Learning a programming language generally involves learning syntax, semantics, idioms, libraries, and tools. While the last two are extremely important for being an effective programmer (to avoid reinventing known solutions or unnecessarily doing things manually), this course does not focus on them much. That can leave the wrong impression that the languages we’re using are “silly” or “theoretical” when it’s really the case that libraries and tools are just less “intellectually interesting” in a course on the similarities and differences of programming languages.

Recall that an ML program is a sequence of bindings, each of which adds to the context (for type-checking) and environment (for evaluating) subsequent bindings. Last time we saw variable bindings; today will be all about function bindings, i.e., how to define and use functions. We’ll also learn how to build up larger pieces of data from smaller ones using pairs and lists.

A function is kind of like a Java method — it is something that is called with arguments and produces a result. Unlike a method, there is no notion of a class, this, etc. We also don’t have things like return statements. Syntactically, we can write a function 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 of types t1, ..., tn. In the function body e, the arguments are bound to x1, ... xn. As always, syntax is just syntax — we must define the typing rules and evaluation rules for function bindings.

To type-check a function binding, we type-check the body e in a context that (in addition to all the earlier bindings that make up the current context) maps x1 to t1, ... xn to tn and x0 to t1 * ... * tn -> t. Because x0 is in the context (and environment, see below), 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 we’ll learn about 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. (The way we are using pair-reading constructs like #1 in this lecture and the first homework require these explicit types.)

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 applied or called (these are synonyms). As expected for recursion, x0 is in the 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).

So, function definitions are only useful with function application. The syntax is e0(e1, ..., en). The typing rules require that e0 has a type that looks like t1*...*tn->t and for 1 ≤ i ≤ n, ei has type ti. Then the whole application has type t. Hopefully, this is not too surprising. For the evaluation rules, we use the environment at the point of the application to evaluate e0 to v0, e1 to v1, ..., en to vn. Then v0 must be a function (it will be assuming the application 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 does not matter in this lecture, but it will be very important later (and we’ll repeat this point).

Putting all this together, we can determine that this code will produce an environment where ans is 64:
fun pow (x:int, y:int) = (* only correct for y >= 0 *)
  if y=0
  then 1
  else x * pow(x,y-1)

fun cube2 (x:int) = pow(x,3)

val ans = cube(4)

Programming languages need a way to build compound data out of simpler data. ML has several ways. The first we will learn about 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 only useful if we can call them, making pairs is only useful if we can later retrieve the pieces. Until we learn pattern-matching, we'll use #1 and #2 to retrieve 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 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, etc.

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)

  (* returning a pair a real pain in Java *)

fun div_mod (x : int, y : int) =
  (x div y, x mod y)

fun sort_pair (pr : int*int) =
  if (#1 pr) > (#2 pr)
  then pr
  else ((#2 pr),(#1 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.

Though we can make 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’s often too restrictive; we often 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 have to have the same type.

The empty list, written [] has 0 elements. It is a value. It has type t list for any type t. 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’s more common to make a list with e1 :: e2. Here e1 evaluates
to an “item of type t” and e2 evaluates to a “list of t’s” 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 them is only useful if we can then do something with them. As
with pairs, we’ll change how we use lists after we learn pattern-matching, but for now we’ll use 3 functions
provided by ML: 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.

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 learn to think this way, many problems becomes
quite a bit simpler in a way almost mind-boggling to people who are used to thinking about while loops and
assignment statements. A great example is a function that takes two lists and produces a list that is one list
appended to the other:

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

This code is 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 still have 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 we took off.

Most Java or C code for list append is more complicated for two reasons. First, the whole “while loops
and field updates” approach to programming often makes you miss simple higher-level algorithms that you
see by thinking recursively. Second, in Java or C we have to ask a very important question: Should the
result of append make a copy of its arguments or should it change its arguments? This is crucial: If I append
[1,2] to [3,4,5], I’ll get some list [1,2,3,4,5] but if later someone can change the [3,4,5] list to be
[3,7,5] is the appended list still [1,2,3,4,5] or is it now [1,2,3,7,5]? This is the essence of why so
much intellectual energy in Java programming is spent on keeping track of how much sharing or aliasing
there is among objects, and why object identity (are two objects the same object or do they just have the
same field values) is so important.

In ML, it doesn’t matter! And that’s a huge advantage of functional programming — because there is
no way to update a list (no assignment statements), the whole idea of sharing and aliasing goes away. A
list [3,4,5] is just that — a list with three elements, 3, 4, and 5. You cannot tell if append copies lists or
shares them.

The same is true for tl. For efficiency reasons, tl doesn’t actually make a copy of the tail of the list, it
just returns it, so I would expect the ML implementation to internally have aliasing here:

val y = tl x (* now y and the tail of x are "the same list" *)

This is more efficient in terms of time (no copying the list) and space (only one list). And since you can’t
tell since lists can’t be mutated, you can forget about this sharing — you get the efficiency without the
complications. It’s a great reason not to use assignment statements.