

Announcements:

- What are the following numbers: 74/2/70/17 (2:30,2:30,3:35,7:30)
 - PS2 due Thursday 9/20 11:59PM
 - Guest lecture on Tuesday 9/25
 - No RDZ office hours next Friday, I am on travel
 - A brief comment about academic integrity
 - You must write your own code. Only exception is PS5,PS6 when you work in pairs.
 - We'll talk about this in more detail next lecture.
 - Recall: rules for simplifying fractions ($a/1$, $0/b$, $a/b + c/d$, etc.)
 - Rewrite rules that get you to a base case
 - Can solve very complex expressions
-

- 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 $1+2 \rightarrow 3$ 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:

$$\begin{array}{ll} \text{if true then } e_1 \text{ else } e_2 & \rightarrow e_1 \\ \text{if false then } e_1 \text{ else } e_2 & \rightarrow e_2 \end{array}$$

- 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.
- Also note that replacing one of the values by, say, a number doesn't work.

- 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` \rightarrow `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:

$$v ::= c \quad | \quad x(v) \quad | \quad (v_1, \dots, v_n) \quad | \quad \text{fun } p \rightarrow e$$

- **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` \rightarrow $e\{v/x\}$

- 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:

$$\begin{aligned} x\{2/x\} &= 2 \\ x\{2/y\} &= x \\ (\text{fun } y \rightarrow x) \{ \text{"hi"}/x \} &= (\text{fun } y \rightarrow \text{"hi"}) \\ f(x) \{ \text{fun } y \rightarrow y / f \} &= (\text{fun } y \rightarrow y)(x) \end{aligned}$$

- 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 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.

- Evaluating functions

- Function applications are the most interesting case. When a function is applied, OCaml does a similar substitution: it substitutes the values passed as arguments into the body of the function. Suppose we define a function `abs` as follows:

```
let abs (r:float): float =
  if r < 0.0 then -. r else r
```

- We would like the evaluation of `abs (2.0+.1.0)` to proceed roughly as follows:

```
abs(2.0+.1.0) →
abs(3.0) →
if 3.0 < 0.0 then ~3.0 else 3.0 →
if false then ~3.0 else 3.0 →
3.0
```

- In fact, we know that declaring a function is really just syntactic sugar for binding a variable to an anonymous function. So when we evaluate the declaration of `abs` above, we are really binding the identifier `abs` to the value `fun r ->> if r < 0.0 then -. r else r`.
- Therefore, the evaluation of a function call proceeds as in the following example:

```
let abs r =
  if r < 0.0 then -. r else r
in
abs(2.0 +. 1.0)
→ (fun r -> if r < 0.0 then -. r else r)(2.0 +. 1.0)
  (* replace occurrences of abs in let body with anonymous function *)
→ (fun r -> if r < 0.0 then -. r else r)(3.0)
→ if 3.0 < 0.0 then -. 3.0 else 3.0
  (* replace occurrences of r in function body with argument 3.0 *)
→ if false then -. 3.0 else 3.0
→ 3.0
```

- We can use the substitution operator to give a more precise rule for what happens when a function is called:

$$(\text{fun } (p) \rightarrow e) (v) \rightarrow e\{v/p\}$$

- Interestingly, this is the same result that we got from the expression `let p = v in e`). So this tells us that we can think of a `let` as sugar for a function application!

- Some caveats

This is a model for how OCaml evaluates. The truth is that OCaml terms are compiled into machine code that executes much more efficiently than rewriting would. But that is much more complex to explain, and not that important for our purposes. The goal here is to allow us as programmers to understand what the program is going to do. We can do that much more clearly in terms of term rewriting than by thinking about the machine code, or for that matter in terms of the transistors in the computer and the electrons inside them. This evaluation model is an *abstraction* that hides complexity you don't need to know about. Understanding how programs execute in terms of these lower levels of abstraction is the topic of other courses, like CS 3420 and CS 4120.

Some aspects of the model should not be taken too literally. For example, you might think that function calls take longer if an argument variable appears many times in the body of the function. It might seem that calls to function `f` are faster if it is defined as `fun f(x) = x*3` rather than as `fun f(x) = x+x+x` because the latter requires three substitutions. Actually the time required to pass arguments to a function typically depends only on the number of arguments. Chances are the definition on the right is at least as fast as that on the left.

The model as given also has one significant weakness: it doesn't explain how recursive functions work. The problem is that a function is in scope within its own body. Mutually recursive functions are also problematic, because each mutually recursive function is in scope within the bodies of all the others.

A way to understand this is that a recursive function is an infinite unfolding of the original definition. For example, in a function for computing factorials,

```
let rec fact(n) = if n = 0 then 1 else n*fact(n-1)
```

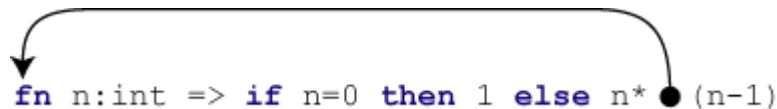
This is the same thing as

```
let rec fact = fun n -> if n = 0 then 1 else n*fact(n-1)
```

. This definition can then be unrolled as many times as we want, so we can think of `fact` as being bound to an infinite anonymous function:

```
fun n -> if n=0 then 1 else
  n*(fun n -> if n = 0 then 1 else
    n*(fun n -> if n = 0 then 1 else
      n*(...) (n-1))
    (n-1))
  (n-1)
```

It's probably easiest to think of it as an anonymous function that hasn't been infinitely unrolled like this, but rather contains a pointer to itself that expands out into the same full anonymous function whenever it is used:



```
fn n:int => if n=0 then 1 else n*(n-1)
```

It is possible to define a purely mathematical substitution-based semantics for recursive function declarations, which you'd see in CS 4110 or 6110. However, we compromise our purity slightly, we can successfully give a simpler semantics.

