CSE341, Fall 2011, Lecture 2 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.

Pieces of a language:

Learning a programming language involves learning *syntax*, *semantics* (evaluation rules and typing rules), *idioms*, *libraries*, and *tools*. While libraries and tools are essential for being an effective programmer (to avoid reinventing available solutions or unnecessarily doing things manually), this course does not focus on them much. That can leave the wrong impression that we're using "silly" or "impractical" languages, but libraries and tools are just less relevant in a course on the conceptual similarities and differences of programming languages.

Function bindings:

Recall that an ML program is a sequence of bindings. Each binding adds to the static environment (for type-checking subsequent bindings) and to the dynamic environment (for evaluating subsequent bindings). The previous lecture introduced variable bindings; this lecture introduces *function bindings*, i.e., how to define and use functions. We will then learn how to build up and use larger pieces of data from smaller ones using *pairs* and *lists*.

A function is sort of like a Java method — it is something that is called with arguments and has a body that produces a result. Unlike a method, there is no notion of a class, `this`, etc. We also don't have things like return statements. A simple example is this function that computes x^y assuming $y \geq 0$:

```
fun pow (x:int, y:int) = (* correct only for y >= 0 *)
    if y=0
    then 1
    else x * pow(x,y-1)
```

Syntax:

The syntax for a function binding looks like this (we'll generalize this definition in later lectures):

```
    fun x0 (x1 : t1, ..., xn : tn) = e
```

This is a binding for a function named `x0`. It takes n arguments `x1`, ... `xn` of types `t1`, ..., `tn` and has an expression `e` for its body. As always, syntax is just syntax — we must define the typing rules and evaluation rules for function bindings. But roughly speaking, in `e`, the arguments are bound to `x1`, ... `xn` and the result of calling `x0` is the result of evaluating `e`.

Type-checking:

To type-check a function binding, we type-check the body `e` in a static environment that (in addition to all the earlier bindings) maps `x1` to `t1`, ... `xn` to `tn` and `x0` to `t1 * ... * tn -> t`. Because `x0` is in the environment, we can make *recursive* function calls, i.e., a function definition can use itself. The syntax of a function type is "argument types" `->` "result type" where the argument types are separated by `*` (which just happens to be the same character used in expressions for multiplication). For the function binding to type-check, the body `e` must have the type `t`, i.e., the result type of `x0`. That makes sense given the evaluation rules below because the result of a function call is the result of evaluating `e`.

But what, exactly, is `t` – we never wrote it down? It can be any type, and it's up to the type-checker (part of the language implementation) to figure out what `t` should be such that using it for the result type of `x0` makes, "everything work out." For now, we will take it as magical, but *type inference* (figuring out types not written down) is a very cool feature of ML discussed in a later lecture. It turns out that in ML you almost never have to write down types. Soon the argument types `t1`, ..., `tn` will also be optional but not until we learn pattern matching in a couple lectures.[1]

After a function binding, `x0` is added to the static environment with its type. The arguments are not added to the top-level static environment — they can be used only in the function body.

Evaluation:

The evaluation rule for a function binding is trivial: *A function is a value* — we simply add `x0` to the environment as a function that can be *called* later. As expected for recursion, `x0` is in the dynamic environment in the function body and for subsequent bindings (but not, unlike in say Java, for preceding bindings, so the order you define functions is very important).

Function calls:

Function bindings are useful only with function call, a new kind of expression. The ***syntax*** is `e0 (e1,...,en)` with the parentheses optional if there is exactly one argument. The ***typing rules*** require that `e0` has a type that looks like `t1*...*tn->t` and for $1 \leq i \leq n$, `ei` has type `ti`. Then the whole call has type `t`. Hopefully, this is not too surprising. For the ***evaluation rules***, we use the environment at the point of the call to evaluate `e0` to `v0`, `e1` to `v1`, ..., `en` to `vn`. Then `v0` must be a function (it will be assuming the call type-checked) and we evaluate the function's body in an environment extended such that the function arguments map to `v1`, ..., `vn`.

Exactly which environment is it we extend with the arguments? The environment that "was current" when the function was *defined*, not the one where it is being called. This distinction will not arise in this lecture, but we will discuss it in great detail later.

Putting all this together, we can determine that this code will produce an environment where `ans` is `64`:

```
fun pow (x:int, y:int) = (* correct only for y >= 0 *)
    if y=0
    then 1
    else x * pow(x,y-1)

fun cube (x:int) =
    pow(x,3)

val ans = cube(4)
```

Pairs, Tuples:

Programming languages need ways to build compound data out of simpler data. The first we will learn about in ML is *pairs*. The ***syntax*** to build a pair is `(e1,e2)` which ***evaluates*** `e1` to `v1` and `e2` to `v2` and makes the pair of values `(v1,v2)`, which is itself a value. Since `v1` and/or `v2` could themselves be pairs (possibly holding other pairs, etc.), we can build data with several "basic" values, not just two, say, integers. The ***type*** of a pair is `t1*t2` where `t1` is the type of the first part and `t2` is the type of the second part.

Just like making functions is useful only if we can call them, making pairs is useful only if we can later retrieve the pieces. Until we learn pattern-matching, we will use `#1` and `#2` to access the first and second part. The typing rule for `#1 e` or `#2 e` should not be a surprise: `e` must have some type that looks like `ta * tb` and then `#1 e` has type `ta` and `#2 e` has type `tb`.

