University of Washington CSE 341: Programming Languages
Unit 1 Reading Notes

Standard Description: This summary covers **some of** *the same material as lecture and section. It can help to read about the material in a narrative style and to have the material for an entire unit of the course in a single document, especially when reviewing the material later. Please report errors in these notes, even typos. This summary is not a sufficient substitute for attending class, reading the associated code, etc.*

Last updated: July 22, 2022

Contents

Welcome to Programming Languages

These reading notes supplement the other essential course materials and, in particular, do not include important course information from the syllabus, course web page, etc. *Read those other documents thoroughly.*

A course titled, "Programming Languages" can mean many different things. For us, it means the opportunity to learn the *fundamental concepts* that appear in one form or another in almost every programming language. We will also get some sense of how these concepts "fit together" to provide what programmers need in a language. And we will use different languages to see how they can take complementary approaches to representing these concepts. All of this is intended to make you a better software developer, in any language.

Many people would say this course "teaches" (parts of) the OCaml and Racket languages, but that is fairly misleading. We will use these languages to learn various paradigms and concepts because they are well-suited to do so. If our goal were just to make you as productive as possible in these languages, the course material would be very different. That said, being able to learn new languages and recognize the similarities and differences across languages is an important goal.

Most of the course will use *functional programming* (both OCaml and Racket are functional languages), which emphasizes immutable data (no assignment statements) and functions, especially functions that take and return other functions. As we will discuss later in the course, functional programming does some things exactly the opposite of object-oriented programming but also has many similarities. Functional programming is not only a very powerful and elegant approach, but learning it helps you better understand all styles of programming.

The conventional thing to do at the very beginning of a course is to motivate the course, which in this case would explain why you should learn functional programming and more generally why it is worth learning different languages, paradigms, and programming-languages concepts. We will largely *delay* this discussion for a few weeks. It is simply too important to cover when most students are more concerned with getting a sense of what the work in the course will be like. More importantly, it is a much easier discussion to have after we have built some shared terminology and experience. Motivation does matter; we promise the delay will be well worth it.

OCaml Expressions and Variable Bindings

So let's just start "learning OCaml" but in a way that teaches core programming-languages concepts rather than just "getting down some code that works." Therefore, pay extremely careful attention to the words used to describe the very, very simple code we start with. We are building a foundation that we will expand very quickly over the first three course units. Do not *yet* try to relate what you see back to what you already know in other languages (Java, Python, C, whatever) as that is likely to lead to struggle. *Do* try to understand exactly why these programs means what they mean.

An OCaml program is a sequence of *bindings*. First the sequence of bindings is *type-checked* and then (assuming the sequence type-checks) the sequence of bindings is *evaluated*. What type (if any) a binding has depends on a *static environment*,[1] which is roughly the types of the preceding bindings in the file. How a binding is evaluated depends on a *dynamic environment*, which is roughly the values of the preceding bindings in the sequence. When we just say *environment*, we usually mean *dynamic environment*. Sometimes *context* is used as a synonym for *static environment*.

There are several kinds of bindings, but for now let's consider only a *variable binding*, which in OCaml has this *syntax*:

```
let x = e;;
```

Here, `let` is a keyword, `x` can be any variable, and `e` can be any *expression*. We will learn many ways to write expressions. The semicolons are optional, and typically omitted, in a file, but necessary in the *read-eval-print loop* to let the *interpreter* know that you are done typing the binding.

We now know a variable binding's syntax (how to write it), but we still need to know its *semantics* (how it type-checks and evaluates). Mostly this depends on the expression `e`. To type-check a variable binding, we use the "current static environment" (the types of preceding bindings) to type-check `e` (which will depend on what kind of expression it is) and produce a "new static environment" that is the current static environment except with `x` having type `t` where `t` is the type of `e`. Evaluation is analogous: To evaluate a variable binding, we use the "current dynamic environment" (the values of preceding bindings) to evaluate `e` (which will depend on what kind of expression it is) and produce a "new dynamic environment" that is the current environment except with `x` having the value `v` where `v` is the result of evaluating `e`.

