Announcements:

- PS2 due Thursday 9/22 11:59PM
- Quiz #2 Thursday 9/22, first ten minutes
    - Coverage through today's lecture

- Inductive correctness proofs

- Code correctness approaches:
    - Hey, it worked on my example!
        - More or less the state of the art
        - MS has very large test suites, lots of automatic tools
        - Lots of run-time failure checks, i.e. assert

    - Convince your partner ("pair programming")

    - Use a model of how the code works!

- We will use the term **verification** to refer to a process that generates high assurance that code works on all inputs and in all environments.

- Testing is a good, cost-effective way of getting assurance
    - but not a verification process in this sense
    - there is no guarantee that the coverage of the tests is sufficient for all uses of the code
    - Obscure paths through code can be catastrophic

- Verification generates a proof that all inputs will result in outputs that conform to the specification.
    - Sometimes only implicit proof
    - Relative to the spec, but specs are easier to debug than code!

- In this lecture, we look at verification based on explicitly but informally proving correctness of the code.

- Verification tends to be expensive and to require thinking carefully about and deeply understanding the code to be verified.
    - In practice, it tends to be applied to code that is important and relatively short.
    - Verification is particularly valuable for critical systems where testing is less effective.
    - Critical systems are quite interesting (space shuttle destruct switch is a nice example)

- Many programs are concurrent (you will write some later this semester).
    - Because their execution is not deterministic, concurrent programs are hard to test
    - Sometimes subtle bugs can only be found by attempting to verify the code formally.

- In fact, tools to help prove programs correct have been getting increasingly effective and some large systems have been fully verified,
    - including compilers, processors and processor emulators, and key pieces of operating systems.

- Another benefit to studying verification is that when you understand what it takes to prove code correct, it will help you reason about your own code (or others')
    - Will write code that is correct more often, based on specs that are more precise and useful.

- Outline of a proof of program correctness,
  - To guest star in P1 and final exam
- Key tools:
  - Induction
  - Substitution model

- Recall our induction recipe
  - Statement to be proved: sum_i=1^n = n(n+1)/2
  - Variable: n
  - Proof of base case, here P[1]
  - Prove P[n] implies P[n+1] for any n
    - Pick an arbitrary n, assume P[n] (I.H.), show P[n+1]
- Think of dominos labeled “P[1]”, “P[2]”, etc. knocking each other over

- Now suppose we want to prove something simple like our factorial program is correct
  - More complex example coming shortly, maybe in section

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

- What is our statement?

-

Example : proof of an inductive sort

We want to prove the correctness of the following insertion sort algorithm. The sorting uses a function `insert` that inserts one element into a sorted list, and a helper function `isort'` that merges an unsorted list into a sorted one, by inserting one element at a time into the sorted part. Functions `insert` and `isort'` are both recursive.

```
let rec insert(e, l): int list =
  match l with
     [] -> [e]
  | x::xs -> if e < x then e::l else
x::(insert(e,xs))

let rec isort' (l1, l2): int list =
  match l1 with
     [] -> l2
  | x::xs -> isort'(xs, insert(x, l2))

let isort(l: int list): int list =
  isort'(l, [])
```

We will prove that `isort` works correctly for lists of arbitrary size. The proof consists of three steps: first prove that `insert` is correct, then prove that `isort'` is correct, and finally prove that `isort` is correct. Each step relies on the result from the previous step. The first two steps require proofs by induction (because the functions in question are recursive). The last step is straightforward.

Correctness proof for insert

We want to prove that for any element e and any list l: 1) the resulting list insert(e, l) is sorted; and 2) that the resulting list insert(e,l) contains all of the elements of l, plus element e.

The notion that a list l is sorted, written sorted(l), is defined as usual: sorted(l) is true if l has at most one element; and sorted(x::xs) holds if x <= e for all elements e being members of xs, and sorted(xs) holds.

The proof is by induction on length of list l. The proof follows the usual format of a proof by induction, i.e., specifying what property we want to prove, what we are inducting on, showing the base case, the inductive step, and clearly specifying when we apply the induction hypothesis (carefully indicating why we can apply it, and showing the lists to which it is applied!).

- **Property to prove**:
  P(n) = for any list l and element e:
  ```
  if      sorted(l) and
          length(l) = n
  then    sorted(insert(e,l)) and
          elements(insert(e,l)) = elements(l) U {e}
  ```

  We want to prove that P(n) holds for all n >= 0.

  Note: elements(l) represents the multiset containing the elements of l. A multiset is a set where duplicates are allowed in the set. Think of a set where each element is annotated with the number of occurences (>= 1).
  The union operation will increase the cardinality of elements in both sets, e.g., {1,2} U {1} = {1,1,2}.

- **Base case**: n = 0. We want to prove that P(0) holds. Let a list l such that sorted(l) and length(l) = 0.
  The only list with length zero is nil, so l= nil. Therefore, insert(e,l) evaluates as follows:

  ```
  insert(e, [])

  -> (evaluation of function application)
  match [] with
    [] -> [e]
  | x::xs -> ...

  -> (pattern matching)
  [e]
  ```

  List [e] has one element, so it is sorted by definition. Hence, insert(e, []) is sorted. Furthermore, elements(insert(e, [])) = elements([e]) = {e} = {e} U Â = {e} U elements([]). Therefore, the base case holds.

- **Inductive step**. Assume that P(n) holds. That is, for any element e and any sorted list of length n, insert(e,l) is sorted and contains all of the elements of l, plus e. This is the induction hypothesis IH.

  We want to prove that P(n+1) also holds. That is, we want to prove that for any e, and any sorted l of length n+1, insert(e,l) is also sorted and contains all elements of l, plus e.

