

# Programming Languages and Techniques (CIS120)

## Lecture 5

January 29, 2014

Datatypes and Trees

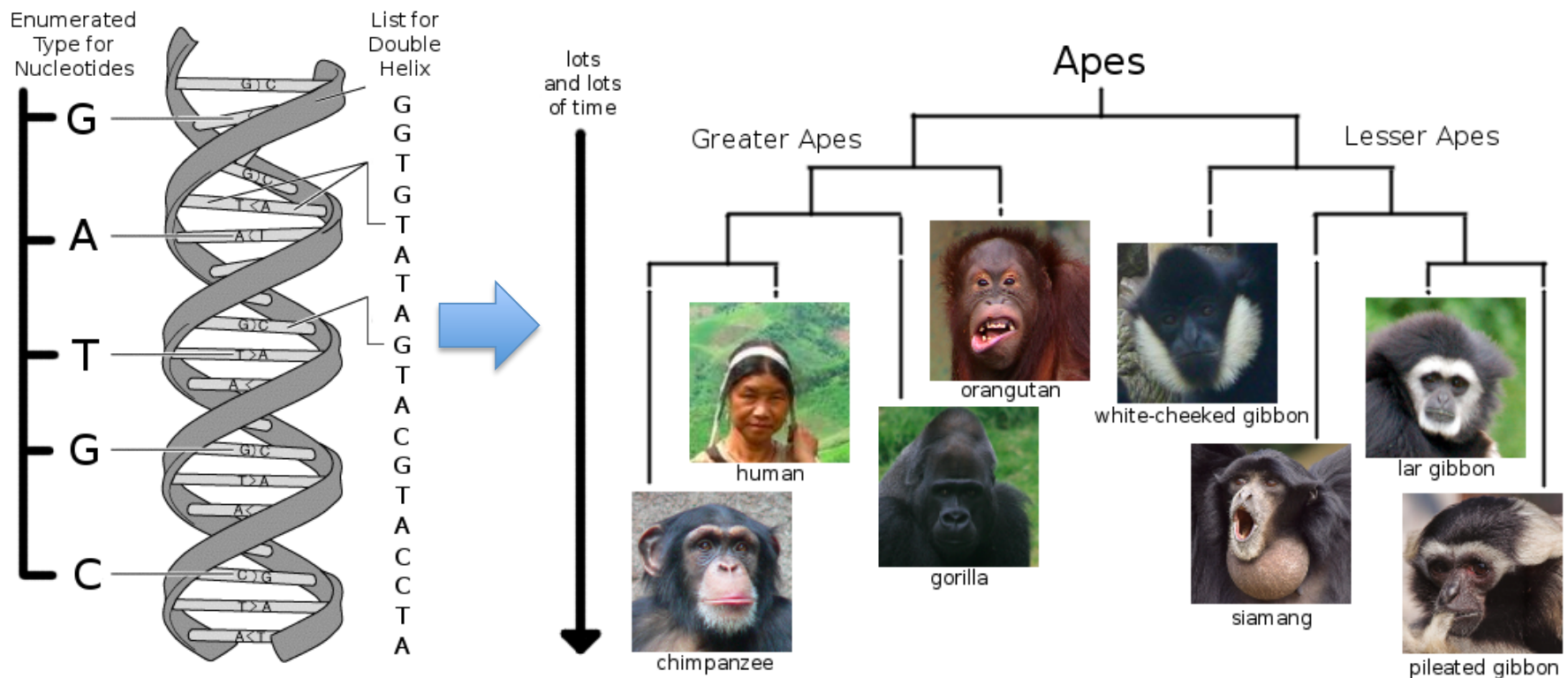
# Announcements

- Submit HW1 by Thursday midnight (hard deadline)
- Homework 2 now available
  - On-time due date: Tuesday, Feb 4<sup>th</sup> at 11:59:59pm
  - Warning: It is a bit more challenging than HW1
  - Get started early, and seek assistance if you get stuck!
- Recitations today and tomorrow
  - 208 and 209 have the most space
- Read Chapters 5 and 6 of the course notes
- Register your clicker ID number on course website
  - You should start seeing “Quizzes” on the submission page
  - Name of quiz is lecture date: TP140127 was Monday
  - If you have “Not submitted” then we don’t have an ID number for your data (that’s 35 of you)

