# A Concepts-Focused Introduction to Functional Programming Using Standard ML Course Notes

Dan Grossman

DRAFT of August 12, 2010

## Contents



## 1 Meta-Introduction: What These Notes Are

These notes cover approximately ten hours of lecture material on functional programming. They assume only that students have completed an introductory programming sequence (CS1 and CS2) in Java, though another object-oriented language would proably be fine and students without object-oriented background would miss out only on occasional discussions that contrast functional (FP) with object-oriented (OOP).

These notes were extracted from a 10-week sophomore-level course on programming languages that I have taught several times at the University of Washington. The full course covers *many* additional topics while using Scheme and Ruby in addition to Standard ML. See http://www.cs.washington.edu/education/courses/cse341/08sp/ for my most recent offering.

The purpose of extracting a ten-hour introduction is to provide other instructors a starting point for covering (some of) the most essential pieces in a coherent fashion without assuming prior knowledge. Sticking within the time bound is painful to say the least, but what "made the cut" is, in my opinion, of the highest value. This is only one of a few sample ten-hour modules that the SIGPLAN Education Board is making available. This is a virtue, recognizing

that there are multiple successful ways to demonstrate the value of teaching functional programming early in an undergraduate curriculum.

Compared to other approaches I am aware of, "my way" has the following characteristics:

- The presentation is informal yet precise. It discusses the rules for type-checking and evaluating expressions in a compositional fashion. It builds up from simple variable bindings to patterns, functions, and basic modules in a way that does not "hand-wave." It emphasizes a few core concepts while dismissing other ones as syntactic sugar (e.g., every function takes exactly one argument).
- It emphasizes datatypes and pattern-matching and how this approach to "one-of types" (better known among PL experts as sum types) complements the OOP approach of subclasses. Pattern-matching comes *before* higher-order functions though no time is compromised in the latter essential topic.
- The presentation of very basic ML modules emphasizes the software-engineering benefits of hiding values and, much more interestingly, the representation of a type.

Given the limitation of only ten hours, these notes give limited "air time" to parametric polymorphism and type inference, two key characteristics of Standard ML. The focus is also not on writing useful or large programs, but rather on writing programs that demonstrate the core concepts of mutation-free and loop-free programming (the former is much more important), pattern-matching, and higher-order functions.

When I teach the course, I recommend the Ullman textbook *Elements of ML Programming, ML'97 Edition* even though, in fact because, I follow these notes rather than the style and order of presentation in the text. Some students do not need this secondary reference.

Alongside these notes I have also prepared homework problems and sample exam questions, as well as slides and code that I use when lecturing. The slides and code may be difficult for instructors other than myself to use, but they may be a useful starting point. (I give most of the lectures primarily using a text editor and writing code.) However, the code includes many more examples than these notes. The slides correspond to the sections in the notes but they do *not* all take the same amount of time (i.e., they do not exactly line up on 1-hour boundaries). Finally, separate from these notes is a document on the reasons I give to students for learning functional programming. There are other equally good justifications available; this is my personal take on the issue.

Not "included" is a primer on installing an ML implementation, using a text editor (such as emacs), saving files, launching the interpreter, etc. These details will probably differ based on local issues. I point to various "getting started" guides that I and other instructors have used, but these (and only these) are specific to my institution.

# 2 ML Variable Bindings and The Essential Pieces of a Programming Language

The purpose of these notes is to use the Standard ML programming language to learn some of the key ideas, definitions, and idioms of *functional programming*, a style of programming that favors using recursion and functions (kind of like methods but without an enclosing object) as values and avoids mutations (assignment statements) and explicit loops. While one program in a functional *style* in many languages, it is much more elegant to do it in a language that encourages this style.

Because these notes assume no prior knowledge of Standard ML, everything is explained from the beginning, assuming only that you have programmed in another language, probably Java, before. However, you are strongly cautioned *not* to try to describe everything you learn in terms of Java. At first everything will seems completely different, like you are learning to program again from the beginning. This is a good thing! That way, you can learn the concepts without your prior knowledge of Java details getting in the way. Then later we can draw some important contrasts with object-oriented programming — only after we understand enough about Standard ML and functional programming to draw appropriate conclusions.

Pay extremely careful attention to the words used to describe the very, very simple code in this section. We are building a foundation that we will expand very quickly over the next few lectures. If you skim the English and guess what the code does, your guess will be correct, but you will miss the point and have trouble in later sections.

An ML program is a sequence of *bindings*. Each binding gets *type-checked* and then (assuming it type-checks) *evaluated*. What type (if any) a binding has depends on a *context*, which is roughly the types of the preceding bindings

in the file. How a binding is evaluated depends on an *environment*, which is roughly the values of the preceding bindings in the file.

Later sections will introduce several interesting kinds of bindings, but this section considers only *variable bindings*. A variable binding in ML has this *syntax*:

```
val x = e;
```

Here, `val` is a keyword, `x` can be any variable, and `e` can be any *expression*. We will learn many ways to write expressions. The semicolon is optional in a file, but necessary in the *read-eval-print loop* (where you can type in programs to have them run) to let the *interpreter* (the thing that runs your program) know that you are done typing the binding.

