Class 23: tons of fun!

Currying
Records
Quiz on types
Equality testing
A little big-O
Trees
Announcement

• Tuesday office hours, normally 1-2 PM, will be 1 - 1:40 tomorrow.
Currying

• A transformation on functions
  • Named for Haskell Curry, a logician
• Suppose we have two functions:
  let f1 : (int, int) => int = (x, y) => x + y + 1;
  let f2: int => (int => int) = x => (y => x + y + 1);

• What's $f1(2, 4)$?
• What's $f2(2)$?
• What's $f2(2)(4)$?
• What's $f1(2)$?
  • Surprise! It's exactly the same as $f2(2)$!
This correspondence is called "Currying".

Some languages go nuts with this; we mostly won't.
  - Automatic currying in Ocaml used to cause lots of problems for CS17 students

When we have a two (or more) argument function, we can "curry" on the first argument.

To curry on the second argument of \( f(x, y, z, w) \)...

```ocaml
let g = (y, x, z, w) => f(x, y, z, w);
g(2) ...
```
A new bit of ReasonML syntax: records

• Lists: expandable ordered collections of data (all the same type)
• Tuples: aggregates of multiple pieces of data (possibly different types)
• Records: like tuples, but with names for the pieces.
• Remember the name-and-nickname database?

ReasonML version 1:

```plaintext
type entry = (string, string);
type database = list(entry)
```
A new bit of ReasonML syntax: records

- Remember the name-and-nickname database?

ReasonML version 2:

```reasonml
type entry = {
    name: string,
    nickname: string
};
type database = list(entry)
let pp = {name: "Porcellus Smith", nickname: "Porky Pig"};
```
Record use

• When a datatype has multiple components, naming them can lead to clarity

```typescript
type ifExprData = {
  test: expression,
  trueResult: expression,
  falseResult: expression
};
```
Example pattern-matching with records

type s = {a:int, b:bool};
let f: s => bool = fun
  | {a:_, b:true} => false
  | {a:_, b:false} => true;
Quiz on types

- Write down the types for each of the following:
  a) (1, 2.5)
  b) let f = fun
      | "abc" => true
      | _ => false;
  c) let g = fun
      | [] => true
      | [[_]] => false
      | _ => true;
Equality testing

• We can test whether numbers or bools or strings or lists of these things are equal

• What about equality of functions?
  • Provably hard! (take CS1010)
  • Racket solve this by saying that "equal?" behavior on function types is undefined. (I think it always returns "false" in practice.)

• For Rackette: you only need to handle non-function types.
A last visit to some big-O stuff
Bootstrapping lemma
Applying bootstrapping
Almost done with ReasonML syntax

• Remaining item: modules, later this week.
• What else is left?
  • All of computer science!
• Particular things we'll look at
  • Stacks and Queues
  • Sorting
  • Trees
    • Binary Search Trees
    • Game Trees
    • Minimax algorithm
Binary trees

- A binary tree is something that looks like this:
Binary tree terms

• "Leaf": one of the filled-in dots at the bottom
• "Node": one of the circular things with a number in it
• The thing in a node could be a number, a string, a list...
  • but all nodes in a tree will contain the same type of value
• Type definition in ReasonML:
  type tree('a) = Leaf | Node('a, tree('a), tree('a))
Binary trees

let s:tree(int) = Node(4, Node(6, Leaf, Node(5, Leaf, Leaf), Node(2, Leaf, Leaf)));
Terminology

- *Depth* of a tree: longest sequence of edges.
  - *Example has depth 3*
  - *Also called height*
- *Root* node: the one at the top
- *Depth* of a node: number of edges to get to the root. Root is at depth 0.
- Typical tree-drawing: omit leaves!
Tree depths
Tree depths (cont)
What makes trees interesting

• Lists attach a single item to something (possibly) more complicated: (cons 1 (list 3 4))
• Trees attach a single item to two things, each possibly more complicated.
• Shift from 1 to 2 introduces complexity and (in some cases) efficiency!
treeContains17

• Recall typedefinition:
  type tree('a) = Leaf | Node('a, tree('a), tree('a))

• Write a procedure:
  let treeContains17 : tree(int) => bool = fun
  | Leaf => ...
  | Node(val, left, right) => ...