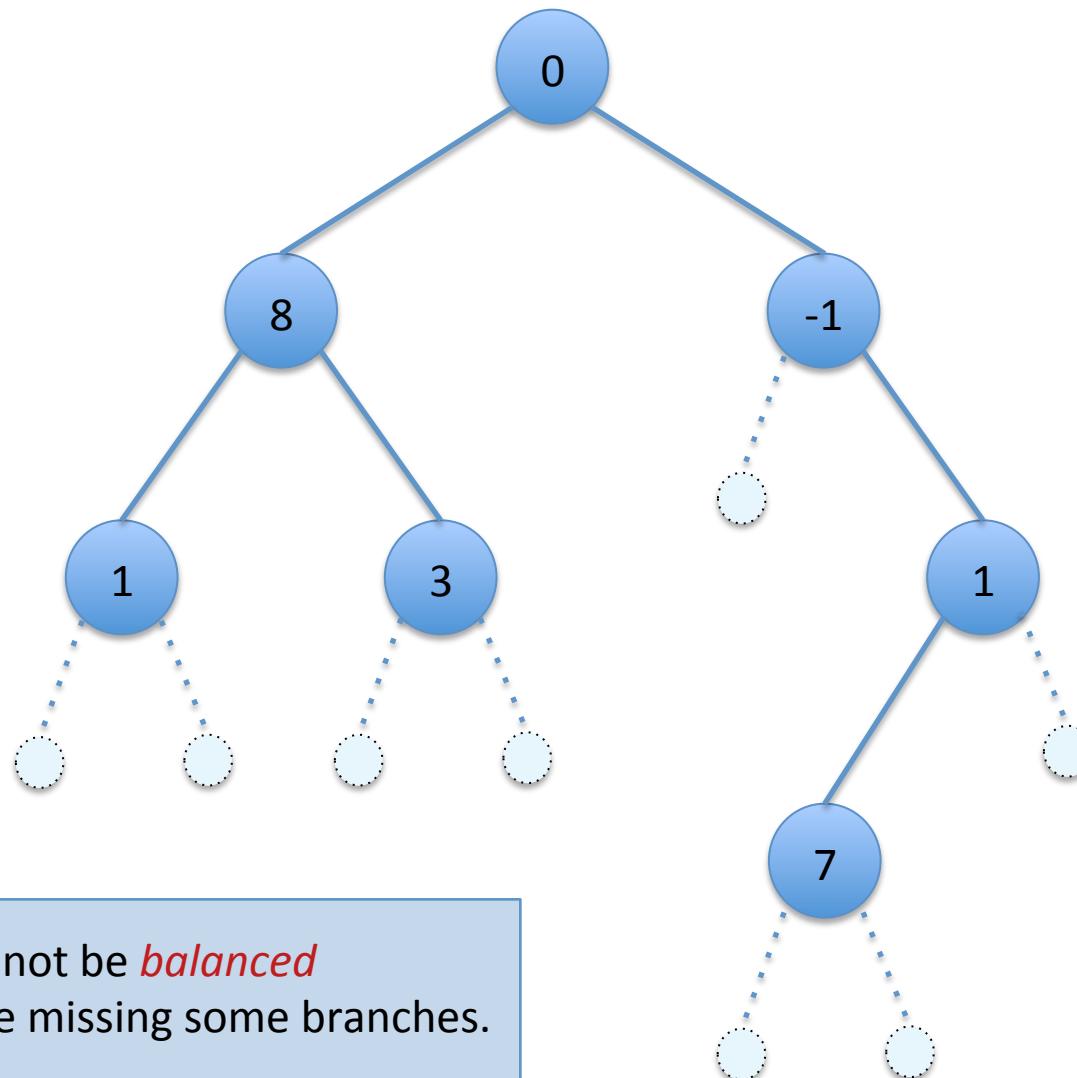
# Binary Trees



A binary tree is either *empty*, or a *node* with at most two children, both of which are also binary trees.

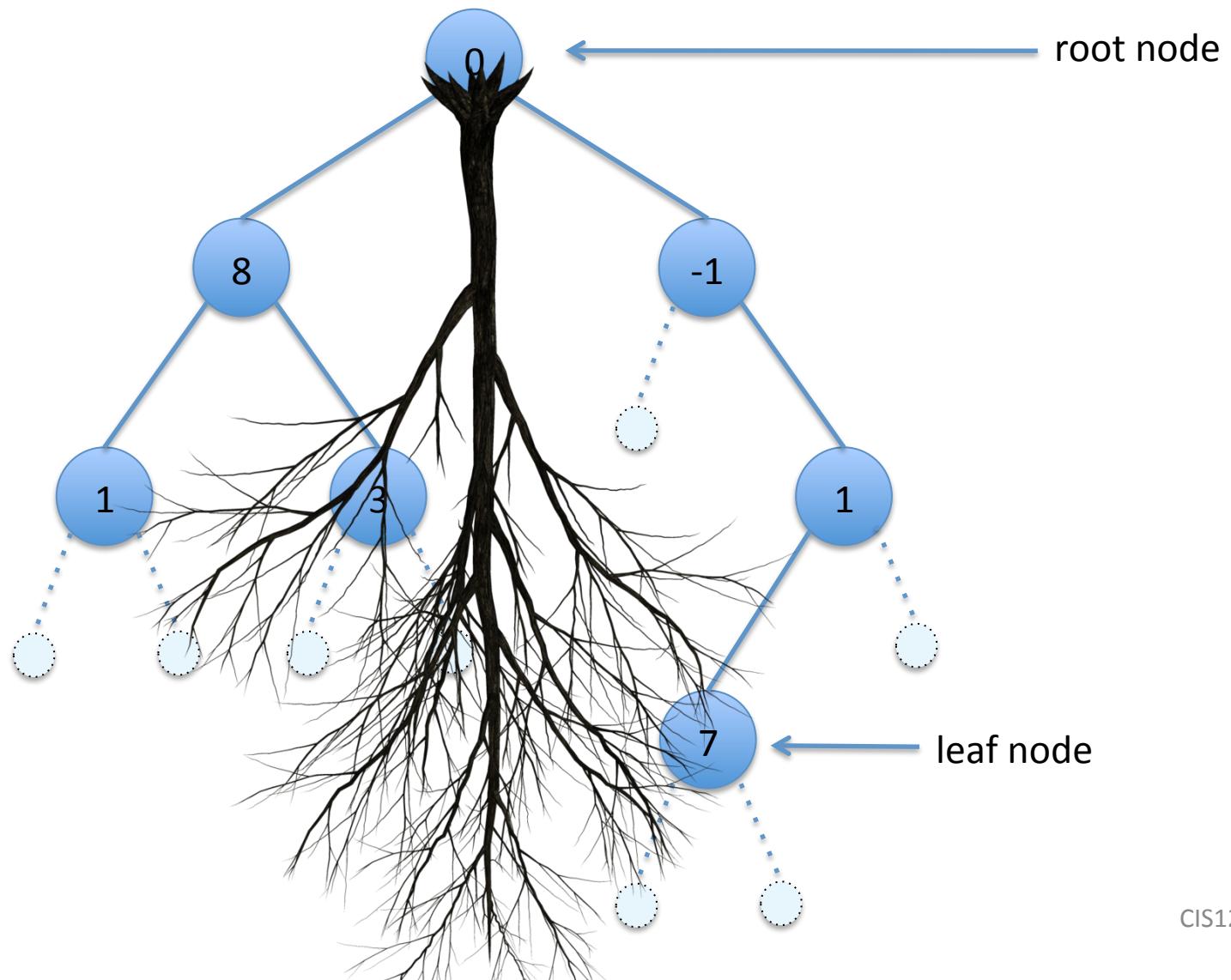
A *leaf* is a node whose children are both *empty*.

# Another Example Tree



Trees need not be *balanced*  
they may be missing some branches.

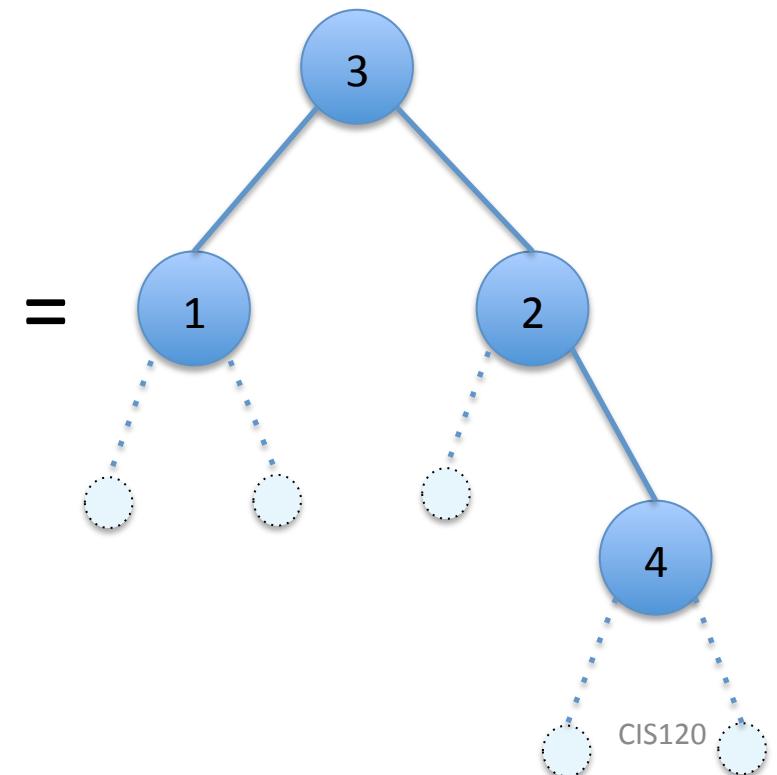
# Drawn Upside Down



# Binary Trees in OCaml

```
type tree =
| Empty
| Node of tree * int * tree
```

```
let t : tree =
Node (Node (Empty, 1, Empty),
3,
Node (Empty, 2,
Node (Empty, 4, Empty)))
```



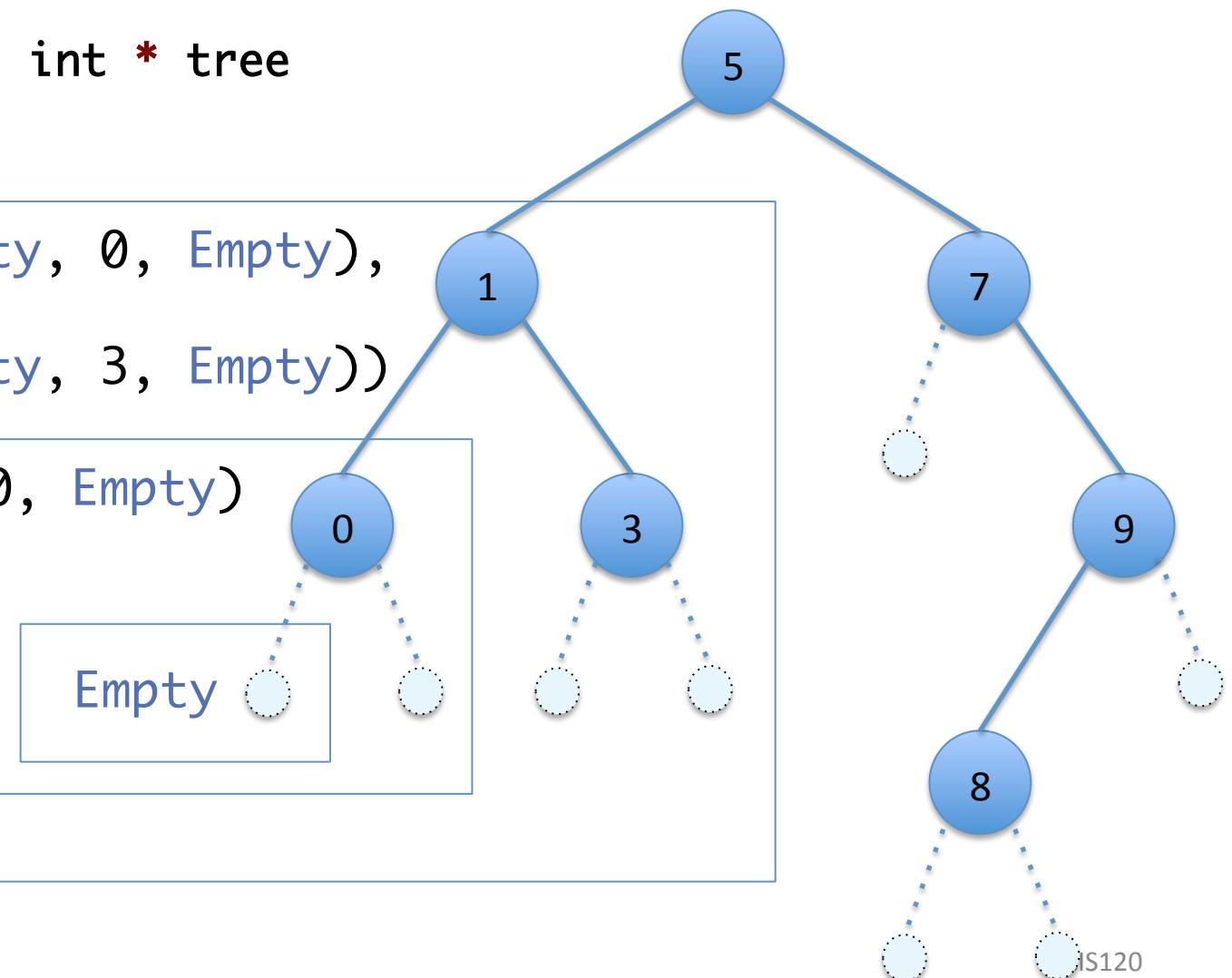
# Representing trees

```
type tree =  
| Empty  
| Node of tree * int * tree
```

```
Node (Node (Empty, 0, Empty),  
      1,  
      Node (Empty, 3, Empty))
```

```
Node (Empty, 0, Empty)
```

Empty

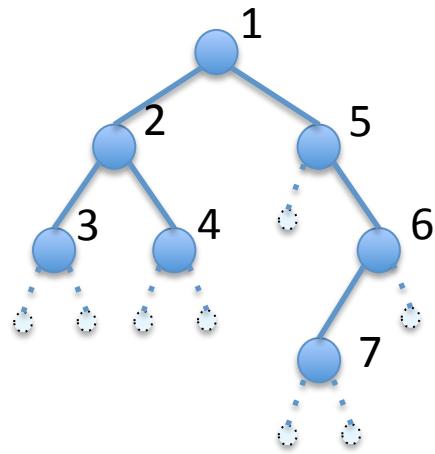


## More on trees

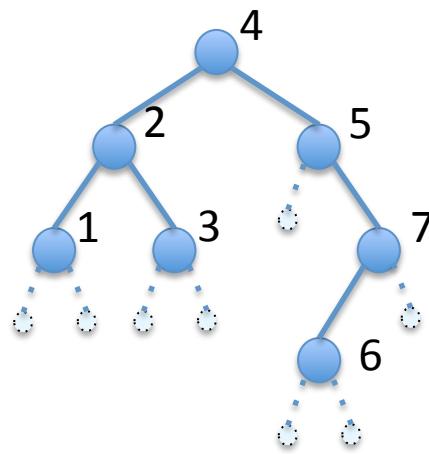
see tree.ml

treeExamples.ml

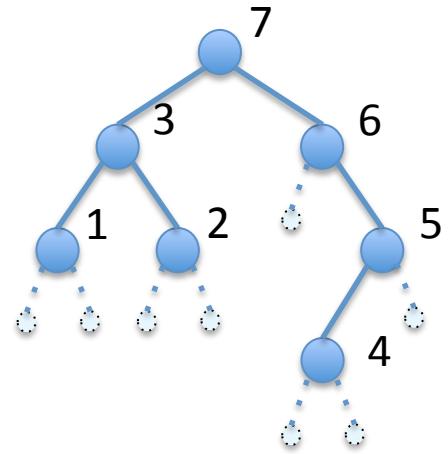
# Recursive Tree Traversals



Pre-Order  
Root – Left – Right



In Order  
Left – Root – Right



Post-Order  
Left – Right – Root

```
(* Code for Pre-Order Traversal *)
let rec f (t:tree) : ... =
begin match t with
| Empty -> ...
| Node(l, x, r) ->
  let root = ... x ... in (* process root *)
  let left = f l in (* recursively process left *)
  let right = f r in (* recursively process right *)
  combine root left right
end
```

The traversals vary the order in which these are computed...

# Trees as Containers

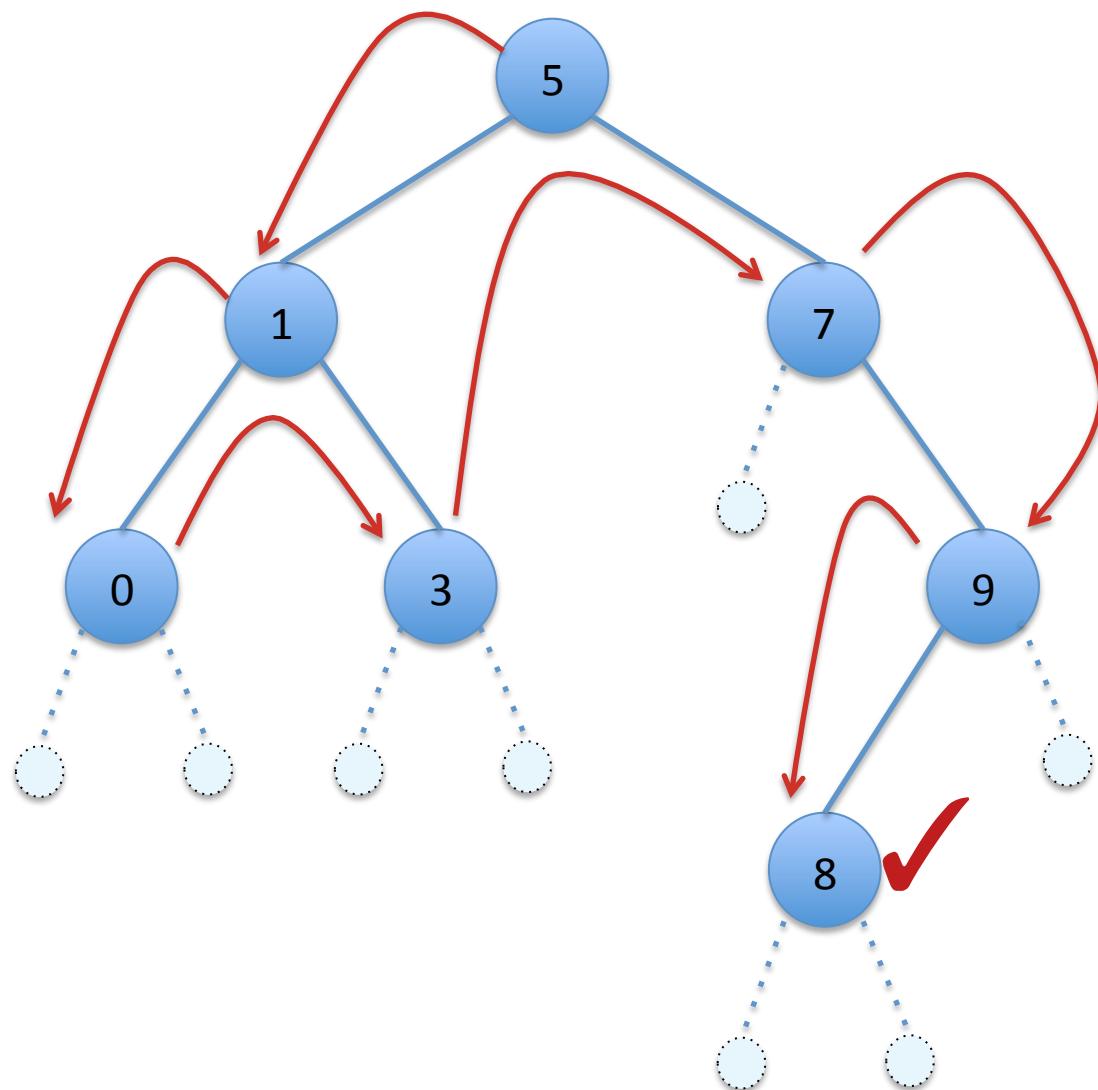
- Like lists, trees aggregate ordered data
- As we did for lists, we can write a function to determine whether the data structure *contains* a particular element

# Searching for Data in a Tree

```
let rec contains (t:tree) (n:int) : bool =
begin match t with
| Empty -> false
| Node(lt,x,rt) -> x = n || (contains lt n) || (contains rt n)
end
```

- This function searches through the tree, looking for n
- In the worst case, it might have to traverse the *entire* tree

# Search during (contains t 8)



# Programming Languages and Techniques (CIS120)

## Lecture 6

September 13th, 2017

Trees, and Binary Search Trees  
(Lecture notes Chapter 7)

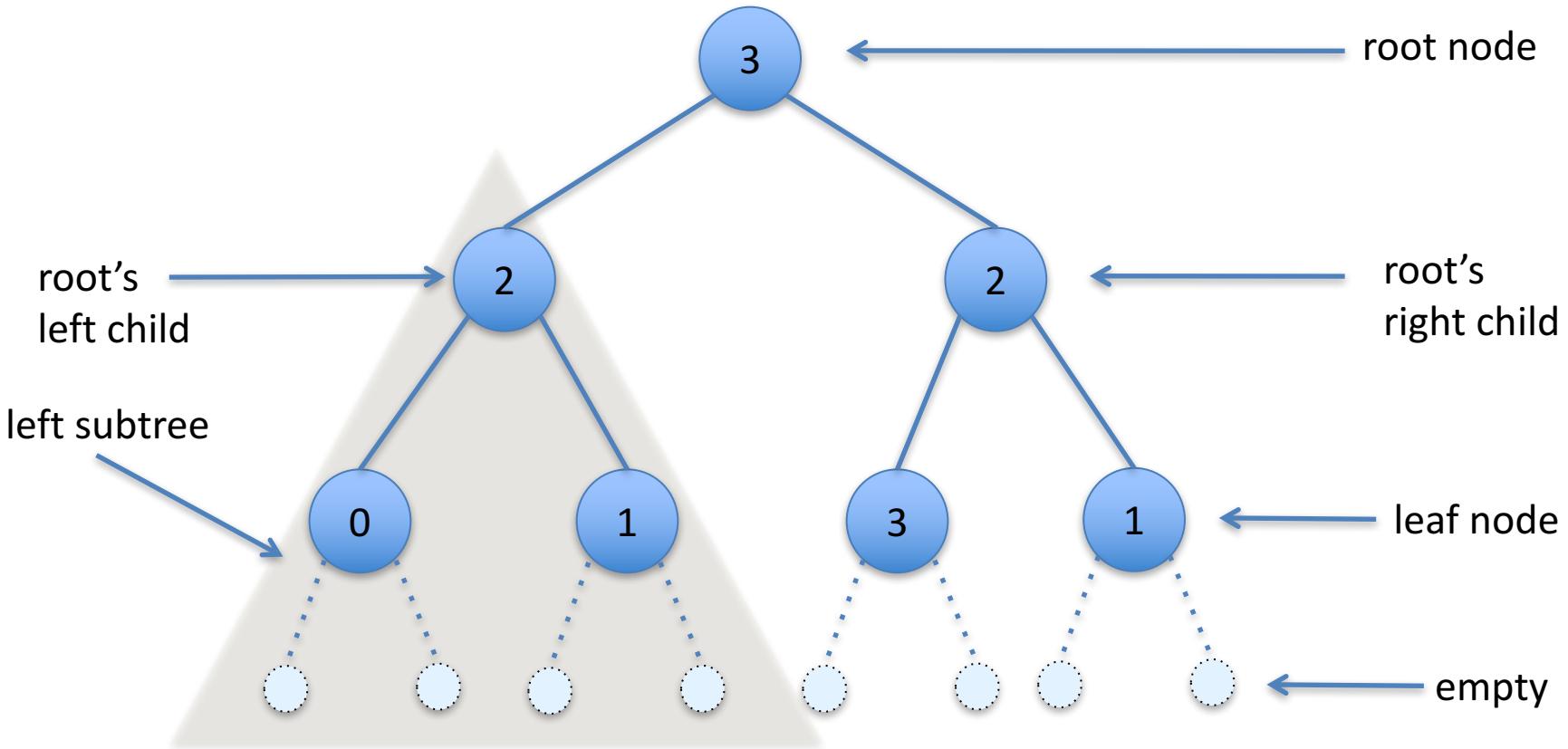
# Announcements

- Homework 2: Computing Human Evolution
  - due Tuesday, September 19<sup>th</sup>
- Reading: Chapter 7
- Please Complete the Entry Survey
  - See the link on Piazza
- Dr. Zdancewic will be away on Friday
  - Dr. Sheth will cover the 11:00am class.

# Binary Trees

trees with (at most) two branches

# Binary Trees



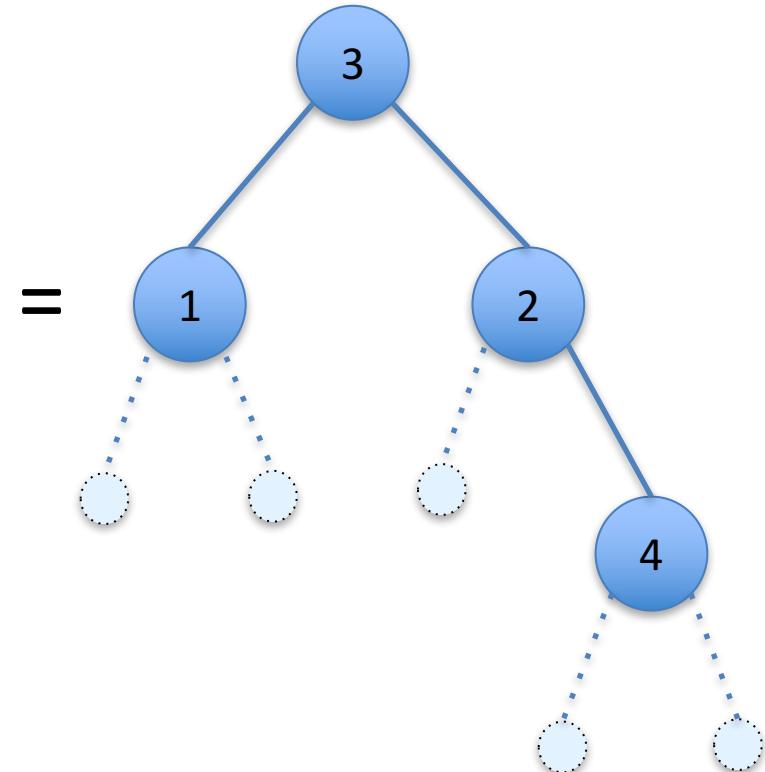
A binary tree is either *empty*, or a *node* with at most two children, both of which are also binary trees.

A *leaf* is a node whose children are both empty.

# Binary Trees in OCaml

```
type tree =
| Empty
| Node of tree * int * tree
```

```
let t : tree =
Node (Node (Empty, 1, Empty),
3,
Node (Empty, 2,
Node (Empty, 4, Empty)))
```



# More Tree Coding

example: size

# Trees as Containers

# Trees as Containers

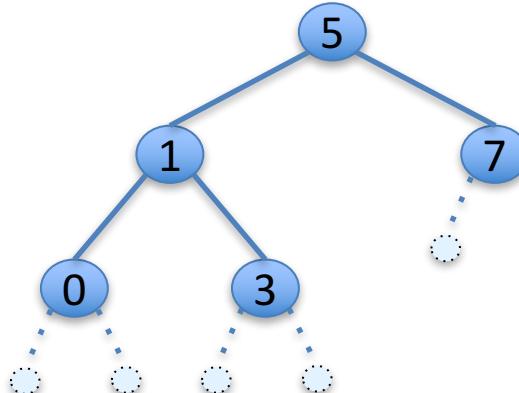
- Like lists, binary trees aggregate data
- As we did for lists, we can write a function to determine whether the data structure *contains* a particular element

```
type tree =
| Empty
| Node of tree * int * tree
```

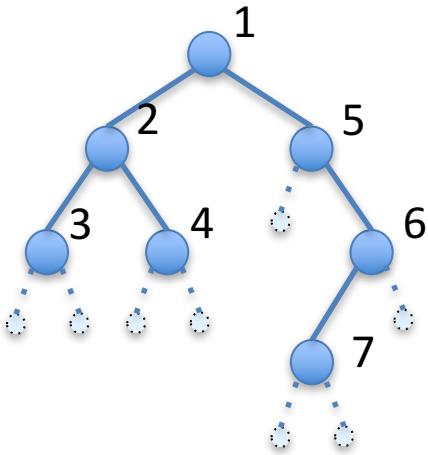
# Searching for Data in a Tree

```
let rec contains (t:tree) (n:int) : bool =
  begin match t with
  | Empty -> false
  | Node(lt,x,rt) -> x = n
    || (contains lt n) || (contains rt n)
  end
```

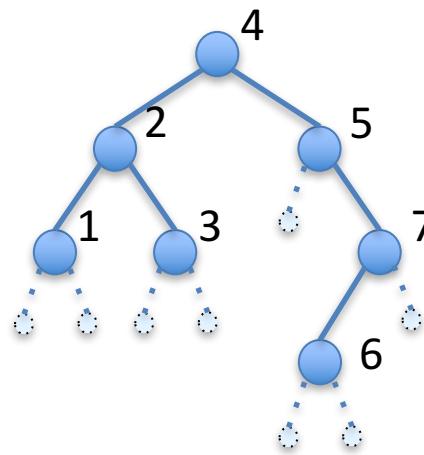
- This function searches through the tree, looking for n
- In the worst case, it might have to traverse the *entire* tree



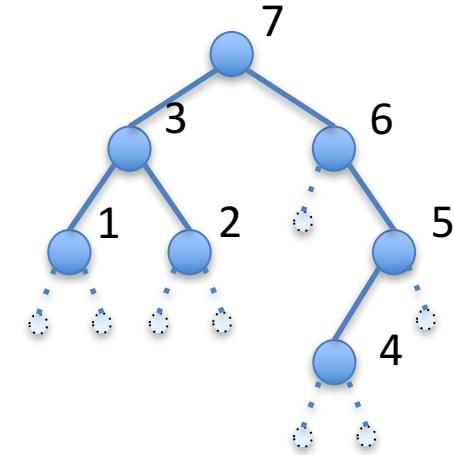
# Recursive Tree Traversals



Pre-Order  
Root – Left – Right



In Order  
Left – Root – Right



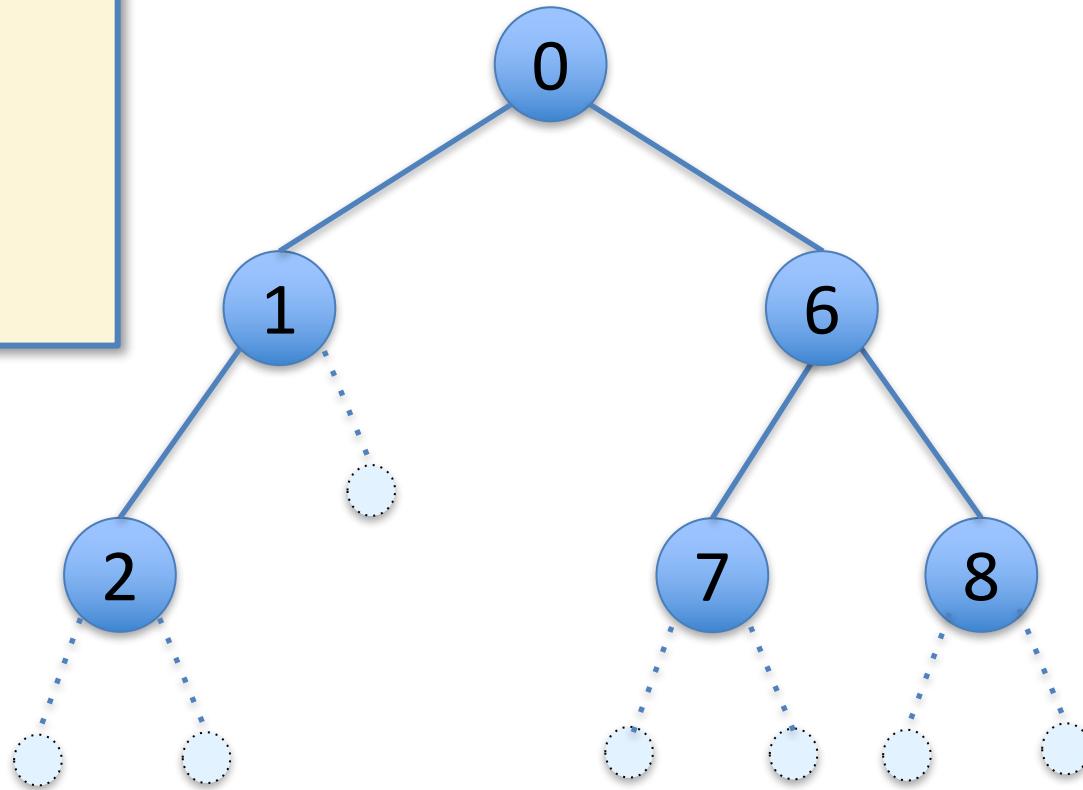
Post-Order  
Left – Right – Root

```
(* Pre-Order Traversal: *)
let rec f (t:tree) : ... =
begin match t with
| Empty -> ...
| Node(l, x, r) ->
  let root = ... x ... in (* process root *)
  let left = f l in (* recursively process left *)
  let right = f r in (* recursively process right *)
  combine root left right
end
```

Different traversals vary the order in which these are computed...

In what sequence will the nodes of this tree be visited by a post-order traversal?

1. [0;1;6;2;7;8]
2. [0;1;2;6;7;8]
3. [2;1;0;7;6;8]
4. [7;8;6;2;1;0]
5. [2;1;7;8;6;0]



Post-Order  
Left – Right – Root

Answer: 5

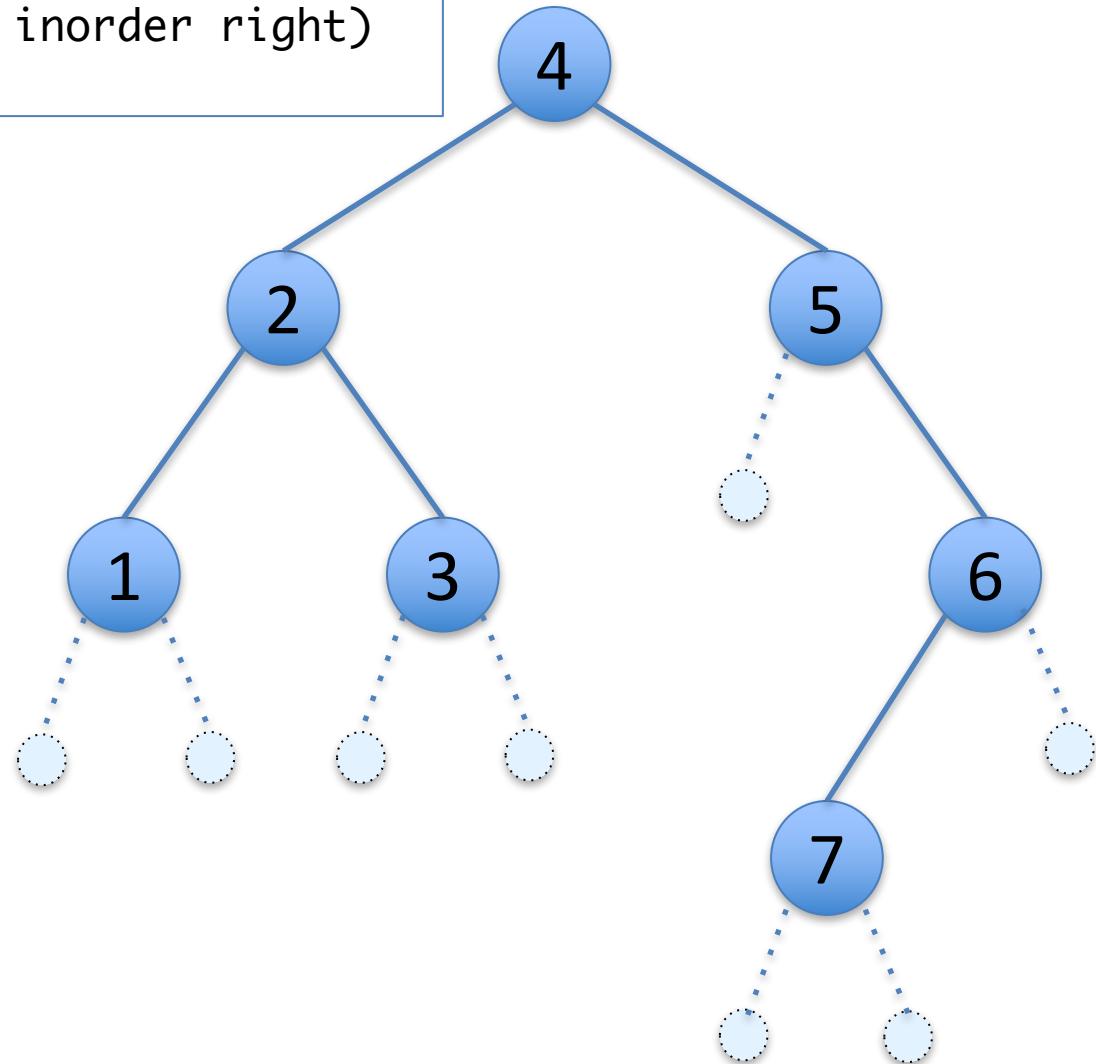
```

let rec inorder (t:tree) : int list =
begin match t with
| Empty -> []
| Node (left, x, right) ->
  inorder left @ (x :: inorder right)
end

```

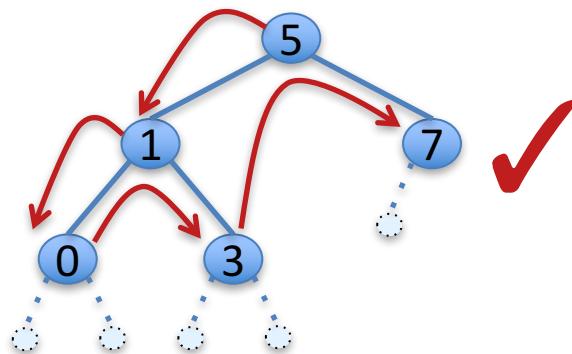
What is the result of applying this function on this tree?

