Programming Languages and Techniques (CIS120)

Lecture 6
Jan 23, 2013

Datatypes and Trees
Announcements

• Homework 2 is up
  – On-time due date: Tuesday, Jan 29th at 11:59:59pm
  – Get started early, and seek assistance if you get stuck!

• Weirich OH today cancelled, email if you would like an appointment (see schedule for TA OH tonight)

• Ask questions on Piazza, but be kind to your TAs.

• Lecture Notes for ch. 1-6 will be posted after class.
Case Study: DNA and 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)

• 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

CIS120 / Spring 2013
Building Datatypes

• Programming languages provide means of creating and manipulating structured data

• We have already seen
  – *primitive datatypes* (int, string, bool, ...)
  – *immutable lists* (int list, string list, string list list, ...)
  – *tuples* (int * int, int * string, ...)
  – *functions* (that define relationships among values)

• How do we build new datatypes from these?
Simple User-defined Datatypes

- OCaml lets programmers define new datatypes

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

```ocaml
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

- Datatypes 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

- 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.
• OCaml also lets us name types, like this:

```ocaml
type helix = nucleotide list

type codon = nucleotide * nucleotide * nucleotide
```

• i.e. a codon is just a triple of nucleotides
• Its scope is the rest of the program.
Datatypes Can Also Carry Data

- 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:

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

• Patterns *bind* variables (e.g. ‘n’) just like lists and tuples
Recursive User-defined Datatypes

• Datatypes can mention themselves!
  – There should be at least one non-recursive ‘base case’
    • Otherwise, how would you build a value for such a datatype?

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

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

CIS120 / Spring 2013
Syntax for User-defined Types

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

- Example values of type `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 []))
```

Constructors (note the capitalization)
Trees are everywhere
Family trees
Organizational charts
Game trees
Expression trees
Natural Language Parse Trees

```
S
   /\   /
  NP   VP
 |     |
D N   V NP
 /   /  /
the chef cooks D N
        /   /
        the soup
```

CIS120 / Spring 2013
File System Directory Structure

- classes
  - cis110
    - 12fa
      - trunk
    - 12su
  - cis120
    - 11fa
    - 11sp
    - 12fa
      - doc
      - exams
  - hw
    - assert.ml
    - assert.mli
    - CommonExportMakefile
    - CommonJavaMakefile
    - CommonMakefile
    - CommonOcamlMakefile
  - hw01
    - export
    - export_html
    - html
    - Makefile
    - notes-from-11fa.txt
    - notes-from-11sp.txt
Domain Name Hierarchy

edu
  - cornell ... upenn
    - cis 
    - seas 
    - wharton ...

com
  - cisco...yahoo

gov
  - nasa ... nsf

mil
  - arpa ... navy ...

org

net
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

```
    0
   / \
  8   -1
 / \   / \
1   3   1  
   /   /   / \
  /   /   /   7
 /   /   /   /
/   /   /   /
```

CIS120 / Spring 2013
Integer Binary Trees in OCaml

```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)))
```

CIS120 / Spring 2013
Demo

see trees.ml