Programming Languages and Techniques (CIS120e)

Lecture 4
Sep 15, 2010

Lists II, Datatypes, Trees
Announcements

• Homework 1 is due today
  – on-time submission ends at 11:59:59pm

• Homework 2 will be available on the web pages later today.
  – Basic concepts will be covered in lecture today and in lab tomorrow.
  – Get started early, and seek assistance if you get stuck!
Recap: Design Pattern for List 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...
     • 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?
     • Is there a direct solution for the empty list?
Demo: Lists
Formalizing the Interface

• Real-world data often has a lot of *structure* to it.
  
  telephone book ~ table of (name, phone number) pairs
  social network ~ nodes (people) and links ("friends")
  blueprint ~ points and lines in a 2D plane

• One crucial step of software design is representing data so that this structure can be exploited.
  – i.e. understanding the problem’s *abstractions*

• Often, the structure of the data suggests useful software design patterns
  – e.g. processing immutable lists by using recursion
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?
DNA Computing Abstractions

• **Nucleotide:**
  – Adenine (A), Guanine (G), Thymine (T), or Cytosine (C)

• **Codon**
  – triple of 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 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, ... )
  – *functions* (that define relationships among values)

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

• OCaml lets programmers define new datatypes

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

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

```ocaml
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 therefore lead to many bugs.

• All modern languages (Java, C#, C++, OCaml,...) provide user-defined datatypes for this reason.
Tuples

• A tuple is a way of grouping together two or more data values (of possibly different types).

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

```ocaml
let my_pair = (3, true)
let my_triple = ("Hello", 5, false)
let my_codon = (A,G,T)
```

• Tuple types are written using ‘*’
  – e.g. my_triple has type:

```
string * int * bool
```
Pattern Matching Tuples

• Tuples can also be taken apart by pattern matching:

```ocaml
let first (x: string * int) : string =
    begin match x with
        | (n, _) -> n
    end

;; assert_eq "" (first ("barbie", 10)) "barbie"
```

• Note how, as with lists and datatypes, the pattern follows the syntax for the corresponding values
Type Abbreviations

- 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 datatypes:

```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’)
- Patterns can *nest* (not shown)
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 my_string_list =
  | Nil
  | Cons of string * my_string_list
```

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