Announcements:

- What are the following numbers: 52/37/19/6 (2:30,3:35,11:15,7:30)
- PS2 due Thursday 9/22 11:59PM
- Quiz #1 back in section Monday
- Quiz #2 at start of class on Thursday 9/22
    - HOP's, and lots of them!
- Guest lectures on Tuesday 9/27 by Prof. Walker White
    - No RDZ office hours, I am on travel

- Substitution model
- What is the value of the following expression?

```
let rec evil(f1, f2, n) =
  let f(x) = 10 + n in
    if n = 1 then f(0) + f1(0) + f2(0)
    else evil(f, f1, n-1)
and dummy(x) = 1000
in
  evil(dummy, dummy, 3)
```

- Some things are obvious
    - We can see that the function evil calls itself recursively, and the result of the function is the result when it is called with n=1.
    - But what are the values returned by the applications of functions f, f1 and f2?

- To figure this out we need a more precise understanding of how ML works

- Evaluation

- The OCaml prompt lets you type either a term or a declaration that binds a variable to a term.

- It evaluates the term to produce a **value**: a term that does not need any further evaluation.
    - Values include not only constants, but tuples of values, variant constructors applied to values, and functions.

- Running an ML program is just *evaluating a term*.
- What happens when we evaluate a term? In an imperative (non-functional) language like Java, we sometimes imagine that there is an idea of a "current statement" that is executing.

- This isn't a very good model for ML; it is better to think of ML programs as being evaluated in the same way that you would evaluate a mathematical expression.
    - For example, if you see an expression like (1+2)*4, you know that you first evaluate the subexpression 1+2, getting a new expression 3*4.
    - Then you evaluate 3*4. ML evaluation works the same way.

- As each point in time, the ML evaluator rewrites the program expression to another expression.
    - Assuming that evaluation terminates, eventually the whole expression is a value and then evaluation stops: the program is done.
    - Or maybe the expression never reduces to a value, in which case you have an infinite loop.

- Rewriting works by performing simple steps called reductions. In the arithmetic example above, the rewrite is performed by doing the reduction 1+2 → 3 within the larger expression, replacing the occurrence of the subexpression 1+2 with the right-hand side of the reduction, 3, therefore rewriting (1+2)*4 to 3*4.

- The next question is which reduction OCaml does. Fortunately, there is a simple rule.

- Evaluation works by always performing the leftmost reduction that is allowed.

    - So we can describe evaluation precisely by simply describing all the allowed reductions.

- OCaml has a bunch of built-in reduction rules that go well beyond simple arithmetic. For example, consider the if expression. It has two important reduction rules:

    `if true then` e_1 `else` e_2 → e_1
    `if false then` e_1 `else` e_2 → e_2

- If the evaluator runs into an if expression, the first thing it does is try to reduce the conditional expression to either true or false. Then it can apply one of the two rules here.

- For example, consider the term

```
if 2=3 then "hello" else "good" ^ "bye"
```

- This term evaluates as follows:

```
if 2=3 then "hello" else "good" ^ "bye"
→ if false then "hello" else "good" ^ "bye"
→ "good" ^ "bye"
→ "goodbye"
```

- Notice that the term "good"^"bye" isn't evaluated to produce the string value "goodbye" until the If term is removed. This is because if is lazy about evaluating its then and else clauses. If it weren't lazy, it wouldn't work very well.

- Evaluating a let term

- The rewrite rule for the `let` expression introduces a new issue: how to deal with the bound variable.
    - In the *substitution* model, the bound variable is replaced with the value that it is bound to.

- Evaluation of the `let` works by first evaluating all of the bindings. Then those bindings are substituted into the body of the `let` expression. For example, here is an evaluation using `let` :

```
let x = 1+4 in x*3  →
let x = 5 in x*3  →
5*3  →
15
```

- Notice that the variable `x` is only substituted once there is a value (5) to substitute.
    - That is, OCaml **eagerly** evaluates the binding for the variable.
- Most languages (e.g., Java) work this way.
    - However, in a lazy language like Haskell, the term `1+4` would be substituted for x instead.
    - This could make a difference if evaluating the term could create an exception, side effect, or an infinite loop.
    - We will play with lazy evaluation later in CS3110

- Therefore, we can write the rule for rewriting `let` roughly as follows:

`let x = v in e` → `e` (with occurrences of *x* replaced by *v*)

- Remember that we use *e* to stand for an arbitrary expression (term), *x* to stand for an arbitrary identifier. We use *v* to stand for a value—that is, a fully evaluated term. By writing *v* in the rule, we indicate that the rewriting rule for `let` cannot be used until the term bound to *x* is fully evaluated. Values can be constants, applications of datatype constructors or tuples to other values, or anonymous function expressions. In fact, we can write a grammar for values:

v ::= *c* | *X*(*v*) | (*v*1,...,*v*n) | `fun` *p* `->` *e*

- Substitution

- When we wrote “with occurrences of x replaced by v”, above, we missed an important but subtle issue. The term *e* may contain occurrences of *x* whose binding occurrence is not this binding *x* = *v*.

- It doesn't make sense to substitute *v* for these occurrences. For example, consider evaluation of the expression:

```
let x:int = 1 in
  let f(x) = x in
    let y = x+1 in
      fun(a:string) -> x*2
```

- The next step of evaluation replaces only the magenta occurrences of x with 1, because these occurrences have the first declaration as their binding occurrence. Notice that the two occurrences of x inside the function f, which are respectively a binding and a bound occurrence, are not replaced. Thus, the result of this rewriting step is:

```
let f(x) = x in
  let y = 1+1 in
    fun(a:string) -> 1*2
```

- Let's write the **substitution** *e*{*v*/*x*} to mean the expression *e* with all *unbound* occurrences of *x* replaced by the value *v*. Then we can restate the rule for evaluating `let` more simply:

`let` *x* = *v* `in` *e* → *e*{*v*/*x*}

- This works because any occurrence of *x* in *e* must be bound by exactly this declaration `let x = v`. Here are some examples of substitution:

```
x{2/x}  =  2
x{2/y}  =  x
(fun y->x) {"hi"/x}  =  (fun y ->"hi")
f(x) { fun y->y / f } =  (fun y->y)(x)
```

- One of the features that makes ML unusual is the ability to write complex patterns containing binding occurrences of variables. Pattern matching in ML causes these variables to be bound in such a way that the pattern matches the supplied value. This is can be a very concise and convenient way of binding variables. We can generalize the notation used above by writing *e*{*v*/*p*} to mean the expression *e* with all *unbound* occurrences of variables appearing in the pattern *p* replaced by the values obtained when *p* is matched against *v*.

- What if a let expression introduces multiple declarations? In this case we must substitute for all the bound variables simultaneously, once their bindings have all been evaluated.