# Datatypes and Trees

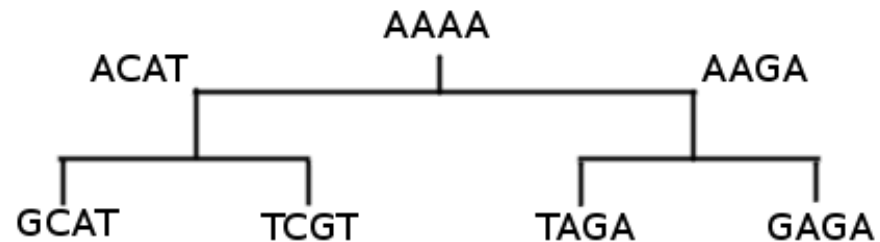
# Case Study: Evolutionary Trees

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



# DNA Computing Abstractions

- Nucleotide
  - Adenine (A), Guanine (G), Thymine (T), or Cytosine (C)
- Codon
  - three nucleotides : e.g. (A,A,T) or (T,G,C)
  - codons map to amino acids and other markers
- Helix
  - a sequence of nucleotides: e.g. AGTCCGATTACAGAGA...
- Phylogenetic tree
  - Binary (2-child) tree with helices (species) at the nodes and leaves



# 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, ...)

# Simple User-Defined Datatypes

- OCaml lets programmers define *new* datatypes

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

Annotations for the `nucleotide` definition:

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

- `type`: 'type' keyword
- `nucleotide`: type name (must be lowercase)
- `A`, `C`, `G`, `T`: constructor names (*tags*) (must be capitalized)

- 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 i.e. it doesn't make sense to do arithmetic like:  
Wednesday - Monday = Tuesday
    - There are *more* integers than days, i.e. "17" isn't a valid day under the representation above, so you must be careful never to pass such invalid "days" to functions that expect days.
  - Conflating integers with days can lead to many bugs.
- All modern languages (Java, C#, C++, OCaml,...) provide user-defined types for this reason.

# Type Abbreviations

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

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



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
```

A blue arrow points from the text 'keyword 'of'' to the 'of' keyword in the 'CodonCount' constructor. A blue bracket points from the text 'Constructors may take a tuple of arguments' to the tuple '(A, G, T)' in the 'CodonCount' constructor.

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

# 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)

*Clickers, please...*

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

What is the type of this expression?

[A;C]

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

Answer: both 2 and 3

*Clickers, please...*

```
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)*

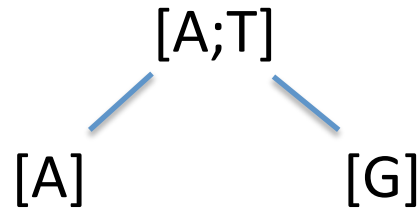
Answer: 4



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

*Clickers, please...*

How would you construct this tree in OCaml?



1. Leaf [A;T]
2. Node (Leaf [G], [A;T], Leaf [A])
3. Node (Leaf [A], [A;T], Leaf [G])
4. Node (Leaf [T], [A;T],  
Node (Leaf [G;C], [G], Leaf []))
5. None of the above