1. []
2. [1;2;3;4;5;6;7]
3. [1;2;3;4;5;7;6]
4. [4;2;1;3;5;6;7]
5. [4]
6. [1;1;1;1;1;1;1]
7. none of the above



Answer: 3

# Searching for Data in a Tree



```
let rec contains (t:tree) (n:int) : bool =
begin match t with
| Empty -> false
| Node(lt,x,rt) -> x = n || (contains lt n) || (contains rt n)
end
```

```
contains (Node(Node(Node (Empty, 0, Empty), 1, Node(Empty, 3, Empty)),
      5, Node (Empty, 7, Empty))) 7
```

```
5 = 7
|| contains (Node(Node (Empty, 0, Empty), 1, Node(Empty, 3, Empty))) 7
|| contains (Node (Empty, 7, Empty)) 7
```

```
(1 = 7 || contains (Node (Empty, 0, Empty)) 7
    || contains (Node(Empty, 3, Empty)) 7)
|| contains (Node (Empty, 7, Empty)) 7
```

```
((0 = 7 || contains Empty 7 || contains Empty 7)
    || contains (Node(Empty, 3, Empty)) 7)
|| contains (Node (Empty, 7, Empty)) 7
```

```
contains (Node(Empty, 3, Empty)) 7
|| contains (Node (Empty, 7, Empty)) 7
```

```
contains (Node (Empty, 7, Empty)) 7
```

# Ordered Trees

Big idea: find things faster by searching less

## *Key Insight:*

*Ordered data can be searched more quickly*

- This is why telephone books are arranged alphabetically
- But requires the ability to focus on (roughly) *half* of the current data

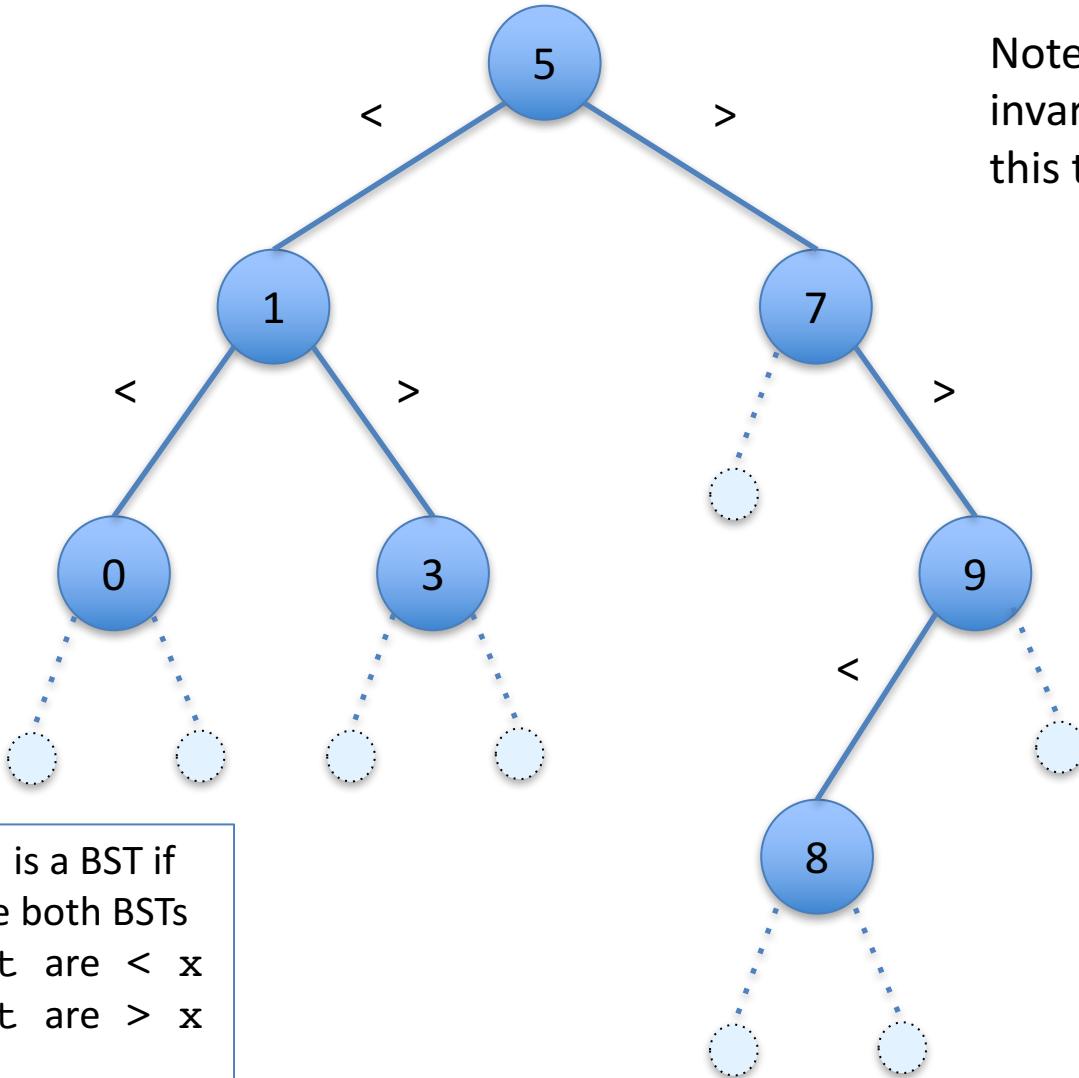
# Binary Search Trees

- A *binary search tree* (BST) is a binary tree with some additional *invariants*\*:
  - $\text{Node}(lt, x, rt)$  is a BST if
    - $lt$  and  $rt$  are both BSTs
    - all nodes of  $lt$  are  $< x$
    - all nodes of  $rt$  are  $> x$
  - $\text{Empty}$  is a BST
- *The BST invariant means that container functions can take time proportional to the **height** instead of the **size** of the tree.*

\*An data structure *invariant* is a set of constraints about the way that the data is organized.

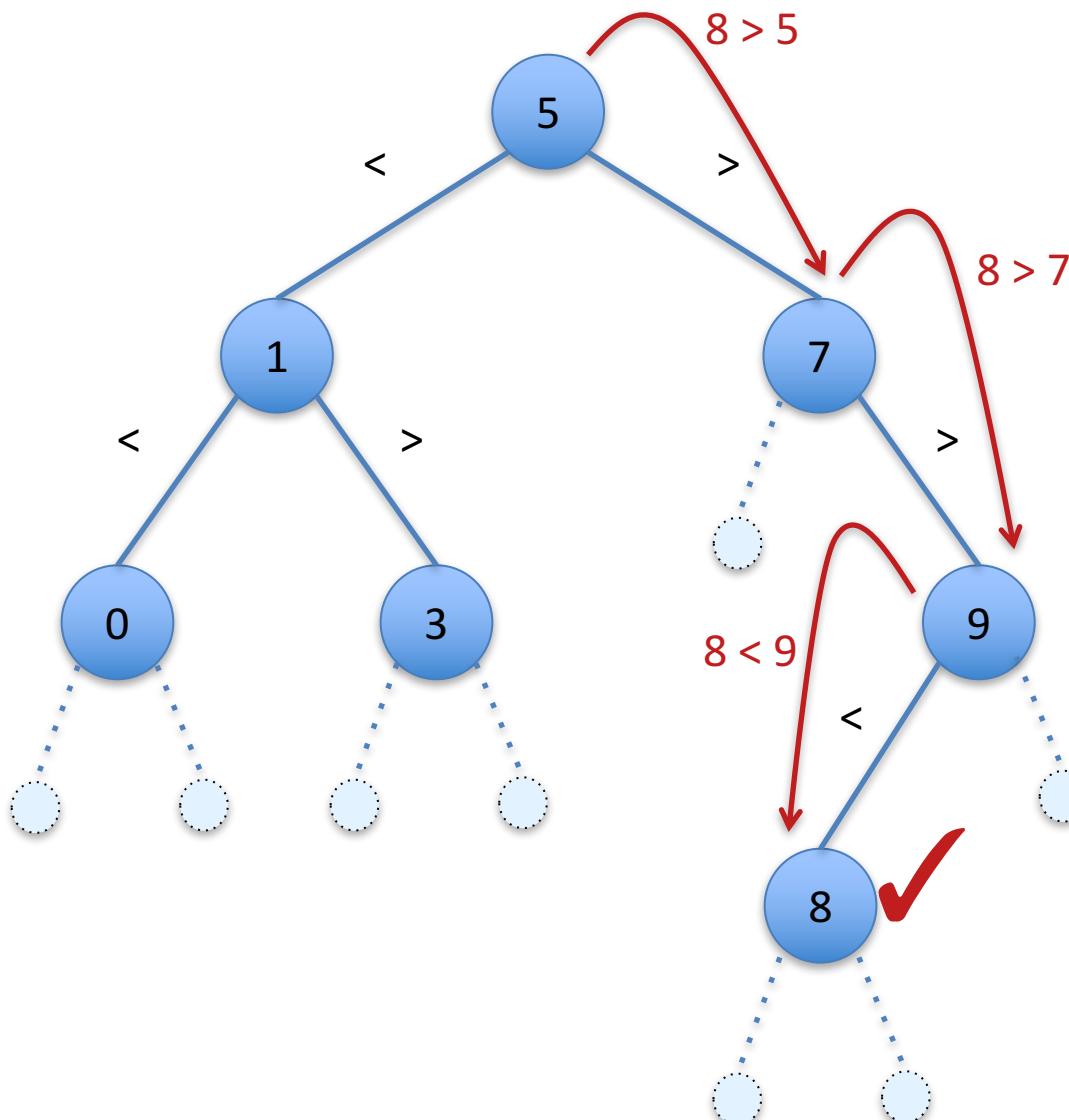
“types” (e.g. list or tree) are one kind of invariant, but we often impose additional constraints.

# An Example Binary Search Tree



- `Node(lt, x, rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are  $< x$
  - all nodes of `rt` are  $> x$
- `Empty` is a BST

# Search in a BST: ( lookup t = 8 )



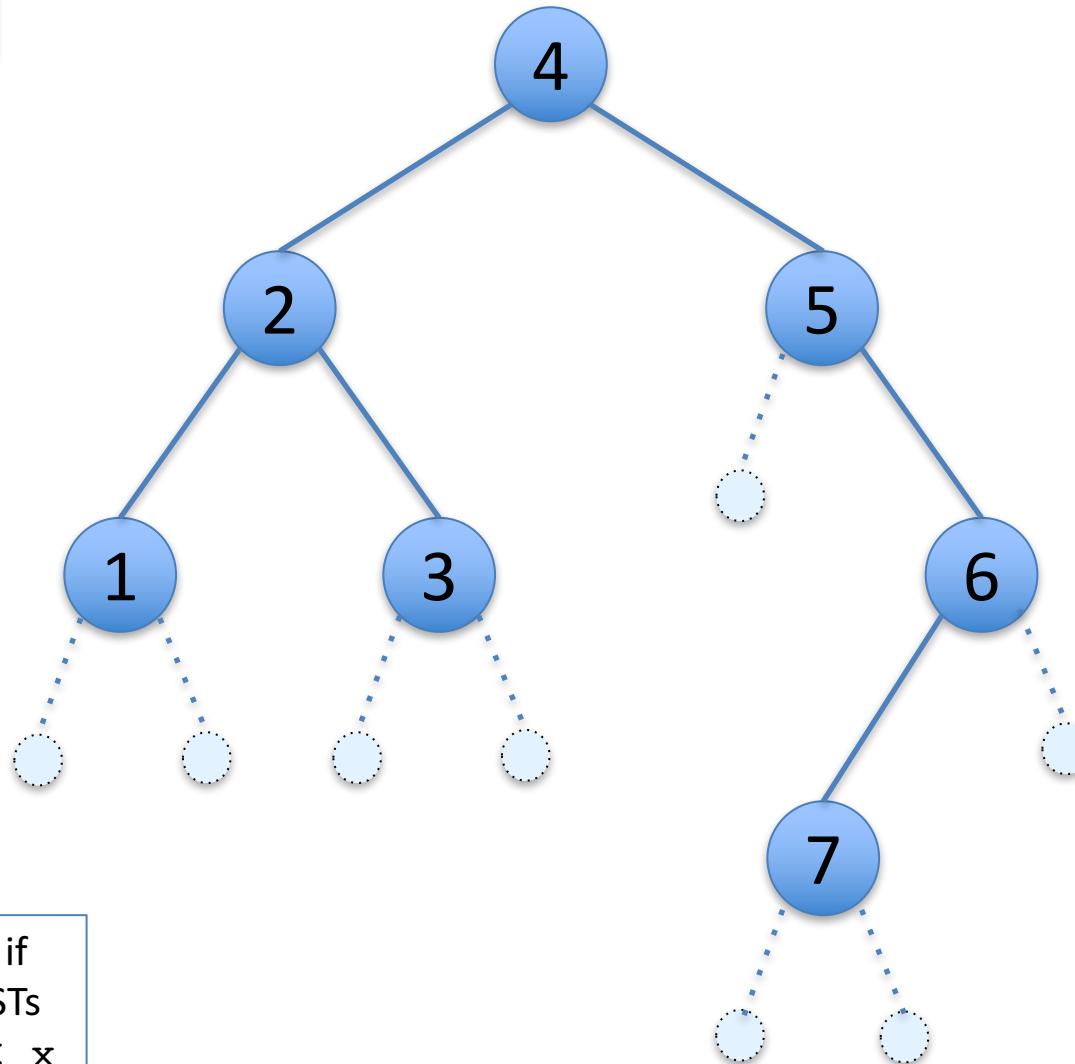
# Searching a BST

```
(* Assumes that t is a BST *)
let rec lookup (t:tree) (n:int) : bool =
begin match t with
| Empty -> false
| Node(lt,x,rt) ->
    if x = n then true
    else if n < x then lookup lt n
    else lookup rt n
end
```

- The BST invariants guide the search.
- Note that `lookup` may return an incorrect answer if the input is *not* a BST!
  - This function *assumes* that the BST invariants hold of `t`.

Is this a BST??

1. yes
2. no

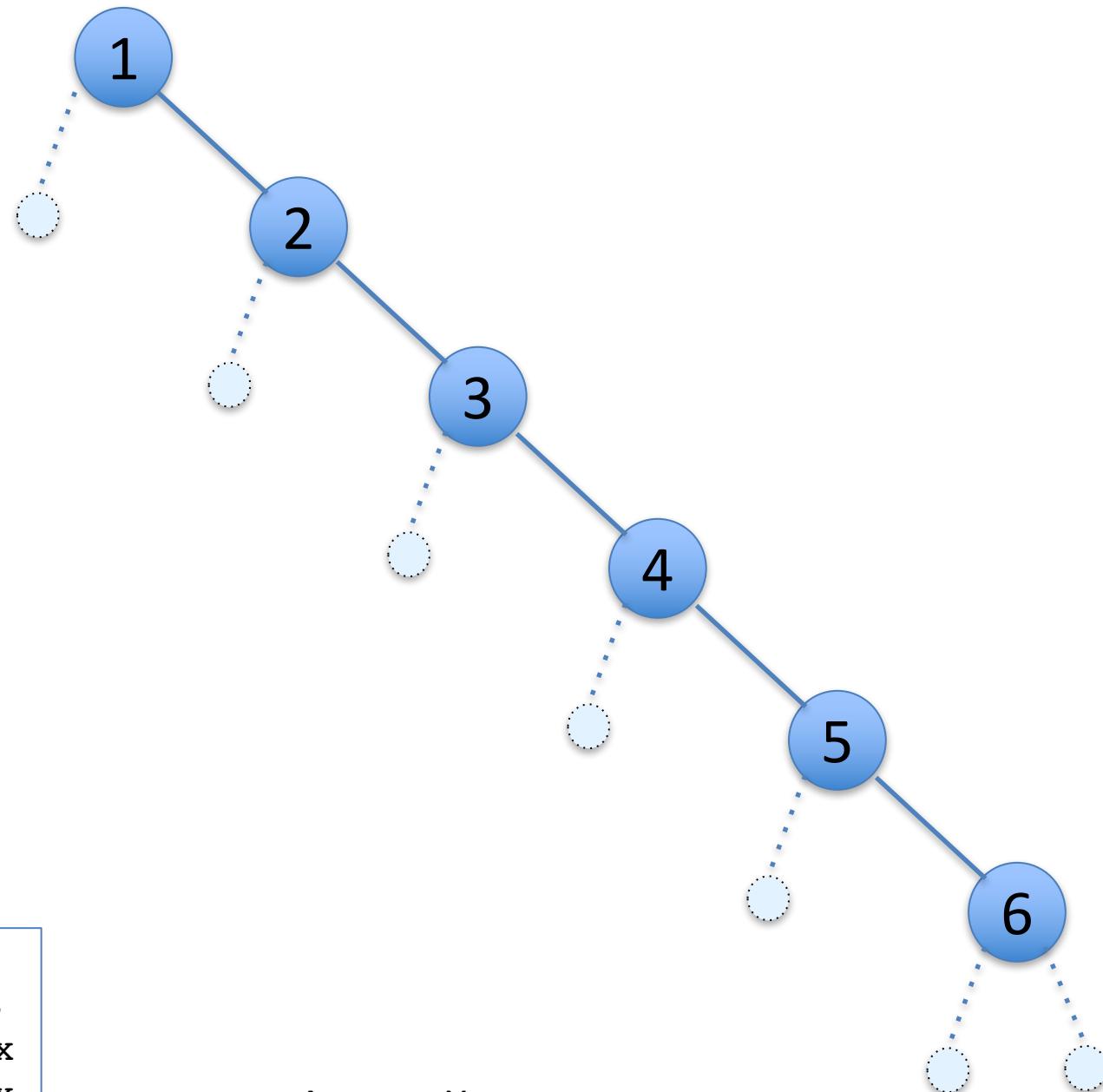


Answer: no, 7 to the left of 6

- `Node(lt, x, rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are  $< x$
  - all nodes of `rt` are  $> x$
- Empty is a BST

Is this a BST??

1. yes
2. no

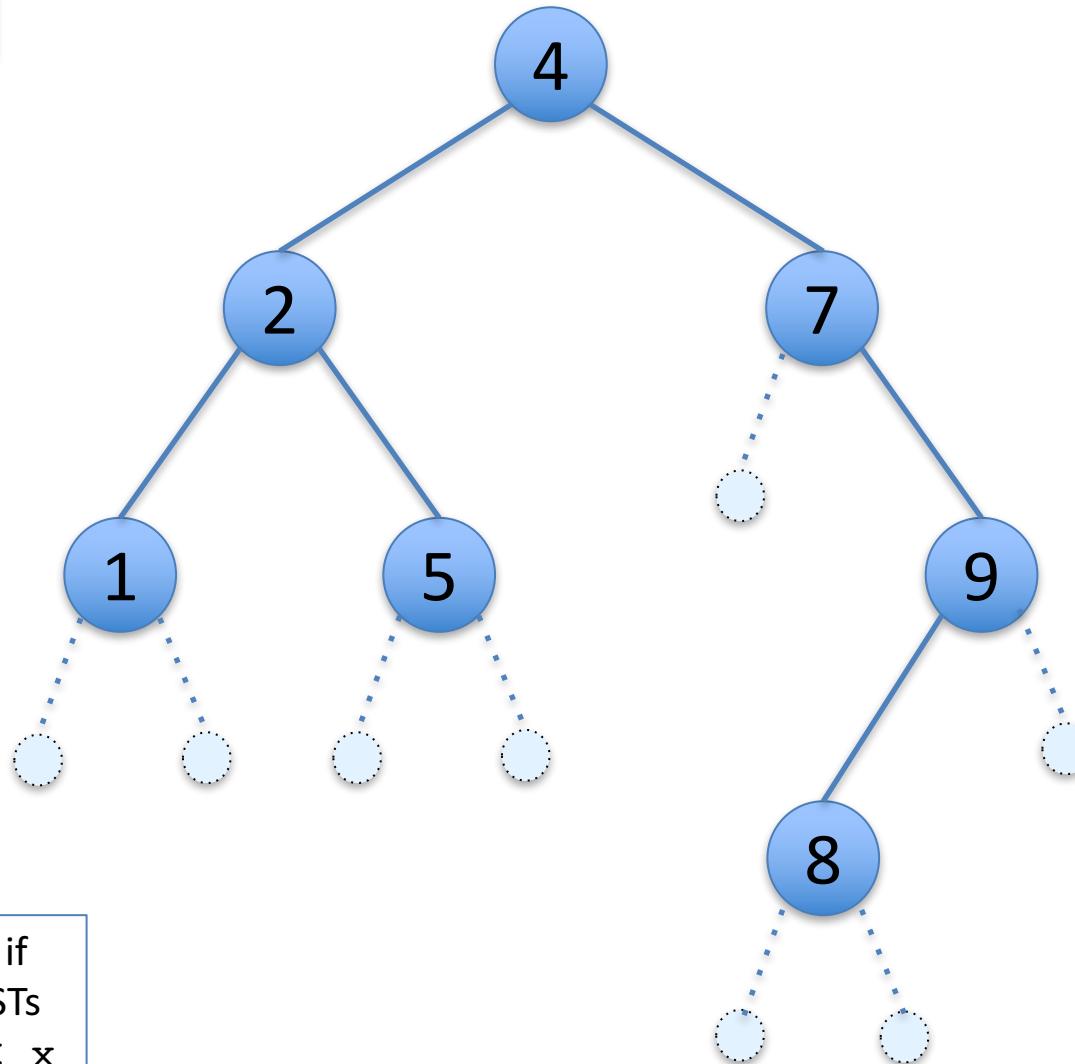


Answer: Yes

- `Node(lt, x, rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are  $< x$
  - all nodes of `rt` are  $> x$
- Empty is a BST

Is this a BST??

1. yes
2. no

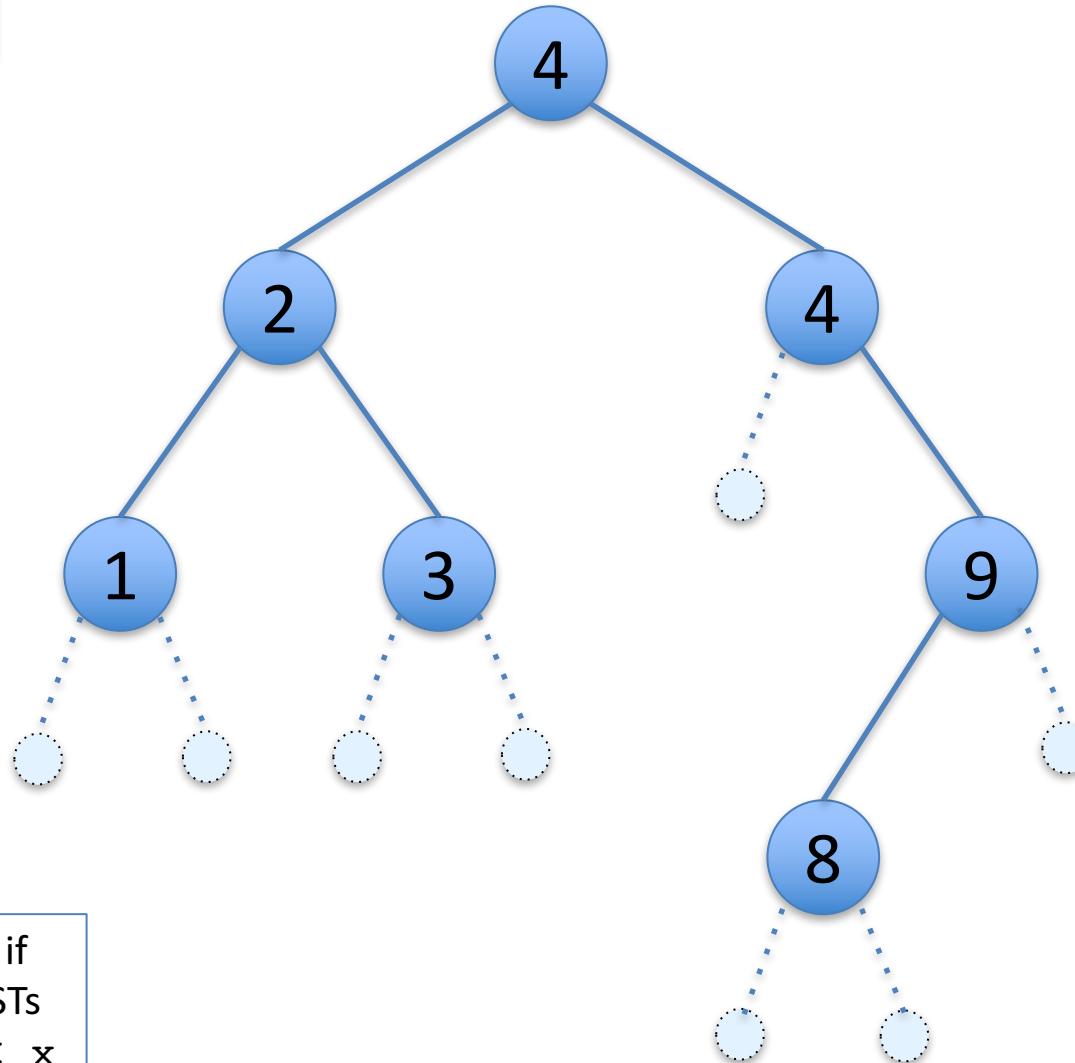


Answer: no, 5 to the left of 4

- `Node(lt, x, rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are  $< x$
  - all nodes of `rt` are  $> x$
- Empty is a BST

Is this a BST??

1. yes
2. no

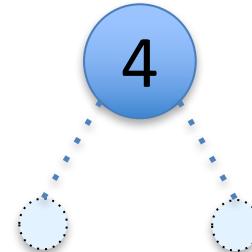


Answer: no, 4 to the right of 4

- `Node(lt, x, rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are  $< x$
  - all nodes of `rt` are  $> x$
- Empty is a BST

Is this a BST??

1. yes
2. no



- `Node(lt, x, rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are  $< x$
  - all nodes of `rt` are  $> x$
- `Empty` is a BST

Answer: yes

Is this a BST??

1. yes
2. no



- `Node(lt, x, rt)` is a BST if
  - `lt` and `rt` are both BSTs
  - all nodes of `lt` are  $< x$
  - all nodes of `rt` are  $> x$
- `Empty` is a BST

Answer: yes

# Manipulating BSTs

Inserting an element

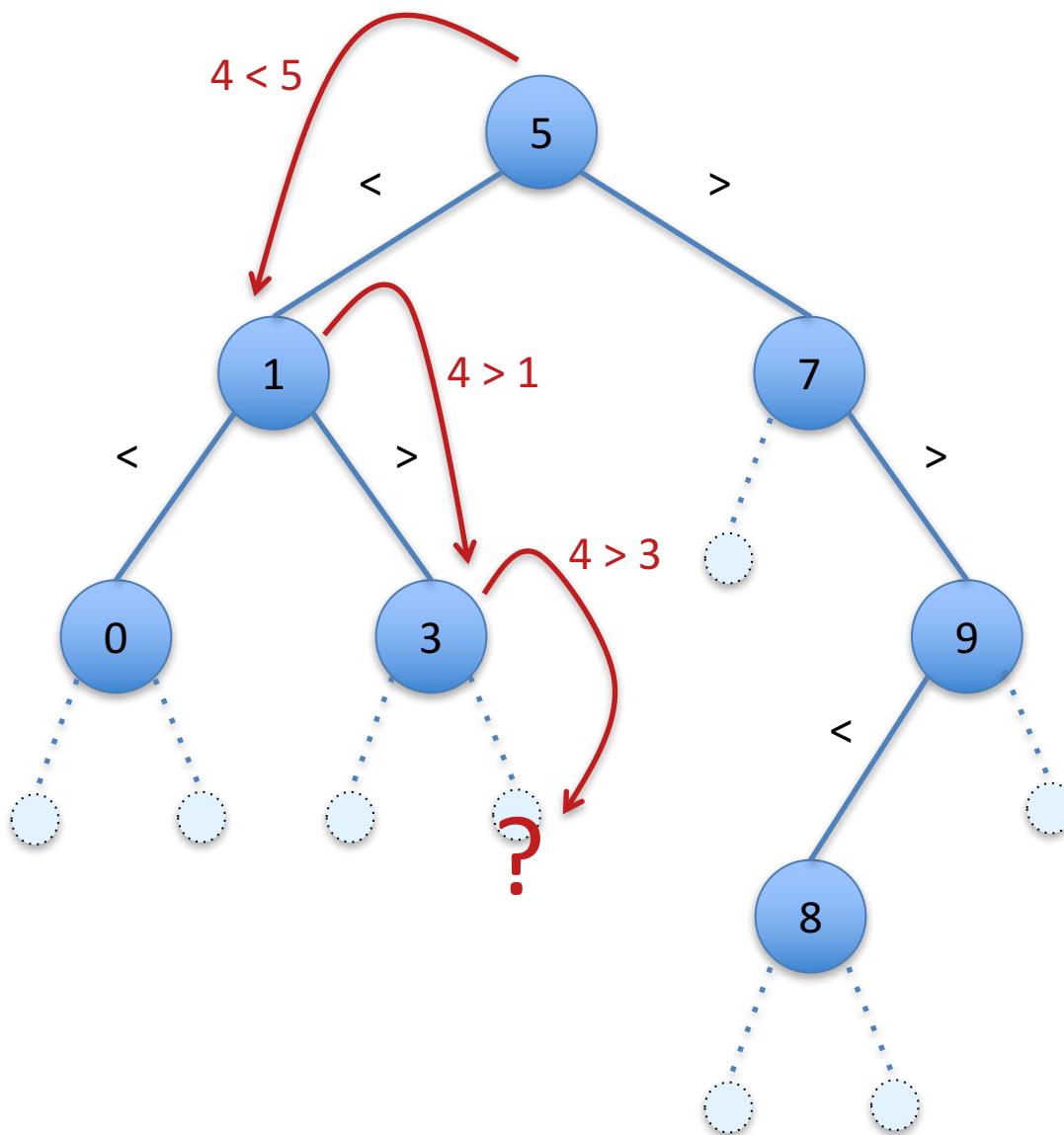
`insert : tree -> int -> tree`

# Inserting into a BST

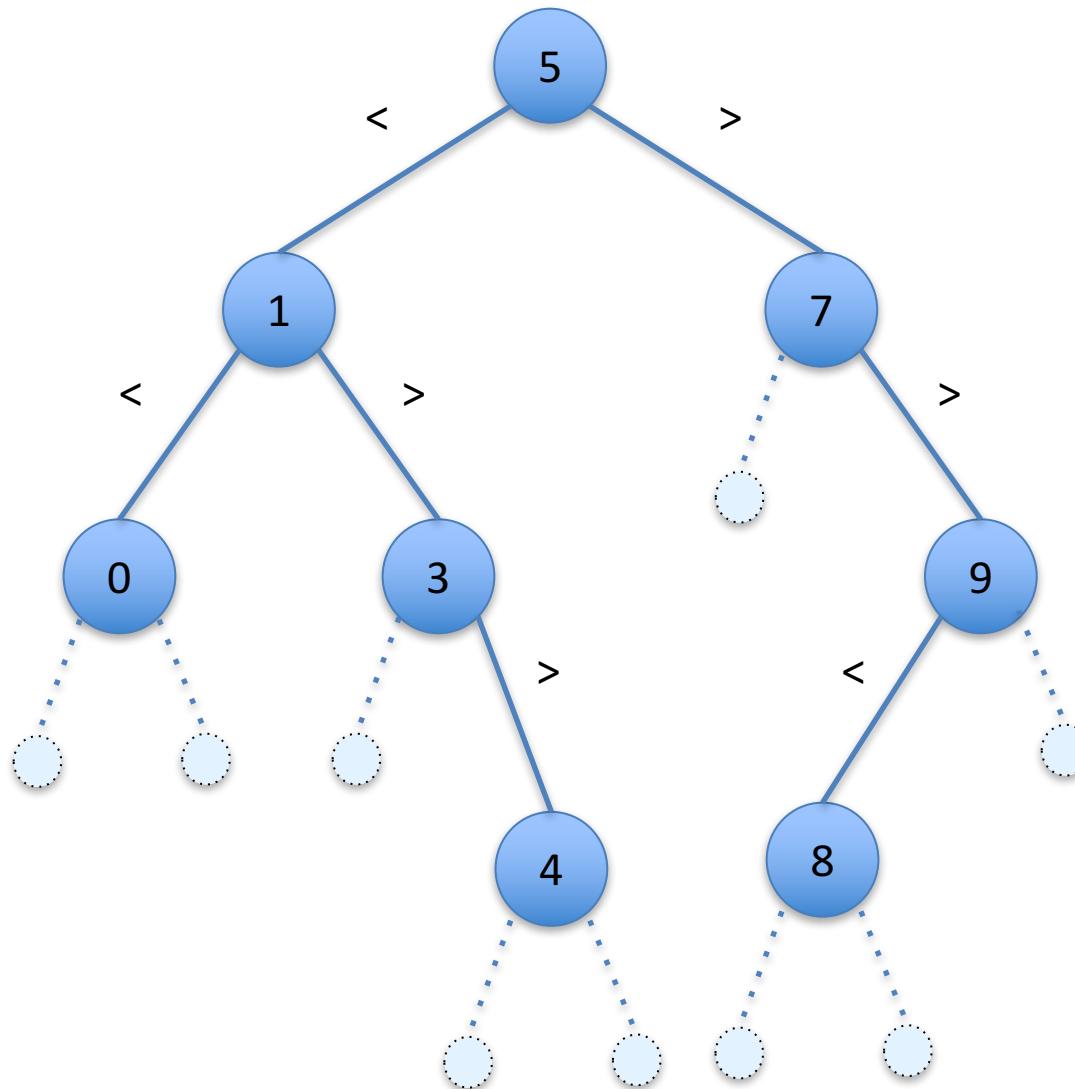
- Suppose we have a BST  $t$  and a new element  $n$ , and we wish to compute a new BST  $t'$  containing all the elements of  $t$  together with  $n$ 
  - Need to make sure the tree we build is really a BST – i.e., make sure to put  $n$  in the right place!
- This gives us a way to build up a BST containing any set of elements we like:
  - Starting from the **Empty** BST, apply this function repeatedly to get the BST we want
  - If insertion *preserves* the BST invariants, then any tree we get from it will be a BST *by construction*
    - No need to check!
  - Later: we can also “rebalance” the tree to make lookup efficient (NOT in CIS 120; see CIS 121)

*First step: find the right place...*

# Inserting a new node: (insert t 4)



# Inserting a new node: (insert t 4)



# Inserting Into a BST

```
(* Insert n into the BST t *)
let rec insert (t:tree) (n:int) : tree =
  begin match t with
    | Empty -> Node(Empty,n,Empty)
    | Node(lt,x,rt) ->
        if x = n then t
        else if n < x then Node(insert lt n, x, rt)
        else Node(lt, x, insert rt n)
  end
```

- Note the similarity to searching the tree.
- Assuming that  $t$  is a BST, the result is also a BST. (Why?)
- Note that the result is a *new* tree with (possibly) one more Node; the original tree is unchanged