  Let e be an arbitrary element, and l a sorted list of length n+1. Therefore, l=h::t, where h is less than all elements in t and t is a sorted list of length n. Also, elements(l) = elements{t} U {h}

  Thanks to the evaluation model by substitution, we have a formal way of describing the execution of insert. According to that model, the evaluation of insert(e,l) proceeds as follows:

  ```
  insert(e, l)

  -> (function evaluation and replacing l with h::t)
  match h::t with
    [] -> [e]
  | x::xs -> if e < x then n::l else x::(insert(e,xs))

  -> (pattern matching)
  if e < h then e::l else h::(insert(e,t))
  ```

  Now we have two possible results depending on the value of the expression "e < h".

  Case 1: If e < h is true, then insert(e,l) = e::l. We have the following: (a) Since h is less than all elements in t,
  and e < h, it means that e is less than all elements in h::t = l. (b) We also know that l is sorted. Together, (a) and (b) imply that e::l is sorted. Therefore, insert(e,l) is sorted.

  Also, elements(insert(e,l)) = elements(e::l) = elements(l) U {e}. So P(n+1) holds in this case.

  Case 2. If h <=e, then insert(e,l) = h::(insert(e,t)). Let l' = insert(e,t).

  Because t is a sorted list of length n, it means that we can **apply the induction hypothesis**. By the IH for element e and list t, the list l'= insert(e,t) is sorted, and elements(l') = elements(t) U {e}.

  Since h::t is sorted, h is less than any element in elements(t). Also, h <= e. Therefore h is less than all elements in l'. Along with the fact that l' is sorted, this means that insert(e,l) = h::l' is sorted.

  Finally, elements(insert(e,l)) = elements(h::insert(e,t)) = elements(h::l') = {h} U elements(l') = {h} U {e} U elements(t) = {e} U {h} U elements(t) = {e} U elements(l). Therefore, P(n+1) holds in this case.

  Since the conclusion of P(n+1) holds for all branches of evaluation, we have proved the inductive step.

We can therefore conclude that P(n) holds for all n >= 0.

Proving correctness of isort'

```
let isort' (l1: int list, l2:int list) =
  match l1 with
    [] -> l2
  | x::xs -> isort'(xs, insert(x, l2))
```

We will now prove by induction on the length of l1 that isort' works correctly, following the same proof format.

- **Property to prove.** We will prove the following property:

  P(n) = for any lists l1 and l2:
  if l1 has length n and
  l2 is sorted
  then:
  isort'(l1, l2) is sorted and
  elements(isort'(l1, l2)) = elements(l1) U elements(l2).

  We prove that P(n) holds for all n >=0 by induction on n (the length of the first list).

- **Base case:** n = 0. Let l1 and l2 such that length(l1) = 0 and l2 is sorted. This means that l1 = nil. According to the substitution model, the evaluation of isort' works as follows:

  ```
  isort' ([], l2)

  -> (evaluation of function application)
  match [] with
    [] -> l2
  | x::xs -> isort'(xs, insert(x, l2))

  -> (pattern matching)
  l2
  ```

  The list l2 is a sorted list by assumption, and also contains all the elements of l2 and [], since the latter is the empty set. Therefore P(0) holds, and we have proven the base case.

- **Induction step.** Assume that P(n) holds. That is, for any l1, l2, such that length(l1) = n and sorted(l2), we have isort'(l1, l2) = l, where l is sorted and contains all the elements of l1 and l2.

  We now want to prove that P(n+1) holds. Let l1 and l2 be two lists such that length(l1) = n+1 and sorted(l2). Hence, l1 = h::t, where t is a list of length n. The evaluation of isort'(l1,l2) proceeds as follows:

```
isort'(l1, l2)

-> (evaluation and replacing l1 with h::t)
match h::t with
  [] -> l2
| x::xs -> isort'(xs, insert(x, l2))

-> (pattern matching)
isort'(t, insert(h, l2))
```

Let l' = insert(h,l2). We know that l2 is sorted. Hence, from the correctness proof for insert, we know that l' is sorted, and elements(l') = elements(l2) U {h}.

We know that length(t) = n. We also know that t and l' are sorted. Therefore, we can apply the induction hypothesis (IH). By applying IH to lists t and l', we get that isort'(t,l') is sorted, and elements(isort'(t,l')) = elements(t) U elements(l').

But elements(l') = elements(l2) U {h}, so elements(isort'(t,l')) = elements(t) U elements(l2) U {h} = elements(l1) U elements(l2). Hence the resulting list l'' = isort'(t,l') is sorted and contains all elements of l1 and l2. This shows that P(n+1) holds.

We conclude that P(n) holds for all n >= 0.

Proving the correctness of isort

It is now straightforward to prove that isort works correctly. We want to show that for any list l, isort(l) is sorted and
elements(isort(l)) = elements(l).

The evaluation of isort(l) is:

```
isort(l)

(by substitution)
-> isort'(l, [])
```

Let l' be the result of evaluating isort'(l, []). From the correctness proof for isort', we know that l' is sorted and elements(l') = elements(l) U elements([]) = elements(l) U Â = elements(l). Hence the final result is sorted and contains all elements of l, Q.E.D..

Modular verification

In our proof that `insert` met its spec, we assumed that the implementation of > met *its* spec. This was good because we didn't have to look at the code of >. This was an example of **modular verification**, in which we were able to verify one small unit of code at a time. Function specifications, abstraction functions, and rep invariants make it possible to verify modules one function at a time, without looking at the code of other functions. If modules have no cyclic dependencies, which OCaml enforces, then the proof of each module can be constructed assuming that every other module satisfies the specs in its interface. The proof of the entire software system is then built from the bottom up.