CSE 341, Winter 2008, Lecture 3 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

Recall that we can now write recursive functions, which is particularly useful / necessary for using lists since the size of a list is not bounded. However, we do not yet have a way to create local variables, which are essential for at least two reasons — writing code in good style and writing code that uses efficient algorithms. Today's topic is ML's let-expressions, which is how we create local variables. Let-expressions are both simpler and more powerful than local variables in many other languages: they can appear anywhere and can have any kind of binding.

Syntactically, a let-expression is:

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

where each `bi` is a binding (so far we have seen variable bindings and function bindings) 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 context/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-expressions (the `e`). The value `e` evaluates to is the value 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 only needed by one other function and is unlikely to be useful elsewhere, it's good style just 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 will evaluate `count`'s body in an environment that is the environment where `count` was defined, extended with bindings for `count`'s arguments. The code above doesn't really exploit this: `count`'s body only uses `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 most 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
```

This 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. So to properly deal with that fact, it would be better to 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 returning a list of integers, using the empty list 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`.

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
```