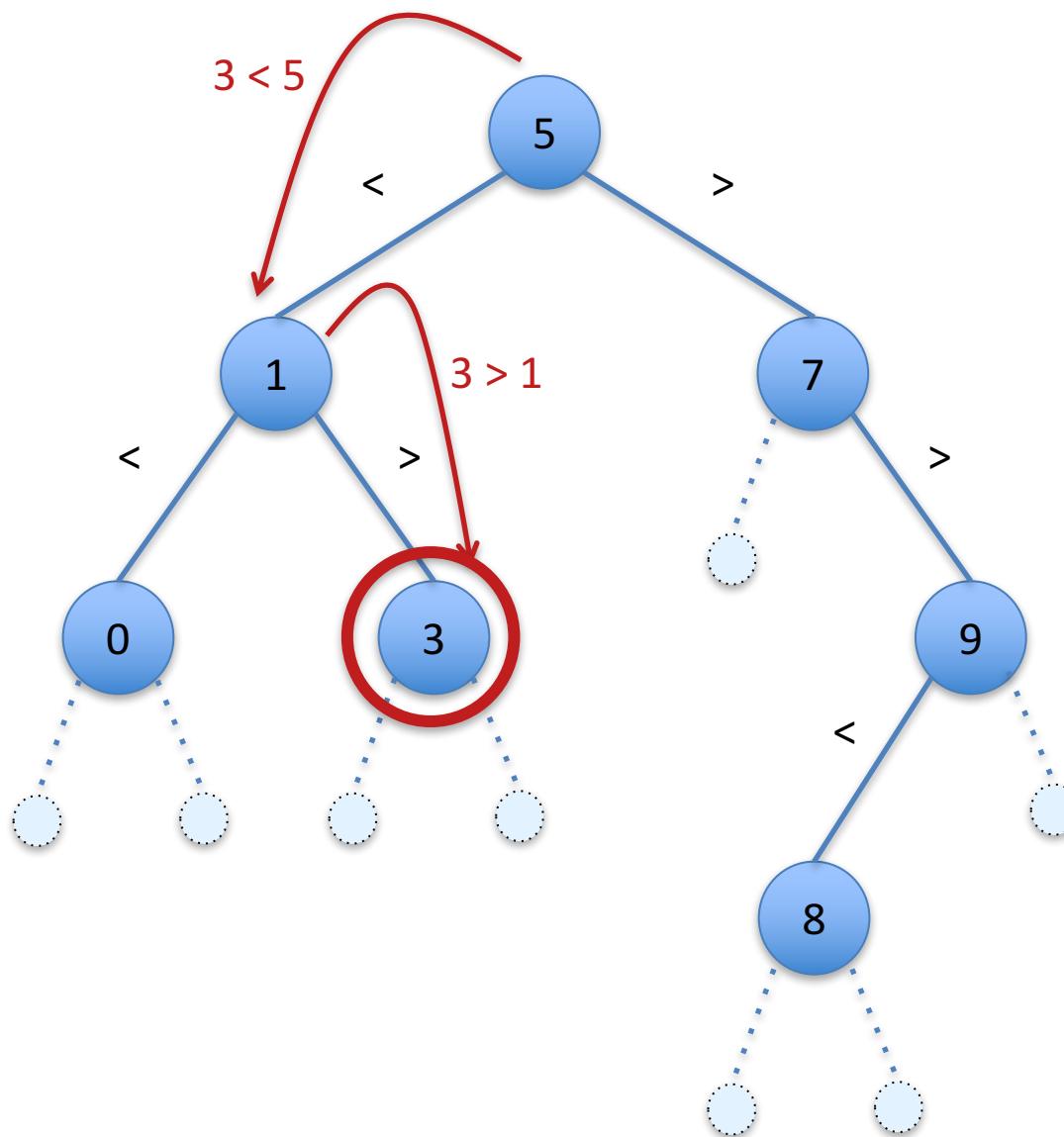


# Manipulating BSTs

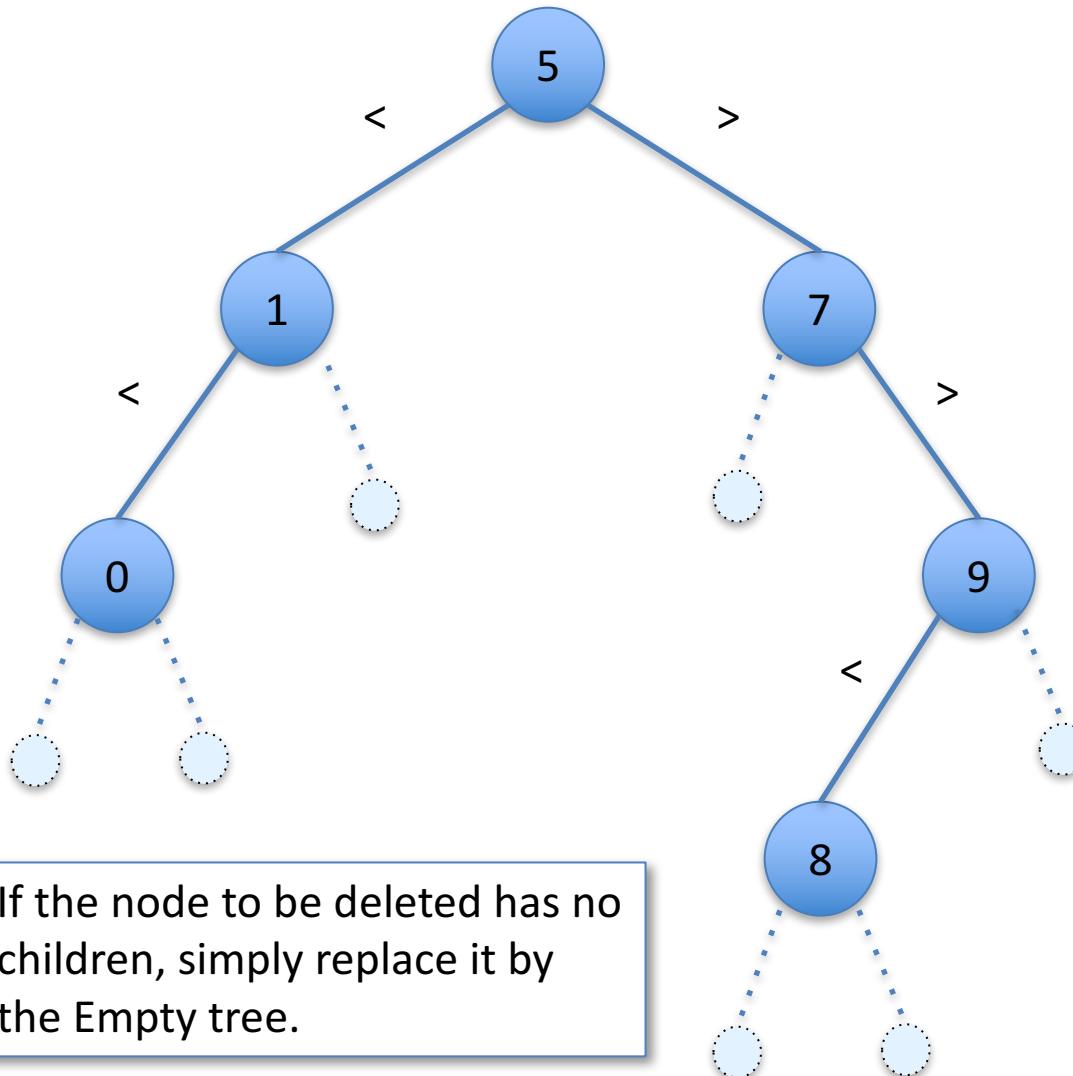
Deleting an element

`delete : tree -> int -> tree`

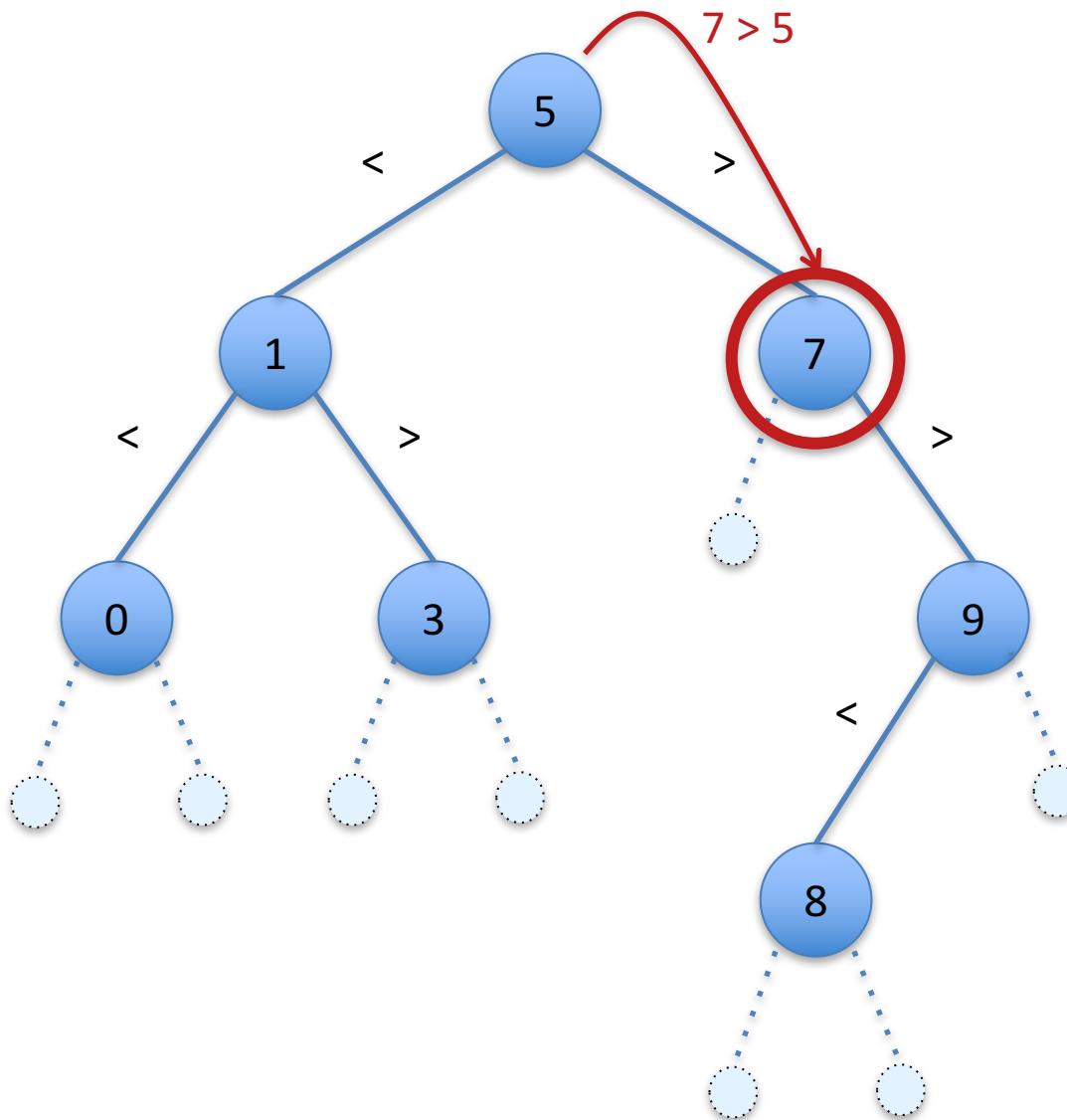
# Deletion – No Children: ( delete t 3 )



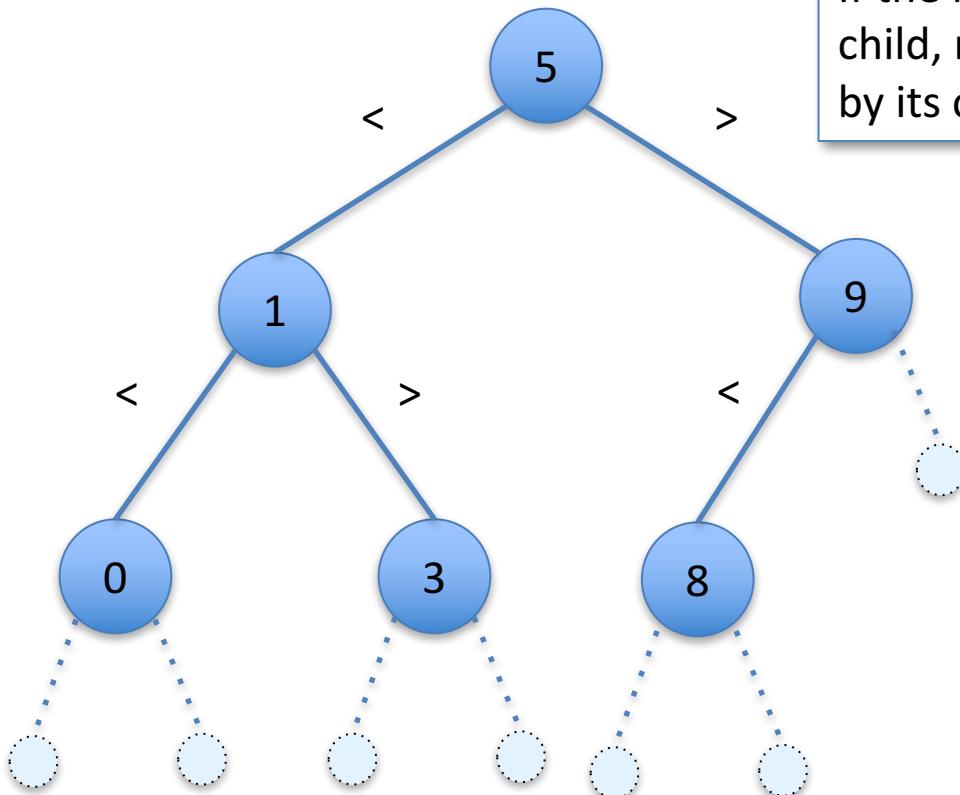
# Deletion – No Children: ( delete t 3 )



# Deletion – One Child: (delete t 7)

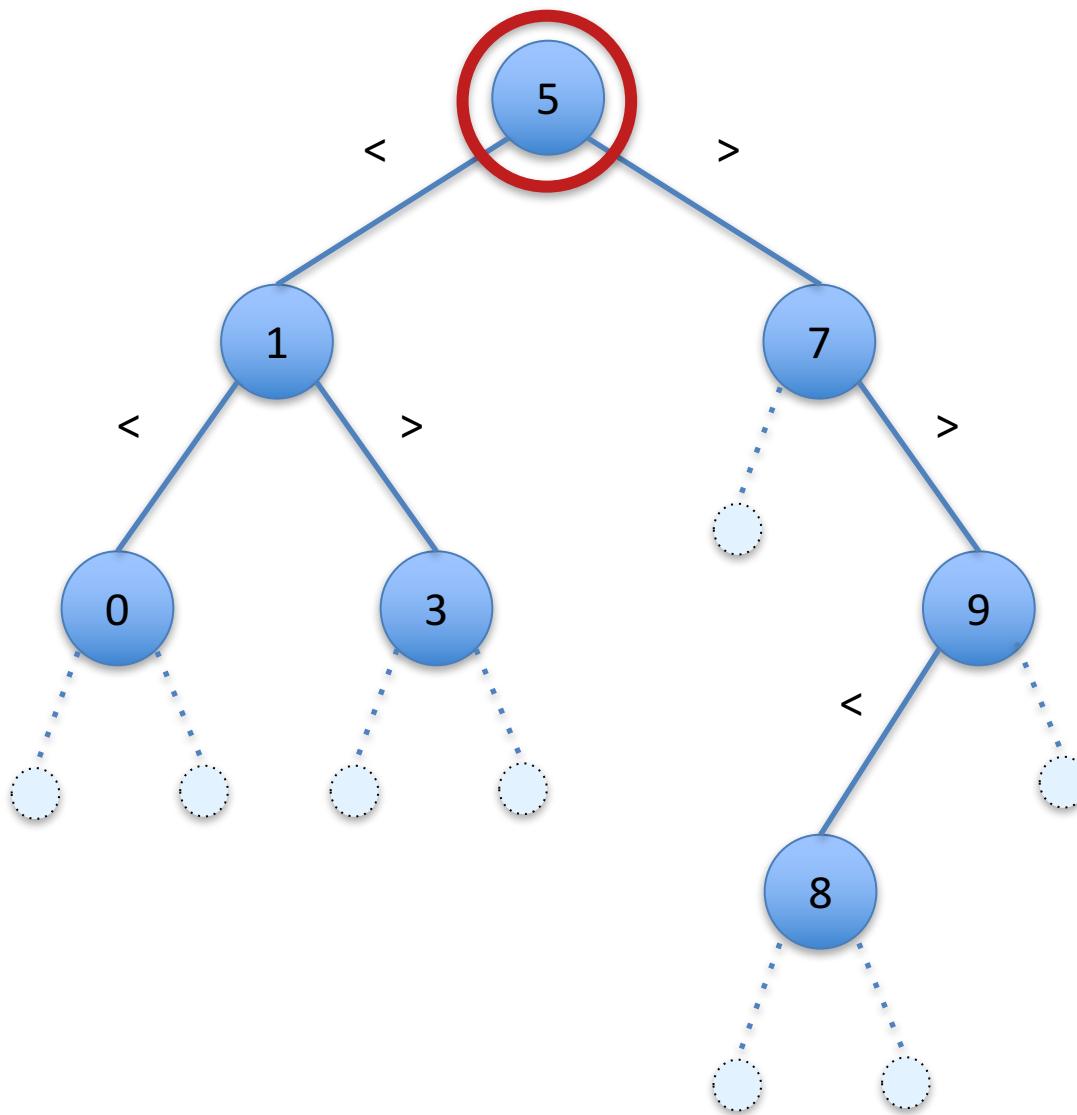


# Deletion – One Child: (delete t 7)

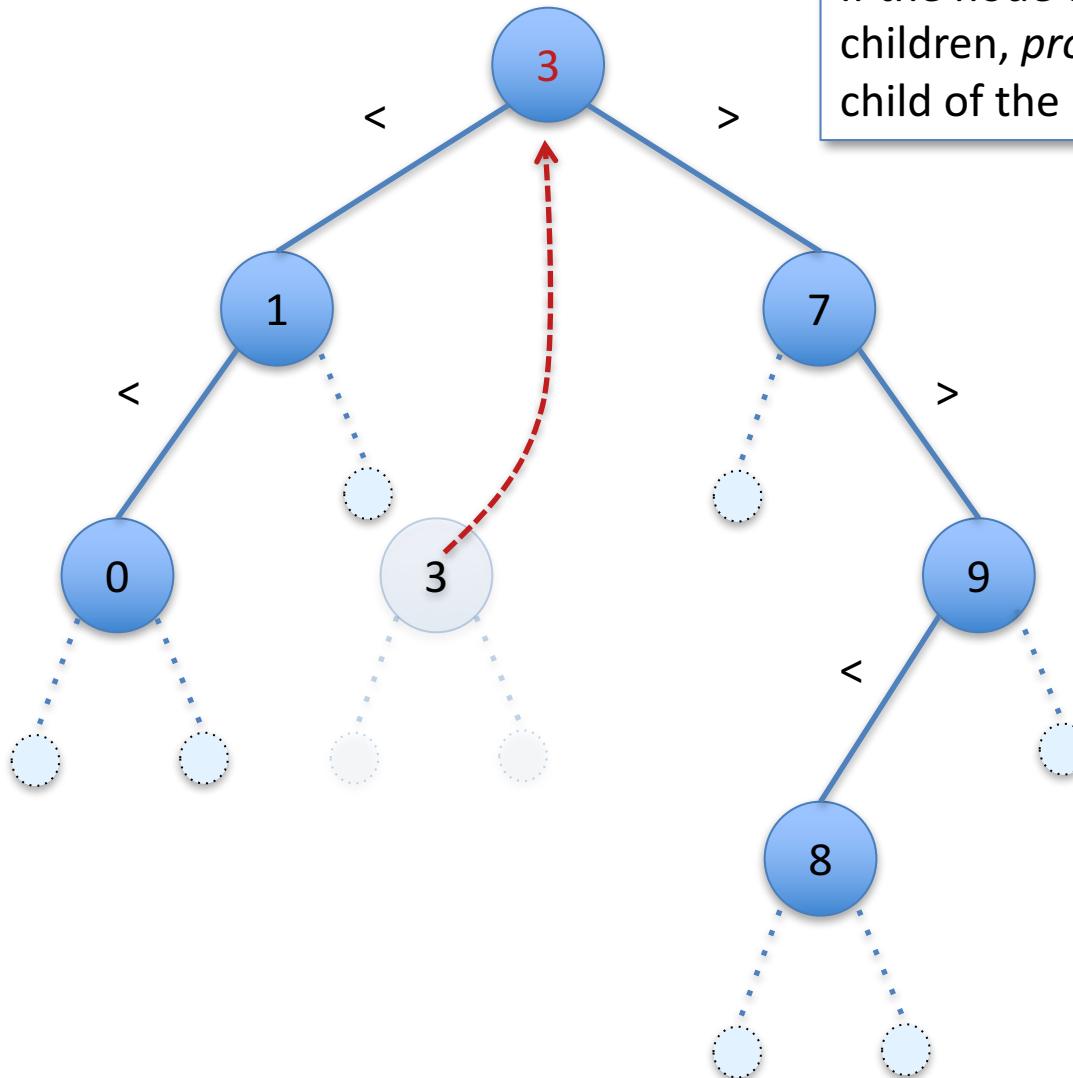


If the node to be delete has one child, replace the deleted node by its child.

# Deletion – Two Children: (delete t 5)



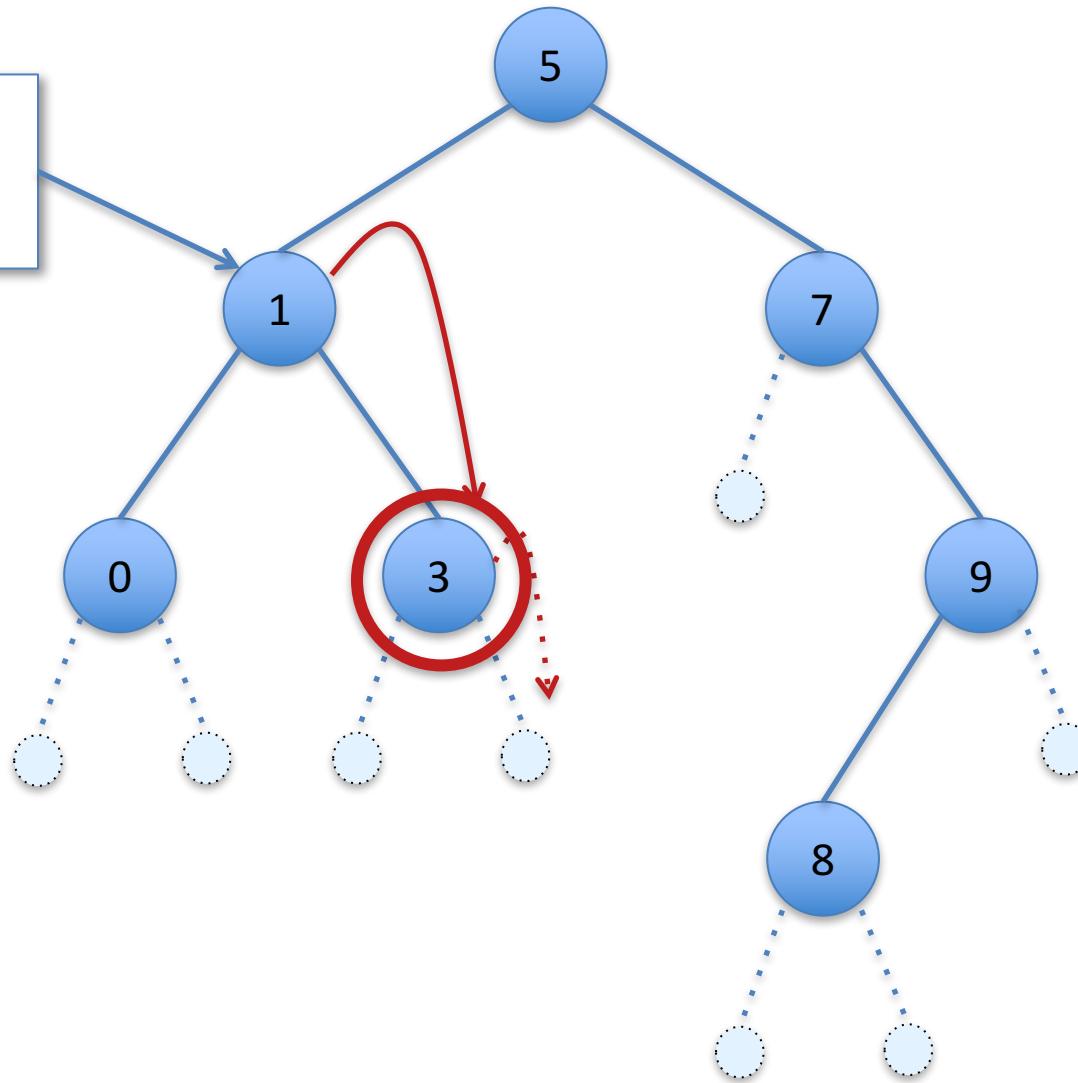
# Deletion – Two Children: (delete t 5)



If the node to be delete has two children, *promote* the maximum child of the left tree.

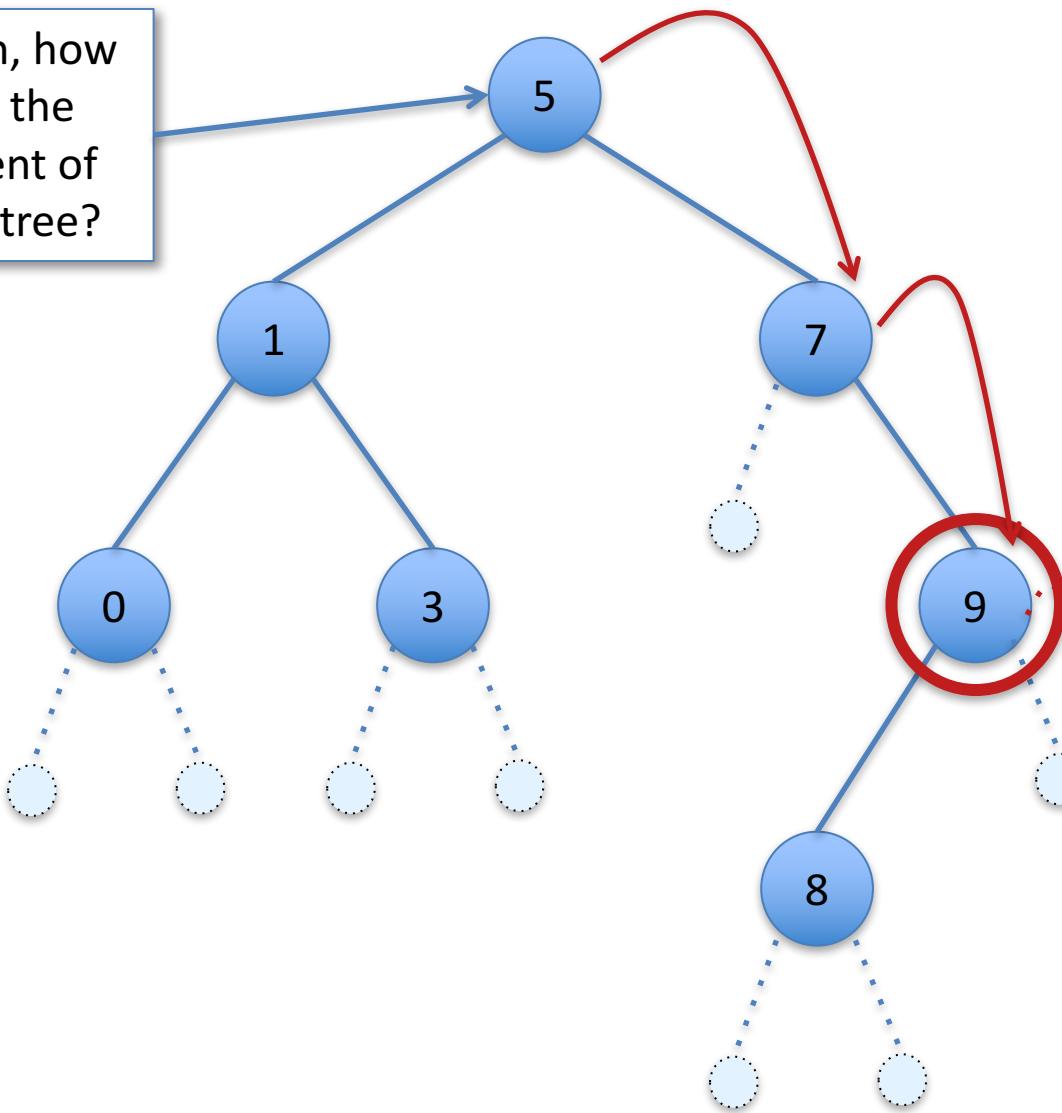
# How to Find the Maximum Element?

What is the max element of this subtree?



# How to Find the Maximum Element?

Just for fun, how do we find the max element of the whole tree?



# Tree Max

```
let rec tree_max (t:tree) : int =
  begin match t with
  | Node(_,x,Empty) -> x
  | Node(_,_,rt) -> tree_max rt
  | _ -> failwith "tree_max called on Empty"
  end
```

- BST invariant guarantees that the maximum-value node is farthest to the right
- Note that `tree_max` is a *partial*\* function
  - Fails when called with an empty tree
- Fortunately, we never need to call `tree_max` on an empty tree
  - This is a consequence of the BST invariants and the case analysis done by the delete function

\* Partial, in this context, means “not defined for all inputs”.

# Code for BST delete

bst.ml

# Deleting From a BST

```
let rec delete (t: tree) (n: int) : tree =
begin match t with
| Empty -> Empty
| Node(lt, x, rt) ->
  if x = n then
    begin match (lt, rt) with
    | (Empty, Empty) -> Empty
    | (Node _, Empty) -> lt
    | (Empty, Node _) -> rt
    | _ -> let m = tree_max lt in
      Node(delete lt m, m, rt)
  end
  else if n < x then Node(delete lt n, x, rt)
  else Node(lt, x, delete rt n)
end
```

See bst.ml

# Subtleties of the Two-Child Case

- Suppose  $\text{Node}(lt, x, rt)$  is to be deleted and  $lt$  and  $rt$  are both themselves nonempty trees.
- Then:
  1. There exists a maximum element,  $m$ , of  $lt$  (Why?)
  2. Every element of  $rt$  is greater than  $m$  (Why?)
- To promote  $m$  we replace the deleted node by:  
 $\text{Node}(\text{delete } lt \ m, m, rt)$ 
  - I.e. we recursively delete  $m$  from  $lt$  and relabel the root node  $m$
  - The resulting tree satisfies the BST invariants

If we insert a label n into a BST and then immediately delete n, do we always get back a tree of exactly the same shape?

1. yes
2. no

Answer: no (what if the node was in the tree to begin with?)

If we insert a value  $n$  into a BST *that does not already contain  $n$*  and then immediately delete  $n$ , do we always get back a tree of exactly the same shape?

1. yes
2. no

Answer: yes

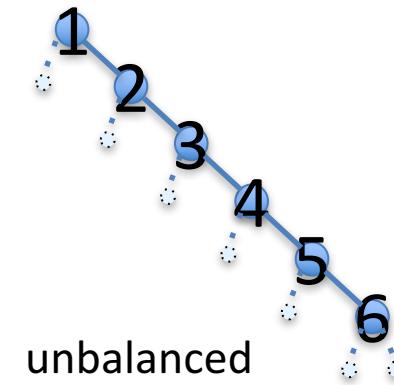
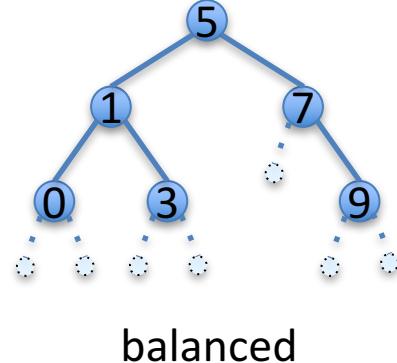
If we delete n from a BST (containing n) and then immediately insert n again, do we always get back a tree of exactly the same shape?

1. yes
2. no

Answer: no (e.g., what if we delete the item at the root node?)

# BST Performance

- lookup takes time proportional to the *height* of the tree.
  - not the *size* of the tree (as it did with contains for unordered trees)
- In a *balanced tree*, the lengths of the paths from the root to each leaf are (almost) *the same*.
  - no leaf is too far from the root
  - the height of the BST is minimized
  - the height of a balanced binary tree is roughly  $\log_2(N)$  where N is the number of nodes in the tree



# Programming Languages and Techniques (CIS120)

Lecture 7

September 15, 2017

Binary Search Trees: Inserting and Deleting  
(Chapters 7 & 8)

# Announcements

- Read Chapters 7 & 8
  - Binary Search Trees
- Next up: generics and first-class functions
- HW2 due *Tuesday* at midnight

# Recap: Trees as Containers

# Binary Search Trees

- A *binary search tree* (BST) is a binary tree with some additional *invariants*:
  - $\text{Node}(lt, x, rt)$  is a BST if
    - $lt$  and  $rt$  are both BSTs
    - all nodes of  $lt$  are  $< x$
    - all nodes of  $rt$  are  $> x$
  - $\text{Empty}$  is a BST
- *The BST invariant means that container functions can take time proportional to the **height** instead of the **size** of the tree.*

# Searching a BST

```
(* Assumes that t is a BST *)
let rec lookup (t:tree) (n:int) : bool =
begin match t with
| Empty -> false
| Node(lt,x,rt) ->
    if x = n then true
    else if n < x then lookup lt n
    else lookup rt n
end
```

- The BST invariants guide the search.
- Note that `lookup` may return an incorrect answer if the input is *not* a BST!
  - This function *assumes* that the BST invariants hold of `t`.

# Manipulating BSTs

Inserting an element

`insert : tree -> int -> tree`

# Inserting Into a BST

```
(* Insert n into the BST t *)
let rec insert (t:tree) (n:int) : tree =
begin match t with
| Empty -> Node(Empty,n,Empty)
| Node(lt,x,rt) ->
  if x = n then t
  else if n < x then Node(insert lt n, x, rt)
  else Node(lt, x, insert rt n)
end
```

- Note the similarity to searching the tree.
- Assuming that t is a BST, the result is also a BST. (Why?)
- Note that the result is a *new* tree with one more Node; the original tree is unchanged

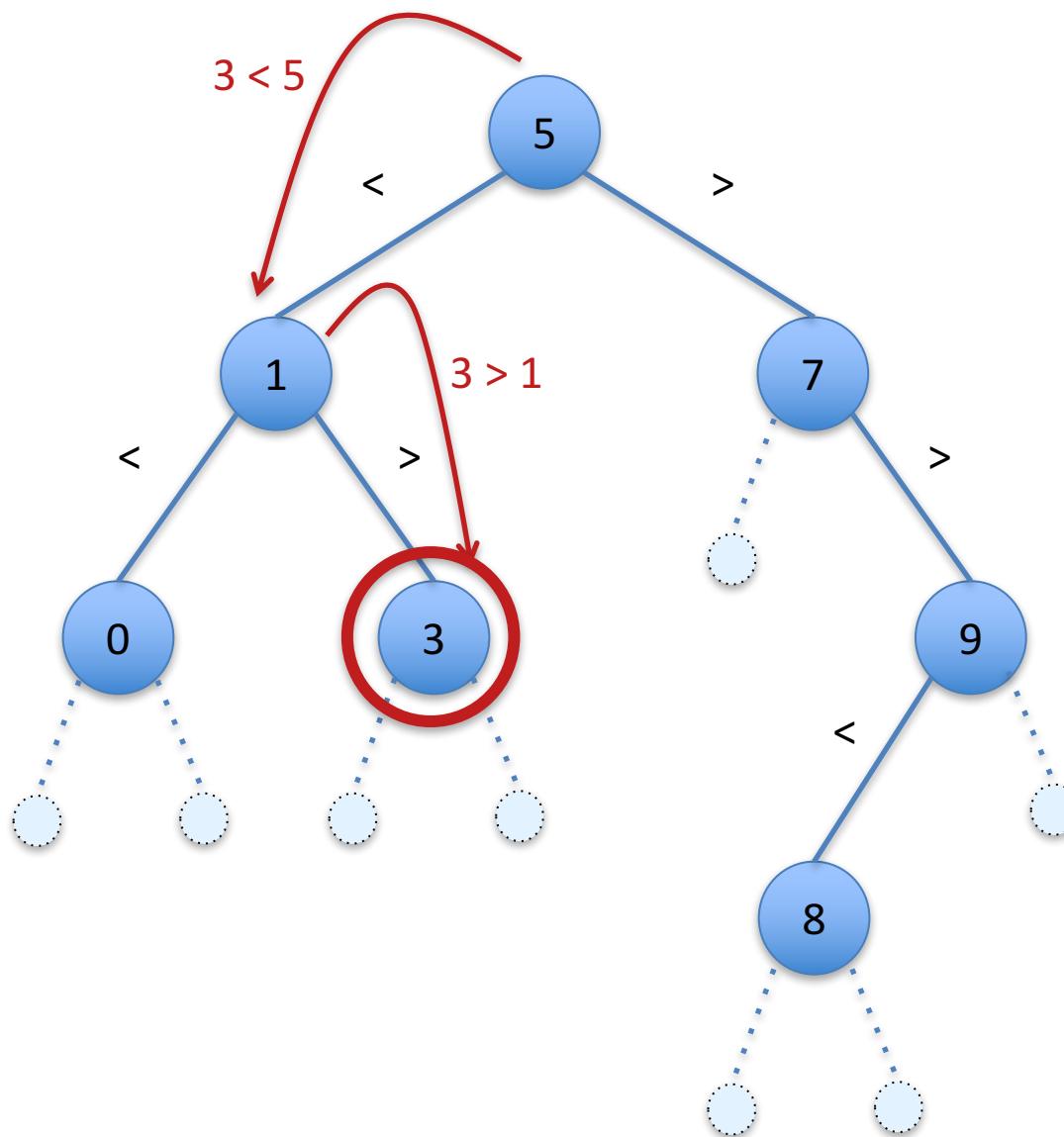


# Manipulating BSTs

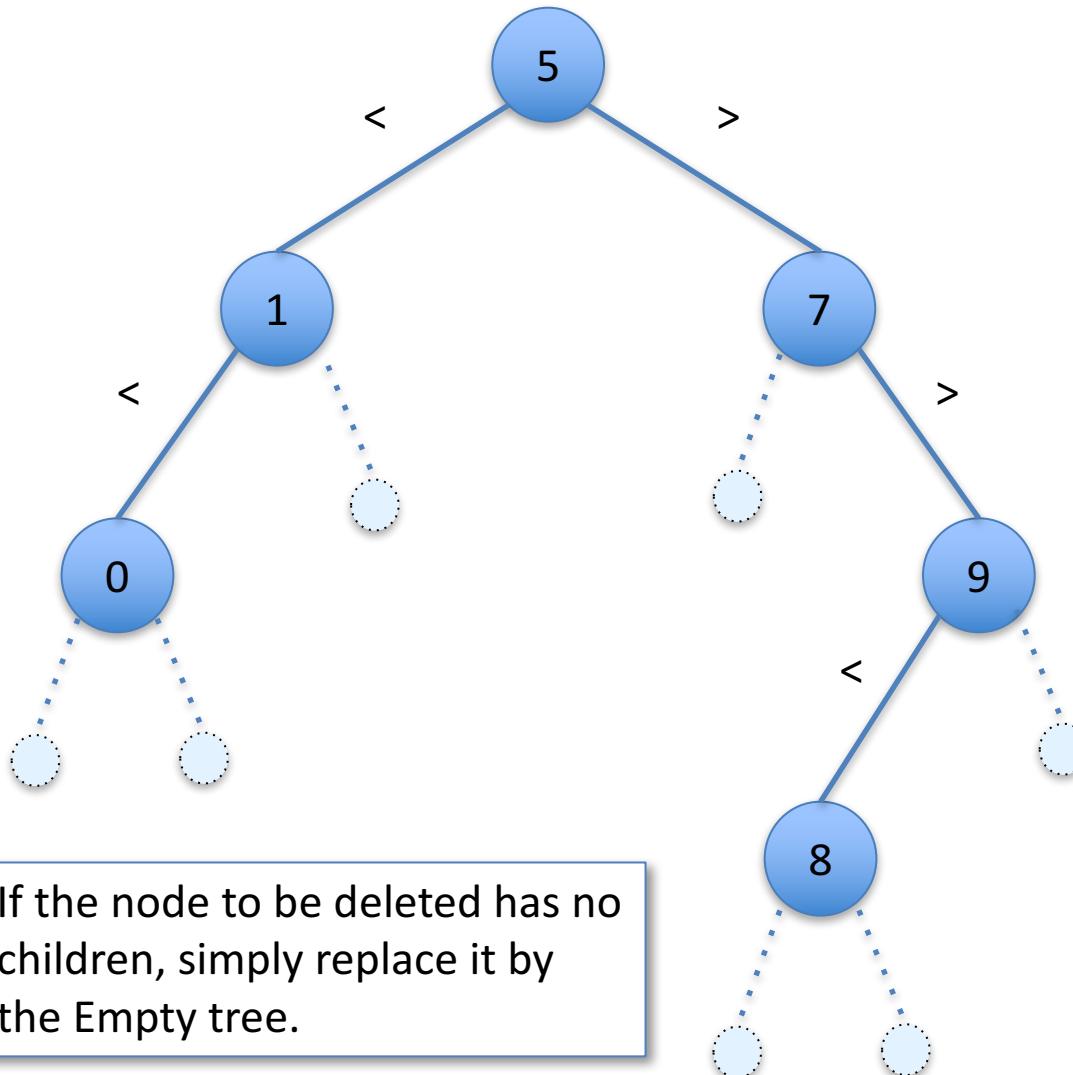
Deleting an element

`delete : tree -> int -> tree`

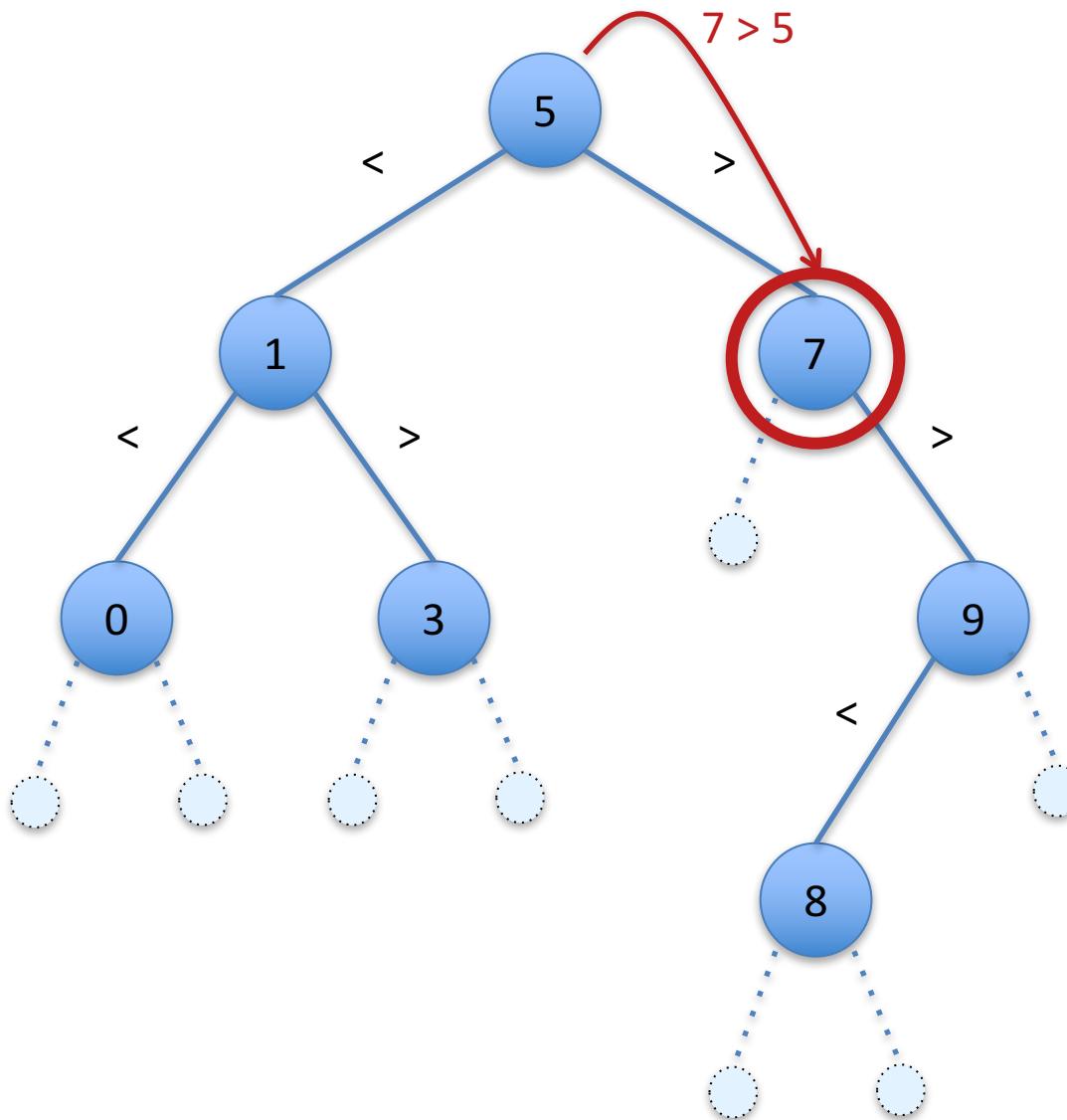
# Deletion – No Children: ( delete t 3 )