A *value* is an expression that, "has no more computation to do," i.e., there is no way to simplify it. As described more generally below, `17` is a value, but `8+9` is not. All values are expressions. Not all expressions are values. There are many expressions, an infinite number in fact, that evaluate to the value `17`. One such expressions is `8+9`.

This whole description of what OCaml programs mean (bindings, expressions, types, values, environments)

[1] The word *static* here has a tenuous connection to its use in Java/C/C++, but too tenuous to explain. Its use here is basically in the sense of, "before the program is running"

may seem awfully theoretical or esoteric or pedantic, but it is exactly the foundation we need to give precise and concise definitions for several different kinds of expressions. Here are several such definitions:

- Integer constants:
  - Syntax: a sequence of digits (plus support for negative numbers)
  - Type-checking: type `int` in any static environment
  - Evaluation: to itself in any dynamic environment (integer constants are values)
- Addition:
  - Syntax: `e1+e2` where `e1` and `e2` are expressions
  - Type-checking: type `int` but only if `e1` and `e2` have type `int` in the same static environment, else does not type-check
  - Evaluation: evaluate `e1` to `v1` and `e2` to `v2` in the same dynamic environment and then produce the sum of `v1` and `v2`
- Variables:
  - Syntax: a sequence of letters, underscores, etc.
  - Type-checking: look up the variable in the current static environment and use that type (if it is not in the environment, then the expression does not type-check)
  - Evaluation: look up the variable in the current dynamic environment and use that value
- Conditionals:
  - Syntax is `if e1 then e2 else e3` where `e1`, `e2`, and `e3` are expressions
  - Type-checking: using the current static environment, a conditional type-checks if and only if (a) `e1` has type `bool` and (b) `e2` and `e3` have the same type. The type of the whole expression is the type of `e2` and `e3`.
  - Evaluation: under the current dynamic environment, evaluate `e1`. If the result is `true`, the result of evaluating `e2` under the current dynamic environment is the overall result. If the result is `false`, the result of evaluating `e3` under the current dynamic environment is the overall result.
- Boolean constants:
  - Syntax: either `true` or `false`
  - Type-checking: type `bool` in any static environment
  - Evaluation: to itself in any dynamic environment (boolean constants are values)
- Less-than comparison:
  - Syntax: `e1 < e2` where `e1` and `e2` are expressions
  - Type-checking: type `bool` but only if `e1` and `e2` have type `int` in the same static environment, else does not type-check. (Actually OCaml allows `e1` and `e2` to have any type provided they have the same type, but we will ignore this for now.)
  - Evaluation: evaluate `e1` to `v1` and `e2` to `v2` in the same dynamic environment and then produce `true` if `v1` is less than `v2` and `false` otherwise

Whenever you learn a new construct in a programming language, you should ask these three questions: What is the syntax? What are the type-checking rules? What are the evaluation rules?

Using #use

When using the read-eval-print loop, it is very convenient to add a sequence of bindings from a file.

```
#use "foo.ml";;
```

does just that. Its effect is to include all the bindings in the file `foo.ml` as though you typed them in.

Variables are Immutable

Bindings are *immutable*. Given `let x = 8+9;;` we produce a dynamic environment where `x` maps to `17`. In this environment, `x` will *always* map to `17`; there is no "assignment statement" in OCaml for changing what `x` maps to. That is very useful if you are using `x`. You *can* have another binding later, say `let x = 19;;`, but that just creates a *different environment* where the later binding for `x` *shadows* the earlier one. This distinction will be extremely important when we define functions that use variables.

