

Programming Languages and Techniques (CIS120)

Lecture 5

September 4th 2015

Datatypes and Trees

Announcements

- No class Monday (Labor Day)
- My office hours: Moved to Tuesday 3:30 – 5:00
- Submit HW1 by midnight tuesday
 - Late policy: 10pt penalty for up to 24 hours
20pt penalty for 24-48 hours
- Register your clicker ID number on course website
 - You should start seeing “Quizzes” on the submission page
 - Name of quiz is lecture date: TP150904 is Today
 - If you have “Not submitted” then we don’t have an ID number for your data
- Read Chapters 5 and 6 of the course notes

More List Programming

see [lists.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 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

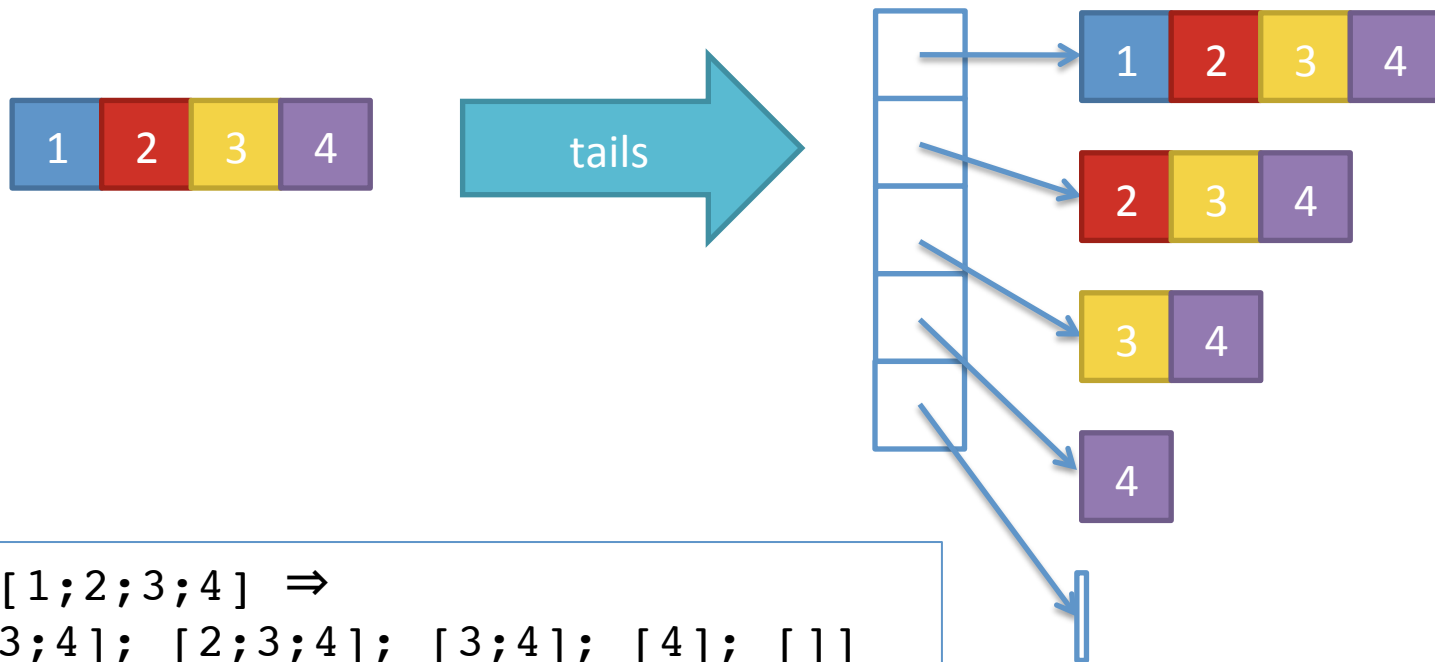
- zip takes two lists of the same length and returns a single list of pairs:

```
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
```