*Clickers, please...*

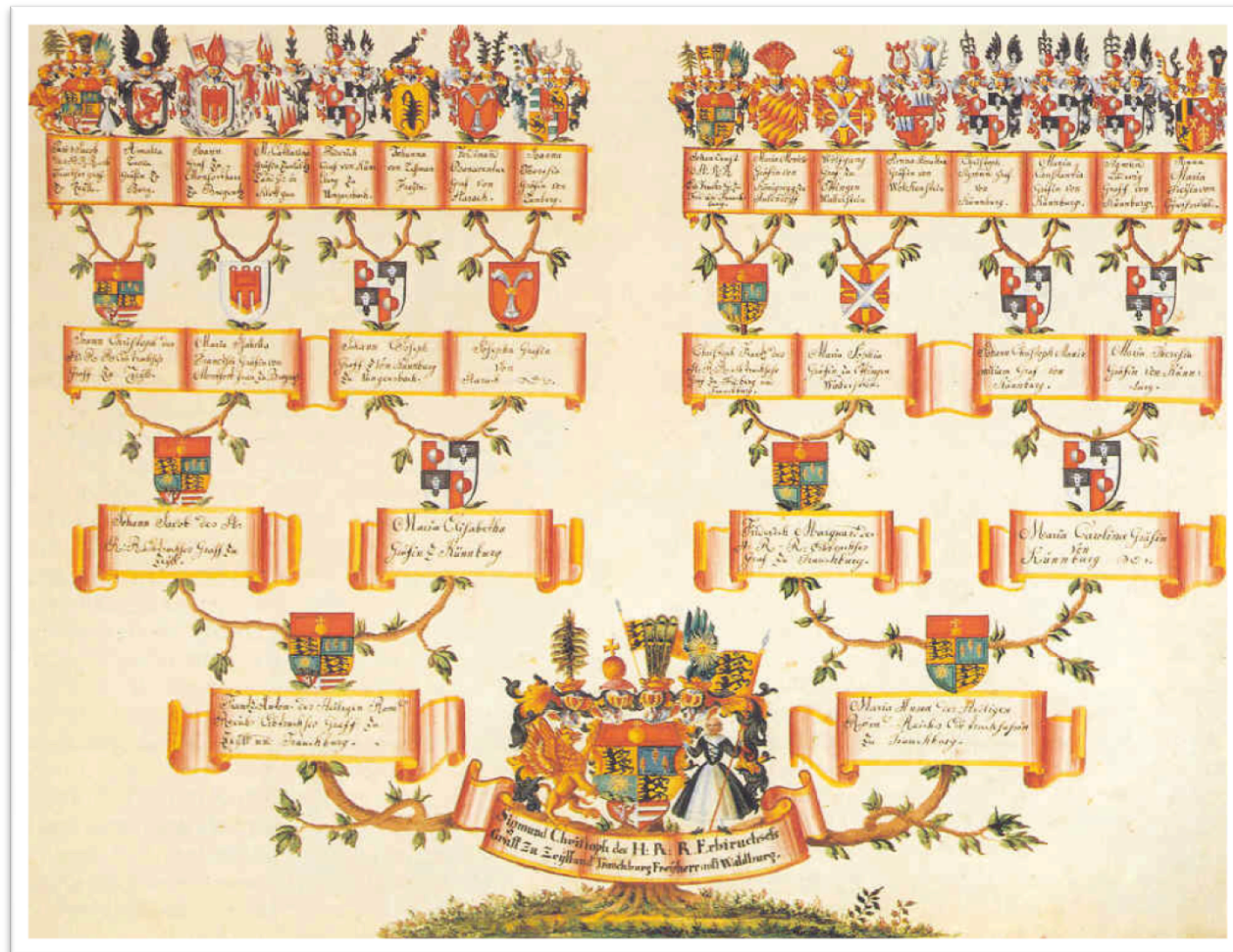
Have you ever programmed with trees before?