Function Bindings

Recall that an OCaml 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). We already introduced variable bindings; we now introduce *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 method in languages like Java — 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 do not have things like return statements. A simple example is this function that computes x^y assuming $y \geq 0$:

```
let rec 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 has several possible variations, some of which we discuss here and some of which we can bring up later. To start, here is one form of correct syntax for a function binding:

```
let rec 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`. Notice how the `pow` definition above fits this syntax: n is 2, `x0` is `pow`, `x1` is `x`, `x2` is `y`, `t1` and `t2` are both `int`, and the body `e` is a conditional expression.

The keyword `rec` is optional as syntax but does affect the semantics of a function binding. If present, then the body `e` can use the "function itself" `x0`, i.e., `x0` is (or can be) recursive. If we omit it from our `pow` example, then our function will no longer type-check because the `pow` in the body (where we have `pow (x,y-1)`) no longer type-checks. Or, more confusingly, if there is some *other* `pow` already in the environment, then without `rec`, we would be referring to that other one.

As a matter of style, use `rec` only when it is needed, i.e., when the function body uses the function being defined.

If a function binding has one argument, then we can use the syntax above, which would be:

```
let rec x0 ((x1 : t1)) = e
```

(or without `rec`), but we can in this case leave off a set of parentheses to instead use:

```
let rec x0 (x1 : t1) = e
```

(again, we could also omit `rec` if the function is not recursive). But this is only for one-argument functions. This syntax will NOT work:

```
let rec x0 (x1 : t1, x2 : t2) = e
```

Soon we will stop writing the types `t1`, ..., `tn` because it turns out they are optional and OCaml can figure them out for us. Once we stop writing them, we also do not need the parenthesis, so this will work:

```
let rec x0 (x1, x2) = 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, if `rec` is used, `x0` to `t1 * ... * tn -> t`. 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 is 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 OCaml discussed later in the course. It turns out that in OCaml you almost never have to write down types. As mentioned above, the argument types `t1`, ..., `tn` are also optional in that we can omit them from the syntax and the type-checker will figure them out anyway, but we will leave them in for now to get you a bit more comfortable with OCaml syntax and 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 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 many languages including Java, for preceding bindings, so the order you define functions is very important).

Function calls:

Function bindings are useful only with function calls, 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*, <u>not</u> the one where it is being called. This distinction will not arise right 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 `ans` is `64`:

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

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

let ans = cube (4) (* can also write cube 4 *)
```

Pairs and Other Tuples

Programming languages 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 `(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 functions `fst` and `snd` to access the first and second part of a pair. (These are just library functions implemented in terms of pattern matching, but we can pretend-for-now they are part of the language.) The typing rule for `fst e` or `snd e` should not be a surprise: `e` must have some type that looks like `ta * tb` and then `fst e` has type `ta` and `snd 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 OCaml, 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.

```
let swap (pr : int*bool) =
  (snd pr, fst pr)

let sum_two_pairs ((pr1 : int*int), (pr2 : int*int)) =
  (fst pr1) + (snd pr1) + (fst pr2) + (snd pr2)

let div_mod ((x : int), (y : int)) =
  (x div y, x mod y)
```

```
let sort_pair (pr : int*int) =
  if (fst pr) < (snd pr) then
    pr
  else
    ((snd pr),(fst pr))
```

In fact, OCaml 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)`. However, the type system gets in our way some here: We cannot use `fst` to retrieve the first part of a triple (e.g., 7) nor `snd` to retrieve the second part (e.g., 9). `fst` and `snd` require their arguments to be pairs, not triples. We might like a way to define functions that take "tuples with at least n parts" for some n, but OCaml's type system does not have this flexibility. Every type system has limitations and this is one of OCaml's limitations. It is usually not too frustrating in practice. In any case, until we learn pattern-matching, let us assume there are corresponding functions `fst3`, `snd3` and `thd3` for getting the first, second, and third components of a triple. We will give you the implementations and later in the course you will understand those implementations and therefore not need the functions anymore.