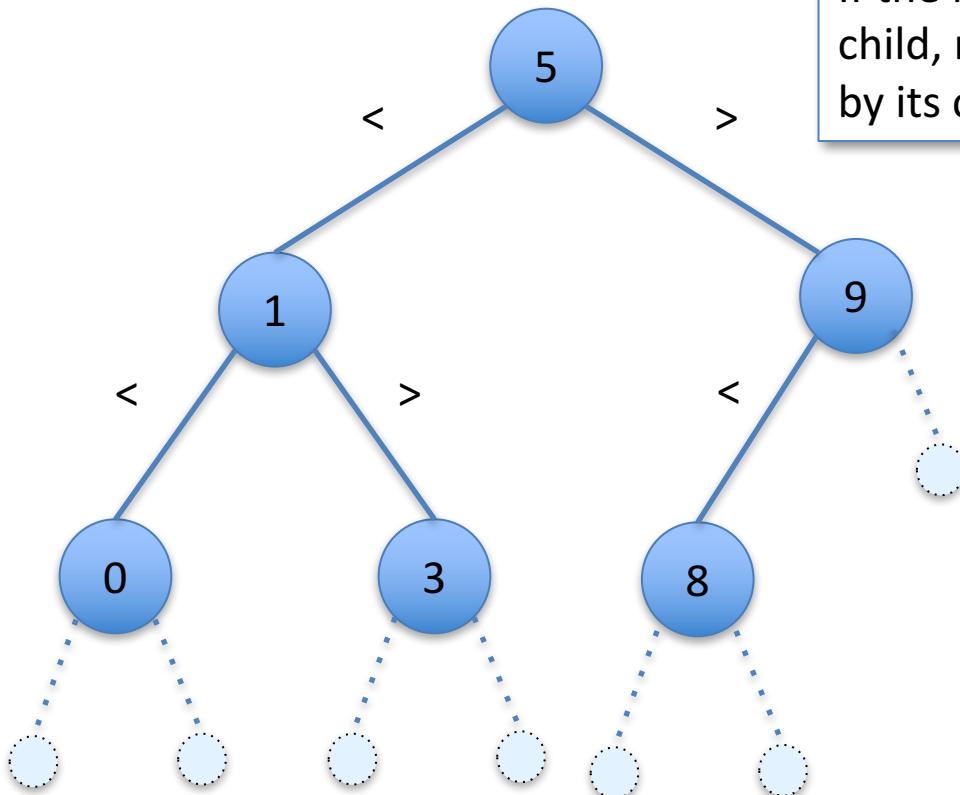
# Deletion – No Children: ( delete t 3 )



# Deletion – One Child: (delete t 7)

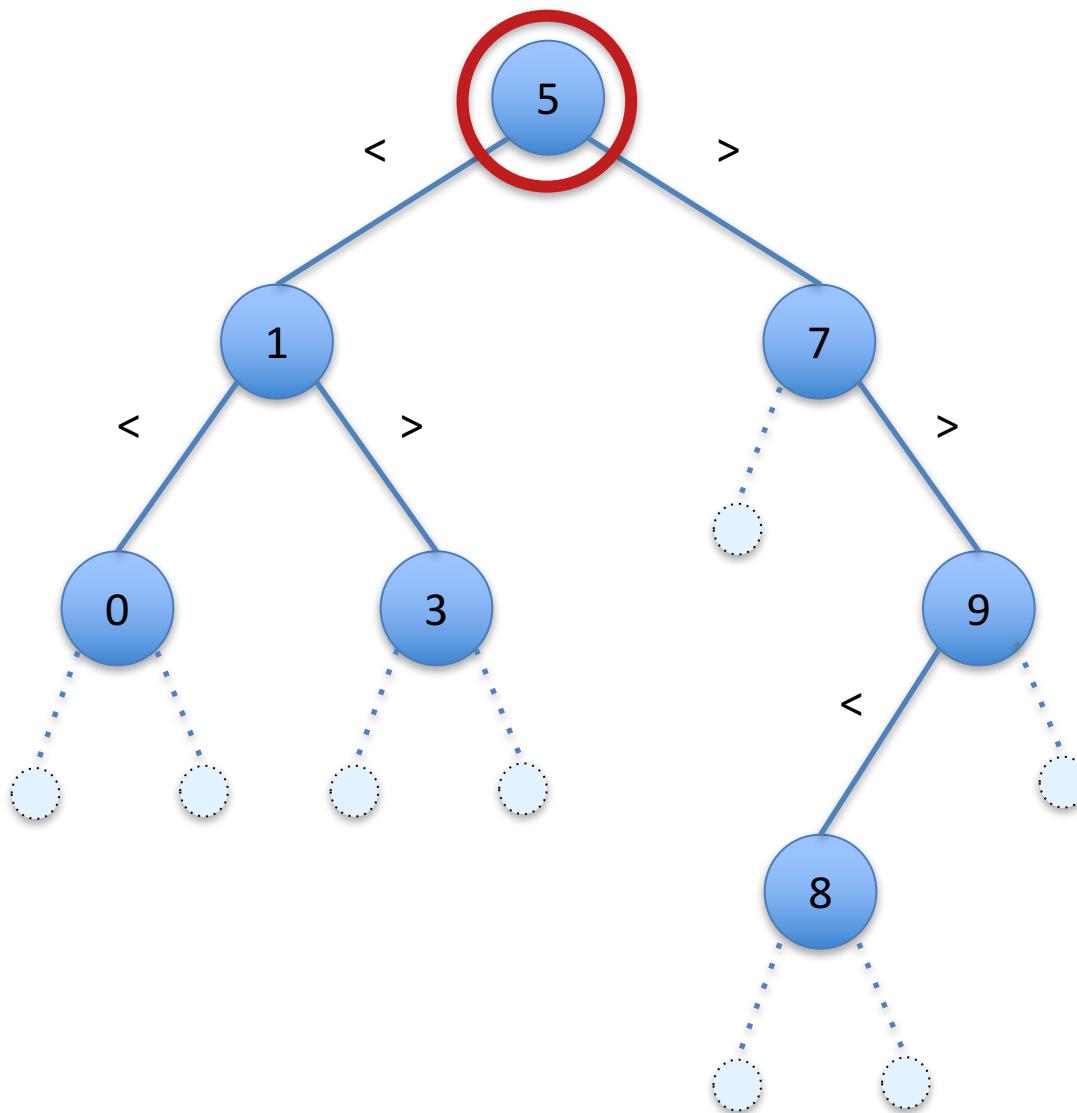


# Deletion – One Child: (delete t 7)

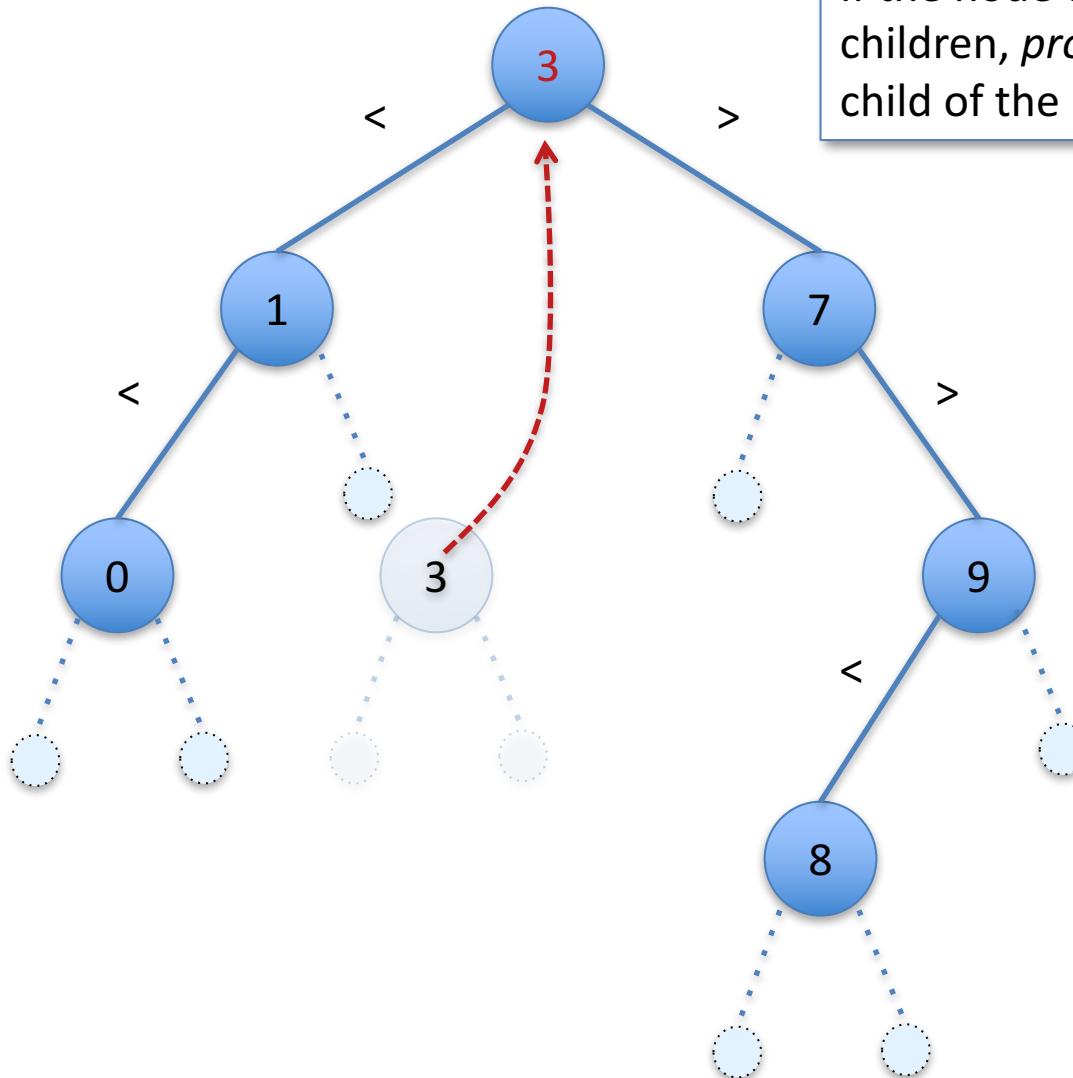


If the node to be delete has one child, replace the deleted node by its child.

# Deletion – Two Children: (delete t 5)



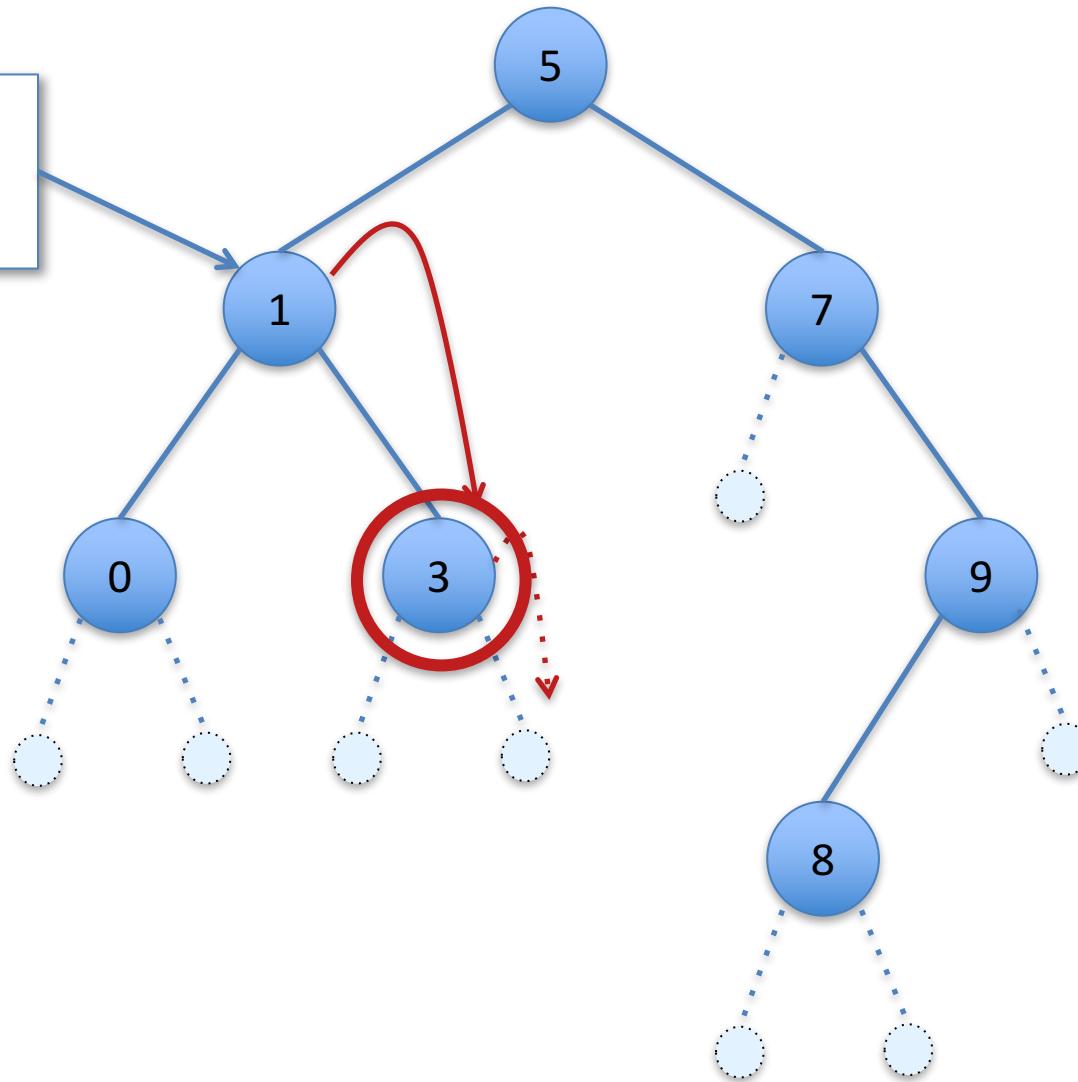
# Deletion – Two Children: (delete t 5)



If the node to be delete has two children, *promote* the maximum child of the left tree.

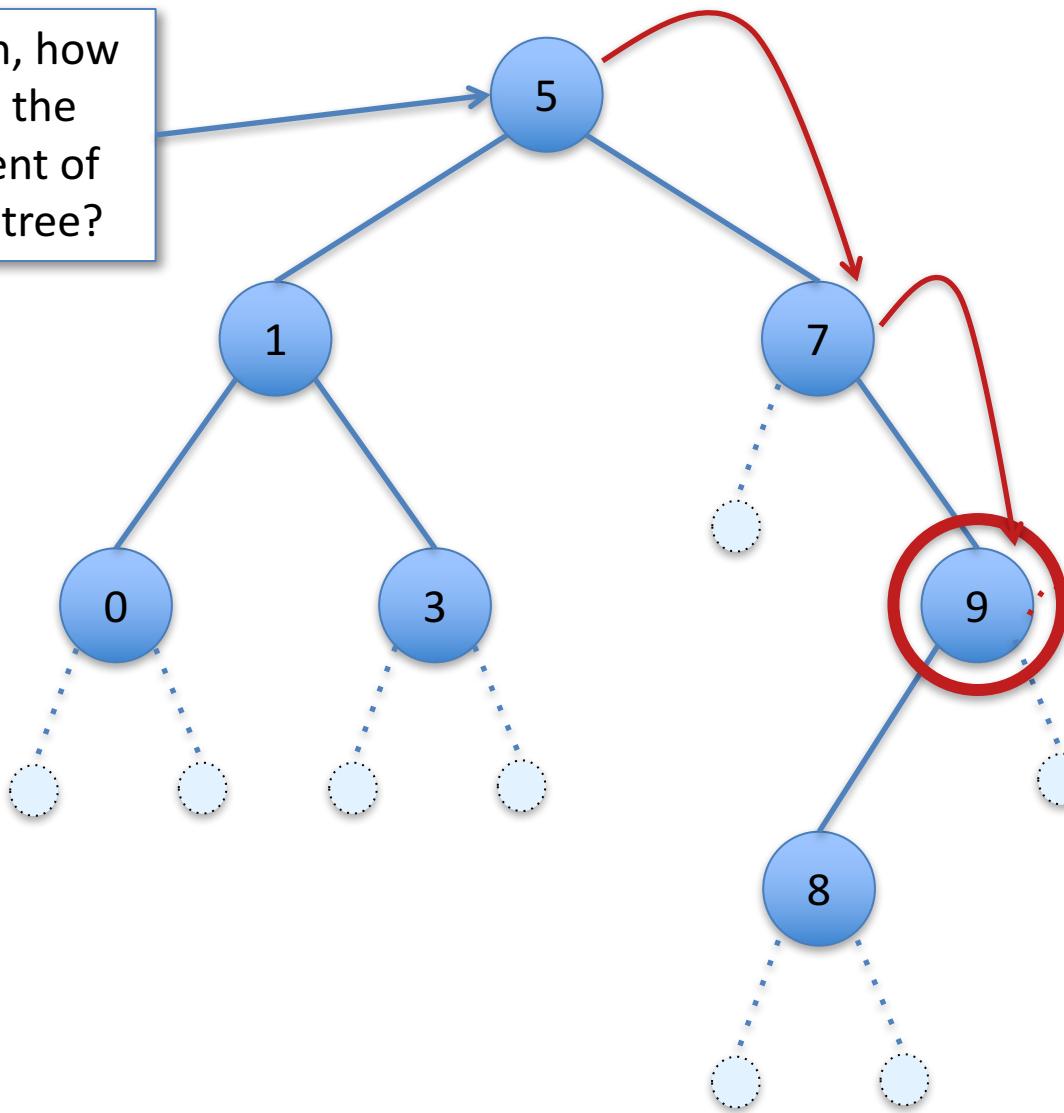
# How to Find the Maximum Element?

What is the max element of this subtree?



# How to Find the Maximum Element?

Just for fun, how do we find the max element of the whole tree?



# Tree Max

```
let rec tree_max (t:tree) : int =
  begin match t with
  | Node(_,x,Empty) -> x
  | Node(_,_,rt) -> tree_max rt
  | _ -> failwith "tree_max called on Empty"
  end
```

- BST invariant guarantees that the maximum-value node is farthest to the right
- Note that `tree_max` is a *partial*\* function
  - Fails when called with an empty tree
- Fortunately, we never need to call `tree_max` on an empty tree
  - This is a consequence of the BST invariants and the case analysis done by the delete function

\* Partial, in this context, means “not defined for all inputs”.

# Code for BST delete

bst.ml

# Deleting From a BST

```
let rec delete (t: tree) (n: int) : tree =
begin match t with
| Empty -> Empty
| Node(lt, x, rt) ->
  if x = n then
    begin match (lt, rt) with
    | (Empty, Empty) -> Empty
    | (Node _, Empty) -> lt
    | (Empty, Node _) -> rt
    | _ -> let m = tree_max lt in
      Node(delete lt m, m, rt)
  end
  else if n < x then Node(delete lt n, x, rt)
  else Node(lt, x, delete rt n)
end
```

See bst.ml

# Subtleties of the Two-Child Case

- Suppose  $\text{Node}(lt, x, rt)$  is to be deleted and  $lt$  and  $rt$  are both themselves nonempty trees.
- Then:
  1. There exists a maximum element,  $m$ , of  $lt$  (Why?)
  2. Every element of  $rt$  is greater than  $m$  (Why?)
- To promote  $m$  we replace the deleted node by:  
 $\text{Node}(\text{delete } lt \ m, m, rt)$ 
  - I.e. we recursively delete  $m$  from  $lt$  and relabel the root node  $m$
  - The resulting tree satisfies the BST invariants

If we insert a label n into a BST and then immediately delete n, do we always get back a tree of exactly the same shape?

1. yes
2. no

Answer: no (what if the node was in the tree to begin with?)

If we insert a value  $n$  into a BST *that does not already contain  $n$*  and then immediately delete  $n$ , do we always get back a tree of exactly the same shape?

1. yes
2. no

Answer: yes

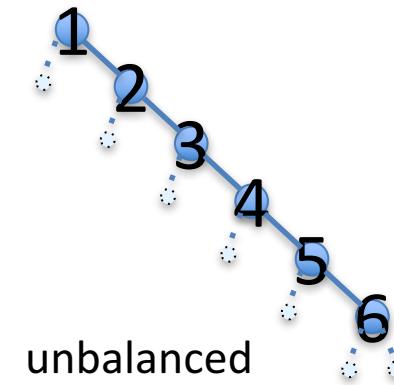
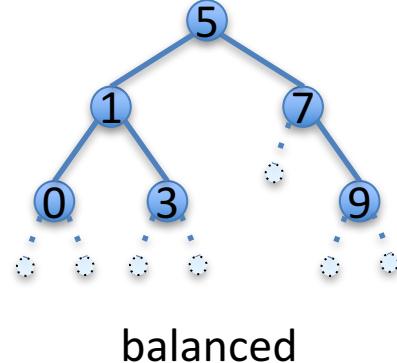
If we delete n from a BST (containing n) and then immediately insert n again, do we always get back a tree of exactly the same shape?

1. yes
2. no

Answer: no (e.g., what if we delete the item at the root node?)

# BST Performance

- lookup takes time proportional to the *height* of the tree.
  - not the *size* of the tree (as it did with contains for unordered trees)
- In a *balanced tree*, the lengths of the paths from the root to each leaf are (almost) *the same*.
  - no leaf is too far from the root
  - the height of the BST is minimized
  - the height of a balanced binary tree is roughly  $\log_2(N)$  where N is the number of nodes in the tree



# Generic Functions and Data

Wow, implementing BSTs took quite a bit of typing...  
Do we have to repeat it all again if we want to use BSTs  
containing strings, or characters, or floats?

or

*How not to repeat yourself, Part I.*

# Structurally Identical Functions

- Observe: many functions on lists, trees, and other datatypes don't depend on the contents, only on the structure.
- Compare: `length` for “`int list`” vs. “`string list`”

```
let rec length (l: int list) : int =
  begin match l with
  | [] -> 0
  | _ :: tl -> 1 + length tl
  end
```

```
let rec length (l: string list) : int =
  begin match l with
  | [] -> 0
  | _ :: tl -> 1 + length tl
  end
```

The functions are *identical*, except for the type annotation.

They are “*generic*” with respect to the input type.

# Notation for Generic Types

- OCaml provides syntax for functions with *generic* types

```
let rec length l:'a list : int =
  begin match l with
    | [] -> 0
    | _::tl -> 1 + (length tl)
  end
```

- Notation: '*a* is a *type variable*; the function `length` can be used on a `t list` for *any type t*.
- Examples:
  - `length [1;2;3]` use length on an `int list`
  - `length [“a”;”b”;”c”]` use length on a `string list`

# Generic List Append

Note that the two input lists must have the *same* type of elements.

The return type is the same as the inputs.

```
let rec append (l1:'a list) (l2:'a list) : 'a list =
  begin match l1 with
    | [] -> l2
    | h::tl -> h::(append tl l2)
  end
```

Pattern matching works over generic types!

In the body of the branch:

h has type 'a

tl has type 'a list

# Generic Zip

Functions can operate over *multiple* generic types.

```
let rec zip (l1:'a list) (l2:'b list) : ('a*'b) list =
  begin match (l1,l2) with
  | (h1::t1, h2::t2) -> (h1,h2)::(zip t1 t2)
  | _ -> []
  end
```

- Distinct type variables can be instantiated differently:

```
zip [1;2;3] ["a";"b";"c"]
```

- Here, '*a* is instantiated to `int`, '*b* to `string`
- Result is

```
[(1,"a");(2,"b");(3,"c")]
of type (int * string) list
```

Intuition: OCaml tracks instantiations of type variables ('*a* and '*b*) and makes sure they are used consistently in each use of the function.

# User-Defined Generic Datatypes

- Recall our integer tree type:

```
type tree =
| Empty
| Node of tree * int * tree
```

- We can define a generic version by adding a type parameter, like this:

```
type 'a tree =
| Empty
| Node of 'a tree * 'a * 'a tree
```

Parameter '*a*  
used here

Note that the recursive  
uses also mention '*a*

# User-Defined Generic Datatypes

- BST operations can be generic too; only change is to the type annotation

(\* Insert n into the BST t \*)

```
let rec insert (t:'a tree) (n:'a) : 'a tree =
begin match t with
| Empty -> Node(Empty,n,Empty)
| Node(lt,x,rt) ->
  if x = n then t
  else if n < x then Node(insert lt n, x, rt)
  else Node(lt, x, insert rt n)
end
```

Equality and comparison are generic — they work for *any* type of data too.

Does the following function typecheck?

```
let f (l : 'a list) : 'b list =
begin match l with
| [] -> true::l
| _::rest -> 1::l
end
```

1. yes
2. no

Answer: no: even though the return type is generic, the two branches must agree (so that 'b can be consistently instantiated).

Does the following function typecheck?

```
let f (x : 'a) : 'a =  
  x + 1  
  
;; print_endline (f "hello")
```

1. yes
2. no

Answer: no, the type annotations and uses of f aren't consistent

# First-class Functions

Higher-order Programs

or

How not to repeat yourself, Part II.

# First-class Functions

- You can pass a function as an *argument* to another function:

```
let twice (f:int->int) (x:int) : int =  
    f (f x)
```

```
let add_one (z:int) : int = z + 1  
let add_two (z:int) : int = z + 2  
let y = twice add_one 3  
let w = twice add_two 3
```

function type: argument of type int  
and result of type int

- You can *return* a function as the result of another function.

```
let make_incr (n:int) : int->int =  
    let helper (x:int) : int =  
        n + x  
    in  
    helper  
let y = twice (make_incr 1) 3
```

# Function Types

- Functions have types that look like this:

$$t_{in} \rightarrow t_{out}$$

- Examples:

`int -> int`

`int -> bool * int`

`int -> int -> int`

*int input*

`(int -> int) -> int`

*function input*

# Function Types

- Functions have types that look like this:

$$t_{in} \rightarrow t_{out}$$

- Examples:

`int -> int`

`int -> (bool * int)`

`int -> (int -> int)`

`(int -> int) -> int`

Parentheses matter!

`int -> int -> int` is equivalent to  
`int -> (int -> int)` but not to  
`(int -> int) -> int`

*int input*

*function input*

# First-class Functions

- You can store functions in data structures

```
let add_one    (x:int) : int = x+1
let add_two    (x:int) : int = x+2
let add_three  (x:int) : int = x+3

let func_list : (int -> int) list =
  [ add_one; add_two; add_three ]
```

A list of  $(\text{int} \rightarrow \text{int})$  functions.

```
let func_list1 : (int -> int) list =
  [ make_incr 1; make_incr 2; make_incr 3 ]
```

# Simplifying First-Class Functions

```
let twice (f:int->int) (x:int) : int =  
  f (f x)
```

```
let add_one (z:int) : int = z + 1
```

twice add\_one 3  
→ add\_one (add\_one 3)  
→ add\_one (3 + 1)  
→ add\_one 4  
→ 4 + 1  
→ 5

*substitute add\_one for f, 3 for x*  
*substitute 3 for z in add\_one*  
 $3+1 \Rightarrow 4$   
*substitute 4 for z in add\_one*  
 $4+1 \Rightarrow 5$

# Simplifying First-Class Functions

```
let make_incr (n:int) : int->int =  
  let helper (x:int) : int = n + x in  
  helper
```

make\_incr 3

*substitute 3 for n*

↪ let helper (x:int) = 3 + x in helper

↪ ???

# Simplifying First-Class Functions

```
let make_incr (n:int) : int->int =  
  let helper (x:int) : int = n + x in  
    helper
```

make\_incr 3

*substitute 3 for n*

↪ let helper (x:int) = 3 + x in helper

↪ fun (x:int) -> 3 + x

Anonymous function value

keyword “fun”

“->” after arguments  
no return type annotation

# Named function values

A standard function definition...

```
let add_one (x:int) : int = x+1
```

really has two parts:

```
let add_one : int->int = fun (x:int) -> x+1
```

define a name for  
the value

create a function value

The two definitions have the same type and behave exactly the same.

# Anonymous functions

```
let add_one (z:int) : int = z + 1
let add_two (z:int) : int = z + 2
let y = twice add_one 3
let w = twice add_two 3
```

```
let y = twice (fun (z:int) -> z+1) 3
let w = twice (fun (z:int) -> z+2) 3
```



# Multiple Arguments

We can decompose a standard function definition

```
let sum (x : int) (y:int) : int = x + y
```

into parts

```
let sum = fun (x:int) -> fun (y:int) -> x + y
```

define a variable with  
that value

create a function value

that returns a function value

The two definitions have the same type and behave exactly the same

```
let sum : int -> int -> int
```

# Partial Application

```
let sum (x:int) (y:int) : int = x + y
```

sum 3

→ (fun (x:int) -> fun (y:int) -> x + y) 3 *definition*

→ fun (y:int) -> 3 + y *substitute 3 for x*

# Programming Languages and Techniques (CIS120)

Lecture 8

September 18, 2017

Generics & Higher-order functions

# Announcements

- Homework 2
  - Due tomorrow night at 11:59pm
- See Piazza for how to enable error-highlighting in Codio
- Homework 3 available soon
  - Practice with BSTs, generic functions, HOFs and *abstract types*
- Reading: Chapters 8, 9, and 10 of the lecture notes

# Generic Functions and Data

Wow, implementing BSTs took quite a bit of typing...  
Do we have to repeat it all again if we want to use BSTs  
containing strings, or characters, or floats?

or

*How not to repeat yourself, Part I.*

# Structurally Identical Functions

- Observe: many functions on lists, trees, and other datatypes don't depend on the contents, only on the structure.
- Compare: `length` for “`int list`” vs. “`string list`”

```
let rec length (l: int list) : int =
  begin match l with
  | [] -> 0
  | _ :: tl -> 1 + length tl
  end
```

```
let rec length (l: string list) : int =
  begin match l with
  | [] -> 0
  | _ :: tl -> 1 + length tl
  end
```

The functions are *identical*, except for the type annotation.

They are “*generic*” with respect to the input type.

# Notation for Generic Types

- OCaml provides syntax for functions with *generic* types

```
let rec length l:'a list : int =
  begin match l with
    | [] -> 0
    | _::tl -> 1 + (length tl)
  end
```

- Notation: '*a* is a *type variable*; the function `length` can be used on a `t list` for *any type t*.
- Examples:
  - `length [1;2;3]` use length on an `int list`
  - `length [“a”;”b”;”c”]` use length on a `string list`

# Generic List Append

Note that the two input lists must have the *same* type of elements.

The return type is the same as the inputs.

```
let rec append (l1:'a list) (l2:'a list) : 'a list =
  begin match l1 with
    | [] -> l2
    | h::tl -> h::(append tl l2)
  end
```

Pattern matching works over generic types!

In the body of the branch:

h has type 'a

tl has type 'a list

# Generic Zip

Functions can operate over *multiple* generic types.

```
let rec zip (l1:'a list) (l2:'b list) : ('a*'b) list =
  begin match (l1,l2) with
  | (h1::t1, h2::t2) -> (h1,h2)::(zip t1 t2)
  | _ -> []
  end
```

- Distinct type variables can be instantiated differently:

```
zip [1;2;3] ["a";"b";"c"]
```

- Here, '*a* is instantiated to `int`, '*b* to `string`
- Result is

```
[(1,"a");(2,"b");(3,"c")]
of type (int * string) list
```

Intuition: OCaml tracks instantiations of type variables ('*a* and '*b*) and makes sure they are used consistently in each use of the function.

# User-Defined Generic Datatypes

- Recall our integer tree type:

```
type tree =
| Empty
| Node of tree * int * tree
```

- We can define a generic version by adding a type parameter, like this:

```
type 'a tree =
| Empty
| Node of 'a tree * 'a * 'a tree
```

Parameter '*a*  
used here

Note that the recursive  
uses also mention '*a*

# User-Defined Generic Datatypes

- BST operations can be generic too; only change is to the type annotation

(\* Insert n into the BST t \*)

```
let rec insert (t:'a tree) (n:'a) : 'a tree =
begin match t with
| Empty -> Node(Empty,n,Empty)
| Node(lt,x,rt) ->
  if x = n then t
  else if n < x then Node(insert lt n, x, rt)
  else Node(lt, x, insert rt n)
end
```

Equality and comparison are generic — they work for *any* type of data too.

Does the following function typecheck?

```
let f (l : 'a list) : 'b list =
begin match l with
| [] -> true::l
| _::rest -> 1::l
end
```

1. yes
2. no

Answer: no: even though the return type is generic, the two branches must agree (so that 'b can be consistently instantiated).

Does the following code typecheck?

```
let f (x : 'a) : 'a =
  x + 1

;; print_endline (f "hello")
```

1. yes
2. no

Answer: no, the type annotations and uses of f aren't consistent.

However it is a bit subtle: without the use (f "hello") the code *would* be correct – so long as all uses of f provide only 'int' the code is consistent! Despite the "generic" type annotation, f really has type int -> int.

# First-class Functions

Higher-order Programs

or

How not to repeat yourself, Part II.

# First-class Functions

- You can pass a function as an *argument* to another function:

```
let twice (f:int->int) (x:int) : int =  
    f (f x)
```

```
let add_one (z:int) : int = z + 1  
let add_two (z:int) : int = z + 2  
let y = twice add_one 3  
let w = twice add_two 3
```

function type: argument of type int  
and result of type int

- You can *return* a function as the result of another function.

```
let make_incr (n:int) : int->int =  
    let helper (x:int) : int =  
        n + x  
    in  
    helper  
let y = twice (make_incr 1) 3
```

# Function Types

- Functions have types that look like this:

$$t_{in} \rightarrow t_{out}$$

- Examples:

`int -> int`

`int -> bool * int`

`int -> int -> int`

*int input*

`(int -> int) -> int`

*function input*

# Function Types

- Functions have types that look like this:

$$t_{in} \rightarrow t_{out}$$

- Examples:

`int -> int`

`int -> (bool * int)`

`int -> (int -> int)`

`(int -> int) -> int`

Parentheses matter!

`int -> int -> int` is equivalent to  
`int -> (int -> int)` but not to  
`(int -> int) -> int`

*int input*

*function input*

# First-class Functions

- You can store functions in data structures

```
let add_one    (x:int) : int = x+1
let add_two    (x:int) : int = x+2
let add_three  (x:int) : int = x+3

let func_list : (int -> int) list =
  [ add_one; add_two; add_three ]
```

A list of  $(\text{int} \rightarrow \text{int})$  functions.

```
let func_list1 : (int -> int) list =
  [ make_incr 1; make_incr 2; make_incr 3 ]
```

# Using A List of Functions

```
let rec compose (fs:(int->int) list) (x:int) :  
int =  
  begin match fs with  
  | [] -> x  
  | f::rest -> f (compose rest x)  
  end  
  
let ans : int = compose func_list1 0
```

# Simplifying First-Class Functions

```
let twice (f:int->int) (x:int) : int =  
  f (f x)
```

```
let add_one (z:int) : int = z + 1
```

twice add\_one 3  
→ add\_one (add\_one 3)  
→ add\_one (3 + 1)  
→ add\_one 4  
→ 4 + 1  
→ 5

*substitute add\_one for f, 3 for x*  
*substitute 3 for z in add\_one*  
 $3+1 \Rightarrow 4$   
*substitute 4 for z in add\_one*  
 $4+1 \Rightarrow 5$

# Simplifying First-Class Functions

```
let make_incr (n:int) : int->int =  
  let helper (x:int) : int = n + x in  
  helper
```

make\_incr 3

*substitute 3 for n*

↪ let helper (x:int) = 3 + x in helper

↪ ???

# Simplifying First-Class Functions

```
let make_incr (n:int) : int->int =  
  let helper (x:int) : int = n + x in  
    helper
```

make\_incr 3

*substitute 3 for n*

↪ let helper (x:int) = 3 + x in helper

↪ fun (x:int) -> 3 + x

Anonymous function value

keyword “fun”

“->” after arguments  
no return type annotation

# Named function values

A standard function definition...

```
let add_one (x:int) : int = x+1
```

really has two parts:

```
let add_one : int->int = fun (x:int) -> x+1
```

define a name for  
the value

create a function value

The two definitions have the same type and behave exactly the same.

# Anonymous functions

```
let add_one (z:int) : int = z + 1
let add_two (z:int) : int = z + 2
let y = twice add_one 3
let w = twice add_two 3
```

```
let y = twice (fun (z:int) -> z+1) 3
let w = twice (fun (z:int) -> z+2) 3
```



# Multiple Arguments

We can decompose a standard function definition

```
let sum (x : int) (y:int) : int = x + y
```

into parts

```
let sum = fun (x:int) -> fun (y:int) -> x + y
```

define a variable with  
that value

create a function value

that returns a function value

The two definitions have the same type and behave exactly the same

```
let sum : int -> int -> int
```

# Partial Application

```
let sum (x:int) (y:int) : int = x + y
```

sum 3

→ (fun (x:int) -> fun (y:int) -> x + y) 3 *definition*

→ fun (y:int) -> 3 + y *substitute 3 for x*

# List transformations

A fundamental design pattern  
using first-class functions

# Phone book example

```
type entry = string * int
let phone_book = [ ("Steve", 2155559092), ... ]  
  
let rec get_names (p : entry list) : string list =
  begin match p with
    | ((name, num)::rest) -> name :: get_names rest
    | [] -> []
  end  
  
let rec get_numbers (p : entry list) : int list =
  begin match p with
    | ((name, num)::rest) -> num :: get_numbers rest
    | [] -> []
  end
```

Can we use first-class functions  
to refactor code to share common  
structure?