1. yes
2. no
3. *not sure*

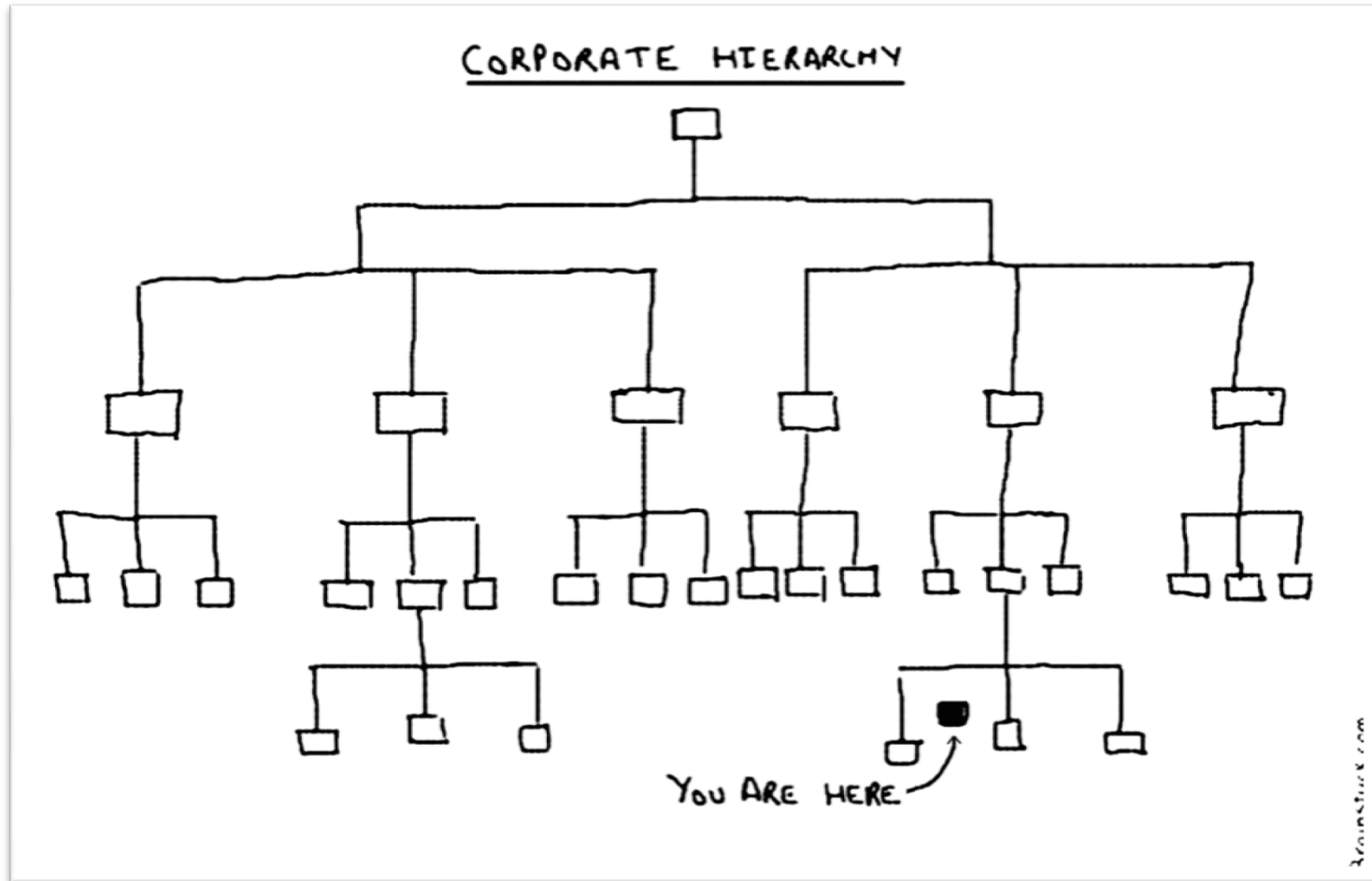
Class answer: about 60% said yes

Trees are everywhere