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`. For example:

```
let p = (7,(true,9))
let q = (7,true,9)
let is_true = fst (snd p)
let also_true = (snd3 q)
(* let nope = snd3 p (* does not type-check *) *)
(* let also_nope = fst (snd q) (* does not type-check *) *)
```

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 is not yet known when we are type-checking (e.g., it might depend on a function argument). OCaml 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 OCaml writes as `'a list` (pronounced "quote a list" or "tick 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 three constructs provided by OCaml (two of which are just library functions).

- `e = []` evaluates to `e` to a list (it must have some list type to type-check) and then produces `true` for empty lists and `false` for nonempty lists. So the type of the overall expression is `bool`. In other words, it tests for empty lists.
- `List.hd` is a function that takes a list and returns the first element of a list, *raising an exception* if the list is empty.
- `List.tl` is a function that takes a list and 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:

```
let rec sum_list (xs : int list) =
  if xs = [] then
    0
  else
    List.hd xs + sum_list (List.tl xs)

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

let rec append ((xs : int list), (ys : int list)) =
  if xs = [] then
    ys
  else
    (List.hd xs) :: append (List.tl xs, ys)
```

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 do not have them share any code. We will learn how to fix that later.

```
let rec sum_pair_list (xs : (int * int) list) =
    if xs = [] then
```

```
      0
    else
      fst (List.hd xs) + snd (List.hd xs) + sum_pair_list (List.tl xs)

let rec firsts (xs : (int * int) list) =
    if xs = [] then
      []
    else
      (fst (List.hd xs)) :: (firsts (List.tl xs))

let rec seconds (xs : (int * int) list) =
    if xs = [] then
      []
    else
      (snd (List.hd xs)) :: (seconds (List.tl xs))

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

Syntactic Sugar

Often languages have syntax they do not need because there is another more general way to write the same thing. One example with lists is that we can write `[3+4; if true then 7 else 9]` and it evaluates to the two-element list `[7;7]`. But we can perform exactly the same evaluation with `(3+4)::(if true then 7 else 9)::[]`. And the latter is more general because we can use `::` as in some of the function examples above, with lists we already have of unknown length whereas the `[e1; e2; ...; en]` syntax produces a list of a particular length.

This is the first of many examples we will see of ***syntactic sugar***. We say, "the `e1; e2; ...; en]` syntax is syntactic sugar for `e1 :: e2 :: ... :: en`. It is *syntactic* because we can describe everything about what something menas just be converting it to some other equivalent syntax. It is *sugar* because it makes the language sweeter. The term *syntactic sugar* is widely used. Syntactic sugar is a great way to keep the key ideas in a programming-language small (making it easier to implement) while giving programmers convenient ways to write things.

Another simpler example is considering `e1 && e2` to be syntactic sugar for `if e1 then e2 else false`. While `e1 && e2` is better style because it is common and directly communicates the desired concept, the language does not really *need* it since without it programmers could just use `if e1 then e2 else false`. In terms of semantics, we do not really need special evaluation rules for `e1 && e2` as we can simply say it's a better way of writing `if e1 then e2 else false`.

When we can describe some feature as syntactic sugar for another feature, we will do so to make the language definition simpler. When we cannot, there must be something fundamentally different or new and we can focus on that difference.

Let Expressions

Let-expressions are an absolutely crucial feature that allows for local variables in a very simple, general, and flexible way. Let-expressions are crucial for style and for efficiency. A let-expression lets us have local variables. In fact, it lets us have local *bindings* of any sort, including function bindings. Because it is a kind of expression, it can appear anywhere an expression can.

Syntactically, a let-expression starts like the bindings we have already seen but then has another expression as its *body* after the `in` keyword as in these variations:

```
let x = e1 in e2
let rec x0 ((x1:t1),(x2:t2),...,(xn:tn)) = e in e2
let x0 ((x1:t1),(x2:t2),...,(xn:tn)) = e in e2
```

Because let-expressions are expressions, in each of the examples above, `e2` can also be a let-expression, so we do not need anything "extra" in our language to define multiple local variables together. For example, we can have:

```
let x = 34 + 9 in
let y = x + 14 in
x - y
```

Here the `e2` for the outer let-expression is simply the expression `let y = x + 14 in x - y`.

The type-checking and semantics of a let-expression are much like the semantics of the top-level bindings in our OCaml program. We first evaluate the binding part and use that "one new binding" in the body `e2`. We call the *scope* of a binding "where it can be used," so the scope of a binding in a let-expression is *just* the body of the let-expression (plus recursion in the binding itself if the `rec` keyword is used). The value `e` evaluates to is the value for the entire let-expression, and, unsurprisingly, the type of `e` is the type for the entire let-expression.

For example, this expression evaluates to 7; notice how one inner binding for `x` *shadows* an outer one.

```
let x = 1 in
(let x = 2 in x+1) + (let y = x+2 in y+1)
```

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).

As indicated, let-expressions can bind functions. If a helper function is needed by only one other function and is unlikely to be useful elsewhere, it is good style to bind it locally. For example, here we use a local helper function to help produce the list `[1,2,...,x]`:

```
let countup_from1 (x:int) =
    let rec count ((from:int), (to:int)) =
       if from = to then
         to :: []
       else
         from :: count (from+1,to)
    in
    count (1,x)
```

However, we can do better. When we evaluate a call to `count`, we evaluate `count`'s body in a dynamic environment that is the environment where `count` was defined, extended with bindings for `count`'s arguments. The code above does not really utilize this: `count`'s body uses only `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:

```
let countup_from1_better (x:int) =
    let rec count (from:int) =
       if from = x then
         x :: []
       else
         from :: count (from+1)
    in
    count 1
```

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 many 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:

```
let rec bad_max (xs : int list) =
  if xs = [] then
    0 (* note: bad style; see below *)
  else if (List.tl xs) = [] then
    List.hd xs
  else if List.hd xs > bad_max (List.tl xs) then
    List.hd xs
  else
    bad_max (List.tl xs)
```

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 (List.tl xs)` 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`.

```
let rec good_max (xs : int list) =
  if xs = [] then
    0 (* note: bad style; see below *)
  else if (List.tl xs) = [] then
    List.hd xs
  else
    (* for style, could also use a let-binding for hd xs *)
    let tl_ans = good_max (List.tl xs) in
    if List.hd xs > tl_ans then
      List.hd xs
    else
      tl_ans
```

Options

The previous 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, but we should deal with this case reasonably. One possibility is to raise an exception; you can learn about OCaml exceptions on your own if you are interested before we discuss them later in the course. Instead, let's 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 return an `int list`, using `[]` 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 OCaml 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`. The type of `None` is `'a option` and the type of `Some e` is `t option` if `e` has type `t`.

Given a value, how do you use it? Just like we have `e = []` to see if a list is empty, we have `e = None` which evaluates to `true` if its argument is `None`. Just like we have `List.hd` and `List.tl` to get parts of lists (raising an exception for the empty list), we have `Option.get` to get the value carried by `Some` (raising an exception for `None`).

Using options, here is a better version with return type `int option`:

```
let rec better_max (xs : int list) =
  if xs = [] then
    None
  else
    let tl_ans = better_max (List.tl xs) in
    if tl_ans = None then
      Some (List.hd xs)
    else if Option.get tl_ans <= List.hd xs then
      Some (List.hd xs)
    else
      tl_ans
```

The version above works just fine and is a reasonable recursive function because it does not repeat any potentially expensive computations. But it is both awkward and a little inefficient to have each recursive call except the last one create an option with `Some` just to have its caller access the value underneath. Here is an alternative approach where we use a local helper function for non-empty lists and then just have the outer function return an option. Notice the helper function would raise an exception if called with `[]`, but since it is defined locally, we can be sure that will never happen.

