• **Announcements:**
  • Section 3, 4 will meet in 211 Upson, MW at 11:15A or 7:30P
  • PS1 due next Tuesday 9/6 11:59PM
    o PS versioning system
  • Office hours are up
  • All problem sets returned in section on Monday
  • Everyone should be in CMS now
  • **Quiz #1 on Thursday 9/8, first 10 minutes of class**

• **Main difference between function and imperative programming:**
  o Imperative programs: statements that do things
    ▪ Formally, C assignments have an LHS and RHS
  o Functional programs: expressions have values
    ▪ A bit like RHS, but closer to math (equational reasoning)
• You can see this even in simple examples like computing the sum of squares through n
  See slides from lecture 1:
int sumsq(int n) {
    y = 0;
    for (x = 1; x <= n; x++) {
        y += x*x;
    }
    return n;
}

let rec sumsq (n:int):int =
   if n=0 then 0
   else n*n + sumsq(n-1)

• What’s the difference? Lots of things
• Mental model for C involves doing things, one at a time
• ML (= SML/OCaml) is more like math: eternal truths
  o Can always substitute equals for equals
  o Example: \( \cos^2 + \sin^2 = 1 \)

• You will hear me say many times that in ML, an expression has a value
• Instead of asking “what does this program print” we ask “what is the value of this expression”
  o Very different question, different way of thinking
What is an expression? There is a simple definition

- Recursive (first of many!)

- identifier x, f (aka variable, name) 
  \[ \text{ex: frob, num} \]

- constant c 
  \[ \text{ex: } 0, \text{"hello"}, 3.14 \]

- binary operator b 
  \[ \text{ex: +, *, .} \]

- unary operator i 
  \[ \text{ex: -, not} \]

- term e 
  \[ x | c | u \circ e | e1 \circ b \circ e2 | \text{if e1 then e2 else e3} | e0(e1, ..., en) | \text{let } \{ \text{rec} \} \circ d \circ \text{in e} | \text{let } \{ \text{rec} \} \circ d1 \circ \text{and } d2 \circ \text{... and } dn \circ \text{in e} \]

- declaration d 
  \[ \text{d} \circ x = e \]

- type t 
  \[ \text{int} | \text{bool} | \text{char} | \text{string} | t1 \circ * \circ t2 \circ \text{... } \circ tn | t1 \circ * \circ t2 \circ \text{... } \circ tn \rightarrow t \]

- Important notes:
  - tuple types: what is the type of (1,2)? (1.0, 2)?
  - function types, plus terms in body

- Writing all the types down is a pain. So ML does type inference

- Example: type of \[ \text{let } f(x, y) = (x = \text{String.length(y)}) \]

- Different kinds of errors
  - Lexical syntax error: 2.0$
  - Grammatical syntax error: let 0 x
  - Run-time error: 2/0
  - Type error: 1 + \text{"a"}, 1 + 2.0
• Huge win of ML: catch errors early!
  o Why is this so important?
  o The finicky ML compiler is very much your friend
  o Once it compiles it tends to run

• Functions are first-class objects (unlike, e.g. Java, C)
• They can be
  o Bound to a variable
  o Passed to a function as an argument
  o Returned as the result of a function
• Related point: not everything needs a name. Consider 1 + (2*3) in any random programming language. What’s the name of that 6?
  o Having to give everything a name is a pain
• You can have anonymous functions via fun
  o Lots of fun in this course...
• This is surprisingly useful!

    let square x = x * x (* is the same as: *)
    let square = fun x -> x * x (* anon function! *)

    (* higher order functions and values *)
    let twice f = fun x -> f (f x)
    let twice f x = f (f x)
    let fourth = twice square
    let fourth = twice (fun x -> x * x)
let z = 3 in z
let z = 3 in z*z

(* parallel binding *)
let z = z +1 and a = z in z*a

(* uncurried *)
let longEnough (str, len) = String.length str >= len

(* curried *)
let longEnough str len = String.length str >= len
(* let rec and embedded lets *)

let isPrime (n : int) : bool = (* Returns true if n has no divisors between m and sqrt(n) inclusive. *)
  let rec noDivisors (m : int) : bool =
    m * m > n || (n mod m != 0 && noDivisors (m + 1))
  in
  n >= 2 && noDivisors 2

(* Computes the square root of x using Heron of Alexandria's algorithm (circa 100 AD). We start with an initial (poor) approximate answer that the square root is 1.0 and then continue improving the guess until we're within delta of the real answer. The improvement is achieved by averaging the current guess with x/guess. The answer is accurate to within delta = 0.0001. *)

let squareRoot (x : float) : float = (* numerical accuracy *)
  let delta = 0.0001
  in

  (* returns true iff the guess is good enough *)
  let goodEnough (guess : float) : bool =
    abs_float (guess *. guess -. x) < delta
  in

  (* return a better guess by averaging it with x/guess *)
  let improve (guess : float) : float =
    (guess +. x /. guess) /. 2.0
  in

  (* Return the square root of x, starting from an initial guess. *)
  let rec tryGuess (guess : float) : float =
    if goodEnough guess then guess
    else tryGuess (improve guess)
  in

  (* start with a guess of 1.0 *)
  tryGuess 1.0