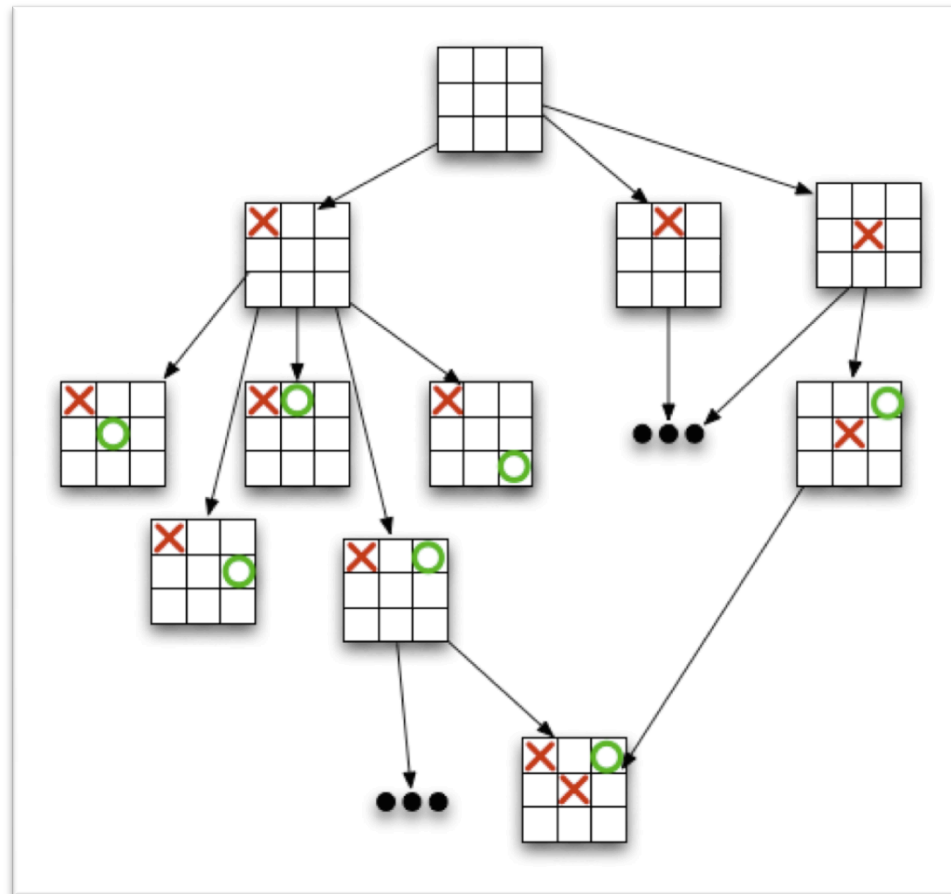
# Family trees



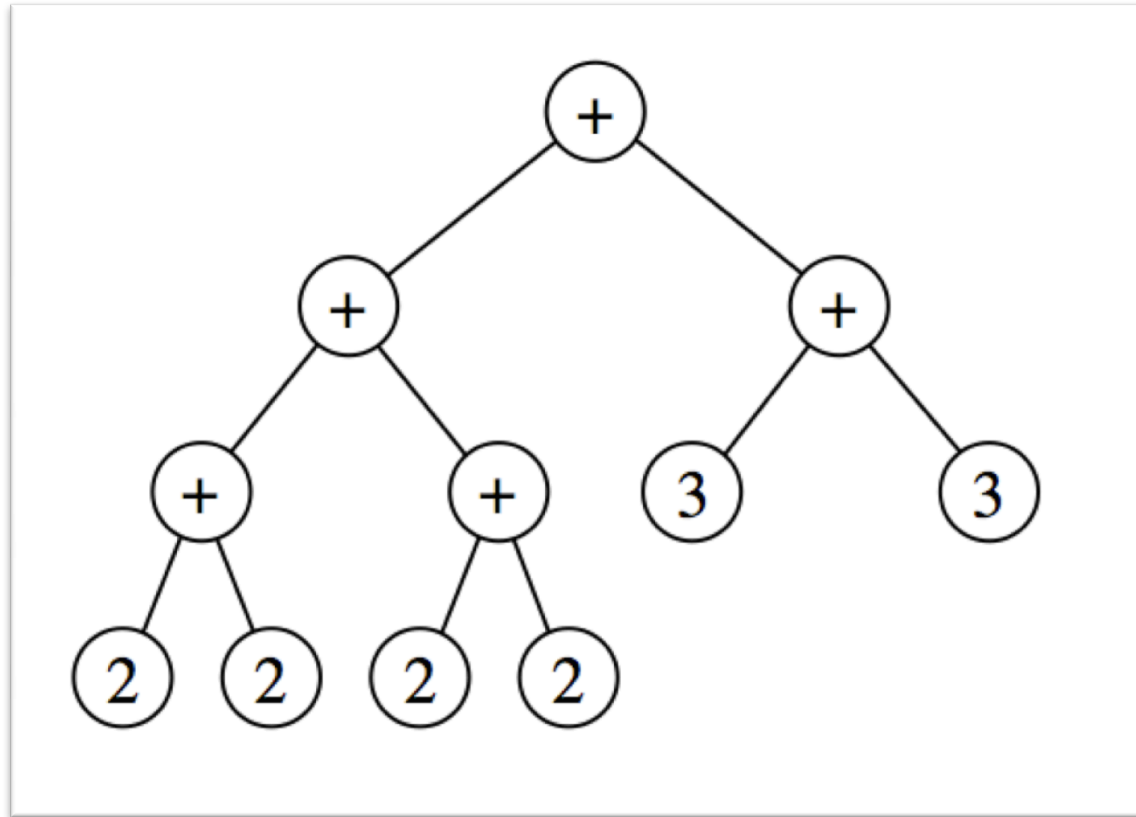
# Organizational charts