# Refactoring

```
let rec helper (f:entry -> 'b) (p:entry list) : 'b list =
begin match p with
| (entry::rest) -> f entry :: helper f rest
| [] -> []
end
```

```
let get_names (p : entry list) : string list =
  helper fst p
let get_numbers (p : entry list) : int list =
  helper snd p
```

fst and snd are functions that access the parts of a tuple:

```
let fst (x,y) = x
let snd (x,y) = y
```

The argument `f` determines what happens with the entry at the head of the list

# Going even more generic

```
let rec helper (f:entry -> 'b) (p:entry list) : 'b list =
begin match p with
| (entry::rest) -> f entry :: helper f rest
| [] -> []
end
```

```
let get_names (p : entry list) : string list =
  helper fst p
let get_numbers (p : entry list) : int list =
  helper snd p
```

Now let's make it work for *all* lists,  
not just lists of entries...

# Going even more generic

```
let rec helper (f:'a -> 'b) (p:'a list) : 'b list =
begin match p with
| (entry::rest) -> f entry :: helper f rest
| [] -> []
end
```

```
let get_names (p : entry list) : string list =
helper fst p
```

```
let get_numbers (p : entry list) : int list =
helper snd p
```

‘a stands for (string\*int)  
‘b stands for int

snd : (string\*int) -> int

# Transforming Lists

```
let rec transform (f:'a -> 'b) (l:'a list) : 'b list =
  begin match l with
  | []    -> []
  | h::t -> (f h)::(transform f t)
  end
```

List transformation (a.k.a. “*mapping* a function across a list”\*)

- foundational function for programming with lists
- occurs over and over again
- part of OCaml standard library (called List.map)

Example of using transform:

```
transform is_engr ["FNCE"; "CIS"; "ENGL"; "DMD"] =
  [false; true; false; true]
```

\*many languages (including OCaml) use the terminology “map” for the function that transforms a list by applying a function to each element. Don’t confuse List.map with “finite map”.

What is the value of this expression?

```
transform (fun (x:int) -> x>0)
[0 ; -1; 1; -2]
```

1. [0; -1; 1; -2]
2. [1]
3. [0; 1]
4. [false; false; true; false]
5. runtime error

Answer: 4

# The ‘fold’ design pattern

# Refactoring code, again

- Is there a pattern in the definition of these two functions?

```
let rec exists (l : bool list) : bool =
begin match l with
| [] -> false
| h :: t -> h || exists t
end
```

*base case:*  
Simple answer when  
the list is empty

```
let rec acid_length (l : acid list) : int =
begin match l with
| [] -> 0
| h :: t -> 1 + acid_length t
end
```

*combine step:*  
Do something with  
the head of the list  
and the result of the  
recursive call

- Can we factor out this pattern using first-class functions?

# Preparation

```
let rec exists (l : bool list) : bool =
begin match l with
| [] -> false
| h :: t -> h || exists t
end
```

```
let rec acid_length (l : acid list) : int =
begin match l with
| [] -> 0
| h :: t -> 1 + acid_length t
end
```

# Preparation

```
let rec helper (l : bool list) : bool =
begin match l with
| [] -> false
| h :: t -> h || helper t
end
```

```
let exists (l : bool list) = helper l
```

```
let rec helper (l : acid list) : int =
begin match l with
| [] -> 0
| h :: t -> 1 + helper t
end
```

```
let acid_length (l : acid list) = helper l
```

# Abstracting with respect to Base

```
let rec helper (l : bool list) : bool =
  begin match l with
  | [] -> false
  | h :: t -> h || helper t
  end
```

```
let exists (l : bool list) = helper l
```

```
let rec helper (l : acid list) : int =
  begin match l with
  | [] -> 0
  | h :: t -> 1 + helper t
  end
```

```
let acid_length (l : acid list) = helper l
```

# Abstracting with respect to Base

```
let rec helper (base : bool) (l : bool list) : bool =
  begin match l with
  | [] -> base
  | h :: t -> h || helper base t
  end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
  begin match l with
  | [] -> base
  | h :: t -> 1 + helper base t
  end

let acid_length (l : acid list) = helper 0 l
```

# Abstracting with respect to Combine

```
let rec helper (base : bool) (l : bool list) : bool =
  begin match l with
  | [] -> base
  | h :: t -> h || helper base t
  end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
  begin match l with
  | [] -> base
  | h :: t -> 1 + helper base t
  end

let acid_length (l : acid list) = helper 0 l
```

# Abstracting with respect to Combine

```
let rec helper (base : bool) (l : bool list) : bool =
  begin match l with
  | [] -> base
  | h :: t -> h || helper base t
  end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
  begin match l with
  | [] -> base
  | h :: t -> 1 + helper base t
  end

let acid_length (l : acid list) = helper 0 l
```

# Abstracting with respect to Combine

```
let rec helper (combine : bool -> bool -> bool)
              (base : bool) (l : bool list) : bool =
  begin match l with
  | [] -> base
  | h :: t -> combine h (helper combine base t)
  end
```

```
let exists (l : bool list) =
  helper (fun (h:bool) (acc:bool) -> h || acc) false l
```

```
let rec helper (combine : acid -> int -> int)
              (base : int) (l : acid list) : int =
  begin match l with
  | [] -> base
  | h :: t -> combine h (helper combine base t)
  end
```

```
let acid_length (l : acid list) =
  helper (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

# Making the Helper Generic

```
let rec helper (combine : 'a -> 'b -> 'b)
              (base : 'b) (l : 'a list) : 'b =
  begin match l with
  | [] -> base
  | h :: t -> combine h (helper combine base t)
  end

let exists (l : bool list) =
  helper (fun (h:bool) (acc:bool) -> h || acc) false l
```

```
let rec helper (combine : 'a -> 'b -> 'b)
              (base : 'b) (l : 'a list) : 'b =
  begin match l with
  | [] -> base
  | h :: t -> combine h (helper combine base t)
  end

let acid_length (l : acid list) =
  helper (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

# List Fold

```
let rec fold (combine: 'a -> 'b -> 'b)
            (base:'b) (l : 'a list) : 'b =
  begin match l with
  | [] -> base
  | x :: t -> combine x (fold combine base t)
  end

let exists (l : bool list) : bool =
  fold (fun (h:bool) (acc:bool) -> h || acc) false l

let acid_length (l : acid list) : int =
  fold (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

- **fold** (a.k.a. Reduce)
  - Like transform, foundational function for programming with lists
  - Captures the pattern of recursion over lists
  - Also part of OCaml standard library (`List.fold_right`)
  - Similar operations for other recursive datatypes (`fold_tree`)

How would you rewrite this function

```
let rec sum (l : int list) : int =  
begin match l with  
| [] -> 0  
| h :: t -> h + sum t  
end
```

using fold? What should be the arguments for base and combine?

1. combine is: (fun (h:int) (acc:int) -> acc + 1)  
base is: 0
2. combine is: (fun (h:int) (acc:int) -> h + acc)  
base is: 0
3. combine is: (fun (h:int) (acc:int) -> h + acc)  
base is: 1
4. sum can't be written with fold.

Answer: 2

How would you rewrite this function

```
let rec reverse (l : int list) : int list =  
begin match l with  
| [] -> []  
| h :: t -> reverse t @ [h]  
end
```

using fold? What should be the arguments for base and combine?

1. combine is: (fun (h:int) (acc:int list) -> h :: acc)  
base is: 0
2. combine is: (fun (h:int) (acc:int list) -> acc @ [h])  
base is: 0
3. combine is: (fun (h:int) (acc:int list) -> acc @ [h])  
base is: []
4. reverse can't be written by with fold.

Answer: 3

# Functions as Data

- We've seen a number of ways in which functions can be treated as data in OCaml
- Everyday programming practice offers many more examples
  - objects bundle “functions” (a.k.a. methods) with data
  - iterators (“cursors” for walking over data structures)
  - event listeners (in GUIs)
  - etc.
- Also heavily used at “large scale”: Google's MapReduce
  - Framework for transforming (mapping) sets of key-value pairs
  - Then “reducing” the results per key of the map
  - Easily distributed to 10,000 machines to execute in parallel!

# Programming Languages and Techniques (CIS120)

## Lecture 9

September 20, 2017

List transformations

Lecture notes: Chapter 10

# Announcements

- HW02 EXTENSION: due *tonight* at midnight
  - due to Codio availability issues
- Homework 3 is available now
  - due *Tuesday, Sept. 26 at 11:59pm*
- Read Chapter 10 of lecture notes

# List transformations

A fundamental design pattern  
using first-class functions

# Phone book example

```
type entry = string * int
let phone_book = [ ("Steve", 2155559092), ... ]  
  
let rec get_names (p : entry list) : string list =
  begin match p with
  | ((name, num)::rest) -> name :: get_names rest
  | [] -> []
  end  
  
let rec get_numbers (p : entry list) : int list =
  begin match p with
  | ((name, num)::rest) -> num :: get_numbers rest
  | [] -> []
  end
```

Can we use first-class functions  
to refactor code to share common  
structure?

# Refactoring

```
let rec helper (f:entry -> 'b) (p:entry list) : 'b list =
begin match p with
| (entry::rest) -> f entry :: helper f rest
| [] -> []
end
```

```
let get_names (p : entry list) : string list =
  helper fst p
let get_numbers (p : entry list) : int list =
  helper snd p
```

fst and snd are functions that access the parts of a tuple:

```
let fst (x,y) = x
let snd (x,y) = y
```

The argument `f` determines what happens with the entry at the head of the list

# Going even more generic

```
let rec helper (f:entry -> 'b) (p:entry list) : 'b list =
begin match p with
| (entry::rest) -> f entry :: helper f rest
| [] -> []
end

let get_names (p : entry list) : string list =
  helper fst p
let get_numbers (p : entry list) : int list =
  helper snd p
```

Now let's make it work for *all* lists,  
not just lists of entries...

# Going even more generic

```
let rec helper (f:'a -> 'b) (p:'a list) : 'b list =
begin match p with
| (entry::rest) -> f entry :: helper f rest
| [] -> []
end
```

```
let get_names (p : entry list) : string list =
helper fst p
let get_numbers (p : entry list) : int list =
helper snd p
```

‘a stands for (string\*int)  
‘b stands for int

snd : (string\*int) -> int

# Transforming Lists

```
let rec transform (f:'a -> 'b) (l:'a list) : 'b list =
  begin match l with
  | []    -> []
  | h::t -> (f h)::(transform f t)
  end
```

List transformation (a.k.a. “*mapping* a function across a list”\*)

- foundational function for programming with lists
- occurs over and over again
- part of OCaml standard library (called List.map)

Example of using transform:

```
transform is_engr ["FNCE"; "CIS"; "ENGL"; "DMD"] =
  [false; true; false; true]
```

\*many languages (including OCaml) use the terminology “map” for the function that transforms a list by applying a function to each element. Don’t confuse List.map with “finite map”.

What is the value of this expression?

```
transform (fun (x:int) -> x>0)  
[0 ; -1; 1; -2]
```

1. [0; -1; 1; -2]
2. [1]
3. [1; 1; 0; 1]
4. [false; false; true; false]
5. runtime error

ANSWER: 4

# The ‘fold’ design pattern

# Refactoring code, again

- Is there a pattern in the definition of these two functions?

```
let rec exists (l : bool list) : bool =
begin match l with
| [] -> false
| h :: t -> h || exists t
end
```

```
let rec acid_length (l : acid list) : int =
begin match l with
| [] -> 0
| h :: t -> 1 + acid_length t
end
```

*base case:*  
Simple answer when  
the list is empty

*combine step:*  
Do something with  
the head of the list  
and the result of the  
recursive call

- Can we factor out this pattern using first-class functions?

# Preparation

```
let rec exists (l : bool list) : bool =
begin match l with
| [] -> false
| h :: t -> h || exists t
end
```

```
let rec acid_length (l : acid list) : int =
begin match l with
| [] -> 0
| h :: t -> 1 + acid_length t
end
```

# Preparation

```
let rec helper (l : bool list) : bool =
begin match l with
| [] -> false
| h :: t -> h || helper t
end

let exists (l : bool list) = helper l
```

```
let rec helper (l : acid list) : int =
begin match l with
| [] -> 0
| h :: t -> 1 + helper t
end

let acid_length (l : acid list) = helper l
```

# Abstracting with respect to Base

```
let rec helper (l : bool list) : bool =
begin match l with
| [] -> false
| h :: t -> h || helper t
end

let exists (l : bool list) = helper l
```

```
let rec helper (l : acid list) : int =
begin match l with
| [] -> 0
| h :: t -> 1 + helper t
end

let acid_length (l : acid list) = helper l
```

# Abstracting with respect to Base

```
let rec helper (base : bool) (l : bool list) : bool =
  begin match l with
  | [] -> base
  | h :: t -> h || helper base t
  end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
  begin match l with
  | [] -> base
  | h :: t -> 1 + helper base t
  end

let acid_length (l : acid list) = helper 0 l
```

# Abstracting with respect to Combine

```
let rec helper (base : bool) (l : bool list) : bool =
  begin match l with
  | [] -> base
  | h :: t -> h || helper base t
  end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
  begin match l with
  | [] -> base
  | h :: t -> 1 + helper base t
  end

let acid_length (l : acid list) = helper 0 l
```

# Abstracting with respect to Combine

```
let rec helper (base : bool) (l : bool list) : bool =
  begin match l with
  | [] -> base
  | h :: t -> h || helper base t
  end

let exists (l : bool list) = helper false l
```

```
let rec helper (base : int) (l : acid list) : int =
  begin match l with
  | [] -> base
  | h :: t -> 1 + helper base t
  end

let acid_length (l : acid list) = helper 0 l
```

# Abstracting with respect to Combine

```
let rec helper (combine : bool -> bool -> bool)
              (base : bool) (l : bool list) : bool =
  begin match l with
  | [] -> base
  | h :: t -> combine h (helper combine base t)
  end

let exists (l : bool list) =
  helper (fun (h:bool) (acc:bool) -> h || acc) false l
```

```
let rec helper (combine : acid -> int -> int)
              (base : int) (l : acid list) : int =
  begin match l with
  | [] -> base
  | h :: t -> combine h (helper combine base t)
  end

let acid_length (l : acid list) =
  helper (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

# Making the Helper Generic

```
let rec helper (combine : 'a -> 'b -> 'b)
              (base : 'b) (l : 'a list) : 'b =
  begin match l with
  | [] -> base
  | h :: t -> combine h (helper combine base t)
  end

let exists (l : bool list) =
  helper (fun (h:bool) (acc:bool) -> h || acc) false l
```

```
let rec helper (combine : 'a -> 'b -> 'b)
              (base : 'b) (l : 'a list) : 'b =
  begin match l with
  | [] -> base
  | h :: t -> combine h (helper combine base t)
  end

let acid_length (l : acid list) =
  helper (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

# List Fold

```
let rec fold (combine: 'a -> 'b -> 'b)
            (base:'b) (l : 'a list) : 'b =
  begin match l with
  | [] -> base
  | x :: t -> combine x (fold combine base t)
  end

let exists (l : bool list) : bool =
  fold (fun (h:bool) (acc:bool) -> h || acc) false l

let acid_length (l : acid list) : int =
  fold (fun (h:acid) (acc:int) -> 1 + acc) 0 l
```

- **fold (a.k.a. Reduce)**
  - Like transform, foundational function for programming with lists
  - Captures the pattern of recursion over lists
  - Also part of OCaml standard library (`List.fold_right`)
  - Similar operations for other recursive datatypes (`fold_tree`)

How would you rewrite this function

```
let rec sum (l : int list) : int =
  begin match l with
  | [] -> 0
  | h :: t -> h + sum t
  end
```

using fold? What should be the arguments for base and combine?

1. combine is: (fun (h:int) (acc:int) -> acc + 1)  
base is: 0
2. combine is: (fun (h:int) (acc:int) -> h + acc)  
base is: 0
3. combine is: (fun (h:int) (acc:int) -> h + acc)  
base is: 1
4. sum can't be written with fold.

Answer: 2

How would you rewrite this function

```
let rec reverse (l : int list) : int list =
  begin match l with
    | [] -> []
    | h :: t -> reverse t @ [h]
  end
```

using fold? What should be the arguments for base and combine?

1. combine is: (fun (h:int) (acc:int list) -> h :: acc)  
base is: 0
2. combine is: (fun (h:int) (acc:int list) -> acc @ [h])  
base is: 0
3. combine is: (fun (h:int) (acc:int list) -> acc @ [h])  
base is: []
4. reverse can't be written by with fold.

Answer: 3

# Functions as Data

- We've seen a number of ways in which functions can be treated as data in OCaml
- Everyday programming practice offers many more examples
  - objects bundle “functions” (a.k.a. methods) with data
  - iterators (“cursors” for walking over data structures)
  - event listeners (in GUIs)
  - etc.
- Also heavily used at “large scale”: Google's MapReduce
  - Framework for transforming (mapping) sets of key-value pairs
  - Then “reducing” the results per key of the map
  - Easily distributed to 10,000 machines to execute in parallel!

# Abstract Collections

Are you familiar with the idea of a *set* from mathematics?

In math, we typically write sets like this:

$\emptyset$  {1,2,3} {true,false}

with operations:

$S \cup T$  for union and

$S \cap T$  for intersection;

we write  $x \in S$  for the predicate

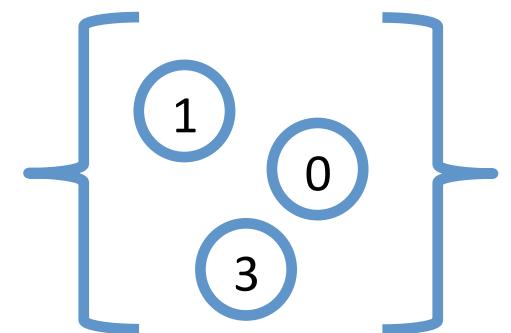
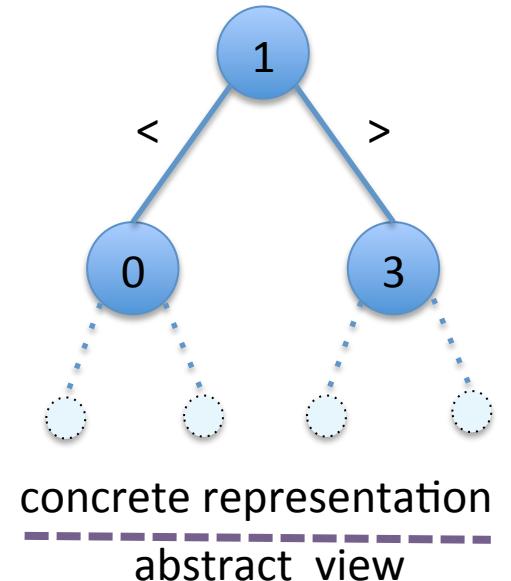
“x is a member of the set S”

# A *set* is an abstraction

- A set is a collection of data
    - we have operations for forming sets of elements
    - we can ask whether elements are in a set
  - A set is a lot like a list, except:
    - Order doesn't matter
    - Duplicates don't matter
    - *It isn't built into OCaml*
  - Sets show up frequently in applications
    - Examples: set of students in a class, set of coordinates in a graph, set of answers to a survey, set of data samples from an experiment, ...
- 
- An element's *presence* or *absence* in the set is all that matters...

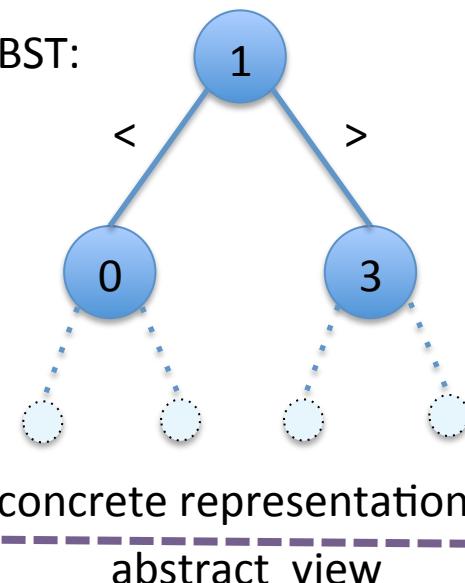
# Abstract type: set

- A BST can *implement (represent)* a *set*
  - there is a way to represent an empty set (*Empty*)
  - there is a way to list all elements contained in the set (*inorder*)
  - there is a way to test membership (*lookup*)
  - Can define union/intersection (with *insert* and *delete*)
- *BSTs are not the only way to implement sets*



# Three Example Representations of Sets

BST:

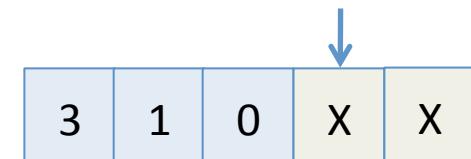


Alternate representation:  
unsorted linked list.

3::0::1::[]

concrete representation  
-----  
abstract view

Alternate representation:  
reverse sorted array with  
Index of next slot.



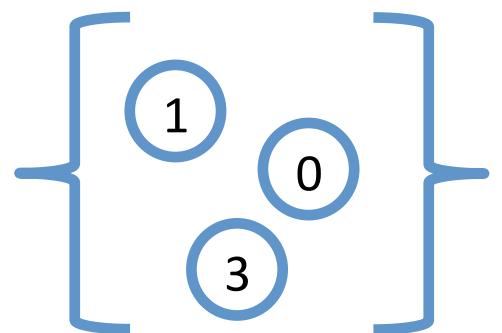
concrete representation  
-----  
abstract view

# Abstract types (e.g. set)

- An abstract type is defined by its *interface* and its *properties*, not its representation.
- **Interface:** defines operations on the type
  - There is an empty set
  - There is a way to add elements to a set to make a bigger set
  - There is a way to list all elements in a set
  - There is a way to test membership
- **Properties:** define how the operations interact with each other
  - Elements that were added can be found in the set
  - Adding an element a second time doesn't change the elements of a set
  - Adding in a different order doesn't change the elements of a set
- **Any type (possibly with invariants) that satisfies the interface and properties can be a set.**



concrete representation  
-----  
abstract view



# Sets in OCaml

# Set Signature

The name of the signature.

```
module type SET = sig
```

The **sig** keyword indicates an interface declaration

```
type 'a set
```

Type declaration has no “right-hand side” – its representation is *abstract*!

```
val empty
```

```
: 'a set
```

```
val add
```

```
: 'a -> 'a set -> 'a set
```

```
val member
```

```
: 'a -> 'a set -> bool
```

```
val equals
```

```
: 'a set -> 'a set -> bool
```

```
val set_of_list : 'a list -> 'a set
```

```
end
```

The interface members are the (only!) means of manipulating the abstract type.

Signature (a.k.a. Interface): defines operations on the type

# Implementing sets

- There are many ways to implement sets.
  - lists, trees, arrays, etc.
- *How do we choose which implementation?*
  - Depends on the needs of the application...
  - How often is ‘member’ used vs. ‘add’?
  - How big can the sets be?
- Many such implementations are of the flavor “a set is a ... with some invariants”
  - A set is a *list* with no repeated elements.
  - A set is a *tree* with no repeated elements
  - A set is a *binary search tree*
  - A set is an *array of bits*, where 0 = absent, 1 = present
- *How do we preserve the invariants of the implementation?*

# A *module* implements an interface

- An implementation of the set interface will look like this:

```
module BSTSet : SET = struct
  ...
  (* implementations of all the operations *)
  ...
end
```

Name of the module

Signature that it implements

The **struct** keyword indicates a module implementation

The diagram illustrates the components of a module implementation. It shows a code snippet within a blue box. Three arrows point from three callout boxes above the code to specific parts of the code:

- An arrow points from the top-left box labeled "Name of the module" to the word "BSTSet" in the first line of the code.
- An arrow points from the middle box labeled "Signature that it implements" to the type annotation ": SET" in the first line of the code.
- An arrow points from the bottom-right box labeled "The **struct** keyword indicates a module implementation" to the keyword "struct" in the second line of the code.

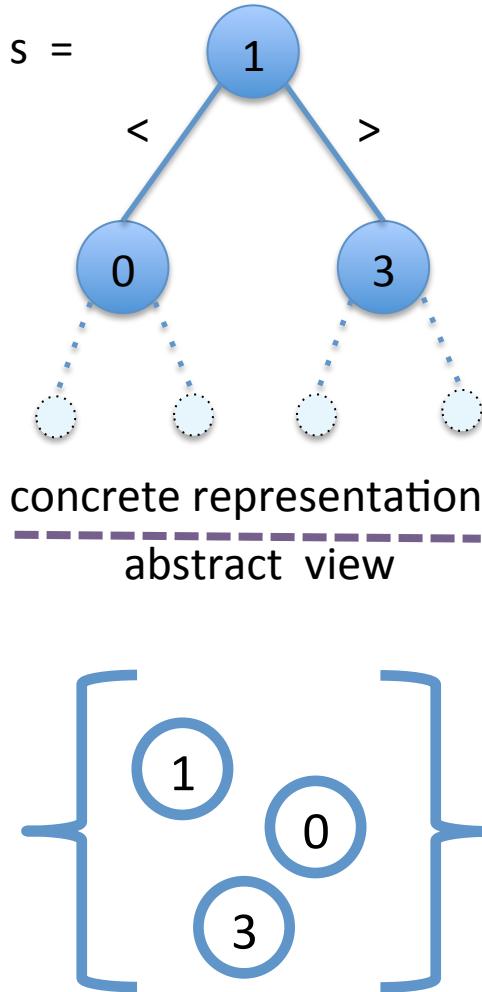
# Implement the set Module

```
module BSTSet : SET = struct  
  
  type 'a tree =  
    | Empty  
    | Node of 'a tree * 'a * 'a tree  
  
  type 'a set = 'a tree  
  
  let empty : 'a set = Empty  
  ...  
end
```

Module must define (give a *concrete representation* to) the type declared in the signature

- The implementation has to include everything promised by the interface
  - It can contain *more* functions and type definitions (e.g. auxiliary or helper functions) but those cannot be used outside the module
  - The types of the provided implementations must match the interface

# Abstract vs. Concrete BSTSet



```
module BSTSet : SET = struct
  type 'a tree = ...
  type 'a set = 'a tree
  let empty : 'a set = Empty
  let add (x:'a) (s:'a set) :'a set =
    ... (* can treat s as a tree *)
end
```

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end
```

```
(* A client of the BSTSet module *)
;; open BSTSet
```

```
let s : int set
= add 0 (add 3 (add 1 empty))
```

# Another Implementation

```
module ULSet : SET =  
struct
```

```
  type 'a set = 'a list
```

A different definition for  
the type set

```
  let empty : 'a set = []
```

```
  ...
```

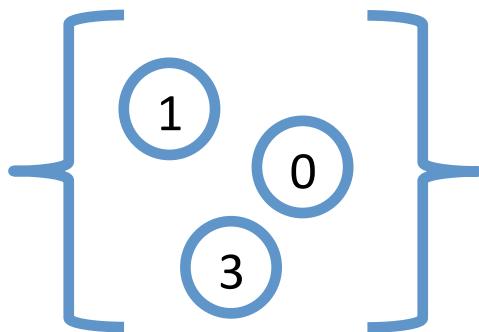
```
end
```

# Abstract vs. Concrete ULSet

`s = 0::3::1::[]`

```
module ULSet : SET = struct
  type 'a set = 'a list
  let empty : 'a set = []
  let add (x:'a) (s:'a set) : 'a set =
    x::s (* can treat s as a list *)
end
```

concrete representation  
-----  
abstract view



```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end
```

(\* A client of the ULSet module \*)  
;; open ULSet

```
let s : int set
= add 0 (add 3 (add 1 empty))
```

Client code doesn't change!

# Programming Languages and Techniques (CIS120)

## Lecture 10

September 22, 2017

Abstract types: Sets

# Announcements

- Homework 3
  - due *Tuesday* at 11:59:59pm
- Midterm 1
  - *Friday, October 13, in class*
  - Covers material through Chapter 13
  - Review materials (old exams) on course website
  - Review session TBA

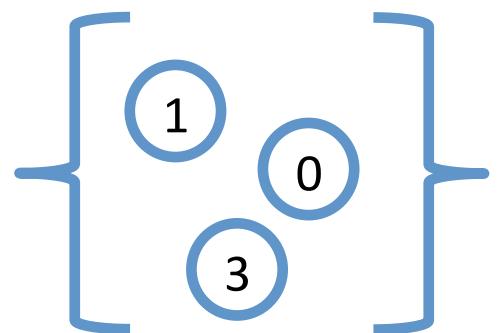
# Abstract Collections

# Abstract types (e.g. set)

- An abstract type is defined by its *interface* and its *properties*, not its representation.
- **Interface:** defines operations on the type
  - There is an empty set
  - There is a way to add elements to a set to make a bigger set
  - There is a way to list all elements in a set
  - There is a way to test membership
- **Properties:** define how the operations interact with each other
  - Elements that were added can be found in the set
  - Adding an element a second time doesn't change the elements of a set
  - Adding in a different order doesn't change the elements of a set
- **Any type (possibly with invariants) that satisfies the interface and properties can be a set.**



concrete representation  
-----  
abstract view



# Sets in OCaml

# Set Signature

The name of the signature.

```
module type SET = sig
```

The **sig** keyword indicates an interface declaration

```
type 'a set
```

Type declaration has no “right-hand side” – its representation is *abstract*!

```
val empty
```

```
: 'a set
```

```
val add
```

```
: 'a -> 'a set -> 'a set
```

```
val member
```

```
: 'a -> 'a set -> bool
```

```
val equals
```

```
: 'a set -> 'a set -> bool
```

```
val set_of_list : 'a list -> 'a set
```

```
end
```

The interface members are the (only!) means of manipulating the abstract type.

Signature (a.k.a. Interface): defines operations on the type

# Implementing sets

- There are many ways to implement sets.
  - lists, trees, arrays, etc.
- *How do we choose which implementation?*
  - Depends on the needs of the application...
  - How often is ‘member’ used vs. ‘add’?
  - How big can the sets be?
- Many such implementations are of the flavor  
“a set is a ... with some *invariants*”
  - A set is a *list* with no repeated elements.
  - A set is a *tree* with no repeated elements
  - A set is a *binary search tree*
  - A set is an *array of bits*, where 0 = absent, 1 = present
- *How do we preserve the invariants of the implementation?*

*Invariant:* a property that remains unchanged when a specified transformation is applied.

# A *module* implements an interface

- An implementation of the set interface will look like this:

```
module BSTSet : SET = struct
  ...
  (* implementations of all the operations *)
  ...
end
```

Name of the module

Signature that it implements

The **struct** keyword indicates a module implementation

The diagram illustrates the components of a module implementation. It shows a code snippet within a blue box. Three arrows point from three callout boxes above the code to specific parts of the code:

- An arrow points from the top-left box labeled "Name of the module" to the word "BSTSet" in the first line of the code.
- An arrow points from the middle box labeled "Signature that it implements" to the type annotation ": SET" in the first line of the code.
- An arrow points from the bottom-right box labeled "The **struct** keyword indicates a module implementation" to the keyword "struct" in the second line of the code.

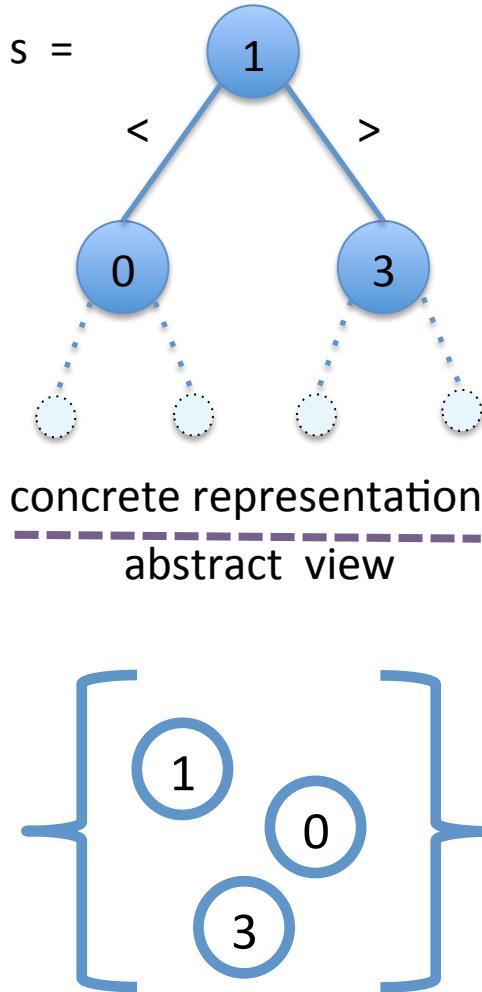
# Implement the set Module

```
module BSTSet : SET = struct  
  
  type 'a tree =  
    | Empty  
    | Node of 'a tree * 'a * 'a tree  
  
  type 'a set = 'a tree  
  
  let empty : 'a set = Empty  
  ...  
end
```

Module must define (give a *concrete representation* to) the type declared in the signature

- The implementation has to include everything promised by the interface
  - It can contain *more* functions and type definitions (e.g. auxiliary or helper functions) but those cannot be used outside the module
  - The types of the provided implementations must match the interface

# Abstract vs. Concrete BSTSet



```
module BSTSet : SET = struct
  type 'a tree = ...
  type 'a set = 'a tree
  let empty : 'a set = Empty
  let add (x:'a) (s:'a set) :'a set =
    ... (* can treat s as a tree *)
end
```

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end
```

```
(* A client of the BSTSet module *)
;; open BSTSet
```

```
let s : int set
= add 0 (add 3 (add 1 empty))
```

# Another Implementation

```
module ULSet : SET =  
struct
```

```
  type 'a set = 'a list
```

A different definition for  
the type set

```
  let empty : 'a set = []
```

```
  ...
```

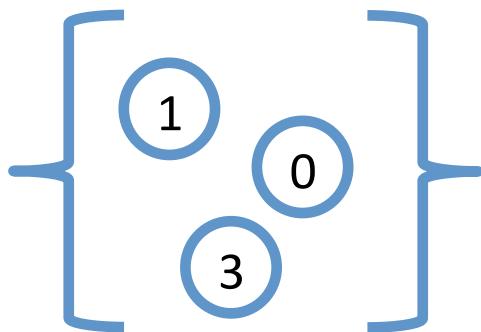
```
end
```

# Abstract vs. Concrete ULSet

`s = 0::3::1::[]`

```
module ULSet : SET = struct
  type 'a set = 'a list
  let empty : 'a set = []
  let add (x:'a) (s:'a set) : 'a set =
    x::s (* can treat s as a list *)
end
```

concrete representation  
-----  
abstract view



```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end
```

(\* A client of the ULSet module \*)  
;; open ULSet

```
let s : int set
= add 0 (add 3 (add 1 empty))
```

Client code doesn't change!

# Completing ULSet

See sets.ml

# Testing (and using) sets

- To use the values defined in the set module use the “dot” syntax:

`ULSet.<member>`

- Note: Module names must be capitalized in OCaml

```
let s1 = ULSet.add 3 ULSet.empty
let s2 = ULSet.add 4 ULSet.empty
let s3 = ULSet.add 4 s1

let test () : bool = (ULSet.member 3 s1)
;; run_test "ULSet.member 3 s1" test

let test () : bool = (ULSet.member 4 s3)
;; run_test "ULSet.member 4 s3" test
```

# Testing (and using) sets

- Alternatively, use “`open`” to bring all of the names defined in the interface into scope.

```
;; open ULSet

let s1 = add 3 empty
let s2 = add 4 empty
let s3 = add 4 s1

let test () : bool = (member 3 s1)
;; run_test "ULSet.member 3 s1" test

let test () : bool = (member 4 s3)
;; run_test "ULSet.member 4 s3" test
```

Does this code type check?

```
;; open BSTSet  
let s1 : int set = add 1 empty
```

1. yes
2. no

```
module type SET = sig  
  type 'a set  
  val empty : 'a set  
  val add   : 'a -> 'a set -> 'a set  
end  
  
module BSTSet : SET = struct  
  type 'a tree =  
    | Empty  
    | Node of 'a tree * 'a * 'a tree  
  type 'a set = 'a tree  
  let empty : 'a set = Empty  
  ...  
end
```

Answer: yes

Does this code type check?

```
;; open BSTSet
let s1 = add 1 empty
let i1 = begin match s1 with
    | Node (_,k,_) -> k
    | Empty -> failwith "impossible"
  end
```

1. yes
2. no

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add   : 'a -> 'a set -> 'a set
end

module BSTSet : SET = struct
  type 'a tree =
    | Empty
    | Node of 'a tree * 'a * 'a tree
  type 'a set = 'a tree
  let empty : 'a set = Empty
  ...
end
```

Answer: no, add constructs a set, not a tree

Does this code type check?

```
;; open BSTSet
let s1 = add 1 empty
let i1 = size s1
```

1. yes
2. no

```
module type SET = sig
  type 'a set
  val empty : 'a set
  val add    : 'a -> 'a set -> 'a set
end

module BSTSet : SET = struct
  type 'a tree =
    | Empty
    | Node of 'a tree * 'a * 'a tree
  type 'a set = 'a tree
  let empty : 'a set = Empty
  let size (t : 'a tree) : int = ...
  ...
end
```

Answer: no, cannot access helper functions outside the module

Does this code type check?

```
;; open BSTSet  
let s1 : int set = Empty
```

1. yes
2. no

```
module type SET = sig  
  type 'a set  
  val empty : 'a set  
  val add   : 'a -> 'a set -> 'a set  
end  
  
module BSTSet : SET = struct  
  type 'a tree =  
    | Empty  
    | Node of 'a tree * 'a * 'a tree  
  type 'a set = 'a tree  
  let empty : 'a set = Empty  
  ...  
end
```

Answer: no, the Empty data constructor is not available outside the module

If a client module works correctly and starts with:

`; ; open ULSet`

will it continue to work if we change that line to:

`; ; open BSTSet`

assuming that ULSet and BSTSet both implement SET and satisfy all of the set properties?

1. yes
2. no

Answer: yes (though performance may be different)

```

module type SET = sig
  type 'a set
  val empty : 'a set
  val add    : 'a -> 'a set -> 'a set
  val member : 'a -> 'a set -> bool
end

module BSTSet : SET = struct
  type 'a tree =
    | Empty
    | Node of 'a tree * 'a * 'a tree
  type 'a set = 'a tree
  let empty : 'a set = Empty
  ...
end

```

Is it possible for a client to call `member` with a tree that is not a BST?

1. yes
2. no

No: the BSTSet operations preserve the BST invariants.  
 there is no way to construct a non-BST tree using the interface.

