Programming Languages and Techniques (CIS120)

Lecture 6
January 27th, 2016

Datatypes and Trees
Announcements

• Great job on HW1!

• Homework 2 is available
  – due Tuesday, February 2\textsuperscript{nd}

• Lecture attendance grade (i.e. clickers)
  – Flexibility for occasional missed lectures due to minor emergencies (i.e. it’s OK to miss a few lectures)

• Please complete the CIS 120 Demographics Survey
  – See Piazza (or this week’s labs)

• Read Chapter 6 and 7
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 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?

Suggested reading:
Dawkins, The Ancestor's Tale: A Pilgrimage to the Dawn of Evolution
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

```ocaml
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

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

‘type’ keyword
(type name (must be lowercase)

constructor names (tags)
(must be capitalized)
Pattern Matching Simple Datatypes

• Datatype values can be analyzed by pattern matching:

```haskell
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* make new abstractions:

```ocaml
let x : codon = (A, C, C)
```

- i.e. a `codon` is the same thing a triple of nucleotides

- Makes code easier to read & write
Data-Carrying Constructors

- Datatype constructors can also carry values

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

- 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:

```plaintext
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
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
What is the type of this expression?

\[(A, \text{"A"})\]

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!

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

• Recursive datatypes can be taken apart by pattern matching (and recursive functions).
Syntax for User-defined Types

define the type of tree

```plaintext
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 []))
```

Example values of type 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

Answer: 3
Have you ever programmed with trees before?

1. yes
2. no
3. not sure
Trees are everywhere
Family trees
Organizational charts
Game trees
Natural-Language Parse Trees

```
S
  NP    VP
    D    N  V    NP
       the  chef cooks D  N
                                the soup
```
CIS120

Filesystem Directory Structure
Binary Trees

A particular form of tree-structured data
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

define the tree type

define the tree

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
Demo

see trees.ml
treeExamples.ml