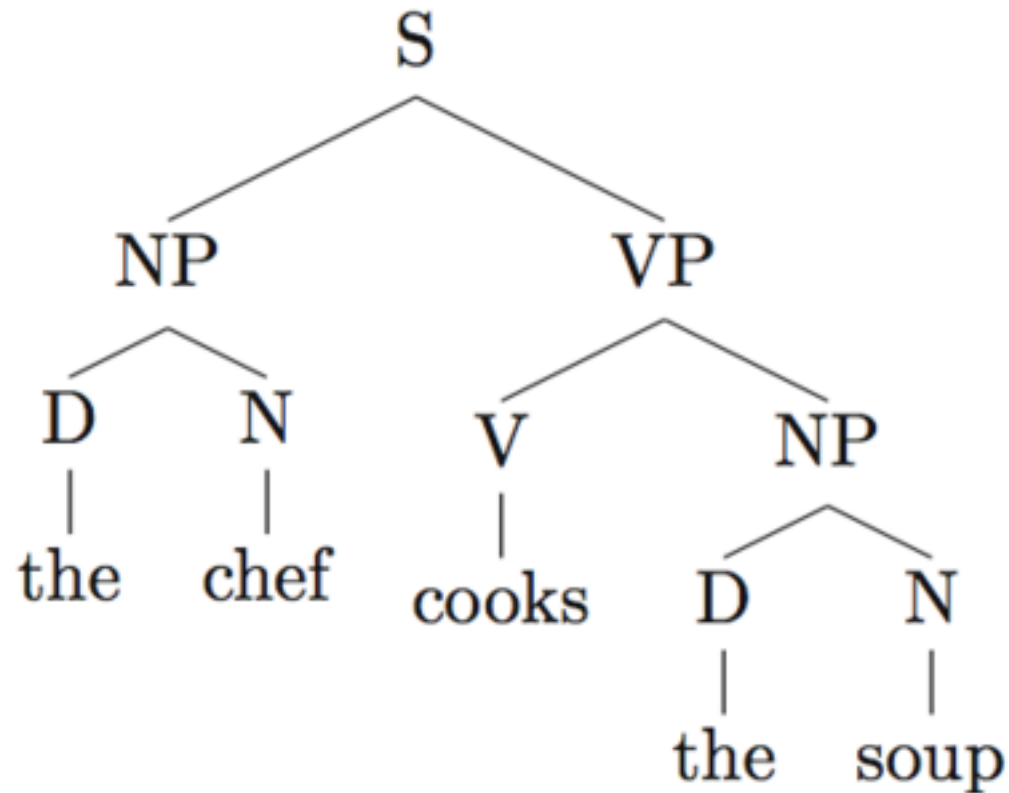
# Game trees



# Expression trees

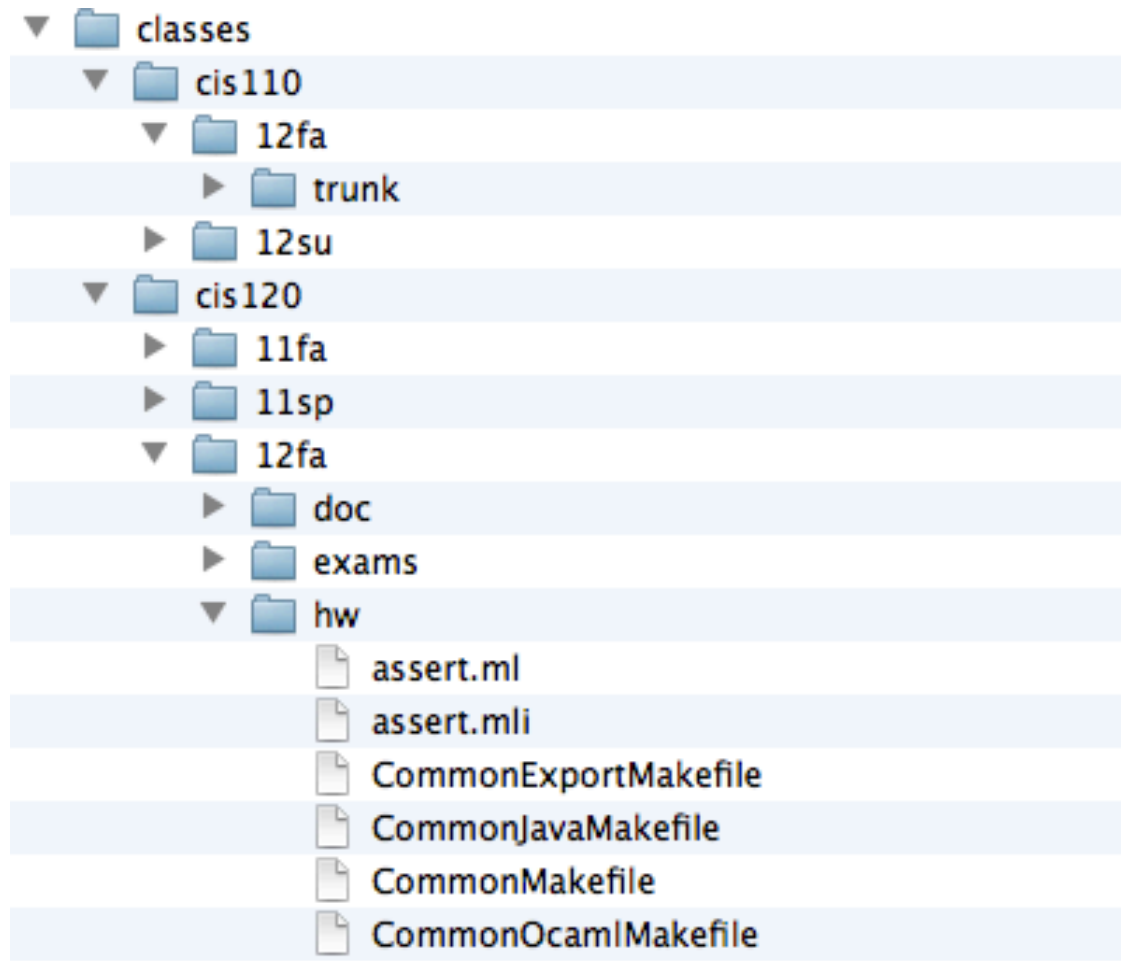


