Announcements:

- PS1 due Today 11:59PM
  - HW back in section Monday
  - Quiz #1 also back Monday
- Staff responsible for PS updates and grading. RDZ for exams, lectures.
- Comment about writing weird code as an exercise, like athletic exercises.

Quiz answers:

Q1: obvious

Q2, Q3: 11

Q4: (int -> int) -> (int * int)

Note that g f abs = (11, 5) while g f square = (11, 7)
• Need to write the simplest solution to a problem
  Important in real life
  o Code that works is simply not good enough
  o “Programs are designed primarily to be read by other humans”
• Important in CS3110
  o Full credit reserved for really the right answer

• Examples (students do this all the time!)

```ml
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
  o Recall that in ML, as opposed to imperative languages, a program “feels” much more like a mathematical definition

```c
#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):
  o 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]
• Induction canonical failures
  o Not following the recipe (what variable are you doing induction on?)
  o Not using the induction hypothesis
  o Not writing down the induction hypothesis

• Fun “buggy induction” proof: all horses are the same color
• Illustrates weak versus strong induction
  o If you are in doubt you can try the first few cases

• Sneak preview: we are going to use induction to prove program correctness, initially of simple recursive functions like factorial
• In order to do this, we will need a precise definition of the evaluation rules in OCaml: SUBSTITUTION MODEL

• One of the things that the substitution model will help us with is to precisely understand functions that combine lists, anonymous functions, and currying. This combination is very common, and a basic skill for prelim #1. To get you started here are some examples.
• A good OCaml sanity check with match: RHS is all same type, as is LHS. When used as the body of a function, RHS is return type.
let rec listprod (lst:int list) = 
   match lst with 
   | [] -> 1 
   | h::t -> h * listprod(t)

let rec listsum (lst:int list) = 
   match lst with 
   | [] -> 0 
   | h::t -> h + listprod(t)

let rec listop (base: int) (f:int->int) (lst:int list) = 
   match lst with 
   | [] -> base 
   | h::t -> f h (listop base f t)

let listsum2 = listop 0 (+)
let mul x y = x * y 
let listprod2 = listop 1 mul

let rec map (f:int->int) (lst:int list) = 
   match lst with 
   | [] -> [] 
   | h::t -> (f h)::(map f t)

let incr z = z + 1
map incr [3;1;1;0]

let rec filter (f:int->bool) (lst:int list) = 
   match lst with 
   | [] -> [] 
   | h::t -> if (f h) then h::(filter f t) else filter f t

filter (fun z -> z > 2) [3;1;1;0]
let rec makefuns (lst: int list) =
  match lst with
  | [] -> []
  | h::t -> (fun z -> z+h)::makefuns(t)

let mf = makefuns [3;1;1;0]

(* Notes:
   - can generalize the base case 0 and operator + *)

let rec dofunsandadd (funs: (int->int) list) (z:int) =
  match funs with
  | [] -> 1
  | h::t -> h(z) + (dofunsandadd t z)

let ans = dofunsandadd mf 9