# Abstract types

BIG IDEA: Hide the *concrete representation* of a type behind an *abstract interface* to preserve invariants

- The interface **restricts** how other parts of the program can interact with the data.
- Benefits:
  - **Safety:** The other parts of the program can't break any invariants
  - **Modularity:** It is possible to change the implementation without changing the rest of the program

# Summary: Abstract Types

- Different programming languages have different ways of letting you define abstract types
- At a minimum, this means providing:
  - A way to specify (write down) an interface
  - A means of hiding implementation details (*encapsulation*)
- In OCaml:
  - Interfaces are specified using a *signature* or *interface*
  - Encapsulation is achieved because the interface can *omit* information
    - type definitions
    - names and types of auxiliary functions
  - Clients *cannot* mention values or types not named in the interface

## Bonus Material: OCaml Details

module and interface files

# .ml and .mli files

- You've already been using signatures and modules in OCaml.
- A series of type and val declarations stored in a file foo.mli is considered as defining a signature FOO
- A series of top-level definitions stored in a file foo.ml is considered as defining a module Foo

foo.mli

```
type t
val z : t
val f : t -> int
```

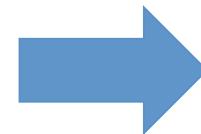
foo.ml

```
type t = int
let z : t = 0
let f (x:t) : int =
  x + 1
```

test.ml

```
;; open Foo
;; print_int
  (Foo.f Foo.z)
```

Files



```
module type FOO = sig
  type t
  val z : t
  val f : t -> int
end
```

```
module Foo : FOO = struct
  type t = int
  let z : t = 0
  let f (x:t) : int =
    x + 1
end
```

```
module Test = struct
  ;; open Foo
  ;; print_int
    (Foo.f Foo.z)
end
```

# Programming Languages and Techniques (CIS120)

## Lecture 11

September 25, 2017

Review: Abstract types: Finite Maps

# Announcements

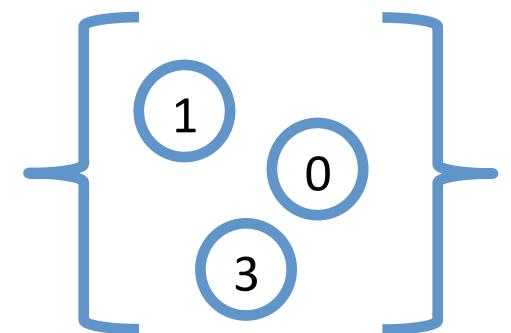
- Homework 3: Extension (due to Codio instabilities)
  - now due *Wednesday* at 11:59:59pm
- Homework 4
  - available soon
  - due on October 10<sup>th</sup>
- Midterm 1
  - *October 13<sup>th</sup> in Class*
  - Where? Last Names:
    - A – M Leidy Labs 10 (Here)
    - N – Z Meyerson Hall B1
  - Covers lecture material through Chapter 13
  - Review materials (old exams) on course website
  - Review session: TBA

# Review: Abstract types (e.g. set)

- An abstract type is defined by its *interface* and its *properties*, not its representation.
- **Interface:** defines operations on the type
  - There is an empty set
  - There is a way to add elements to a set to make a bigger set
  - There is a way to list all elements in a set
  - There is a way to test membership
- **Properties:** define how the operations interact with each other
  - Elements that were added can be found in the set
  - Adding an element a second time doesn't change the elements of a set
  - Adding in a different order doesn't change the elements of a set
- Any type (possibly with invariants) that satisfies the interface and properties can be a set.
- *Clients of an implementation can only access what is explicitly in the abstract type's interface*



concrete representation  
-----  
abstract view



# Finite Maps

*Another example of **abstract datatype interfaces**  
& **concrete implementations***

# Motivating Scenario

- Suppose you were writing some course-management software and needed to look up the lab section for a student given the student's PennKey?
  - Students might add/drop the course
  - Students might switch lab sections
  - Students should be in only *one* lab section
- How would you do it? What data structure would you use?

# Example

| Key        | Value |
|------------|-------|
| “mitch”    | 15    |
| “benjamin” | 05    |
| “ezaan”    | 10    |
| “likat”    | 15    |
| ...        | ...   |

- Each key is associated with a value.
  - No two keys are identical
  - Values can be repeated
- Given the key "mitch" we want to find lookup the value 15

# Finite Maps

- A *finite map* (a.k.a. *dictionary*), is a collection of *bindings* from distinct *keys* to *values*.
  - Operations to *add* & *remove* bindings, *test* for key membership, *look up* a value by its key
- Example: a `(string, int)` map might map a PennKey to the lab section.
  - The map type is generic in two arguments
- Like sets, finite maps appear in many settings to map:
  - domain names to IP addresses
  - words to their definitions (a dictionary)
  - user names to passwords
  - game character unique identifiers to dialog trees
  - ...

# Tests for Finite Map abstract type

```
;; open Assoc

(* Specifying the properties of the MAP abstract type via test cases. *)

(* A simple map with one element. *)
let m1 : (int, string) map =
  add 1 "uno" empty

(* list entries for this simple map *)
;; run_test "entries m1" (fun () -> entries m1 = [(1,"uno")])

(* access value for key in the map *)
;; run_test "find 1 m1" (fun () -> (get 1 m1) = "uno")

(* find for value that does not exist in the map? *)
;; run_failing_test "find 2 m1" (fun () -> (get 2 m1) = "dos" )

let m2 : (int, string) map = add 1 "un" m1

(* find after redefining value, should be new value *)
;; run_test "find 1 m2" (fun () -> (get 1 m2) = "un")

(* entries after redefining value, should only show new value *)
;; run_test "entries m2" (fun () -> entries m2 = [(1, "un")])

(* test membership *)
;; run_test "mem test" (fun () ->
  mem "b" (add "a" 3 (add "b" 4 empty)))
```

# Signature: Finite Map

```
module type MAP = sig
  type ('k, 'v) map
  val empty    : ('k, 'v) map
  val add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
  val remove   : 'k          -> ('k, 'v) map -> ('k, 'v) map
  val mem      : 'k -> ('k, 'v) map -> bool
  val get      : 'k -> ('k, 'v) map -> 'v
  val entries : ('k, 'v) map -> ('k * 'v) list
  val equals   : ('k, 'v) map -> ('k, 'v) map -> bool
end
```

# Implementation: Ordered Lists

```
module Assoc : MAP = struct

  (* Represent a finite map as a list of pairs. *)
  (* Representation invariant:
  (*   - no duplicate keys (helps get, remove)      *)
  (*   - keys are sorted (helps equals, helps get)  *)

  type ('k, 'v) map = ('k * 'v) list

  let empty : ('k, 'v) map = []

  let rec mem (key:'k) (m : ('k, 'v) map) : bool =
    begin match m with
    | [] -> false
    | (k,v)::rest ->
      (key >= k) &&
      (key = k) || (mem key rest)
    end

  ;;; run_test "mem test" (fun () ->
  mem "b" [("a",3); ("b",4)])

```

# Implementation: Ordered Lists

```
let rec get (key:'k) (m : ('k,'v) map) : 'v =
begin match m with
| [] -> failwith "key not found"
| (k,v)::rest ->
  if key < k then failwith "key not found"
  else if key = k then v
  else get key rest
end

let rec remove (key:'k) (m : ('k,'v) map) : ('k,'v) map =
begin match m with
| [] -> []
| (k,v)::rest ->
  if key < k then m
  else if key = k then rest
  else (k,v)::remove key rest
end
```

# Completing module implementation

finiteMap.ml

# Summary: Abstract Types

- Different programming languages have different ways of letting you define abstract types
- At a minimum, this means providing:
  - A way to specify (write down) an interface
  - A means of hiding implementation details (*encapsulation*)
- In OCaml:
  - Interfaces are specified using a *signature* or *interface*
  - Encapsulation is achieved because the interface can *omit* information
    - type definitions
    - names and types of auxiliary functions
  - Clients *cannot* mention values or types not named in the interface

# .ml and .mli files

- You've already been using signatures and modules in OCaml.
- A series of type and val declarations stored in a file foo.mli is considered as defining a signature FOO
- A series of top-level definitions stored in a file foo.ml is considered as defining a module Foo

foo.mli

```
type t
val z : t
val f : t -> int
```

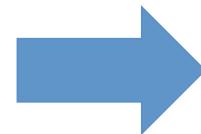
foo.ml

```
type t = int
let z : t = 0
let f (x:t) : int =
  x + 1
```

test.ml

```
;; open Foo
;; print_int
  (Foo.f Foo.z)
```

Files



```
module type FOO = sig
  type t
  val z : t
  val f : t -> int
end
```

```
module Foo : FOO = struct
  type t = int
  let z : t = 0
  let f (x:t) : int =
    x + 1
end
```

```
module Test = struct
  ;; open Foo
  ;; print_int
    (Foo.f Foo.z)
end
```

# Dealing with Partiality\*

\*A function is said to be *partial* if it is not defined for all inputs.

Which of these is a function that calculates the maximum value in a (generic) list:

1.

```
let rec list_max (l:'a list) : 'a =
begin match l with
| [] -> []
| h :: t -> max h (list_max t)
end
```

2.

```
let rec list_max (l:'a list) : 'a =
fold max 0 l
```

3.

```
let rec list_max (l:'a list) : 'a =
begin match l with
| h :: t -> max h (list_max t)
end
```

4. None of the above

Answer: 4

# Quiz answer

- `list_max` isn't defined for the empty list!

```
let rec list_max (l:'a list) : 'a =
  begin match l with
    | [] -> failwith "empty list"
    | [h] -> h
    | h::t -> max h (list_max t)
  end
```

# Client of list\_max

```
(* string_of_max calls list_max *)
let string_of_max (x:int list) : string =
  string_of_int (list_max x)
```

- Oops! `string_of_max` will fail if given `[]`
- Not so easy to debug if `string_of_max` is written by one person and `list_max` is written by another.
- Interface of `list_max` is not very informative

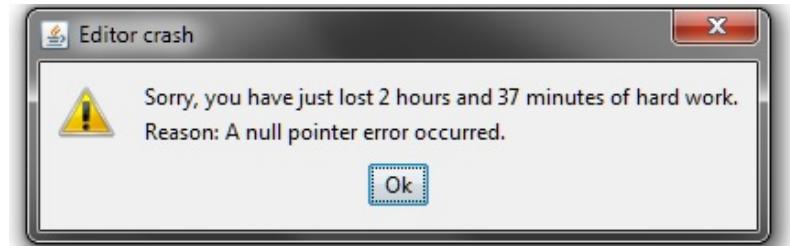
`val list_max : int list -> int`

# Solutions to Partiality: Option 1

- Abort the program:  
`failwith "an error message"`
  - Whenever it is called, `failwith` halts the program and reports the error message it is given.
- This solution is appropriate whenever you *know* that a certain case is impossible
  - The compiler isn't smart enough to figure out that the case is impossible...
  - Often happens when there is an invariant on a data structure
  - `failwith` is also useful to “stub out” unimplemented parts of your program.
- Languages (e.g. OCaml, Java) support *exception handling facilities* to let programs recover from such failures.
  - We'll talk about these when we get to Java

# Solutions to Partiality: Option 2

- Return a *default or error value*
  - e.g. define `list_max []` to be `-1`
  - Error codes used often in C programs
  - `null` used often in Java
- But...
  - What if `-1` (or whatever default you choose) really *is* the maximum value?
  - Can lead to many bugs if the default isn't handled properly by the callers.
  - *IMPOSSIBLE* to implement generically!
    - No way to generically create a sensible default value for every possible type
  - Sir Tony Hoare, Turing Award winner and inventor of `null` calls it his “*billion dollar mistake*”!
- Defaults should be avoided if possible



# Optional values

Solutions to Partiality: Option 3

# Option Types

- Define a generic datatype of *optional values*:

```
type 'a option =
| None
| Some of 'a
```

- A “partial” function returns an option

```
let list_max (l:list) : int option = ...
```

- Contrast this with “null”, a “legal” return value of any type
  - caller can accidentally forget to check whether null was used; results in NullPointerExceptions or crashes
- Modern language designs (e.g. Apple's Swift, Mozilla's Rust) distinguish between the type String (definitely not null) and String? (optional string)

## Example: list\_max

- A function that returns the maximum value of a list as an option (None if the list is empty)

```
let list_max (l:'a list) : 'a option =
begin match l with
| [] -> None
| x::tl -> Some (fold max x tl)
end
```

# Revised client of list\_max

```
(* string_of_max calls list_max *)
let string_of_max (l:int list) : string =
  begin match (list_max l) with
    | None -> "no maximum"
    | Some m -> string_of_int m
  end
```

- `string_of_max` will never fail
- The type of `list_max` makes it explicit that a *client* must check for partiality.

**val list\_max : int list -> int option**

What is the type of this function?

```
let head (x: _____) : _____ =  
begin match x with  
| [] -> None  
| h :: t -> Some h  
end
```

1. 'a list -> 'a
2. 'a list -> 'a list
3. 'a list -> 'b option
4. 'a list -> 'a option
5. None of the above

Answer: 4

What is the value of this expression?

```
let head (x: 'a list) : 'a option =  
begin match x with  
| [] -> None  
| h :: t -> Some h  
end in  
  
[ head [1]; head [] ]
```

1. [ 1 ; 0 ]
2. 1
3. [Some 1; None]
4. [None; None]
5. None of the above

Answer: 3

# Revising the MAP interface

```
module type MAP = sig  
  
  type ('k, 'v) map  
  
  val empty    : ('k, 'v) map  
  val add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map  
  val remove   : 'k           -> ('k, 'v) map -> ('k, 'v) map  
  val mem      : 'k -> ('k, 'v) map -> bool  
  val get      : 'k -> ('k, 'v) map -> 'v option  
  val entries  : ('k, 'v) map -> ('k * 'v) list  
  val equals   : ('k, 'v) map -> ('k, 'v) map -> bool  
  
end
```

get returns an optional 'v.  
Now its type isn't a lie!

# Programming Languages and Techniques (CIS120)

## Lecture 12

September 27, 2017

Partiality, Sequencing, Records  
Chapters 12, 13

# Announcements

- Homework 3: Extension (due to Codio instabilities)
  - due *TONIGHT* at 11:59:59pm
- Homework 4
  - available now due on October 10<sup>th</sup>
- Midterm 1
  - *October 13<sup>th</sup> in Class*
  - Where? Last Names:
    - A – M Leidy Labs 10 (Here)
    - N – Z Meyerson Hall B1
  - Covers lecture material through Chapter 13
  - Review materials (old exams) on course website
  - Review session: TBA
- Dr. Sheth will be traveling 9/29 and 10/4  
(Dr. Zdancewic will cover)

# Dealing with Partiality\*

\*A function is said to be *partial* if it is not defined for all inputs.

Which of these is a function that calculates the maximum value in a (generic) list:

1.

```
let rec list_max (l:'a list) : 'a =
begin match l with
| [] -> []
| h :: t -> max h (list_max t)
end
```

2.

```
let rec list_max (l:'a list) : 'a =
fold max 0 l
```

3.

```
let rec list_max (l:'a list) : 'a =
begin match l with
| h :: t -> max h (list_max t)
end
```

4. None of the above

Answer: 4

# Quiz answer

- `list_max` isn't defined for the empty list!

```
let rec list_max (l:'a list) : 'a =
  begin match l with
    | [] -> failwith "empty list"
    | [h] -> h
    | h::t -> max h (list_max t)
  end
```

# Client of list\_max

```
(* string_of_max calls list_max *)
let string_of_max (x:int list) : string =
  string_of_int (list_max x)
```

- Oops! `string_of_max` will fail if given `[]`
- Not so easy to debug if `string_of_max` is written by one person and `list_max` is written by another.
- Interface of `list_max` is not very informative

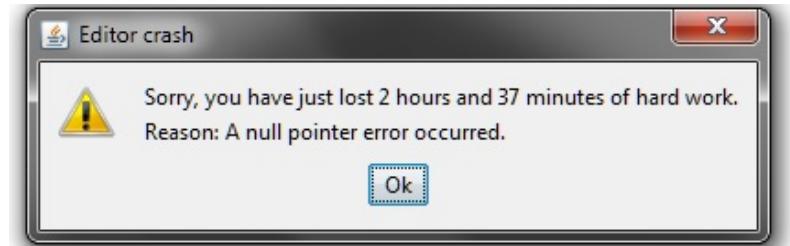
`val list_max : int list -> int`

# Solutions to Partiality: Option 1

- Abort the program:  
`failwith "an error message"`
  - Whenever it is called, `failwith` halts the program and reports the error message it is given.
- This solution is appropriate whenever you *know* that a certain case is impossible
  - The compiler isn't smart enough to figure out that the case is impossible...
  - Often happens when there is an invariant on a data structure
  - `failwith` is also useful to “stub out” unimplemented parts of your program.
- Languages (e.g. OCaml, Java) support *exception handling facilities* to let programs recover from such failures.
  - We'll talk about these when we get to Java

# Solutions to Partiality: Option 2

- Return a *default or error value*
  - e.g. define `list_max []` to be `-1`
  - Error codes used often in C programs
  - `null` used often in Java
- But...
  - What if `-1` (or whatever default you choose) really *is* the maximum value?
  - Can lead to many bugs if the default isn't handled properly by the callers.
  - *IMPOSSIBLE* to implement generically!
    - No way to generically create a sensible default value for every possible type
  - Sir Tony Hoare, Turing Award winner and inventor of `null` calls it his "*billion dollar mistake*"!
- Defaults should be avoided if possible



# Optional values

Solutions to Partiality: Option 3

# Option Types

- Define a generic datatype of *optional values*:

```
type 'a option =
| None
| Some of 'a
```

- A “partial” function returns an option

```
let list_max (l:list) : int option = ...
```

- Contrast this with “null”, a “legal” return value of any type
  - caller can accidentally forget to check whether null was used; results in NullPointerExceptions or crashes
- Modern language designs (e.g. Apple's Swift, Mozilla's Rust) distinguish between the type String (definitely not null) and String? (optional string)

## Example: list\_max

- A function that returns the maximum value of a list as an option (None if the list is empty)

```
let list_max (l:'a list) : 'a option =
begin match l with
| [] -> None
| x::tl -> Some (fold max x tl)
end
```

# Revised client of list\_max

```
(* string_of_max calls list_max *)
let string_of_max (l:int list) : string =
  begin match (list_max l) with
    | None -> "no maximum"
    | Some m -> string_of_int m
  end
```

- `string_of_max` will never fail
- The type of `list_max` makes it explicit that a *client* must check for partiality.

**val list\_max : int list -> int option**

## What is the type of this function?

```
let head (x: _____) : _____ =  
begin match x with  
| [] -> None  
| h :: t -> Some h  
end
```

1. 'a list -> 'a
2. 'a list -> 'a list
3. 'a list -> 'b option
4. 'a list -> 'a option
4. None of the above

Answer: 4

What is the value of this expression?

```
let head (x: 'a list) : 'a option =  
begin match x with  
| [] -> None  
| h :: t -> Some h  
end in  
  
[ head [1]; head [] ]
```

1. [ 1 ; 0 ]
2. 1
3. [Some 1; None]
4. [None; None]
5. None of the above

Answer: 3

# Revising the MAP interface

```
module type MAP = sig
  type ('k, 'v) map
  val empty    : ('k, 'v) map
  val add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
  val remove   : 'k           -> ('k, 'v) map -> ('k, 'v) map
  val mem      : 'k -> ('k, 'v) map -> bool
  val get      : 'k -> ('k, 'v) map -> 'v option
  val entries  : ('k, 'v) map -> ('k * 'v) list
  val equals   : ('k, 'v) map -> ('k, 'v) map -> bool
end
```

get returns an optional 'v.  
Now its type isn't a lie!

# Commands, Sequencing and Unit

What is the type of print\_string?



# Sequencing Commands and Expressions

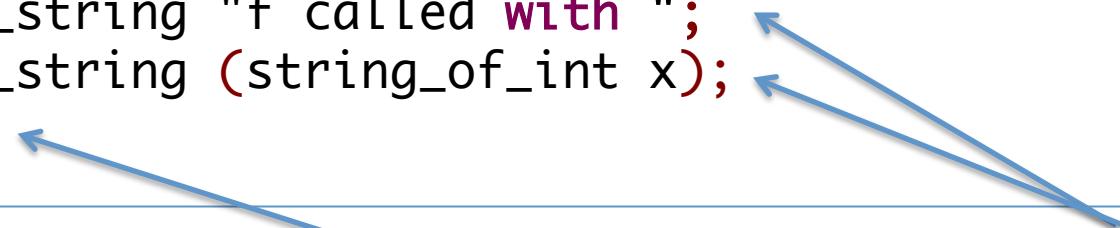
We can *sequence* commands inside expressions using ‘;’

- unlike in C, Java, etc., ‘;’ doesn’t terminate a statement it *separates* a command from an expression

```
let f (x:int) : int =
    print_string "f called with ";
    print_string (string_of_int x);
    x + x
```

do *not* use ‘;’ here!

note the use of ‘;’ here



The distinction between commands & expressions is artificial.

- `print_string` is a function of type: `string -> unit`
- Commands are actually just expressions of type: `unit`

# unit: the trivial type

- Similar to "void" in Java or C
- For functions that don't take any arguments

```
let f () : int = 3  
let y : int = f ()
```

```
val f : unit -> int  
val y : int
```

- Also for functions that don't return anything, such as testing and printing functions a.k.a *commands*:

```
(* run_test : string -> (unit -> bool) -> unit *)  
;; run_test “TestName” test  
  
(* print_string : string -> unit *)  
;; print_string “Hello, world!”
```

# unit: the boring type

- Actually, `()` is a value just like any other value (a 0-ary tuple)
- For functions that don't take any interesting arguments

```
let f () : int = 3  
let y : int = f ()
```

```
val f : unit -> int  
val y : int
```

- Also for functions that don't return anything interesting, such as testing and printing functions a.k.a *commands*:

```
(* run_test : string -> (unit -> bool) -> unit *)  
;; run_test “TestName” test  
  
(* print_string : string -> unit *)  
;; print_string “Hello, world!”
```

# unit: the first-class type

- Can define values of type unit

```
let x : unit = ()
```

```
val x : unit
```

- Can pattern match unit (even in function definitions)

```
let z = begin match x with  
| () -> 4  
end
```

```
fun () -> 3
```

- Is the result of an implicit else branch:

```
;; if z <> 4 then  
failwith "oops"
```



```
;; if z <> 4 then  
failwith "oops"  
else ()
```

# Sequencing Commands and Expressions

- Expressions of type unit are useful because of their *side effects* – they "do" stuff
  - e.g. printing, changing the value of mutable state

```
let f (x:int) : int =
  print_string "f called with ";
  print_string (string_of_int x);
  x + x
```

do *not* use ';' here!

note the use of ';' here

- We can think of ';' as an infix function of type:  
 $\text{unit} \rightarrow 'a \rightarrow 'a$

What is the type of `f` in the following program:

```
let f (x:int) =  
    print_int (x + x)
```

1. `unit -> int`
2. `unit -> unit`
3. `int -> unit`
4. `int -> int`
5. `f` is ill typed

Answer: 3

What is the type of `f` in the following program:

```
let f (x:int) =  
  (print_int x);  
  (x + x)
```

1. `unit -> int`
2. `unit -> unit`
3. `int -> unit`
4. `int -> int`
5. `f` is ill typed

Answer: 4

# Records

# Immutable Records

- Records are like tuples with named fields:

```
(* a type for representing colors *)
type rgb = {r:int; g:int; b:int;}

(* some example rgb values *)
let red    : rgb = {r=255; g=0;   b=0;}
let blue   : rgb = {r=0;   g=0;   b=255;}
let green  : rgb = {r=0;   g=255; b=0;}
let black  : rgb = {r=0;   g=0;   b=0;}
let white  : rgb = {r=255; g=255; b=255;}
```

Curly braces around record.  
Semicolons after record components.

- The type `rgb` is a record with three fields: `r`, `g`, and `b`
  - fields can have any types; they don't all have to be the same
- Record values are created using this notation:  
 $\{field1=val1; field2=val2;...\}$

# Field Projection

- The value in a record field can be obtained by using “dot” notation: `record.field`

```
(* a type for representing colors *)
type rgb = {r:int; g:int; b:int;}

(* using 'dot' notation to project out components *)
(* calculate the average of two colors *)
let average_rgb (c1:rgb) (c2:rgb) : rgb =
  {r = (c1.r + c2.r) / 2;
   g = (c1.g + c2.g) / 2;
   b = (c1.b + c2.b) / 2;}
```

# OCaml Record Syntax

OCaml provides convenient syntax for working with records:

```
let f {r ; g} = r + g
```

```
let mk_rgb (r:int) (g:int) (b:int) =
  {r; g; b}
```

# Programming Languages and Techniques (CIS120)

Lecture 13

September 29, 2017

Mutable State & Abstract Stack Machine  
Chapters 14 & 15

# Announcements

- Homework 4
  - due on October 10<sup>th</sup>
- Midterm 1
  - *October 13<sup>th</sup> in Class*
  - Where? Last Names:
    - A – M Leidy Labs 10 (Here)
    - N – Z Meyerson Hall B1
  - Covers lecture material through Chapter 13
  - Review materials (old exams) on course website
  - Review session: TBA

# Records

# Immutable Records

- Records are like tuples with named fields:

```
(* a type for representing colors *)
type rgb = {r:int; g:int; b:int;}

(* some example rgb values *)
let red    : rgb = {r=255; g=0;   b=0;}
let blue   : rgb = {r=0;   g=0;   b=255;}
let green  : rgb = {r=0;   g=255; b=0;}
let black  : rgb = {r=0;   g=0;   b=0;}
let white  : rgb = {r=255; g=255; b=255;}
```

Curly braces around record.  
Semicolons after record components.

- The type `rgb` is a record with three fields: `r`, `g`, and `b`
  - fields can have any types; they don't all have to be the same
- Record values are created using this notation:

`{field1=val1; field2=val2;...}`

# Field Projection

- The value in a record field can be obtained by using “dot” notation: `record.field`

```
(* a type for representing colors *)
type rgb = {r:int; g:int; b:int;}

(* using 'dot' notation to project out components *)
(* calculate the average of two colors *)
let average_rgb (c1:rgb) (c2:rgb) : rgb =
  {r = (c1.r + c2.r) / 2;
   g = (c1.g + c2.g) / 2;
   b = (c1.b + c2.b) / 2;}
```

# Recap

# Why Pure Functional Programming?

- Simplicity
  - small language: arithmetic, local variables, recursive functions, datatypes, pattern matching, generic types/functions and modules
  - simple *substitution* model of computation
- Persistent data structures
  - Nothing changes; retains all intermediate results
  - Good for version control, fault tolerance, etc.
- Typecheckers give more helpful errors
  - Once your program compiles, it needs less testing
  - Options vs. NullPointerException
- Easier to parallelize and distribute
  - No implicit interactions between parts of the program.
  - All of the behavior of a function is specified by its arguments



*Being vs Doing*



# Mutable State

# *Mutable Record Fields*

- By default, all record fields are *immutable*—once initialized, they can never be modified.
- OCaml supports *mutable* fields that can be imperatively updated by the “set” command: record.field <- val

note the ‘mutable’ keyword

```
type point = {mutable x:int; mutable y:int}

let p0 = {x=0; y=0}
(* set the x coord of p0 to 17 *)
;; p0.x <- 17
;; print_endline ("p0.x = " ^ (string_of_int p0.x))
```

in-place update of p0.x

# Defining new Commands

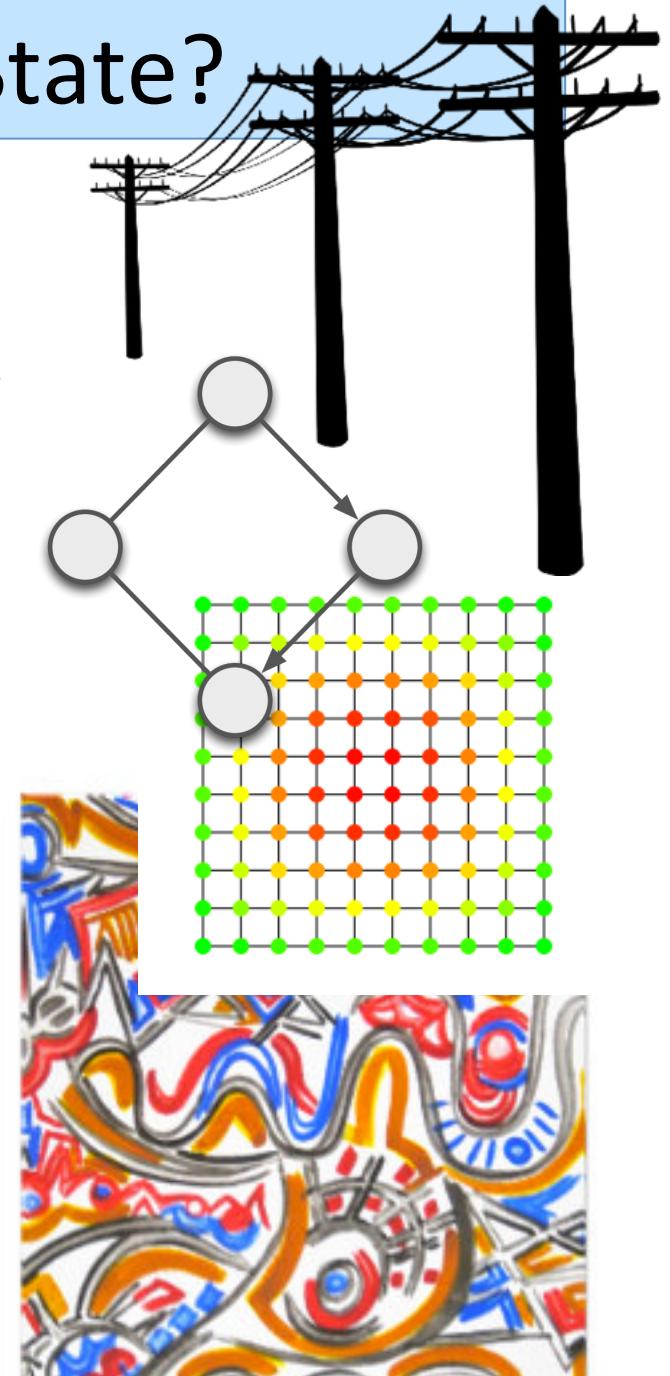
- Functions can assign to mutable record fields
- Note that the return type of ‘`<-`’ is unit

```
type point = {mutable x:int; mutable y:int}

(* a command to shift a point by dx,dy *)
let shift (p:point) (dx:int) (dy:int) : unit =
    p.x <- p.x + dx;
    p.y <- p.y + dy
```

# Why Use Mutable State?

- Action at a distance
  - allow remote parts of a program to communicate / share information without threading the information through all the points in between
- Data structures with explicit sharing
  - e.g. graphs
  - without mutation, it is only possible to build trees – no cycles
- Efficiency/Performance
  - some data structures have imperative versions with better asymptotic efficiency than the best declarative version
- Re-using space (in-place update)
- Random-access data (arrays)
- Direct manipulation of hardware
  - device drivers, displays, etc.



# Different views of imperative programming

## Java (and C, C++, C#)

- Null is contained in (almost) every type. Partial functions can return **null**.
- Code is a sequence of **statements** that do something, sometimes using expressions to compute values.
- References are **mutable** by default, must be explicitly declared to be constant

## OCaml (and Haskell, etc.)

- No null. Partiality must be made explicit with **options**.
- Code is an **expression** that has a value. Sometimes computing that value has other effects.
- References are **immutable** by default, must be explicitly declared to be mutable

```
type point = {mutable x:int; mutable y:int}

(* a command to shift a point by dx,dy *)
let shift (p:point) (dx:int) (dy:int) : unit =
    p.x <- p.x + dx;
    p.y <- p.y + dy
```

What answer does the following function produce when called?

```
let f (p1:point) : int =
    p1.x <- 17;
    p1.x
```

1. 17
2. something else
3. sometimes 17 and sometimes something else
4. f is ill typed

ANSWER: 1

```
type point = {mutable x:int; mutable y:int}

(* a command to shift a point by dx,dy *)
let shift (p:point) (dx:int) (dy:int) : unit =
    p.x <- p.x + dx;
    p.y <- p.y + dy
```

What answer does the following function produce when called?

```
let f (p1:point) (p2:point) : int =
    p1.x <- 17;
    p2.x <- 34;
    p1.x
```

1. 17
2. something else
3. sometimes 17 and sometimes something else
4. f is ill typed

ANSWER: 3

# The Challenge of Mutable State: Aliasing

- What does this function return?

```
let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  p2.x <- 42;
  p1.x
```

```
(* Consider this call to f: *)
let p0 = {x=0; y=0} in
  f p0 p0
```

Two identifiers are said to be *aliases* if they both name the *same* mutable record. Inside `f`, the identifiers `p1` and `p2` might or might not be aliased, depending on which arguments are passed in.

# Opening a Whole New Can of Worms\*



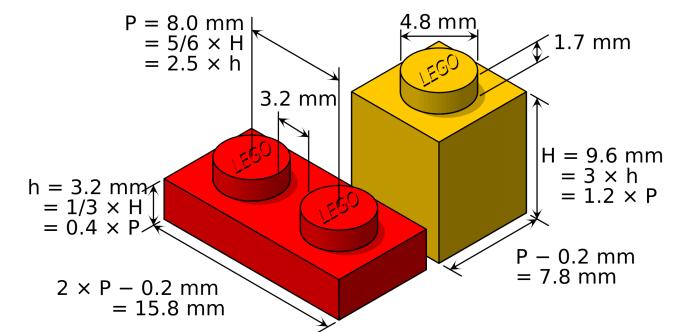
\*t-shirt courtesy of  
[ahrefs.com](http://ahrefs.com)

# Modeling State

Location, Location, Location!

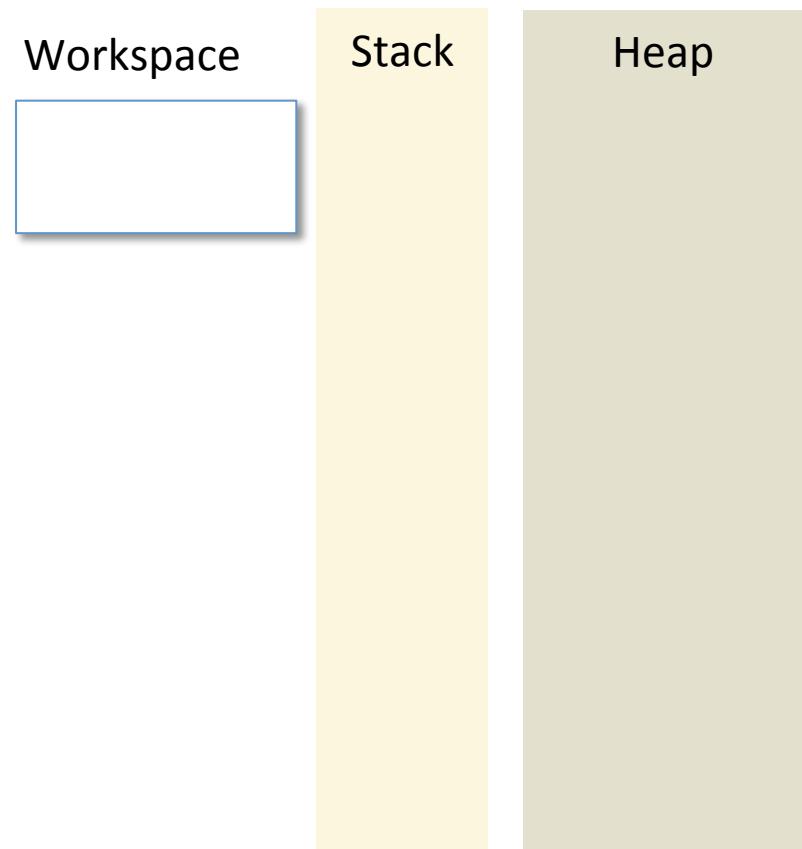
# Need for a New Computation Model

- Substitution works well for value oriented programming.
  - "Observable" behavior of a value is *completely* determined by its structure.
  - Pure functions are *referentially transparent*: two different calls to the same function with the same argument always yield the same results.
  - These properties justify the the "replace equals by equals" model.
- But... mutable state makes the *location* (or *identity*) of values matter.
  - Observable behavior of the program can depend on *where* the value is in memory.
  - Two calls to the same function don't necessarily do the same thing, even with identical inputs.



# Abstract Stack Machine

- Three “spaces”
  - workspace
    - the expression the computer is currently working on simplifying
  - stack
    - temporary storage for `let` bindings and partially simplified expressions
  - heap
    - storage area for large data structures
- Initial state:
  - workspace contains whole program
  - stack and heap are empty
- Machine operation:
  - In each step, choose next part of the workspace expression and simplify it
  - (Sometimes this will also involve changes to the stack and/or heap)
  - Stop when there are no more simplifications to be done



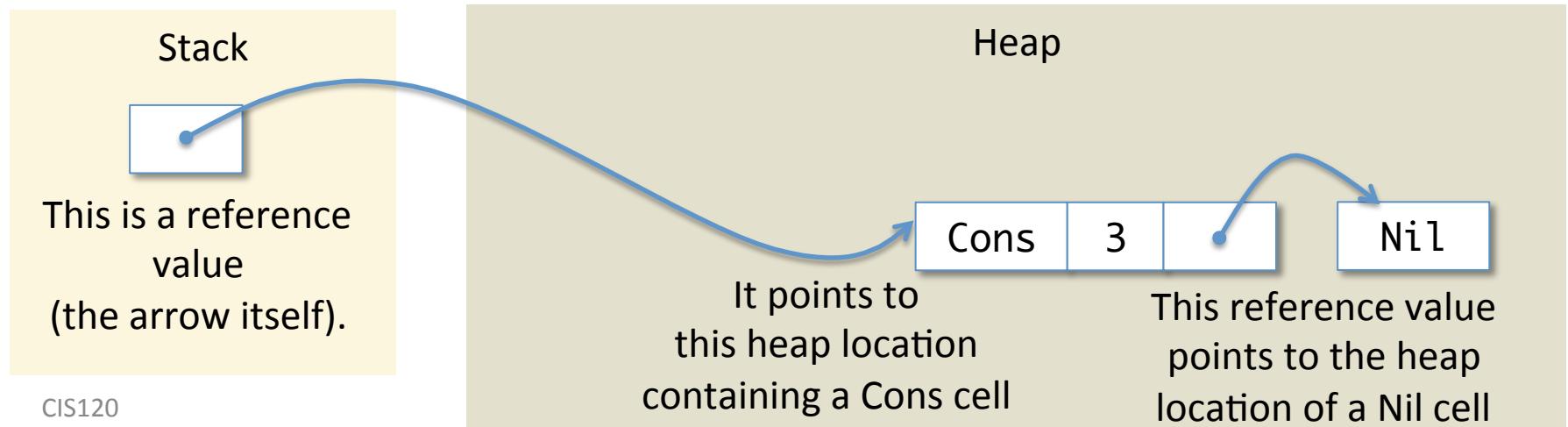
# Values and References

A *value* is either:

- a *primitive value* like an integer, or,
- a *reference* to a location in the heap

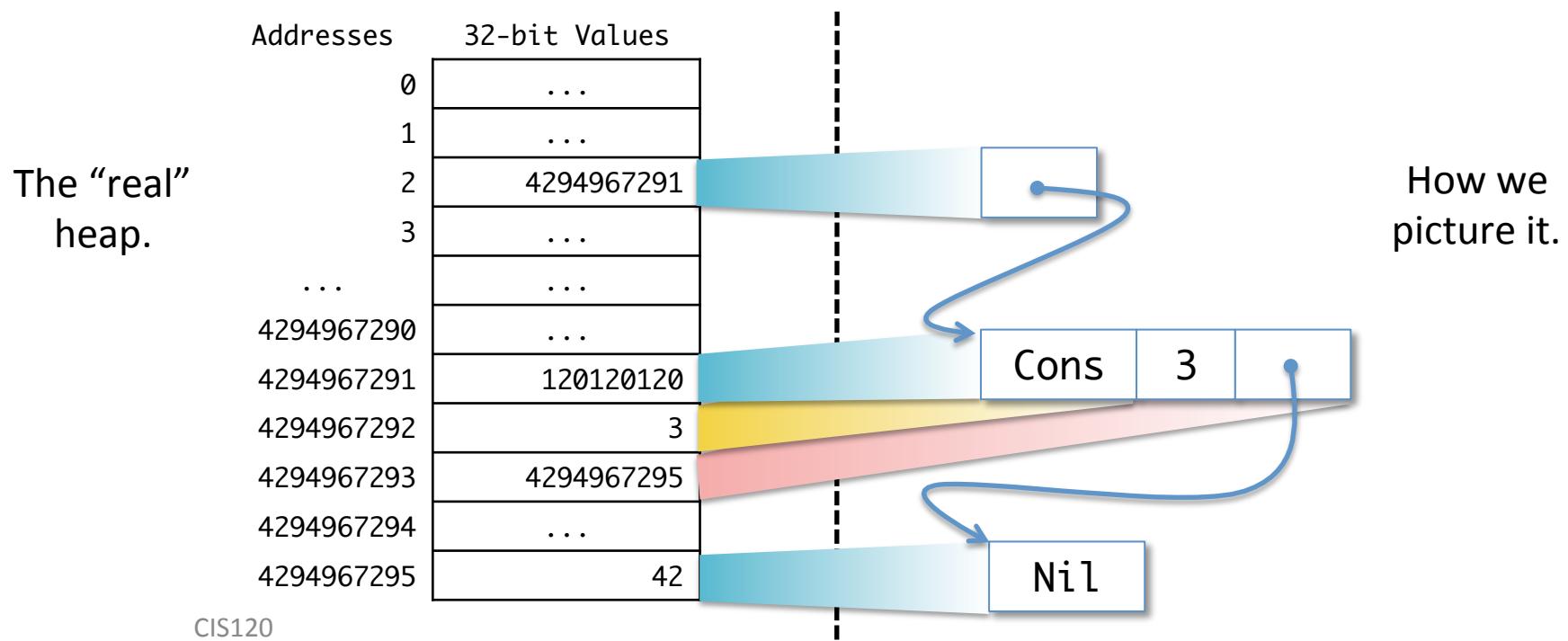
A reference is the *address* of a piece of data in the heap. We draw a reference value as an “arrow”:

- The start of the arrow is the reference itself (i.e. the address).
- The arrow “points” to the value located at the reference’s address.