```
let better_max2 (xs : int list) =
  if xs = [] then
    None
  else
    let rec max_nonempty (xs : int list) = (* fine to assume argument nonempty because it is local *)
      if (List.tl xs) = [] (* xs must not be [] *) then
        List.hd xs
      else
         let tl_ans = max_nonempty (List.tl xs) in
           if List.hd xs > tl_ans then
```

```
          List.hd xs
        else
          tl_ans
  in
  Some (max_nonempty xs)
```

Some Other Expressions and Operators

OCaml has all the arithmetic and logical operators you need, but the syntax is sometimes different than in most languages. Here is a brief list of some additional forms of expressions we will find useful:

- `e1 && e2` is logical-and: It evaluates `e2` only if `e1` evaluates to `true`. The result is `true` if `e1` and `e2` evaluate to true. Naturally, `e1` and `e2` must both have type `bool` and the entire expression also has type `bool`. Using `e1 && e2` is generally better style than the equivalent `if e1 then e2 else false`.
- `e1 || e2` is logical-or: It evaluates `e2` only if `e1` evaluates to `false`. The result is `true` if `e1` or `e2` evaluates to true. Naturally, `e1` and `e2` must both have type `bool` and the entire expression also has type `bool`. Using `e1 || e2` is generally better style than the equivalent `if e1 then true else e2`.
- `not e` is logical-negation. `not` is just a provided function of type `bool->bool` that we could have defined ourselves as `let not x = if x then false else true`. In many languages, such expressions are written `!e`, but in OCaml the `!` operator means something else (related to mutable variables, which we use rarely).
- You can compare many values, including integers, for equality using `e1 = e2`.
- Instead of writing `not (e1 = e2)` to see if two numbers are different, better style is `e1 <> e2`. In many languages, the syntax is `e1 != e2`, whereas ML's `<>` can be remembered as, "less than or greater than." As discussed at the end of this unit, `==` and `!=` exist but do *not* behave as you expect, so do *not* use them. (It turns out for numbers and booleans they will behave the same as `=`, so there is less danger in those and some other cases, but still, keep things simple and avoid using them.)
- The other arithmetic comparisons have the same syntax as in most languages: `>`, `<`, `>=`, `<=`.
- OCaml has floating-point numbers, but they are distinct from integers, require different operations, and there is no implicit conversion between ints and floats. For example, you can write `3.0 +. 4.0`, where `+.` is an operator that takes two floating-point numbers and sums them. But you cannot write `3.0 + 4.0`, nor `3.0 +. 4`, etc. You can write `3.0 +. Int.to_float 4`, evaluates to 7 and `(Float.to_int 3.9) + 2`, which evaluates to 5 (because conversion to int rounds-toward-zero rather than rounding to nearest).

Lack of Mutation and Benefits Thereof

In OCaml, there is no way to *change* the contents of a binding, a tuple, or a list. If `x` maps to some value like the list of pairs `[(3,4),(7,9)]` in some environment, then `x` will forever map to that list in that environment. There is no assignment statement that changes `x` to map to a different list. (You can introduce a new binding that shadows `x`, but that will not affect any code that looks up the "original" `x` in an environment.) There is no assignment statement that lets you change the head or tail of a list. And there is no assignment statement that lets you change the contents of a tuple. So we have constructs for building compound data and accessing the pieces, but no constructs for *mutating* the data we have built.

This is a really powerful feature! That may surprise you: how can a language *not* having something be a feature? Because if there is no such feature, then when you are writing *your code* you can rely on *no other code* doing something that would make your code wrong, incomplete, or difficult to use. Having *immutable data* is probably the most important "non-feature" a language can have, and it is one of the main contributions of functional programming.

While there are various advantages to immutable data, here we will focus on a big one: it makes sharing and aliasing irrelevant. Let's reconsider two examples from above before picking on Java (and every other language where mutable data is the norm and assignment statements run rampant).