When a `let rec` is evaluated, **fresh** variables are generated for each variable bound in the `let rec`. Variables are fresh if they appear nowhere else in the program. A global binding is then generated between the fresh variable(s) and the function body or bodies in the `let rec`, but with the occurrences of the original variables replaced with the corresponding fresh variables.

More precisely,

```
let rec f = fun x -> e in e' → e'{f'/f}
  (with global binding f' = fun x -> e{f'/f}, f' fresh)
```

The name `f'` stands for the expression that the arrow points to the graphical representation above. If evaluation ever hits `f'`, it is replaced with its global binding. For example, consider this difficult-to-understand code that is similar to the example above:

```
let rec f(g,n) =
  if n=1 then g(0)
  else g(0) + f((fun x->n), n-1)
in
  f((fun x->10), 3)
```

Can you predict what the result will be? It evaluates as follows. If you can follow this then you really understand the substitution model!

```
f'((fun x->10), 3)      (with f' = fun (g,n) ->
                        if n=1 then g(0)
                        else g(0) + f'((fun x->n), n-1))
→
(fun (g,n) ->
  if n=1 then g(0)
  else g(0) + f'((fun x->n), n-1)) ((fun x->10), 3)
→
  if 3=1 then (fun x->10)(0)
  else (fun x->10)(0) + f'((fun x->3), 3-1)
→
    if false then (fun x->10)(0)
    else (fun x->10)(0) + f'((fun x->3), 3-1)
→ (fun x->10)(0) + f'((fun x->3), 3-1)
→ 10 + f'((fun x->3), 3-1)
→ 10 + (fun (g,n) -> ...) ((fun x->3), 3-1)
→ 10 + (fun (g,n) -> ...) ((fun x->3), 2)
→ 10 + if 2=1 then (fun x->3)(0) else (fun x->3)(0) + f'(fun x->2, 2-1)
→ 10 + if false then (fun x->3)(0) else (fun x->3)(0) + f'(fun x->2, 2-1)
→ 10 + (fun x->3)(0) + f'(fun x->2, 2-1)
```

```

→ 10 + 3 + f'(fun x->2, 2-1)
→ 10 + 3 + (fun (g,n) -> ...) (fun x->2, 2-1)
→ 10 + 3 + (fun (g,n) -> ...) (fun x->2, 1)
→ 10 + 3 + if 1=1 then (fun x->2) (0) else ...
→ 10 + 3 + (fun x->2) (0)
→ 10 + 3 + 2
→ 10 + 5
→ 15

```

In general, there might be multiple functions defined in a `let rec`. These are evaluated as follows:

```

let rec f1 = fun x1 -> e1
      and f2 = fun x2 -> e2
      ...
      and fn = fun xn -> en
in e' → e'{f'1/f1, ..., f'n/fn}
      (with global bindings f'1 = fun x1 -> e{f'1/f1, ..., f'n/fn}, ...
                          f'n = fun xn -> e{f'1/f1, ..., f'n/fn},
                          all fi fresh)

```

•

- Tricky example revisited

Now we have the tools to return to the tricky example from above. Let's first consider an easier case, where the third parameter is 1 rather than 3 as above:

```
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, 1)
```

The full substitution model would have us replace all the identifiers in the body `evil(dummy, dummy, 1)` with their values, but it can be a useful shorthand to write down the symbol `evil`, while knowing that means that we have to bind `f` and then substitute into the body of the if-then-else in the definition of `evil`. For instance,

```
evil(fun(x)->1000, fun(x)->1000, 1)

→let f(x) = 10 + 1 in
  if 1 = 1 then f(0) + (fun(x)->1000)(0) + (fun(x)->1000)(0)
  else evil(f, (fun(x)->1000), 1-1)

→(fun(x)->10+1)(0) + (fun(x)->1000)(0) + (fun(x)->1000)(0)

→2011
```

Now if we consider the case where `evil` is called with `n=2` rather than `n=1`, things get a bit more interesting. Here we will write down just the reduction steps corresponding to the recursive calls to `evil` and the calculation of the final return value.

```
evil(fun(x)->1000, fun(x)->1000, 2)

→evil(fun(x)->10+2, fun(x)->1000, 1)

→(fun(x)->10+1)(0) + (fun(x)->10+2)(0) + (fun(x)->1000)(0)

→1023
```

- Lexical vs. dynamic scoping

Because variable names are substituted immediately throughout their scope when a function is applied or a `let` is evaluated. This means that whenever we see a variable, how that variable is bound is immediately clear: the variable is bound at the closest enclosing binding occurrence that can be seen in the program text. This rule is called **lexical scoping**. Let us apply this to the tricky example from earlier.

The key question is what the variable `n` means within the functions `f`, `f1`, `f2`. Even though these variables are all bound to the function `f`, they are bound to versions of the function `f` that occurred in three different scopes, where the variable `n` was bound to 1, 2, and 3 respectively. For example, on the first entry to `evil`, the value 3 is substituted for the variable `n` within the function `f` (which ultimately becomes `f2` on the third application on `evil`).

The most common alternative to lexical scoping is called **dynamic scoping**. In dynamic scoping, a variable is bound to the most recently bound version of the variable, and function values do not record how their variables such as `n` are bound. For example, in the language Perl, the equivalent of the example code would print 33 rather than 36, because the most recently bound value for the variable `n` is 1. Dynamic scoping can be confusing because the meaning of a function depends on the context where it is used. Most modern languages use lexical scoping.

-