1 Expressions, names, values, and types

OCaml has an interactive toplevel, AKA a read-eval-print loop, that we can use to play around in the language. To start it, log into your account on ieng6 and type `rlwrap ocaml`. The `rlwrap` makes things like up-arrow work as they do in Bash, instead of printing garbage like `^[A`. You can type an expression followed by `;;` to evaluate it:

```
# 1 + 2;;
- : int = 3
```

OCaml says that the result of the expression `1 + 2` has type `int`, and the value is `3`. I can bind this value to a name, say `x`, for use later:

```
# let x = 1 + 2;;
val x : int = 3
# x + x;;
- : int = 6
```

As you can see, the format of the toplevel output is `<name> : <type> = <value>`. If you didn’t bind any name, it just says `-`. When you do bind a name to some value at the top level in this way, that name will stand for that value in the rest of your session (until you hit `^D` to quit), unless you rebind it later.

You can also bind names to values locally (i.e., for a single expression) using the `let-in` construct:

```
# let y = 2 + 3 in y * y;;
- : int = 25
# y;;
Error: Unbound value y
```

Notice how `y` is not defined outside of that local context (between the `in` and `;;`).

There is a good reason I’m using terms like “bind” and “name” instead of “assign” and “variable”. Although I can bind some name `x` to a value, do a bunch of computation, and then rebind `x` to another value, **this is not the same as updating the contents of a variable** in C or Java. Think of it as creating an entirely new `x` that is distinct from the first one. When we say `x` after this event, we’ll be referring to the new `x`, but conceptually the old `x` is still there, unchanged. The difference here, and the consequences of it, will become abundantly clear in a few lectures. At a very high level, it means you can’t write the usual `for/while loop` with these name-value bindings. That’s OK though, since we won’t be writing any of those loops in our OCaml programs.

Enough about binding, let’s see some more computations. What about some mixed integer and floating-point arithmetic?
OCaml does not allow this. Why the hell not? It makes more sense than you may think. If you’ve taken a computer architecture class, you know that the CPU has physical hardware to add two integers together, and to add two floats together, but not to add a float to an integer. The C language (and many descendants) handles this problem with implicit type conversions. That is, in C the compiler will handle 1 + 2.1 by promoting the 1 to a floating-point 1.0 and using the FP adder. To understand a C program, you have to remember how all these implicit conversions work. To understand an OCaml program, you don’t, because there are none. The + is the addition operator for integers, and there’s a separate operator +. to add floats. It is an error to use + with a float, or +. with an int. If you really want such a conversion, you must specify it yourself:

```
# 1 + (truncate 2.1);;
- : int = 3
```

Here we have just seen a function call in OCaml. Function `truncate` does the obvious thing based on its name: it throws away the fractional part of a float to turn it into an int. We can examine its type at the toplevel:

```
# truncate;;
- : float -> int = <fun>
```

The type float -> int means “function that takes a float and returns an int”. To call it, you just separate the function and the argument with a space. That is, you don’t write f(x), but f x. Function call has higher precedence than binary operators, so I actually didn’t need the parentheses for 1 + truncate 2.1.

In fact, we have been calling functions the whole time. Binary operators like +, *, and +. are just functions, to see this put parens around them:

```
# ( + );;
- : int -> int -> int = <fun>
# ( * );;
- : int -> int -> int = <fun>
# ( +. );;
- : float -> float -> float = <fun>
```

We’ve seen in lecture why function types look this way with all the arrows: ( + ) takes one int argument for the left hand side and returns another function that takes the second int for the right hand side, performs the addition LHS + RHS, and returns the int answer. But as a shortcut, you can think of int -> int -> int as “function that takes two ints and returns an int”. Notice how, like I said, + and * only work on ints, and +. only works on floats. While this may seem annoying coming from C or Java, it enables a very nice feature of OCaml called type inference. Like Java, OCaml is a strongly typed language, and it requires that your programs type-check at compile time before it will ever run them. But unlike Java, I have not had to declare any types in my programs! If I want to say let a = b + c, I don’t have to specify that a is an int, because + will always return int, so OCaml already knows. Likewise it knows that b and c must be ints.
There’s one more thing to see here: notice how functions have the value <fun>. I will defer to later lectures for an explanation of what this value means. For now, simply observe that it is a value. The consequences of this are important: if functions are values just like 2 and 9.3 and "foobar", then we can pass functions around as arguments, and even return functions as results of other functions. This is why OCaml is called a functional programming language; functions will be very pervasive in your programs.

Speaking of pervasive, this is a good opportunity to mention the OCaml library documentation linked on the class site. It has a bunch of modules that you should not use at this point in the class. But there is one module that you can use; in fact, you will have a difficult time doing anything without it. It’s called Pervasives and it is open in all programs, meaning you can refer to whatever without typing Pervasives.whatever. This module is where all the standard binary operators are defined, along with functions like truncate, printing routines, and much more. You might want to keep the URL to it handy in case you forget how any of the built-in OCaml functions work. Here it is: http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html

2 Writing a simple function

Since I claim this is a functional programming language, I’d better show you how to write a function. There are actually multiple ways, but here is one. This is a standard “first function”, the recursive factorial:

```ocaml
# let rec factorial n =
   if n = 0 then 1
   else n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 10;;
- : int = 3628800
```

I must say let rec to write a recursive function. Putting n on the left of the = means it’s a parameter. There is no need for a “return” keyword, because the function returns the value of whatever expression is on the right side of the =. Otherwise, this function directly encodes the recursive definition of factorial given in math textbooks. Notice how again I did not have to declare any types; OCaml just figures them out. Also, if-then-else is an expression, unlike in C and Java where it is a statement (which modifies control flow but does not return a value). That means I can do this:

```ocaml
# let x = if true then "foo" else "bar";;
val x : string = "foo"
```

It also means that the then and else branches of an if expression must have the same type. If I use different types, OCaml’s type checker will complain, even if only one branch could ever be taken at run time:

```ocaml
# let x = if true then "foo" else 7;;
Error: This expression has type int but an expression was expected of type string
```

To understand why it complains, consider how the type checker works. You’ll learn this in much more detail, but basically, in this case it checks whether the expression true has type bool. Of course it does, so it checks whether the expressions in the two branches have the same type. Because the then branch has type string, it expects the else branch to have the same type, but the expression 7 actually has type int. You know very well that the expression true (the series of letters t-r-u-e that you wrote down) will always evaluate to the value true (in its mathematical sense). But the type checker is not evaluating any of your expressions. It doesn’t run your program, even if your program happens to be simple. Consider what
would happen if I hadn’t written `true` as the guard, but instead `blah x y z`. It might be the case that this function call always evaluates to `true` or always `false`, but the type checker would have to do a lot more work to figure that out. And that’s no good, because the type checking algorithm is part of the compiler, so it has to handle arbitrary programs as input and produce an output (either OK or Error) quickly. But for an arbitrary function `blah`, there is no procedure that can tell us whether `blah` ever returns a value at all, let alone discriminate among different values that might be returned, like `true` or `false`. If we had such an algorithm, then we would have a decision procedure for the famous Halting Problem, which Turing proved undecidable in 1936.