```
let sort_pair (pr : int*int) =
  if (fst pr) < (snd pr) then
    pr
  else
    ((snd pr),(fst pr))
```

In `sort_pair`, we clearly build and return a new pair in the else-branch, but in the then-branch, do we return a *copy* of the pair referred to by `pr` or do we return an *alias*, where a caller like:

```
val x = (3,4)
val y = sort_pair x
```

would now have `x` and `y` be aliases for the *same* pair? The answer is *you cannot tell* — there is no construct in OCaml that can figure out whether or not `x` and `y` are aliases, and no reason to worry that they might be. *If* we had mutation, life would be different. Suppose we could say, "change the second part of the pair `x` is bound to so that it holds 5 instead of 4." Then we would have to wonder if `snd y` would be 4 or 5.

In case you are curious, we would expect that the code above would create aliasing: by returning `pr`, the `sort_pair` function would return an alias to its argument. That is more efficient than this version, which would create another pair with exactly the same contents:

```
let sort_pair (pr : int*int) =
  if (fst pr) < (snd pr) then
    ((fst pr),(snd pr))
  else
    ((snd pr),(fst pr))
```

Making the new pair `((fst pr), (snd pr))` is bad style, since `pr` is simpler and will do just as well. Yet in languages with mutation, programmers make copies like this all the time, exactly to prevent aliasing where doing an assignment using one variable like `x` causes unexpected changes to using another variable like `y`. In OCaml, no users of `sort_pair` can ever tell whether we return a new pair or not.

Our second example is our elegant function for list append:

```
let rec append ((xs : int list), (ys : int list)) =
  if xs = [] then
    ys
  else
    (List.hd xs) :: append (List.tl xs, ys)
```

We can ask a similar question: Does the list returned *share* any elements with the arguments? Again the answer does not matter because no caller can tell. And again the answer happens to be yes: we build a new list that "reuses" all the elements of `ys`. This saves space, but would be very confusing if someone could later mutate `ys`. Saving space is a nice advantage of immutable data, but so is simply not having to worry about whether things are aliased or not when writing down elegant algorithms.

In fact, `List.tl` itself thankfully introduces aliasing (though you cannot tell): it returns (an alias to) the tail of the list, which is always "cheap," rather than making a copy of the tail of the list, which is "expensive" for long lists.

The `append` example is very similar to the `sort_pair` example, but it is even more compelling because it is hard to keep track of potential aliasing if you have many lists of potentially large lengths. If I append `[1,2]` to `[3,4,5]`, I will 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]`?

In the analogous Java program, this is a crucial question, which is why Java programmers *must obsess* over when references to old objects are used and when new objects are created. There are times when obsessing over aliasing is the right thing to do and times when avoiding mutation is the right thing to do — functional programming will help you get better at the latter.

For a final example, the following Java is the key idea behind an actual security hole in an important (and subsequently fixed) Java library. Suppose we are maintaining permissions for who is allowed to access something like a file on the disk. It is fine to let everyone see *who has permission*, but clearly only those that do have permission can actually use the resource. Consider this wrong code (some parts omitted if not relevant):

```
class ProtectedResource {
   private Resource theResource = ...;
   private String[] allowedUsers = ...;
   public String[] getAllowedUsers() {
      return allowedUsers;
   }
   public String currentUser() { ... }
   public void useTheResource() {
      for(int i=0; i < allowedUsers.length; i++) {
         if(currentUser().equals(allowedUsers[i])) {
             ... // access allowed: use it
             return;
         }
      }
      throw new IllegalAccessException();
   }
}
```

Can you find the problem? Here it is: `getAllowedUsers` returns an alias to the `allowedUsers` array, so any user can gain access by doing `getAllowedUsers()[0] = currentUser()`. Oops! This would not be possible if we had some sort of array in Java that did not allow its contents to be updated. Instead, in Java we often have to remember to *make a copy*. The correction below shows an explicit loop to show in detail what must be done, but better style would be to use a library method like `System.arraycopy` or similar methods in the `Arrays` class — these library methods exist because array copying is necessarily common, in part due to mutation.

