Programming Languages and Techniques (CIS120)

Lecture 5

Jan 23, 2012

Tuples, Datatypes and Binary Trees

Announcements

- Homework 1 due at midnight tonight.
- Homework 2 will soon be up on the web pages.
  - On-time due date: Monday, Jan 30th at 11:59:59pm
  - Get started early, and seek assistance if you get stuck!
- My office hours canceled this week.

Tuples and Patterns

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:

```
let my_pair = (3, true)
let my_triple = ("Hello", 5, false)
let my_quaduple = (1,2,"three",false)
```

- 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:

```
let first (x: string * int) : string =
  begin match x with
  | (left, right) -> left
  end

first ("b", 10)
⇒
"b"
```

- Note how, as with lists, the pattern follows the syntax for the corresponding values

Mixing Tuples and Lists

- Tuples and lists can mix freely:

```
[(1,"a"); (2,"b"); (3,"c")]
            : (int * string) list
```

```
([1;2;3], ["a"; "b"; "c"])
            : (int list) * (string list)
```


Nested Patterns

- So far, we've seen simple patterns:
  - `[]`
  - `x::tl`
  - `(a,b,c)`
- Like expressions, patterns can *nest*:
  - `x::[]` *matches lists of length 1*
  - `x::(y::tl)` *matches lists of length at least 2*
  - `(x::xs, y::ys)` *matches pairs of non-empty lists*
- A useful pattern is the wildcard pattern: _
  - `_::tl` *matches a non-empty list, but only names tail*
  - `(_,x)` *matches a pair, but only names the 2nd part*

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
```

Exhaustive Matches

- Case analysis is *exhaustive* if every value being matched against can fit some branch's pattern.
- Example of a *non-exhaustive* match:

```
let sum_two (l : int list) : int =
  begin match l with
  | x::y::_ -> x+y
  end
```

- OCaml will give you a warning and show an example of what isn't covered by your cases.
  - in this example, there is no case for [], or for a singleton list
- The wildcard pattern and failwith are useful tools for ensuring match coverage.

Unused Branches

- The branches in a match expression are considered in order from top to bottom.
- If you have "redundant" matches, then some later branches might not be reachable.
  - OCaml will give you a warning

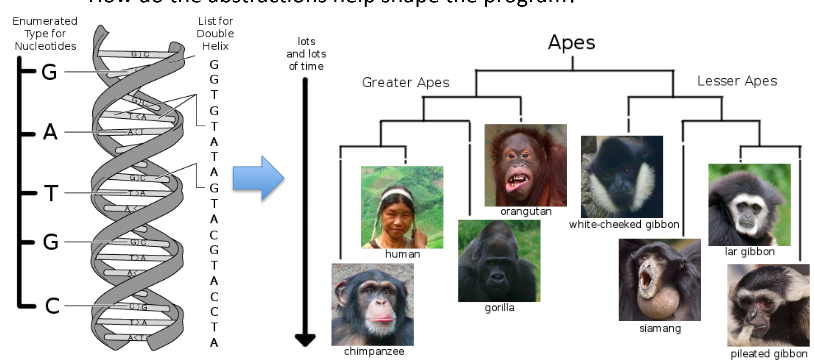
```
let bad_cases (l : int list) : int =
  begin match l with
  | [] -> 0
  | x::_ -> x
  | x::y::tl -> x + y     (* unreachable *)
  end
```

This case matches more lists than that one does.

Datatypes and Trees

Case Study: DNA and Evolutionary Trees

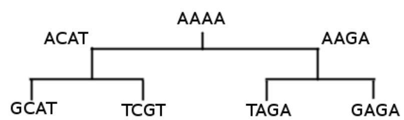
- Problem: reconstruct evolutionary trees from biological data.
  - What are the relevant abstractions?
  - How can we use the language features to define them?
  - How do the abstractions help shape the program?



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



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

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

‘type’ keyword

type name (must be lowercase)

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

- Datatypes 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
- 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, like this:

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

type keyword

type name

definition in terms of existing types

- 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

```
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
```

- Patterns *bind* variables (e.g. 'n') just like lists

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?

```
type my_string_list =
  | Nil
  | Cons of string * my_string_list
```

base case (nonrecursive)

Cons 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 my_string_list =
  | Nil
  | Cons of string * my_string_list
```

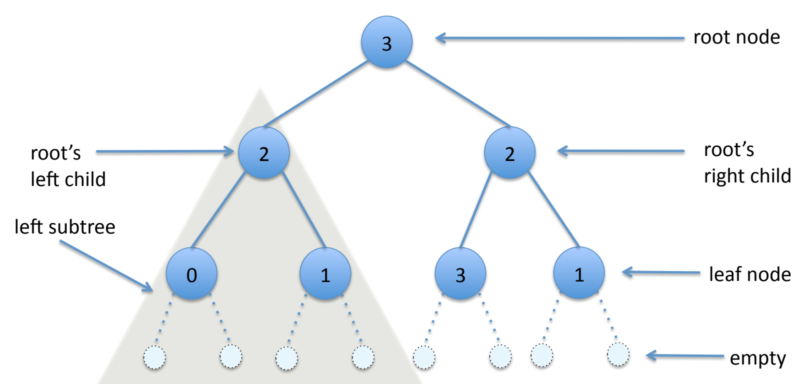
- Example values of type `my_string_list`

```
Nil
Cons(“hello”, Nil)
Cons(“a”, Cons(“b”, Cons(“c”, Nil)))
```

Constructors (note the capitalization)

Binary Trees

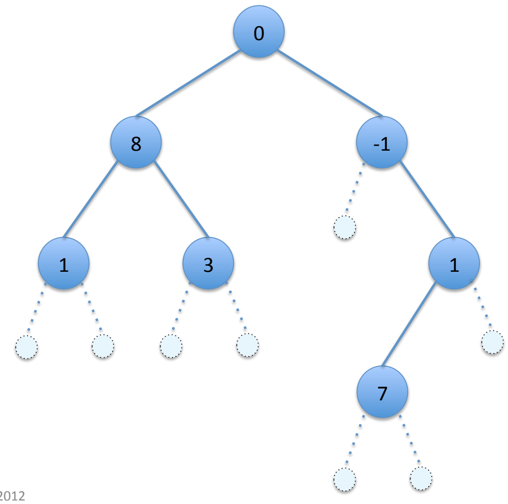
Binary Trees



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.

Another Example Tree



Basic Tree Concepts

- *Size*: the total number of nodes in the trees
- *Height*: the length of the longest path from the root to a leaf
- *Traversal*: A pattern of visiting the nodes of the tree.
  - In order: left-child, node, right child
  - Pre order: node, left-child, right child
  - Post order: left-child, right child, node
  - Level order: in order of distance from the root

Demo: Binary Trees