# References as an Abstraction

- In a real computer, the memory consists of an array of 32-bit words, numbered  $0 \dots 2^{32}-1$  (for a 32-bit machine)
  - A reference is just an address that tells you where to look up a value
  - Data structures are usually laid out in contiguous blocks of memory
  - Constructor tags are just numbers chosen by the compiler  
e.g. Nil = 42 and Cons = 120120120



# The ASM: let, variables, operators, and if expressions

# Simplification

Workspace

```
let x = 10 + 12 in  
let y = 2 + x in  
  if x > 23 then 3 else 4
```

Stack

Heap

# What to simplify next?

- At each step, the ASM finds the left-most *ready* subexpression in the workspace
- An expression involving a primitive operator (eg “**+**”) is *ready* if all its arguments are values.
  - Expression is replaced with its result
- A let expression **let**  $x$  :  $t = e$  **in**  $body$  is *ready* if  $e$  is a value.
  - A new binding for  $x$  to  $e$  is added at the end of the stack
  - $body$  is left in the workspace
- A variable is always *ready*.
  - The variable is replaced by its binding in the stack, searching from the most recent bindings.
- A conditional expression **if**  $e$  **then**  $e_1$  **else**  $e_2$  is *ready* if  $e$  is either **true** or **false**
  - The workspace is replaced with either  $e_1$  (if  $e$  is **True**) or  $e_2$  (if  $e$  is **False**)

# Simplification

Workspace

```
let x = 10 + 12 in  
let y = 2 + x in  
  if x > 23 then 3 else 4
```

Stack

Heap

# Simplification

Workspace

```
let x = 22 in  
let y = 2 + x in  
  if x > 23 then 3 else 4
```

Stack

Heap

# Simplification

Workspace

```
let x = 22 in  
let y = 2 + x in  
  if x > 23 then 3 else 4
```

Stack

Heap

# Simplification

Workspace

```
let y = 2 + x in  
  if x > 23 then 3 else 4
```

Stack

|   |    |
|---|----|
| x | 22 |
|---|----|

Heap

# Simplification

Workspace

```
let y = 2 + x in  
  if x > 23 then 3 else 4
```

Stack

|   |    |
|---|----|
| x | 22 |
|---|----|

Heap

x is not a value: so look it up in the stack

# Simplification

Workspace

```
let y = 2 + 22 in  
  if x > 23 then 3 else 4
```

Stack

|   |    |
|---|----|
| x | 22 |
|---|----|

Heap

# Simplification

Workspace

```
let y = 2 + 22 in  
  if x > 23 then 3 else 4
```

Stack

|   |    |
|---|----|
| x | 22 |
|---|----|

Heap

# Simplification

Workspace

```
let y = 24 in  
  if x > 23 then 3 else 4
```

Stack

|   |    |
|---|----|
| x | 22 |
|---|----|

Heap

# Simplification

Workspace

```
let y = 24 in  
  if x > 23 then 3 else 4
```

Stack

|   |    |
|---|----|
| x | 22 |
|---|----|

Heap

# Simplification

Workspace

```
if x > 23 then 3 else 4
```

Stack

|   |    |
|---|----|
| x | 22 |
| y | 24 |

Heap

# Simplification

Workspace

```
if x > 23 then 3 else 4
```

Stack

|   |    |
|---|----|
| x | 22 |
| y | 24 |

Heap



Looking up `x` in the stack proceeds from most recent entries to the least recent entries – the “top” (most recent part) of the stack is toward the bottom of the diagram.

# Simplification

Workspace

```
if 22 > 23 then 3 else 4
```

Stack

|   |    |
|---|----|
| x | 22 |
| y | 24 |

Heap

# Simplification

Workspace

```
if 22 > 23 then 3 else 4
```

Stack

|   |    |
|---|----|
| x | 22 |
| y | 24 |

Heap

# Simplification

Workspace

```
if false then 3 else 4
```

Stack

|   |    |
|---|----|
| x | 22 |
| y | 24 |

Heap

# Simplification

Workspace

```
if false then 3 else 4
```

Stack

|   |    |
|---|----|
| x | 22 |
| y | 24 |

Heap

# Simplification

Workspace

4

Stack

|   |    |
|---|----|
| x | 22 |
| y | 24 |

Heap



What does the Stack look like after simplifying the following code on the workspace?

```
let z = 20 in  
let w = 2 + z in  
    w
```

Stack

|   |    |
|---|----|
| z | 22 |
|---|----|

|   |       |
|---|-------|
| w | 2 + z |
|---|-------|

1.

Stack

|   |    |
|---|----|
| z | 20 |
|---|----|

|   |    |
|---|----|
| w | 22 |
|---|----|

2.

Stack

|   |    |
|---|----|
| w | 22 |
|---|----|

|   |    |
|---|----|
| w | 22 |
|---|----|

|   |    |
|---|----|
| z | 20 |
|---|----|

3.

Stack

ANSWER: 2

What does the Stack look like after simplifying the following code on the workspace?

```
let z = 20 in  
let z = 2 + z in  
    z
```

Stack

|   |    |
|---|----|
| z | 22 |
|---|----|

|   |    |
|---|----|
| z | 20 |
|---|----|

1.

Stack

|   |    |
|---|----|
| z | 20 |
|---|----|

|   |    |
|---|----|
| z | 22 |
|---|----|

2.

Stack

|   |    |
|---|----|
| z | 22 |
|---|----|

|   |    |
|---|----|
| z | 22 |
|---|----|

3.

Stack

|   |    |
|---|----|
| z | 22 |
|---|----|

|   |    |
|---|----|
| z | 22 |
|---|----|

4.

ANSWER: 2

# Mutable Records

- The reason for introducing the ASM model is to make heap locations and sharing *explicit*.
  - Now we can say what it means to mutate a heap value *in place*.

```
type point = {mutable x:int; mutable y:int}

let p1 : point = {x=1; y=1;}
let p2 : point = p1
let ans : int = (p2.x <- 17; p1.x)
```

- We draw a record in the heap like this:
  - The doubled outlines indicate that those cells are mutable
  - Everything else is immutable
  - (field names don't actually take up space)

|   |   |
|---|---|
| x | 1 |
| y | 1 |

A point record  
in the heap.

# Allocate a Record

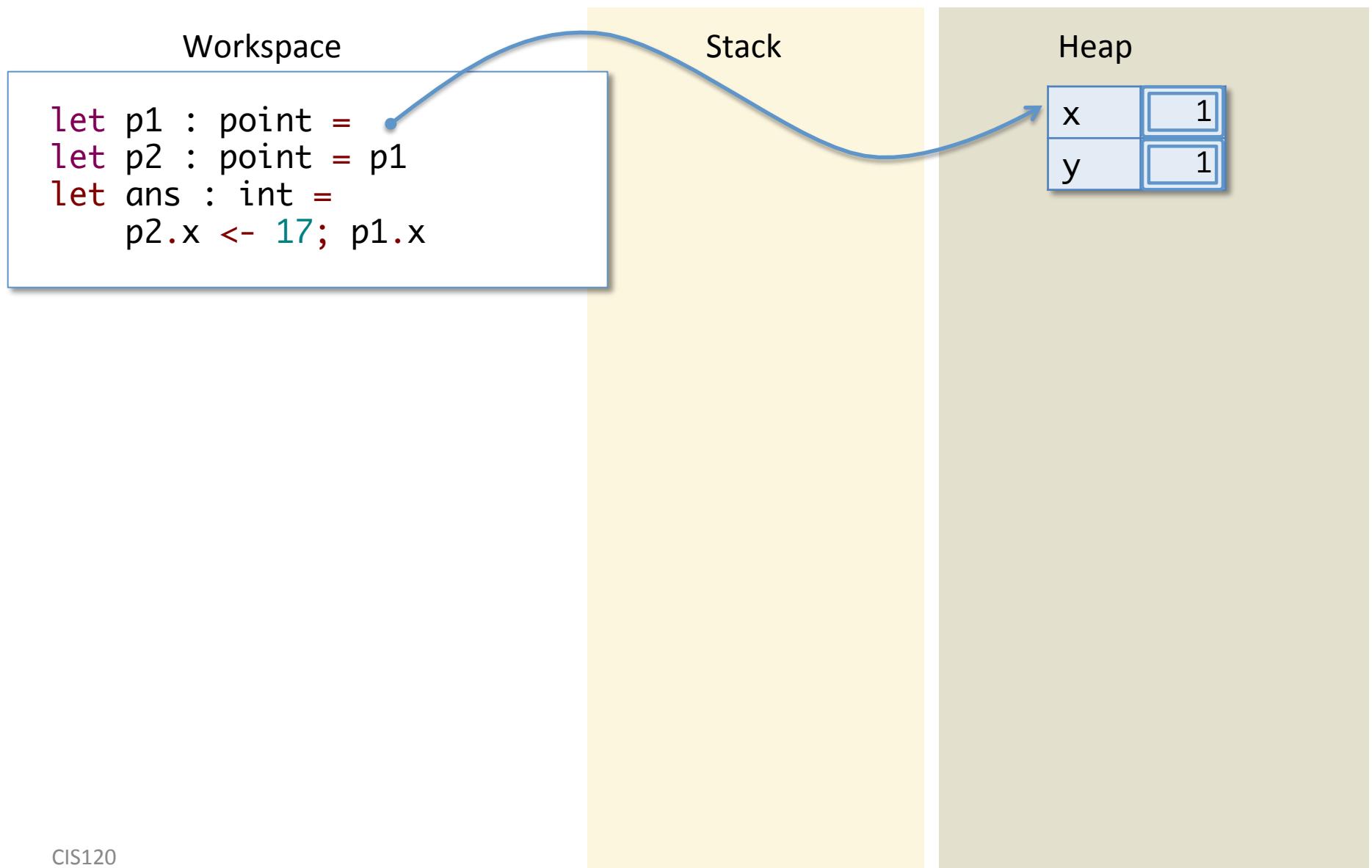
Workspace

```
let p1 : point = {x=1; y=1;}
let p2 : point = p1
let ans : int =
  p2.x <- 17; p1.x
```

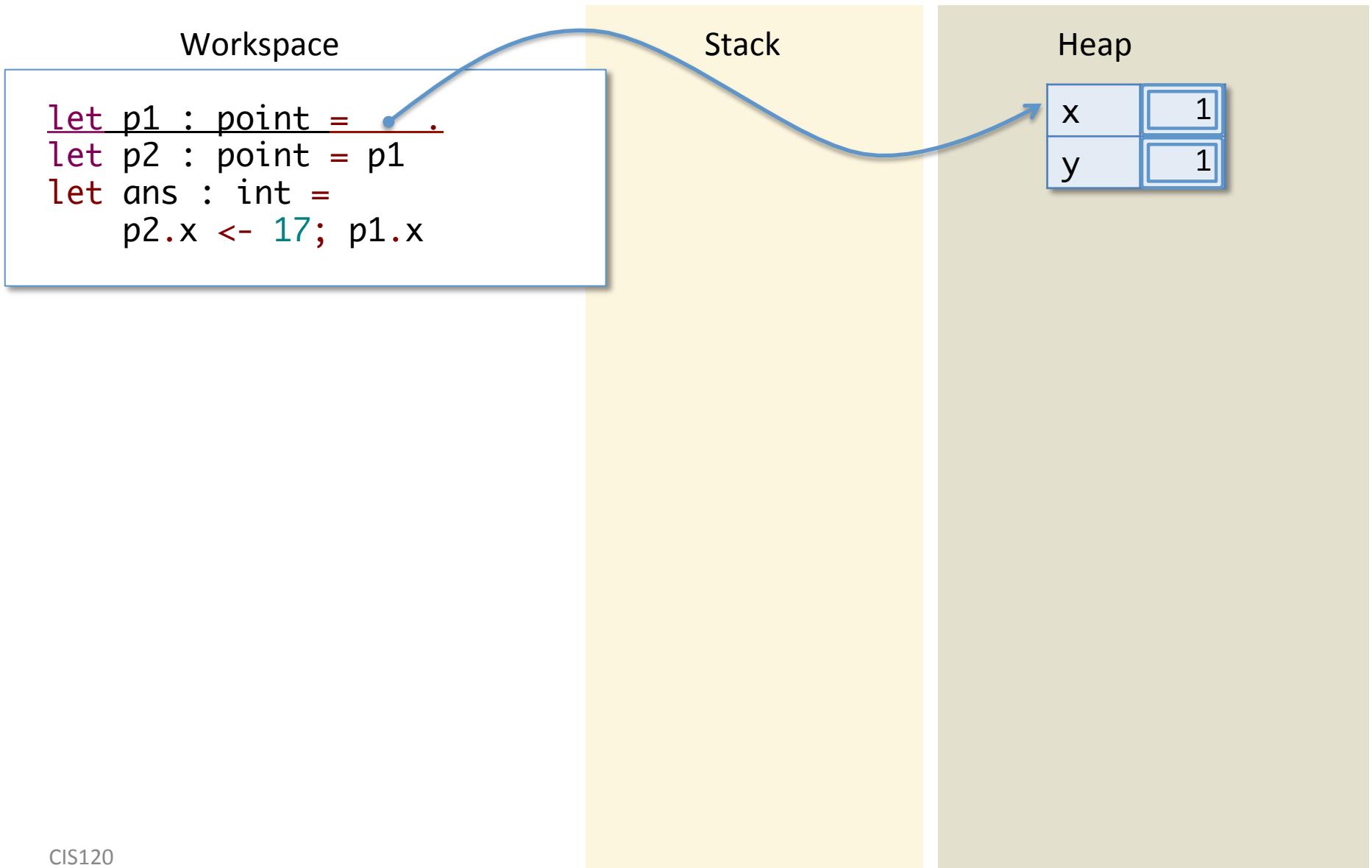
Stack

Heap

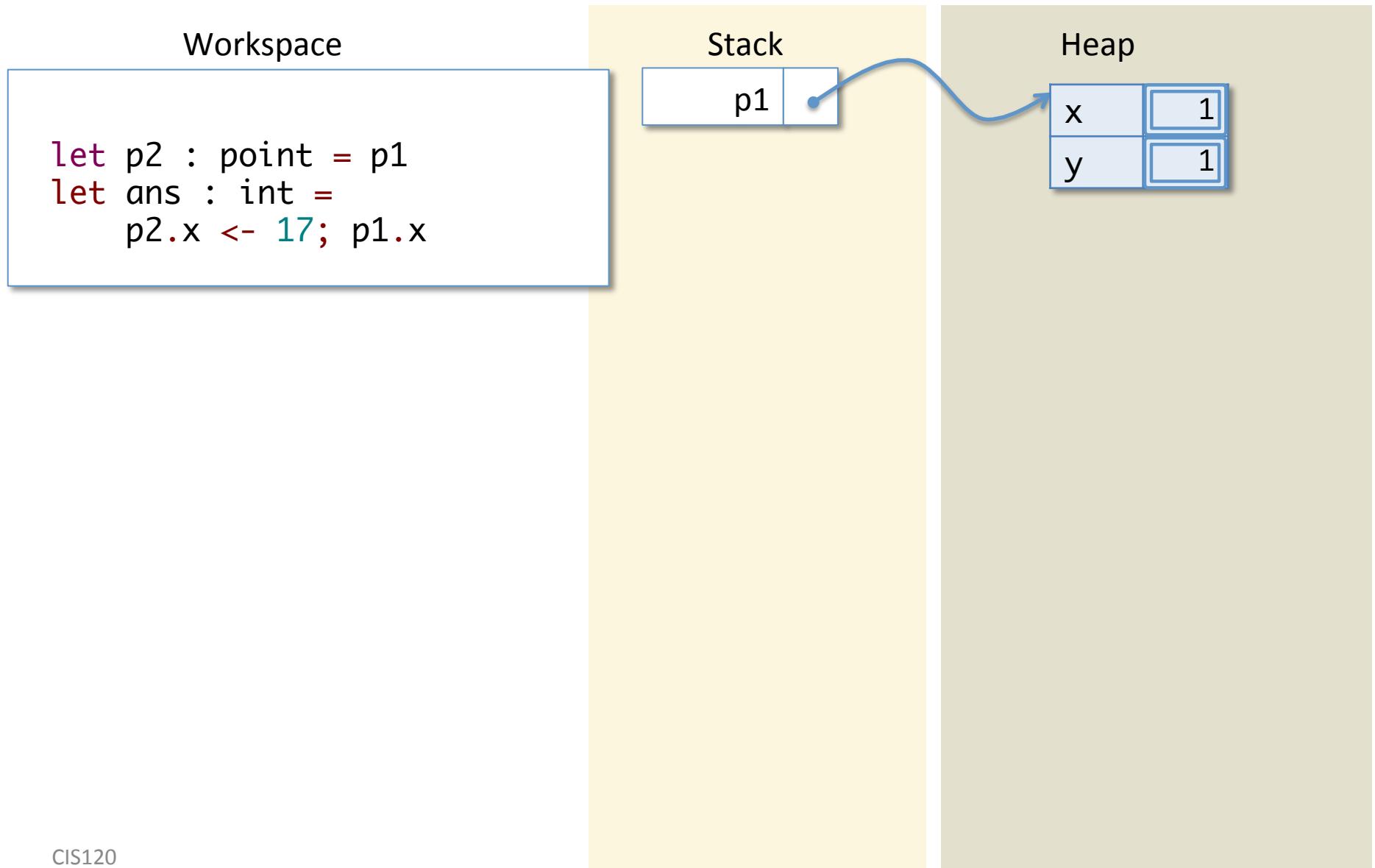
# Allocate a Record



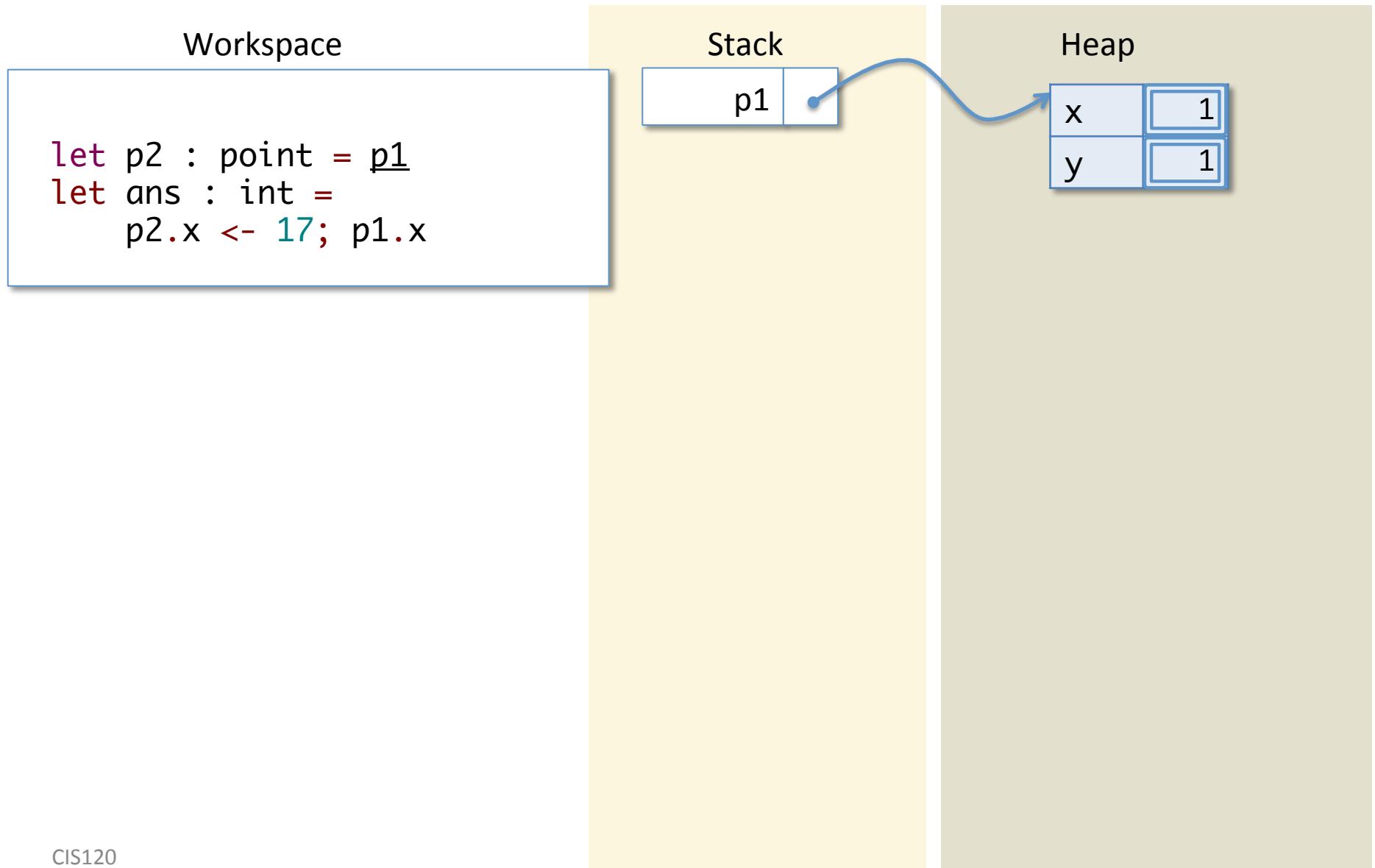
# Let Expression



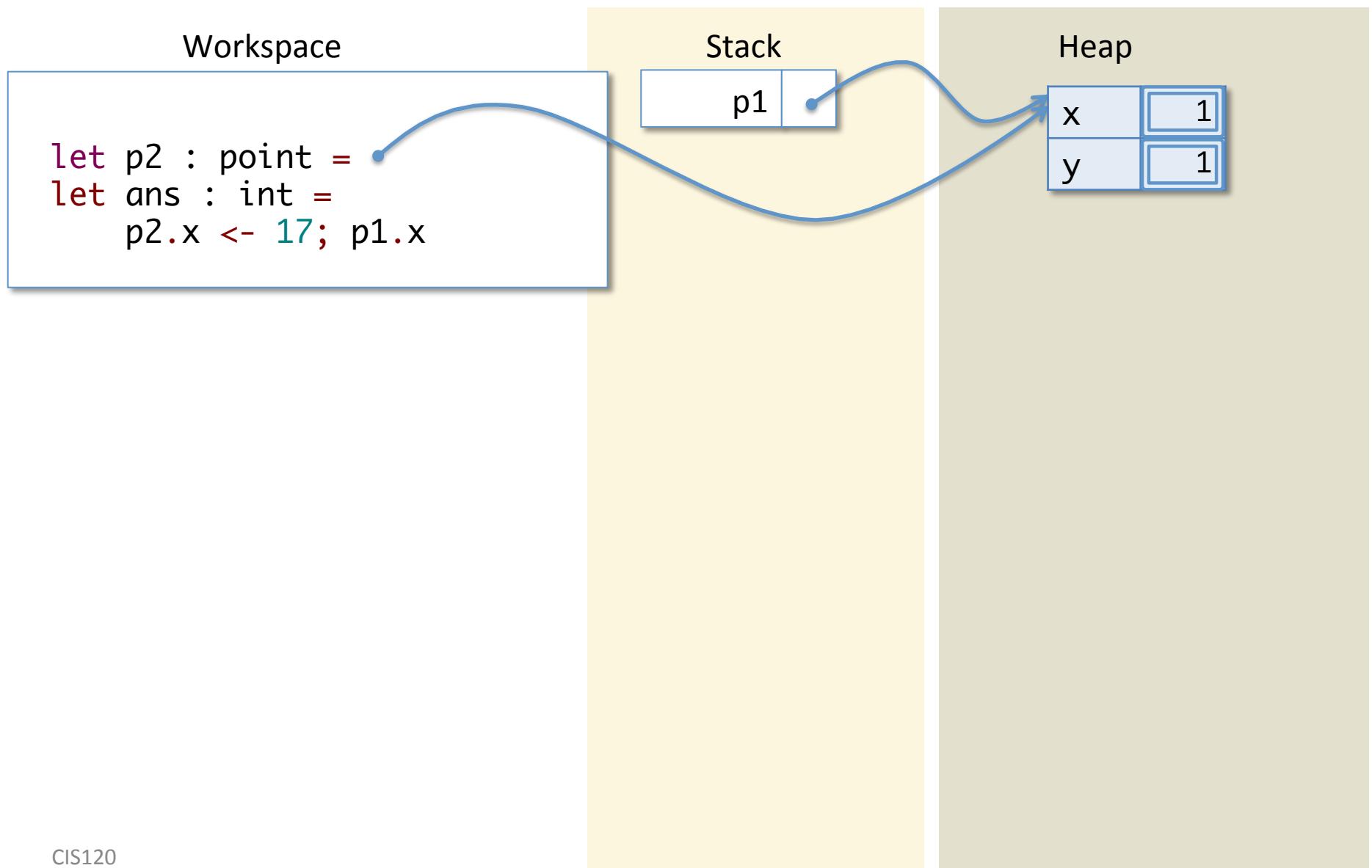
# Push p1



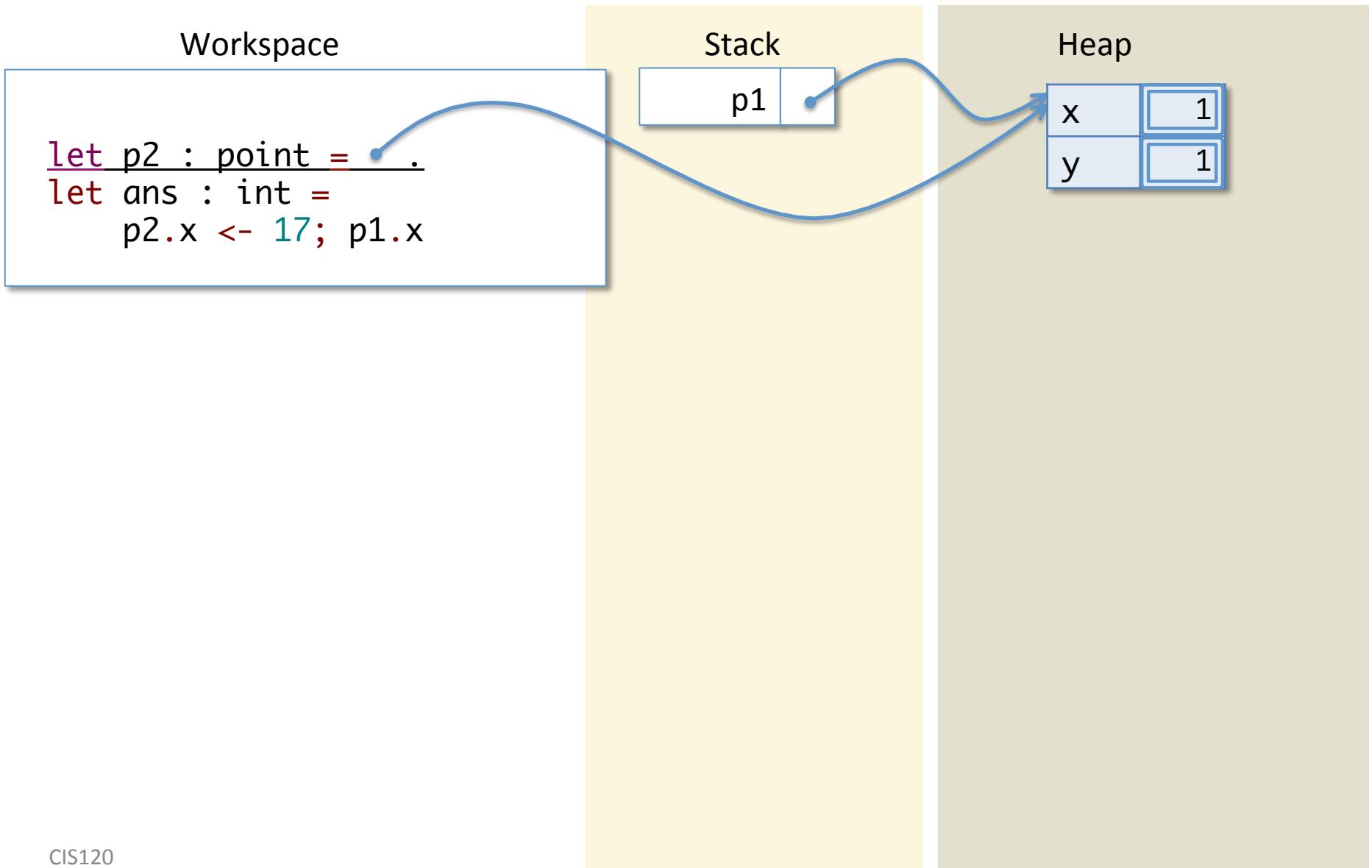
# Look Up 'p1'



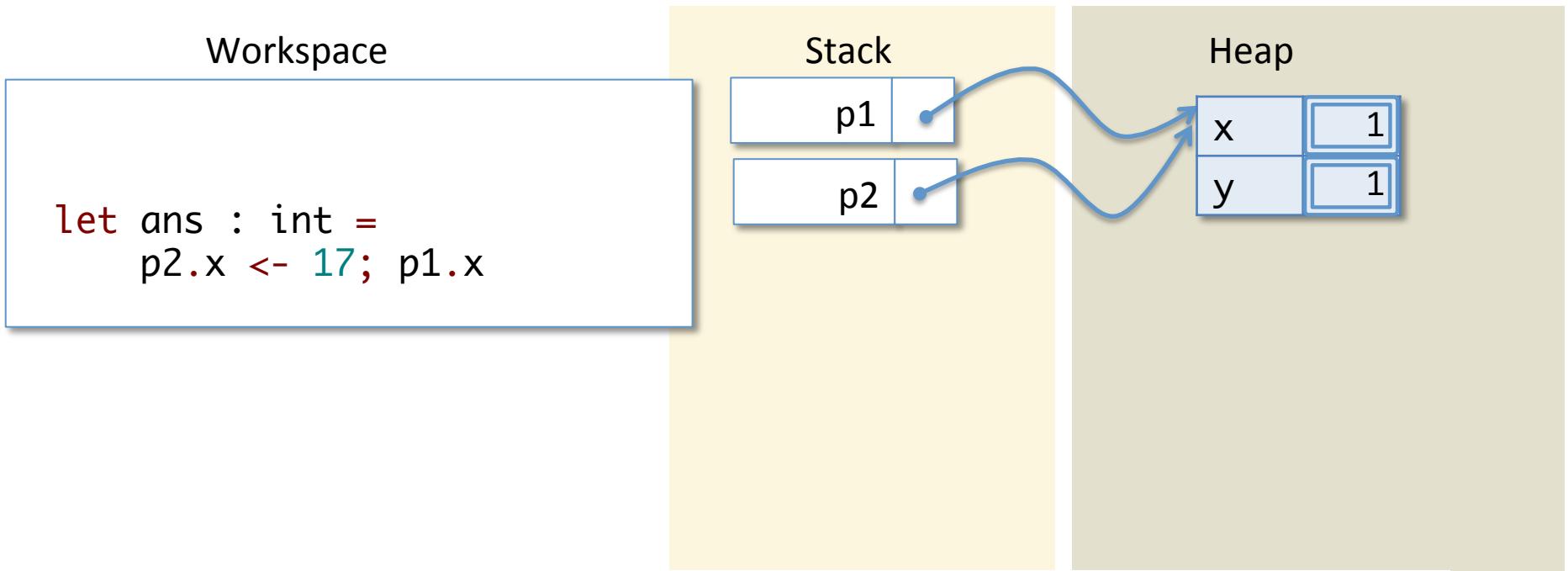
# Look Up 'p1'



# Let Expression

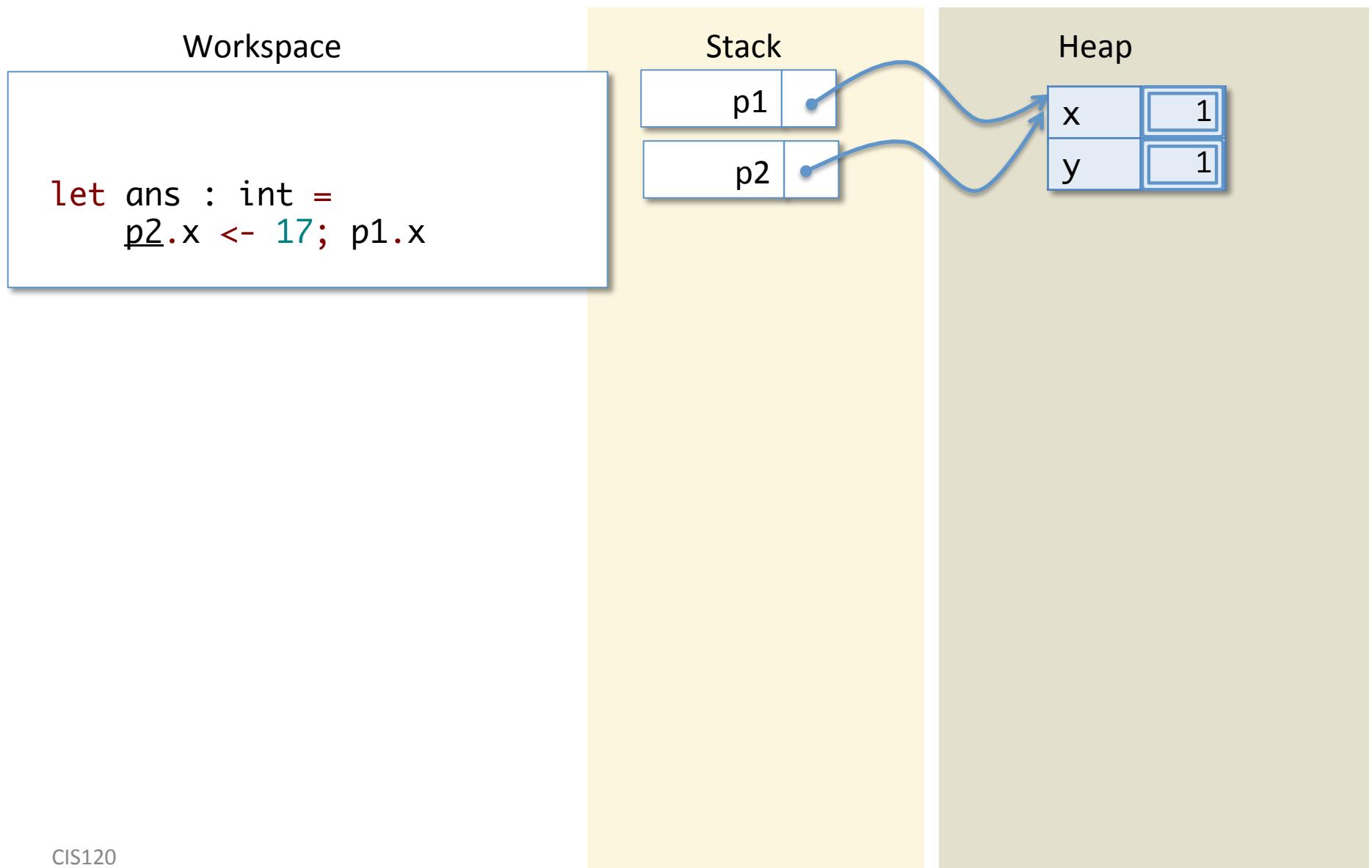


# Push p2

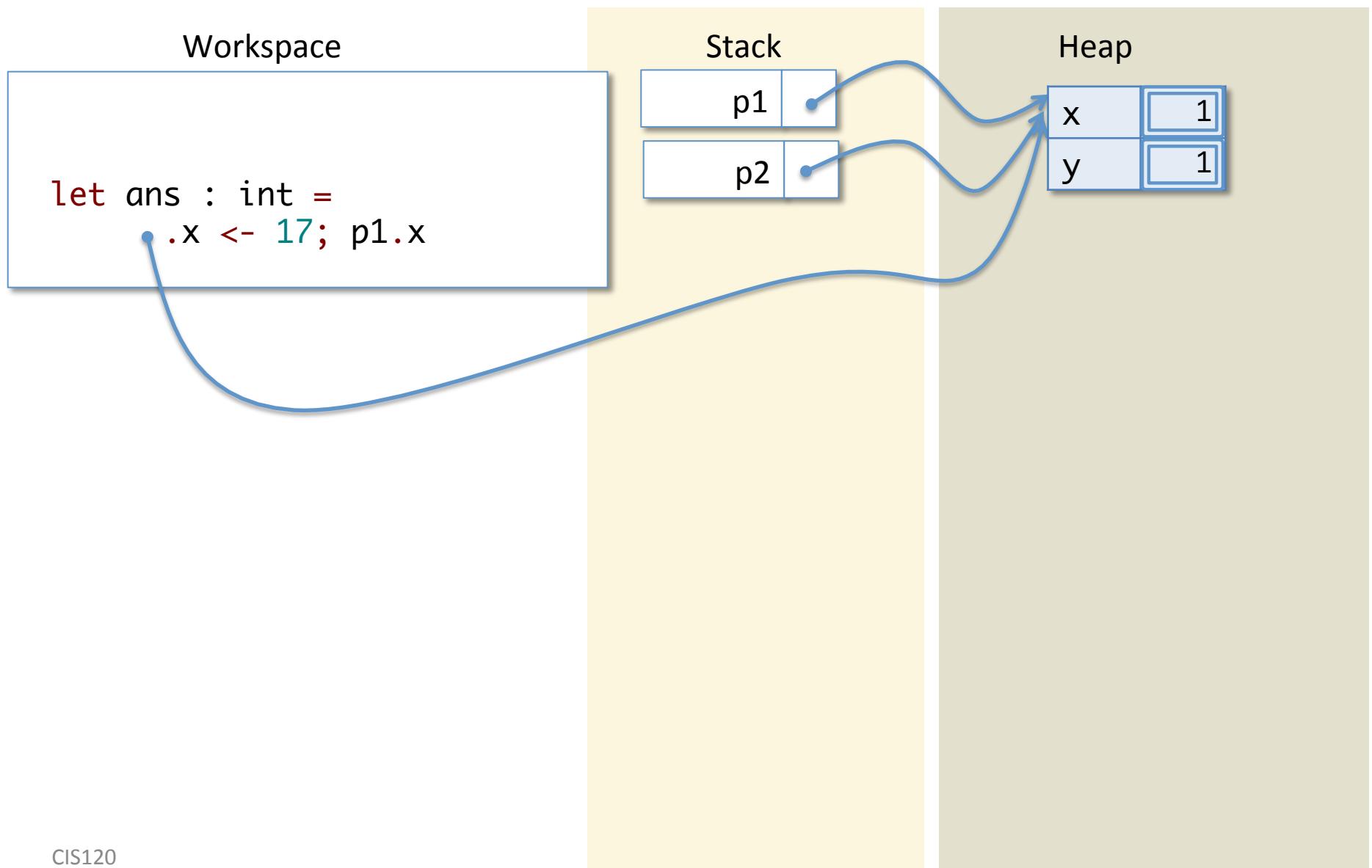


Note: `p1` and `p2` are references to the *same* heap record.  
They are *aliases* – two different names for the *same thing*.