```
public String[] getAllowedUsers() {
   String[] copy = new String[allowedUsers.length];
   for(int i=0; i < allowedUsers.length; i++)
```

```
      copy[i] = allowedUsers[i];
   return copy;
}
```

Structural Equality and = versus ==

OCaml has two equality operators that take two expressions of the same type and produce a boolean:

- `e1 = e2` compares two values *structurally* (see explanation below) and is a fine thing to use when structural comparison is what you want. The operator `e1 <> e2` is simply defined as `not (e1 = e2)`, so it is also fine to use.
- `e1 == e2`, again as explained below, is only somewhat specified for immutable data and is "basically a bad idea" that "is really reference equality but without promising to be," so you should not use it. The operator `e1 != e2` is simply defined as `not (e1 == e2)`, so you should not use it either.

Understanding the difference between `=` and `==` is interesting and relevant to the preceding discussion on aliasing.

Let us first describe what `e1 = e2` means. It evaluates `e1` and `e2` to values and then compares the structure (basically the shape, including recursively the shape of its pieces). These expressions all evaluate to `true`:

- `34 = 34`
- `(1,2) = (1,2)`
- `[3;4;5] = [3;4;5]`
- `[] = []`
- `None` = `None`
- `Some 7` = `Some 7`

The type-checker requires the two expressions in `x = y` to have the same type, which makes sense because it would be rare for expressions of different types to possibly have the same structure.

Note we have used structural equality with our tests like `x = []`. This evaluates `x` to a value and compares that value to `[]`. Any non-empty list will cause this expression to evaluate to `false`. Structural equality is just more general than that. We can ask `[3;4;5;7] = foo` and get `true` if and only if `foo` is exactly the four element list with 3, 4, 5, and 7 in that order.

Structural equality can sometimes be useful, but it is a bit strange that this one particular notion of equality is built into OCaml. Notice this definition is about the contents/shape of values and has nothing to do with the possibility of aliasing. However, consider `x = y` where `x` and `y` happen to be aliases for the same immutable data. Even though we have no way of *knowing* they are aliases, if they are, then surely `x = y` will evaluate to `true`: Any value must have the same contents/shape as itself after all! On the other hand, if `x` and `y` are not aliases, then `x = y` could evaluate to `true` or `false`.

What then is `e1 == e2`? To be honest, it is reference equality though the language definition, wisely, goes to great pains not to promise that. The idea is that it evaluates to `true` if and only if `e1` and `e2` are aliases. If the type of `e1` and `e2` is mutable (including having some mutable parts even if other parts are immutable), then aliasing matters and it makes sense to have some way to check for aliasing. Also, checking reference equality is very efficient, with no need to traverse the data comparing contents/shape.

But if OCaml promised `==` meant reference equality even for immutable data, then some of the advantages of "clients cannot tell if there are aliases" would go away. We still would not have the dangers of mutating alised data, but both programmers and the language implementation ("the compiler") would have to be very careful about creating copies versus aliases because of the possibility of clients using `==` even when aliasing should not matter. So OCaml's designers reached a pragmatic compromise (or you might call it a hack). The official definition of `==` promises almost nothing for immutable data. *All* it guarantees is that if `e1==e2` evaluates to `true`, then `e1=e2` evaluates to `true`. Any code that assumes more than that about the behavior of `==` on immutable data is wrong and risks getting surprising results. With that definition (or "definition") of `==`, we succeed at keeping any notion of aliasing out of the language definition, but at the same time it makes `==` essentially useless for immutable data, since you should just use `=`. It does mean that if you accidentally use `==` in some common situations like for numbers or `e1 == []`, it will happen to work, but for tuples, non-empty lists, etc., it may not necessarily have the behavior you expect.