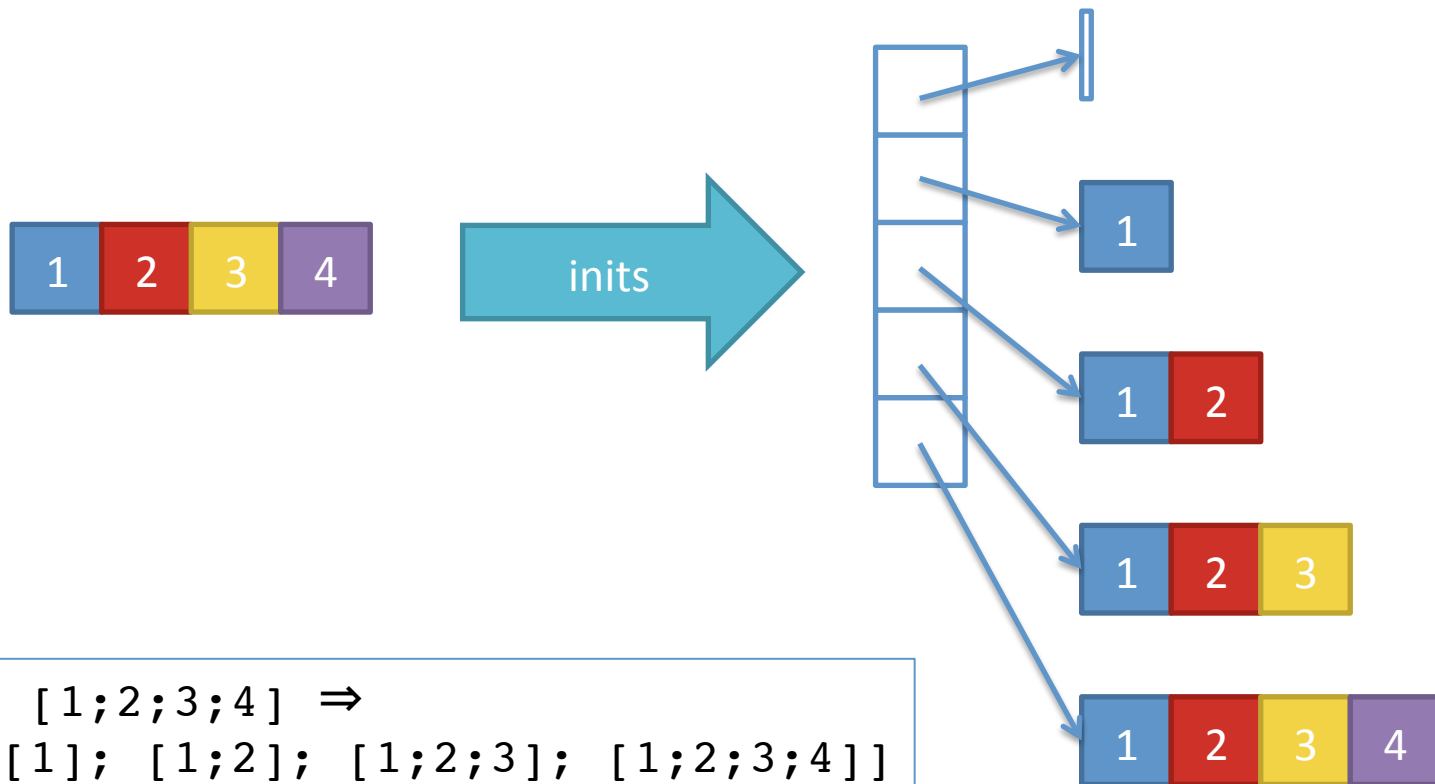
tails

- Design problem: Given a list of integers, produce all suffixes of a given list, starting with the full list and removing the first element at each step



