1 Pieces of a language

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

2 Shadowing

When a variable is bound with \( \text{val } \text{x} = \text{e} \), expression \( \text{e} \) is evaluated in the current dynamic context to a value \( \text{v} \), then the dynamic context is extended with a new binding for \( \text{x} \) to \( \text{v} \). But what happens if \( \text{x} \) was already bound in the current dynamic context? e.g.,

\begin{verbatim}
val a = 10
val b = a + 1
val a = 0
\end{verbatim}

What value is \( \text{b} \) bound to after that program?

The answer is easy enough to figure out, if we stop to think about the semantics of variable bindings. After the first binding above, the dynamic environment binds \( \text{a} \) to 10. To evaluate the second binding, the run-time looks up the current binding of \( \text{a} \) in the dynamic environment, finds that it’s bound to 10, evaluates the addition to get 11, then binds \( \text{b} \) to 11. Finally, to evaluate the third-binding, the run-time \textit{re-binds} \( \text{a} \) in the dynamic environment to 0. The previous binding is no longer visible, so we say that it is \textit{shadowed}. Moreover, \( \text{b} \)'s binding is not touched by this shadowing; \( \text{b} \) is still bound to 11, because nothing in the semantics of evaluating variable bindings says to re-evaluate bindings.

What this means is that \textit{variable bindings are not assignment statements}, unlike what you’re used to in Java and C.

3 Function bindings

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

A function is similar to a Java method, in that it is called with arguments and has a body that produces a result. Unlike a method, there is no notion of a class, \texttt{this}, etc. Nor are there return statements. A simple example is this function that computes \( x^y \) assuming \( y \geq 0 \):

\begin{verbatim}
(* requires: y >= 0 *)
(* returns: x^y *)
fun pow (x:int, y:int) =
  if y = 0
    1
  else
    x * pow (x, y - 1)
\end{verbatim}
then 1
else x * pow(x,y-1)

Syntax. Here’s the syntax for function bindings:

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

This binding is for a function named x0. It takes n arguments x1, ... xn of types t1, ..., tn and has an expression e for its body. Roughly speaking, in e, the arguments are bound to x1, ... xn and the result of calling x0 is the result of evaluating e. But to express that more precisely, let’s define the semantics of functions.

Type-checking. To type-check a function binding, the compiler type-checks 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 write recursive function calls—that is, a function binding can refer to itself. The syntax of a function type is “argument types” → “result type” where the argument types are separated by * (which, somewhat confusingly, 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—that is, 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? So far, we haven’t said. Actually, it turns out that the type-checker (a part of the compiler) figures out what t should be, so that everything works out. For now, type inference (figuring out types not written down) we seem magical. But it is a very cool feature of ML we’ll discuss in a later lecture.

It turns out that in ML you almost never have to write down types. 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 already a value. So the run-time simply adds x0 to the environment as a function that can be called later. As you would expect for recursion, x0 is in the dynamic environment in the function body and for subsequent bindings. However, unlike in Java, x0 is not the environment for preceding bindings. So the order in which you define functions is very important.

4 Function calls

After a function has been created by a function binding, it can be invoked with a function call. Function calls are a new kind of expression. Their syntax is

```
e0 (e1, ..., en)
```

and the parentheses are optional if there is exactly one argument. Their typing rules require that e0 has type t1*...*tn→t. Furthermore, for 1 ≤ i ≤ n, ei must have type ti. The type of the entire function call expression is then type t. For the evaluation rules, the run-time uses the environment at the point of the call to evaluate e0 to v0, ..., and en to vn. Then v0 must be a function (and it will be, assuming the call type-checked) and the run-time evaluates the function’s body in an environment extended such that the

---

1We’ll generalize this definition in a later lecture.

2In fact, function argument types can also be optional. But for these first few lectures, and the first homework, we have to explicitly write down argument types. After we learn pattern matching, we’ll usually be able to omit argument types. Right now they’re required because of certain constructs we’re using, such as the pair destructor #1 coming up later in this lecture.
function arguments map to \( v_1, \ldots, v_n \). Exactly which dynamic environment does the run-time extend with the arguments? The environment that was “current” when the function was defined, not the environment in which the function is called. This distinction won’t be too important for now, but we will discuss it in great detail later.

Putting all this together, we can determine that this code will produce an environment where \( \text{ans} \) is bound to 64:

\[
\begin{align*}
(* \text{ requires: } & y \geq 0 *) \\
(* \text{ returns: } & x^y *) \\
\text{fun } & \text{pow } (x: \text{int}, y: \text{int}) = \\
& \text{if } y = 0 \\
& \text{then } 1 \\
& \text{else } x \times \text{pow}(x, y-1) \\
(* \text{ returns: } & x^3 *) \\
\text{fun } & \text{cube } (x: \text{int}) = \\
& \text{pow}(x, 3) \\
\text{val } & \text{ans } = \text{cube}(4)
\end{align*}
\]

5 Pairs and Tuples

Programmers need ways to build compound data out of simpler data. The first way we will learn about in ML is pairs. The syntax to build a pair is \((e_1, e_2)\). The run-time evaluates a pair by first evaluating \( e_1 \) to \( v_1 \) and \( e_2 \) to \( v_2 \), then creating the pair of values \((v_1, v_2)\), which is itself a value. Values \( v_1 \) and \( v_2 \) could themselves be pairs, and those pairs could contain other pairs, etc. The type of a pair is \( t_1 \times t_2 \) where \( t_1 \) is the type of the first part and \( t_2 \) is the type of the second part.

To retrieve the parts of a pair, we use \#1 and \#2 to access the first and second part.\(^4\) The typing rules for these accessors is simple: assuming \( e \) has type \( t_a \times t_b \), then \#1 \( e \) has type \( t_a \) and \#2 \( e \) has type \( t_b \).

Here are some example functions using pairs. Function \text{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, returning two integers from a function requires a lot of work (defining a class, writing a constructor, creating a new object, initializing its fields, and writing a return statement).

\[
\begin{align*}
\text{fun } & \text{swap } (pr : \text{int*int}) = \\
& (\#2 \text{ pr}, \#1 \text{ pr}) \\
\text{fun } & \text{sum_two_pairs } (pr_1 : \text{int*int}, pr_2 : \text{int*int}) = \\
& (\#1 \text{ pr}_1) + (\#2 \text{ pr}_1) + (\#1 \text{ pr}_2) + (\#2 \text{ pr}_2) \\
\text{fun } & \text{div_mod } (x : \text{int}, y : \text{int}) = \\
& (x \text{ div } y, x \text{ mod } y) \\
\text{fun } & \text{sort_pair } (pr : \text{int*int}) = \\
& \text{if } (\#1 \text{ pr}) > (\#2 \text{ pr}) \\
& \text{then } pr \\
& \text{else swap pr } (* \text{ or could write: else } ((\#2 \text{ pr}),(\#1 \text{ pr}))*)
\end{align*}
\]

\(^3\)We’ll stop bold-facing syntax, type checking, and evaluation now, but you should keep watch for them with each new language feature you learn.

\(^4\)Later we’ll learn a much better way to do this using pattern matching. But one thing at a time.
ML actually supports *tuples*, which are more general than pairs. Tuples allow 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). You would retrieve the parts of a triple with #1 e, #2 e, and #3 e, assuming that 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)` and is different from
- ((7,true),9), which has type `(int * bool) * int`. And both are different from
- (7,true,9), which has type `int*bool*int`. 