CSE341, Fall 2011, Lecture 3 Summary

Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

This lecture has three topics:

- Let-expressions, an absolutely crucial feature that allows for local variables in a very simple, general and flexible way. It is crucial for style and for efficiency.
- Options, which are a way to build data that has 0 or 1 items. We could use 0-element and 1-element lists instead, but using options is better style because it makes clear that the number of items must be 0 or 1.
- Benefits of not being able to mutate (i.e., assign to) variables and parts of data structures.

Let expressions

A let-expression lets us have local variables. In fact, it lets us have local *bindings* of any sort, including function bindings. Because it is a kind of expression, it can appear anywhere an expression can.

Syntactically, a let-expression is:

```
let b1 b2 ... bn in e end
```

where each `bi` is a binding and `e` is an expression.

The type-checking and semantics of a let-expression is much like the semantics of the top-level bindings in our ML program. We evaluate each binding in turn, creating a larger environment for the subsequent bindings. So we can use all the earlier bindings for the later ones, and we can use them all for `e`. We call the *scope* of a binding "where it can be used," so the scope of a binding in a let-expression is the later bindings in that let-expression and the "body" of the let-expression (the `e`). The value `e` evaluates to is the value for the entire let-expression, and, unsurprisingly, the type of `e` is the type for the entire let-expression.

For example, this expression evaluates to 7; notice how one inner binding for `x` *shadows* an outer one.

```
let val x = 1
in
    (let val x = 2 in x+1 end) + (let val y = x+2 in y+1 end)
end
```

Also notice how let-expressions are expressions so they can appear as a subexpression in an addition (though this example is silly and bad style because it is hard to read).

Let-expressions can bind functions too, since functions are just another kind of binding. If a helper function is needed by only one other function and is unlikely to be useful elsewhere, it is good style to bind it locally. For example, here we use a local helper function to help produce the list `[1,2,...,x]`:

```
fun countup_from1 (x:int) =
    let fun count (from:int, to:int) =
            if from=to
            then to::[]
```

```
            else from :: count(from+1,to)
    in
        count(1,x)
    end
```

However, we can do better. When we evaluate a call to `count`, we evaluate `count`'s body in a dynamic environment that is the environment where `count` was defined, extended with bindings for `count`'s arguments. The code above does not really utilize this: `count`'s body uses only `from`, `to`, and `count` (for recursion). It could also use `x`, since that is in the environment when `count` is defined. Then we do not need `to` at all, since in the code above it always has the same value as `x`. So this is better style:

```
fun countup_from1_better (x:int) =
    let fun count (from:int) =
            if from=x
            then x::[]
            else from :: count(from+1)
    in
        count 1
    end
```

This technique — define a local function that uses other variables in scope — is a hugely common and convenient thing to do in functional programming. It is a shame that many non-functional languages have little or no support for doing something like it.

Local variables are often good style for keeping code readable. They can be much more important than that when they bind to the *results of* potentially expensive computations. For example, consider this code that does not use let-expressions:

```
fun bad_max (lst : int list) =
    if null lst
    then 0
    else if null (tl lst)
    then hd lst
    else if hd lst > bad_max(tl lst)
    then hd lst
    else bad_max(tl lst)
```

If you call `bad_max` with `countup_from1 30`, it will make approximately 2^{30} (over one billion) recursive calls to itself. The reason is an "exponential blowup" — the code calls `bad_max(tl lst)` twice and each of those calls call `bad_max` two more times (so four total) and so on. This sort of programming "error" can be difficult to detect because it can depend on your test data (if the list counts down, the algorithm makes only 30 recursive calls instead of 2^{30}).

We can use let-expressions to avoid repeated computations. This version computes the max of the tail of the list once and stores the resulting value in `tl_ans`.

```
fun good_max (lst : int list) =
    if null lst
    then 0
    else if null (tl lst)
    then hd lst
    else
```

```
(* for style, could also use a let-binding for hd lst *)
let val tl_ans = good_max(tl lst)
in
    if hd lst > tl_ans
    then hd lst
    else tl_ans
end
```

Options

The previous example does not properly handle the empty list — it returns 0. This is bad style because 0 is really not the maximum value of 0 numbers. There is no good answer, but we should deal with this case reasonably. One possibility is to raise an exception; you can learn about SML exceptions on your own if you are interested. Instead, let's change the return type to either return the maximum number or indicate the input list was empty so there is no maximum. Given the constructs we have, we could "code this up" by return an `int list`, using `[]` if the input was the empty list and a list with one integer (the maximum) if the input list was not empty.

While that works, lists are "overkill" — we will always return a list with 0 or 1 elements. So a list is not really a precise description of what we are returning. The ML library has "options" which are a precise description: an option value has either 0 or 1 thing: `NONE` is an option value "carrying nothing" whereas `SOME e` evaluates `e` to a value `v` and becomes the option carrying the one value `v`. The type of `NONE` is `'a option` and the type of `SOME e` is `t option` if `e` has type `t`.

Given a value, how do you use it? Just like we have `null` to see if a list is empty, we have `isSome` which evaluates to `false` if its argument is `NONE`. Just like we have `hd` and `tl` to get parts of lists (raising an exception for the empty list), we have `valOf` to get the value carried by `SOME` (raising an exception for `NONE`).

Using options, here is a better version with return type `int option`:

```
fun better_max (lst : int list) =
    if null lst
    then NONE
    else
        let val tl_ans = better_max(tl lst)
        in if isSome tl_ans andalso valOf tl_ans > hd lst
           then tl_ans
           else SOME (hd lst)
        end
```

The version above works just fine and is a reasonable recursive function because it does not repeat any potentially expensive computations. But it is both awkward and a little inefficient to have each recursive call except the last one create an option with `SOME` just to have its caller access the value underneath. Here is an alternative approach where we use a local helper function for non-empty lists and then just have the outer function return an option. Notice the helper function would raise an exception if called with `[]`, but since it is defined locally, we can be sure that will never happen.

```
fun better_max2 (lst : int list) =
    if null lst
    then NONE
    else let (* fine to assume argument nonempty because it is local *)
             fun max_nonempty (lst : int list) =
                 if null (tl lst) (* lst better not be [] *)
```