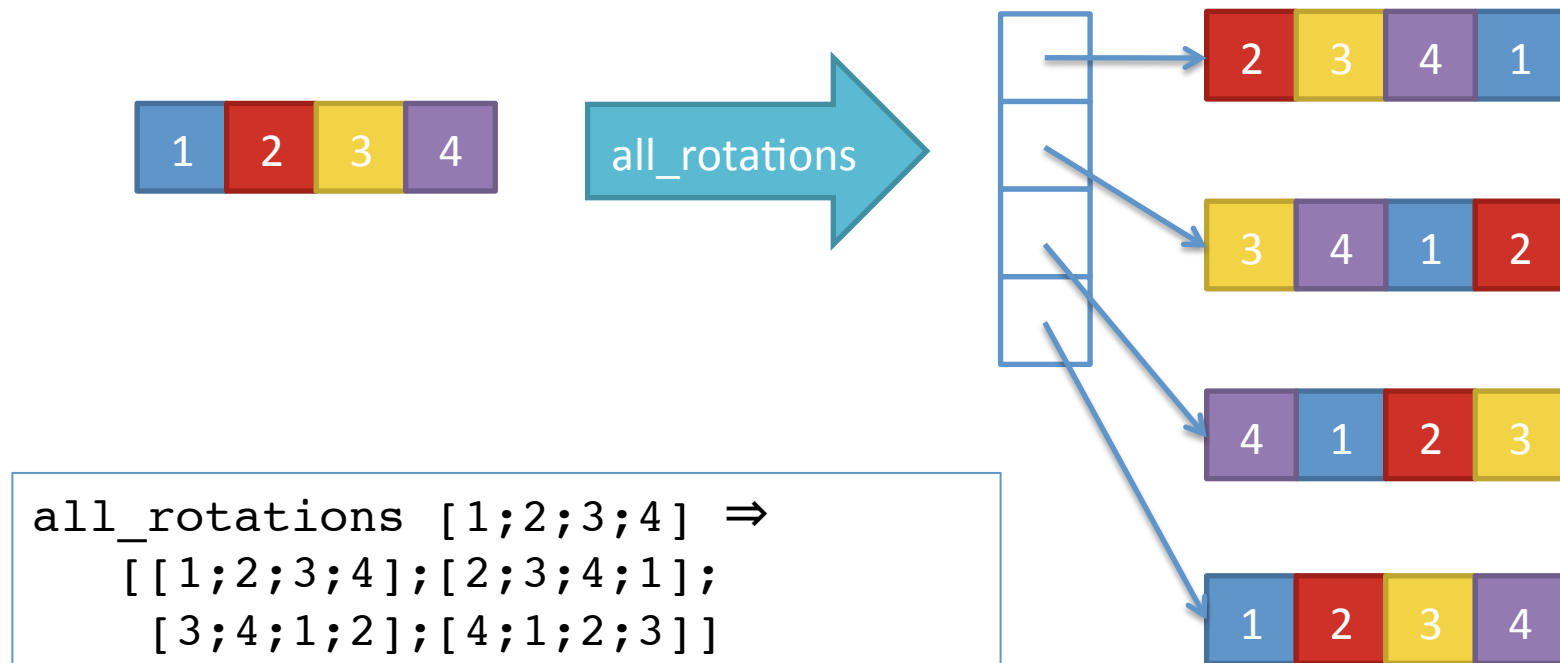
inits

- Design problem: Given a list, produce all *initial prefixes* of the list.



Challenge: All rotations

- Design problem: Given a list, produce all *rotations* of the list.