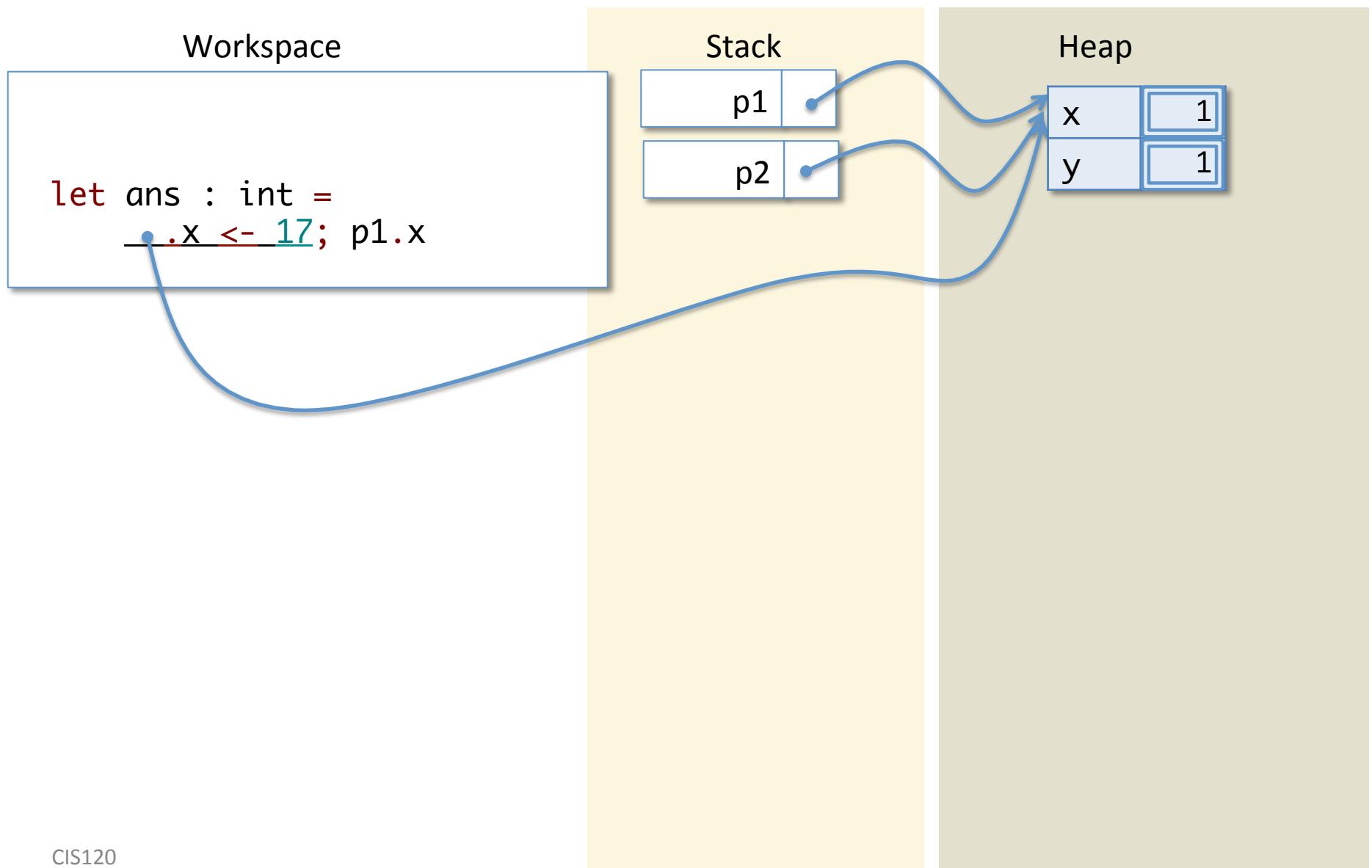
# Look Up 'p2'



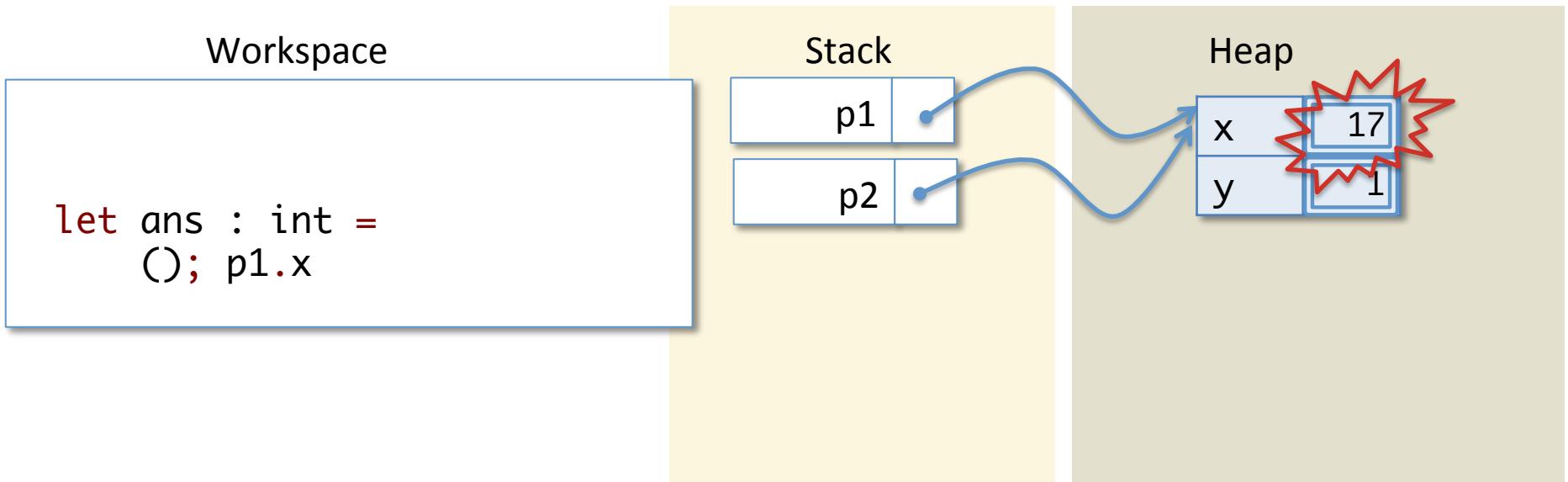
# Look Up 'p2'



# Assign to x field

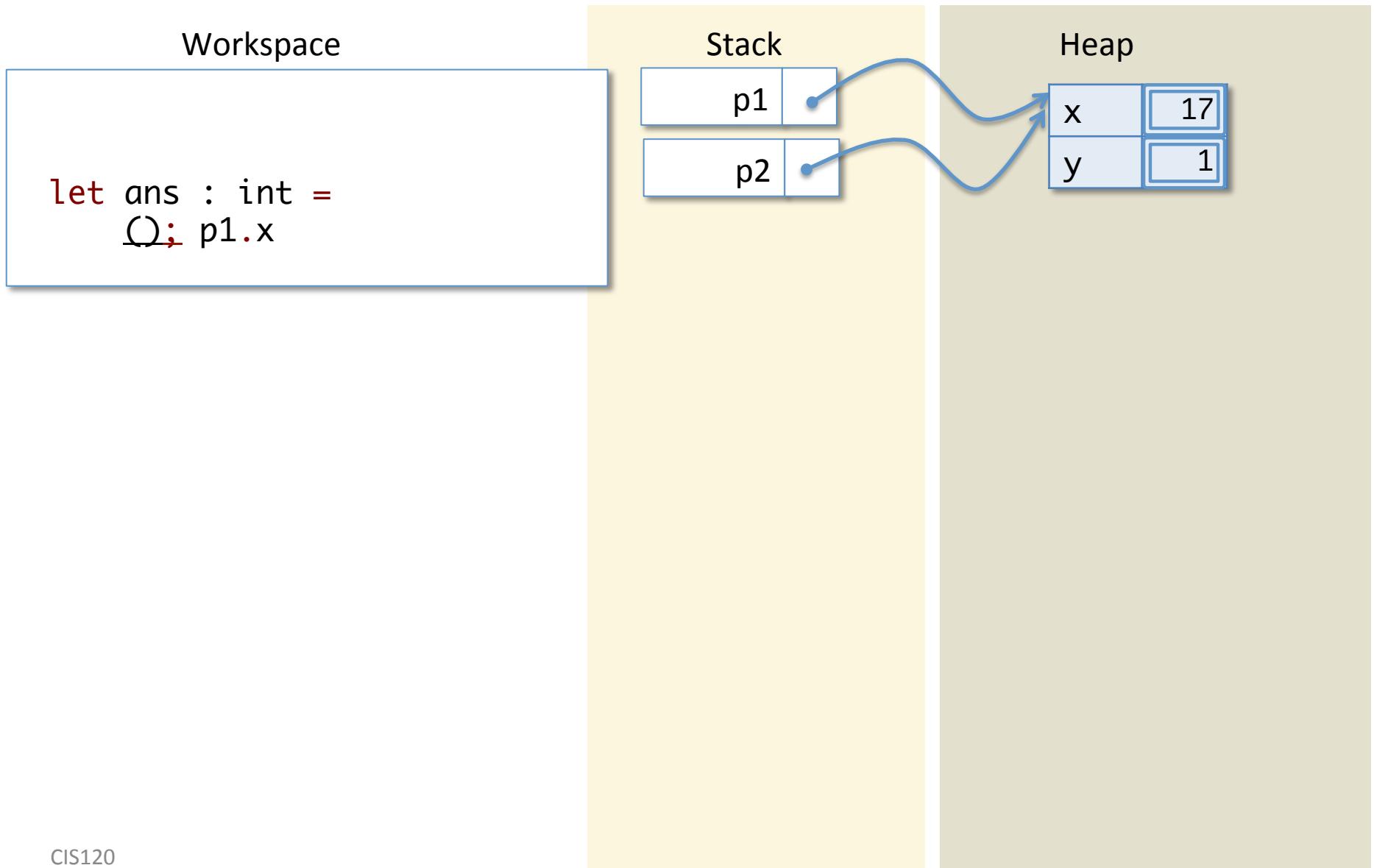


# Assign to x field

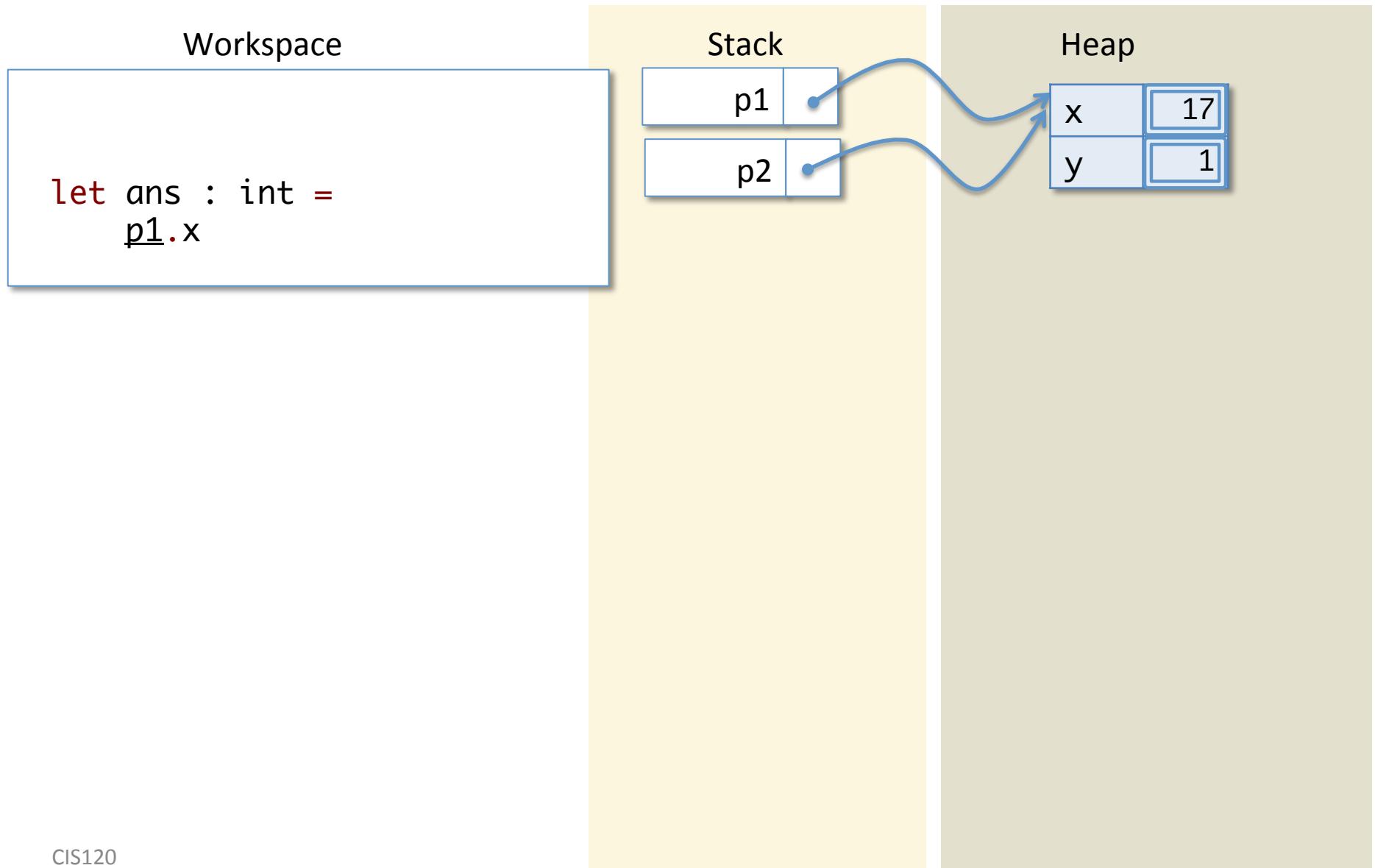


This is the step in which the ‘imperative’ update occurs. The mutable field x has been modified in place to contain the value 17.

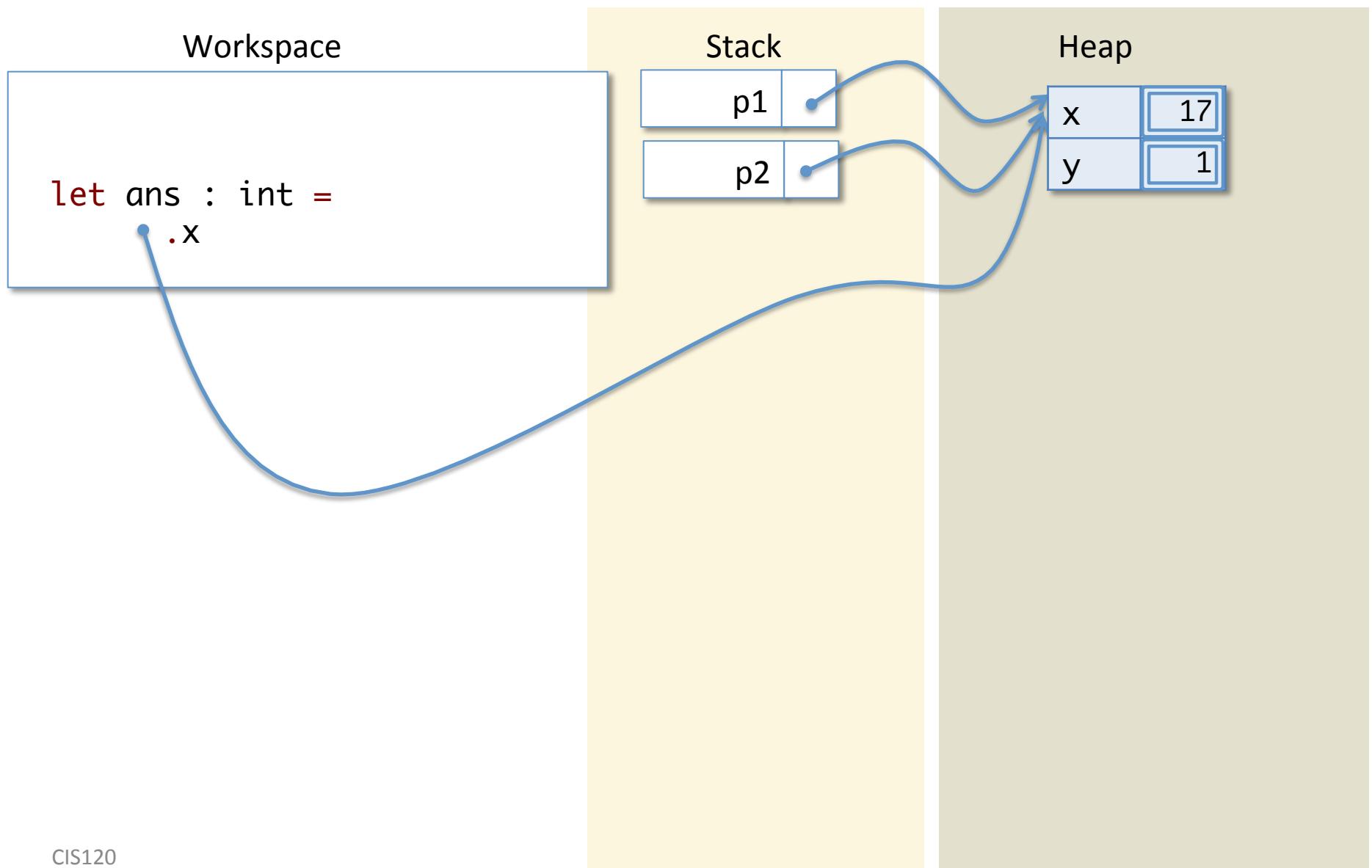
# Sequence ';' Discards Unit



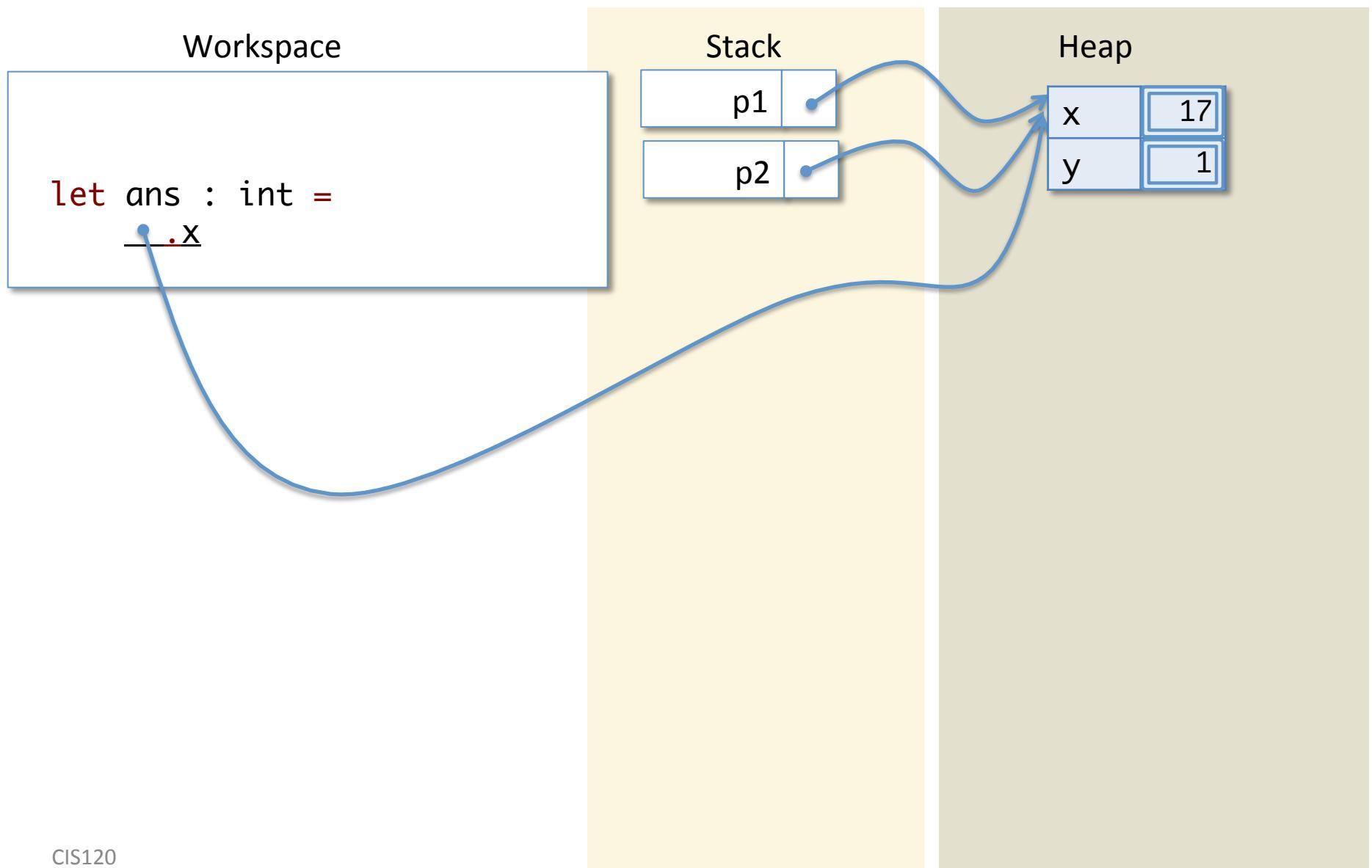
# Look Up 'p1'



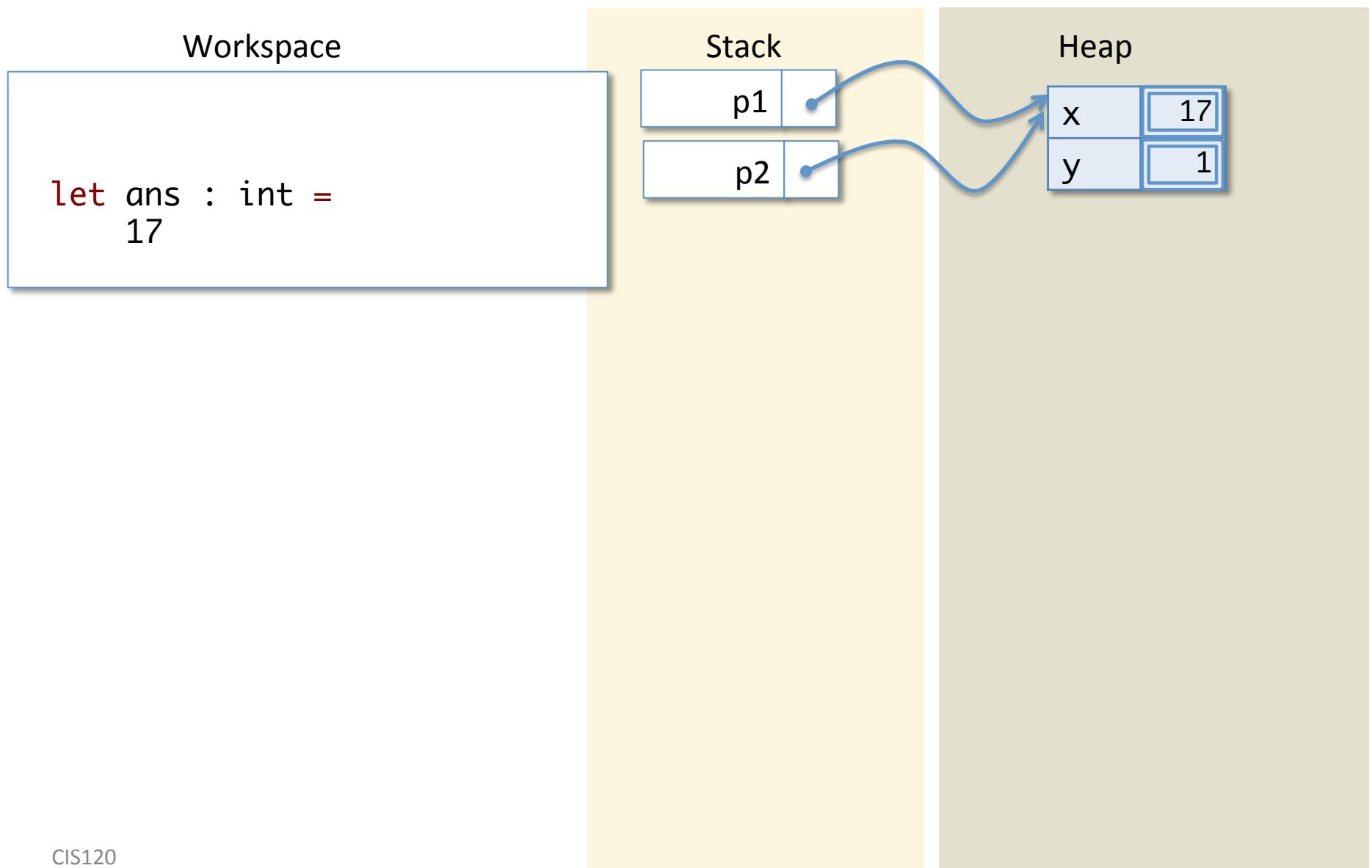
# Look Up 'p1'



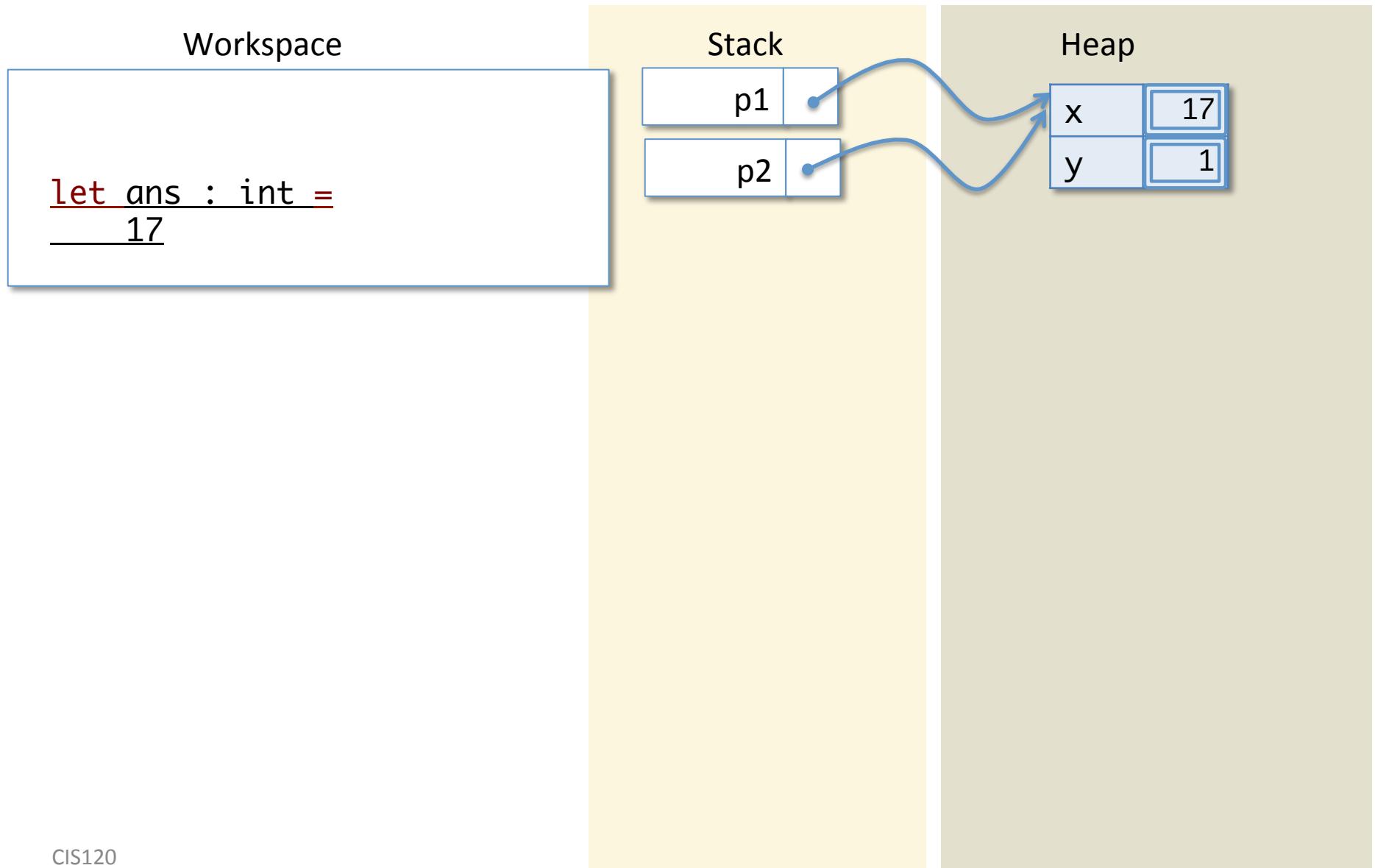
# Project the 'x' field



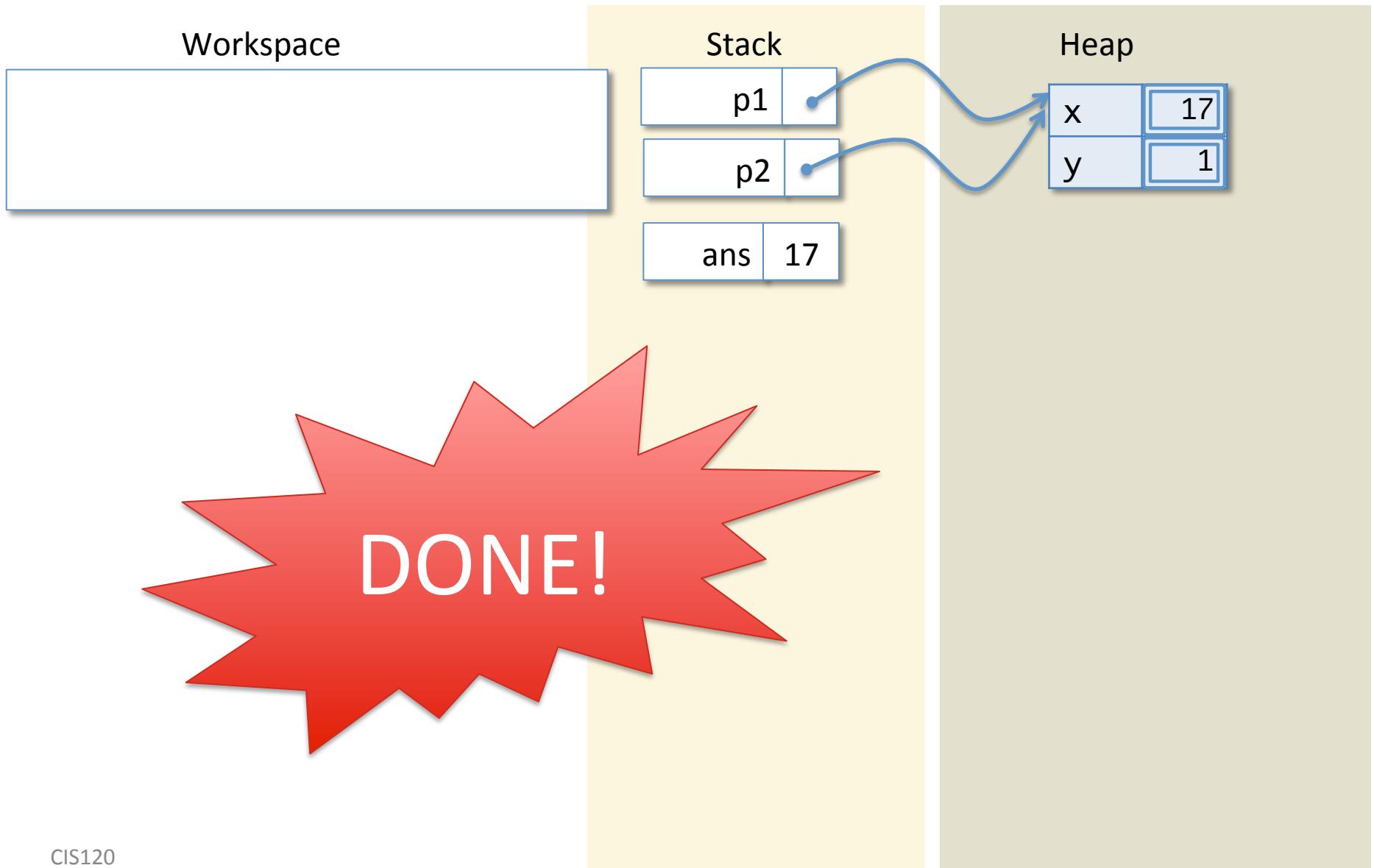
# Project the 'x' field



# Let Expression



# Push ans



What answer does the following function produce when called?

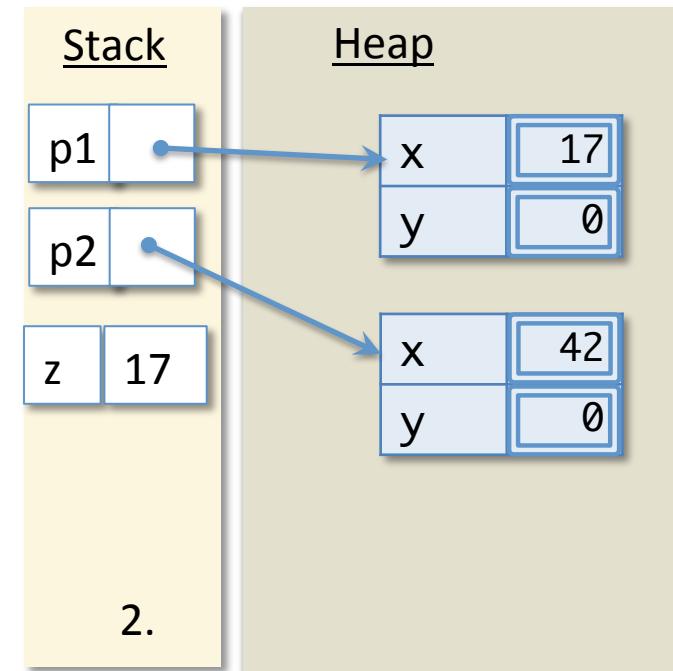
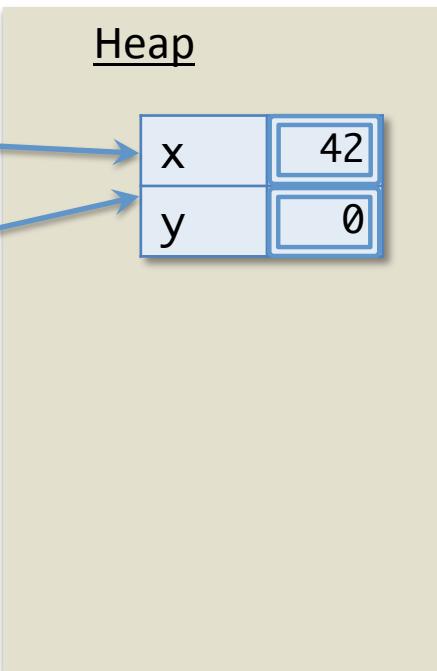
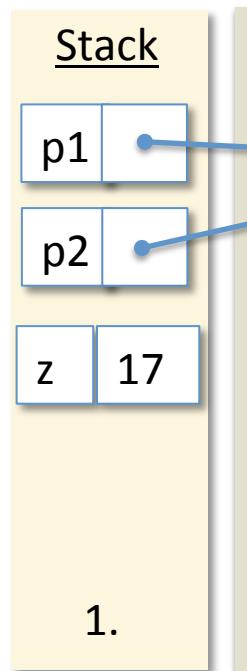
```
let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  let z = p1.x in
  p2.x <- 42;
  z
```

1. 17
2. 42
3. sometimes 17 and sometimes 42
4. f is ill typed

Answer: 17

What do the Stack and Heap look like after simplifying the following code on the workspace?

```
let p1 = {x=0; y=0} in
let p2 = p1 in
p1.x <- 17;
let z = p1.x in
p2.x <- 42;
p1.x
```



# Ah... Refs!

OCaml provides syntax for working with updatable *references*:

`type 'a ref = {mutable contents:'a}`

`ref e`       $\equiv$       `{contents = e}`      has type `t ref` when  $(e : t)$

`e1 := e2`       $\equiv$       `(e1).contents <- e2`      has type `unit` when  
 $(e1 : t \text{ ref})$  and  $(e2 : t)$

`!e`       $\equiv$       `(e).contents`      has type `t` when  $(e : t \text{ ref})$

"is defined to be"  
(not Ocaml syntax)

Ocaml  
"syntactic sugar"

equivalent expressions

type constraints

# Comparison To Java (or C, C++, ...)

Java

```
int f() {  
    int x = 3;  
    x = x + 1;  
    return x;  
}
```

OCaml

```
let f () : int =  
    let x = ref 3 in  
    x := !x + 1;  
    !x
```

- $x$  has type int
- meaning on left of  $=$  different than on right
- *implicit* dereference

- $x$  has type (int ref)
- use  $:=$  for update
- *explicit* dereference