# Natural-Language Parse Trees

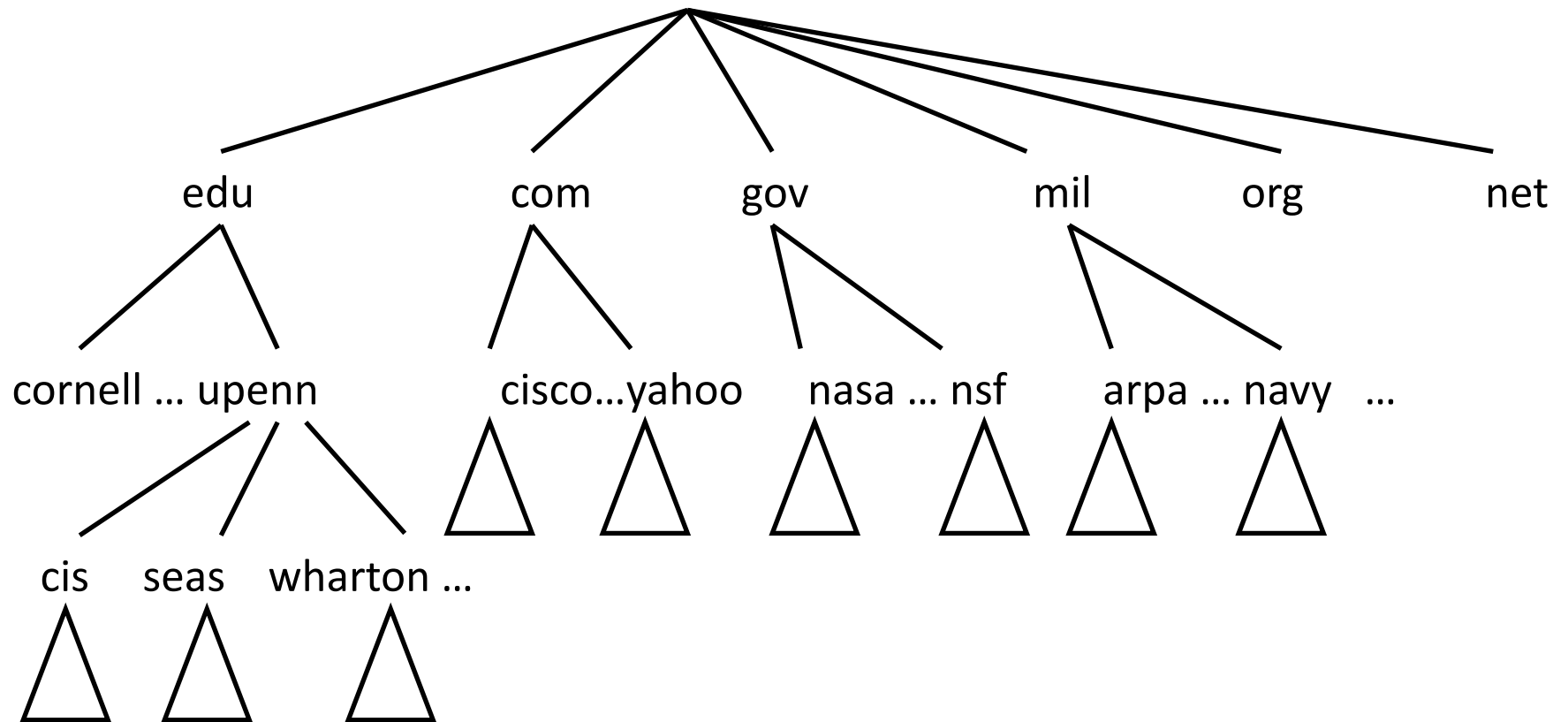




# Filesystem Directory Structure

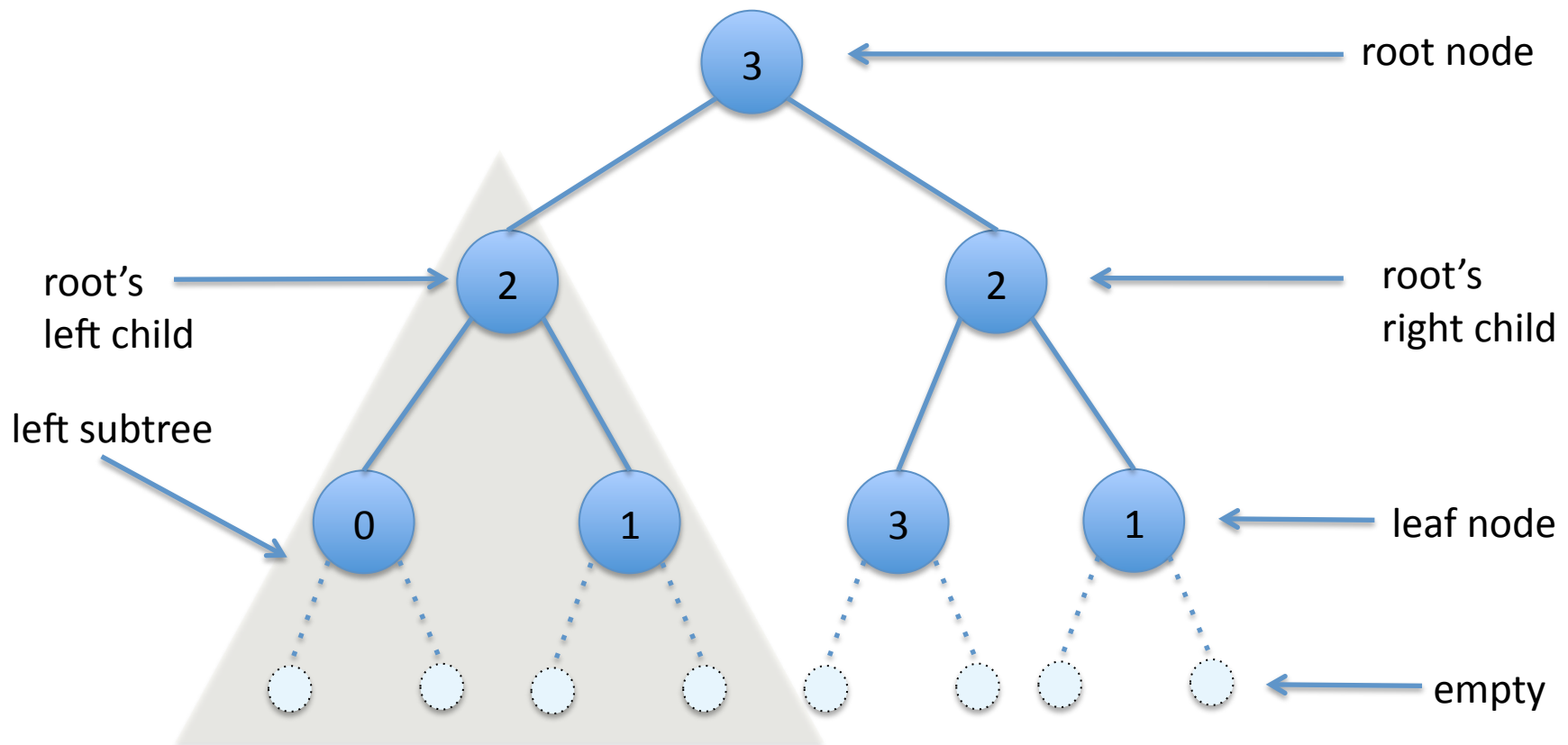


# Domain Name Hierarchy



# Binary Trees

# 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)))
```