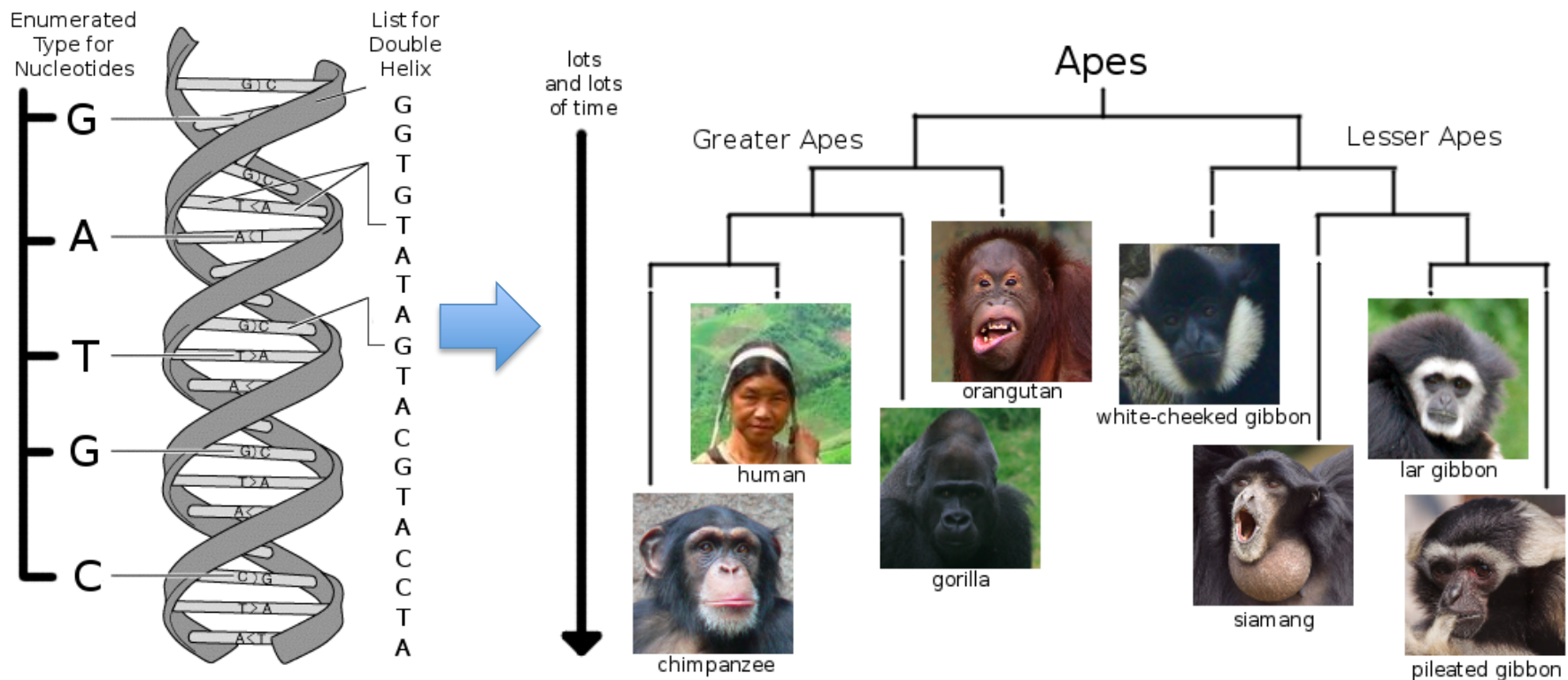
Datatypes and Trees

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

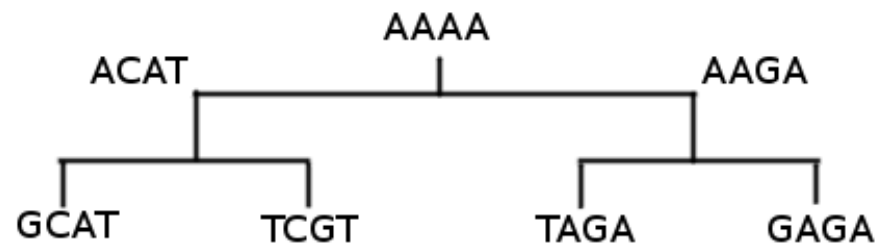
Case Study: Evolutionary Trees

- 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)
- 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
```

Annotations for the `nucleotide` definition:

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

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

- 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.
 - Many *scripting* languages (PHP, Javascript, Perl, Python,...) violate such abstractions (`true == 1 == "1"`), leading to much 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 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
```

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")
```

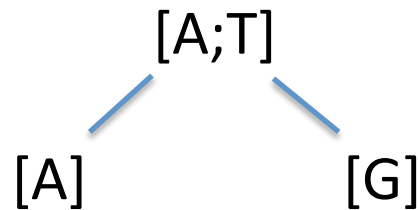
1. nucleotide
2. nucleotide list
3. helix
4. nucleotide * string
5. string * string
6. *none (expression is ill typed)*

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