Here are several example functions using pairs. `div_mod` is perhaps the most interesting because it uses a

[1] The way we are using pair-reading constructs like `#1` in this lecture and the first homework requires these explicit types.

pair to return an answer that has two parts. This is quite pleasant in ML, whereas in Java (for example) returning two integers from a function requires defining a class, writing a constructor, creating a new object, initializing its fields, and writing a return statement.

```
fun swap (pr : int*bool) =
    (#2 pr, #1 pr)

fun sum_two_pairs (pr1 : int*int, pr2 : int*int) =
    (#1 pr1) + (#2 pr1) + (#1 pr2) + (#2 pr2)

fun div_mod (x : int, y : int) =  (* note: returning a pair a real pain in Java *)
    (x div y, x mod y)

fun sort_pair (pr : int*int) =
    if (#1 pr) > (#2 pr)
    then pr
    else ((#2 pr),(#1 pr)) (* or could write: else swap pr *)
```

In fact, ML supports *tuples* by allowing any number of parts. For example, a 3-tuple (i.e., a triple) of integers has type `int*int*int`. An example is `(7,9,11)` and you retrieve the parts with `#1 e`, `#2 e`, and `#3 e` where `e` is an expression that evaluates to a triple.

Pairs and tuples can be nested however you want. For example, `(7,(true,9))` is a value of type `int * (bool * int)`, which is different from `((7,true),9)` which has type `(int * bool) * int` or `(7,true,9)` which has type `int*bool*int`.

Lists:

Though we can nest pairs of pairs (or tuples) as deep as we want, for any variable that has a pair, any function that returns a pair, etc. there has to be a type for a pair and that type will determine the amount of "real data." Even with tuples the type specifies how many parts it has. That is often too restrictive; we may need a list of data (say integers) and the length of the list isn't yet known when we're type-checking (it might depend on a function argument). ML has *lists*, which are more flexible than pairs because they can have any length, but less flexible because all the elements of any particular list must have the same type.

The empty list, with syntax `[]`, has 0 elements. It is a value, so like all values it evaluates to itself immediately. It can have type `t list` for *any* type `t`, which ML writes as `'a list` (pronounced "quote a list" or "alpha list"). In general, the type `t list` describes lists where all the elements in the list have type `t`. That holds for `[]` no matter what `t` is.

A non-empty list with n values is written `[v1,v2,...,vn]`. You can make a list with `[e1,...,en]` where each expression is evaluated to a value. It is more common to make a list with `e1 :: e2`, pronounced "`e1` consed onto `e2`." Here `e1` evaluates to an "item of type `t`" and `e2` evaluates to a "list of `t` values" and the result is a new list that starts with the result of `e1` and then is all the elements in `e2`.

As with functions and pairs, making lists is useful only if we can then do something with them. As with pairs, we will change how we use lists after we learn pattern-matching, but for now we will use 3 functions provided by ML. Each takes a list as an argument.

- `null` evaluates to `true` for empty lists and `false` for nonempty lists.
- `hd` returns the first element of a list, *raising an exception* if the list is empty.
- `tl` returns the tail of a list (a list like its argument but without the first element), raising an exception if the list is empty.

Here are some simple examples of functions that take or return lists:

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

fun countdown (x : int) =
    if x=0
    then []
    else x :: countdown(x-1)

fun append (lst1 : int list, lst2 : int list) =
    if null lst1
    then lst2
    else hd(lst1) :: append(tl(lst1), lst2)
```

Functions that make and use lists are almost always recursive because a list has an unknown length. To write a recursive function, the thought process involves thinking about the *base case* — for example, what should the answer be for an empty list — and the *recursive case* — how can the answer be expressed in terms of the answer for the rest of the list.

When you think this way, many problems become much simpler in a way that surprises people who are used to thinking about while loops and assignment statements. A great example is the append function above that takes two lists and produces a list that is one list appended to the other. This code implements an elegant recursive algorithm: If the first list is empty, then we can append by just evaluating to the second list. Otherwise, we can append the tail of the first list to the second list. That is almost the right answer, but we need to "cons on" (using `::` has been called "consing" for decades) the first element of the first list. There is nothing magical here — we keep making recursive calls with shorter and shorter first lists and then as the recursive calls complete we add back on the list elements removed for the recursive calls.

Finally, we can combine pairs and lists however we want without having to add any new features to our language. For example, here are several functions that take a list of pairs of integers. Notice how the last function reuses earlier functions to allow for a very short solution. This is very common in functional programming. In fact, it should bother us that `firsts` and `seconds` are so similar but we don't have them share any code. We will learn how to fix that later.

```
fun sum_pair_list (lst : (int * int) list) =
    if null lst
    then 0
    else #1 (hd(lst)) + #2 (hd(lst)) + sum_pair_list(tl(lst))

fun firsts (lst : (int * int) list) =
    if null lst
    then []
    else (#1 (hd lst))::(firsts(tl lst))

fun seconds (lst : (int * int) list) =
    if null lst
    then []
    else (#2 (hd lst))::(seconds(tl lst))

fun sum_pair_list2 (lst : (int * int) list) =
    (sum_list (firsts lst)) + (sum_list (seconds lst))
```