Announcements:

- PS1 due Today 11:59PM
  - Solutions in class tomorrow
  - HW back in section Monday
- Quiz #1 on Thursday, first 10 minutes of class
  - Coverage includes today's material but not tomorrow's
  - Also returned Monday
- RDZ office hours: Tuesday after class, but today 4-5, in 4158 Upson
  - Best problem set resource: course staff
  - Best course/exam resource: Yours Truly

- Need to write the simplest solution to a problem
  Important in real life
  - Code that works is simply not good enough
  - “Programs are designed primarily to be read by other humans”
- Important in CS3110
  - Full credit reserved for really the right answer

- Examples:

```
let rec fact(z) =
 if z = 1
  then 1
 else if z = 2
  then 2
 else
  z*fact(z-1)

let rec inclist (lst: int list) =
  match lst
  with
 | [] -> []
 | [h] -> [h+1]
 | h::t -> h+1::inclist(t)
```

- For CS3110 this is a particularly important lesson because we are going to PROVE code is correct
  - Recall that in ML, as opposed to imperative languages, a program “feels” much more like a mathematical definition

http://en.wikipedia.org/wiki/International_Obfuscated_C_Code_Contest

```
#define _ -F<00||--F-OO--;
int F=00,OO=00;main(){F_OO();printf("%1.3f\n",4.*-F/OO/OO);}F_OO()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_
            _-_-_-_
}
```

- The main tool used for proofs in CS is mathematical induction
- Today we will do it, briefly, for mathematical formulae
- We will use it for programs in a week or so

- Induction recipe (one of the very few things you should memorize in CS3110):
    - Example: 1+2+…n = n(n+1)/2
    - 1. Statement to be proven
        - For any natural number n, the sum from 1 to n is n(n+1)/2
    - 2. Variable we are doing induction on: n
        - Easy in this case, not always so trivial
    - Call this P[n]. Note that it is a sentence about the integer n
        - Not an Ocaml function!
    - 3. Prove base case, typically P[1] or P[0]
    - 4. Prove that (for any n) (P[n] => P[n+1])
        - Pick an n, assume P[n] is true (I.H.), prove P[n+1] follows
        - Not the same as prove that (for any n)P[n] => P[n+1]

- Currying and higher order functions
- Suppose we want to compute x + sqrt(y)

```
let try(x,y) = x +. sqrt(y)
```

  - This is short for

```
let try z = match z with (x,y) -> x +. sqrt(y)
```

  - Type is (float * float) -> float
- Alternate form, "Currying", named after logician (not food!)
- Type will be float->float->float
  - What the heck is this?
  - Compare float->float, like sqrt
    - Function that takes a float, returns a float
    - Let's call this a "floatfun", just to give it a name (slang)
  - Such things are FIRST CLASS OBJECTS
  - Higher order procedures!
- Now we are talking about a function that **returns** a floatfun given a float
  - How to build such a thing in OCaml?

```
let tryc x y = x +. sqrt(y)
```

- This is syntactic sugar for

```
let tryc = fun(x) -> fun(y) -> x +. sqrt(y)
```

  - Which is harder to read
- What is the advantage? Let's get back to this in a second.

- Simpler example:

```
let plus x y = x + y
```

or with all the types written explicitly:

```
let plus (x : int) (y : int) : int = x + y
```

Notice that there is no comma between the parameters. Similarly, when applying a curried function, we write no comma:

```
plus 2 3 = 2 + 3 = 5
```

The curried declaration above is syntactic sugar for the creation of a **higher-order function**. It stands for:

```
let plus = fun (x : int) -> fun (y : int) -> x + y
```

Evaluation of `plus 2 3` proceeds as follows:

```
plus 2 3
= ((fun (x : int) -> fun (y : int) -> x + y) 2) 3
= (fun (y : int) -> 2 + y) 3
= 2 + 3
= 5
```

So `plus` is really a function that takes in an `int` as an argument, and returns a new function of type `int -> int`. Therefore, the type of `plus` is `int -> (int -> int)`. We can write this simply as `int -> int -> int` because the type operator `->` is right-associative.

It turns out that we can view binary operators like `+` as functions, and they are curried just like `plus`:

```
# (+);;
- : int -> int -> int = <fun>
# (+) 2 3;;
- : int = 5
# let next = (+) 1;;
val next : int -> int = <fun>
# next 7;;
- : int = 8;
```

- So, how does this help us?
- plus 2 adds 2 to its arg, but without recomputing 2 (so what?)
- How about plus (slowfun 2)?