We now know a variable binding's syntax, but we still need to know how it type-checks and evaluates. Mostly this depends on the expression `e`. To type-check a variable binding, we use the "current context" (the types of preceding bindings) to type-check `e` (which will depend on what kind of expression it is) and produce a "new context" that is the current context 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 environment" (the values of preceding bindings) to evaluate `e` (which will depend on what kind of expression it is) and produce a "new 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 we'll see soon, `17` is a value, but `8+9` is not. All values are expressions, not all expressions are values.

This whole description of what ML programs mean (bindings, expressions, types, values, contexts, environments) may seem awfully theoretical or esoteric, but it's exactly the foundation we need to give precise and concise definitions for several different kinds of expressions. Without further ado, here are several kinds of expressions:

- Integer constants: Syntax is a sequence of digits. Type-checking is type `int` in any context. Evaluation is to itself in any environment (it's a value).
- Addition: Syntax is `e1+e2` where `e1` and `e2` are expressions. Type-checking is type `int` but only if `e1` and `e2` have type `int` (using the same context). Evaluation is to evaluate `e1` to `v1` and `e2` to `v2` (using the same environment) and then produce the sum of `v1` and `v2`.
- Variables: Syntax is a sequence of letters, underscores, etc. Type-checking is to look up the variable in the context and use that type. Evaluation is to look up the variable in the environment and use that value.
- Conditionals: Syntax is `if e1 then e2 else e3` where `e1`, `e2`, and `e3` are expressions. Type-checking under a context is to type-check `e1`, `e2`, and `e3` under the same context. `e1` must have type `bool`. `e2` and `e3` must have the same type, call it `t`. Then the type of the whole expression is `t`. Evaluation under ane environment is to evaluate `e1` under the same environment. If the result is `true`, the result of evaluating `e2` under the same environment is the overall result. If the result is `false`, the result of evaluating `e3` under the same environment is the overall result.
- Boolean constants: Syntax is `true` or `false`. Type-checking is `bool` in any context. Evaluation is to itself in any environment (it's a value).
- Less-than comparison: Syntax is `e1 < e2` where `e1` and `e2` are expressions. Type-checking is type `bool` but only if `e1` and `e2` have type `int` (using the same context). Evaluation is to evaluate `e1` to `v1` and `e2` to `v2` (using the same environment) and then produce `true` if `v1` is a smaller number than `v2` and false otherwise.

When using the read-eval-print loop, it's very convenient to add a sequence of bindings from a file. `use "foo.sml"` does just that. Its type is `unit` and its result is `()` (the only value of type `unit`), but its effect is to extend the context and environment with all the bindings in the file `"foo.sml"`.

Bindings are *immutable*. Given `val x = 8+9;` we produce an environment where `x` maps to `17`. In this environment, `x` will *always* map to `17`; there is no "assignment statement" in ML for changing what `x` maps to. That is very useful if you are using `x`. You *can* have another binding later, say `val 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.

While we haven't even learned enough of ML yet to really think of it as a programming language (we really need functions to do anything interesting), we have enough to list the essential "pieces" necessary for defining and learning a programming language:

- Syntax: How do you write the various parts of the language?
- Semantics: What do the various language features mean? For example, how are expressions evaluated?
- Idioms: What are the common approaches to using the language features to express computations?
- Libraries: What has already been written for you? How do you do things you couldn't do without library support (like access files)?
- Tools: What is available for manipulating programs in the language (compilers, read-eval-print loops, debuggers, ...)

The focus in these notes is on *semantics* and *idioms*. While syntax is important — you have to get it right when you program — it is not particularly interesting. Libraries and tools are important when writing "real" programs, but they are less central to the core ideas in a programming language. That can leave the wrong impression that ML is "silly" or "theoretical" when it's really the case that libraries and tools are just less "intellectually interesting" when learning the similarities and differences among programming languages.

# 3 Functions, Pairs, Lists, and the Benefits of Mutation-Freedom

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. We will now learn how to write 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.

## 3.1 Functions

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 worth learning about. 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 Sections 5 and 6. (The way we are using pair-reading constructs like `#1` in this section requires 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 $1 \leq i \leq n$, `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 much yet, 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)
```

## 3.2 Pairs

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.

## 3.3 Lists

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.

## 3.4 Recursive Functions Over Lists — And A Key Benefit of No Mutation

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 we 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.

# 4 Let-Bindings and Options

We do not yet have a way to create local variables, which are essential for at least two reasons — writing code in good style and writing code that uses efficient algorithms. The main topic in this section is ML's let-expressions, which is how we create local variables. Let-expressions are both simpler and more powerful than local variables in many other languages: they can appear anywhere and can have any kind of binding. This is a nice example of elegant language design, not restricting where a useful feature can be used.

Syntactically, a let-expression is:

```
let b1 b2 ... bn in e end
```

where each `bi` is a binding (so far we have seen variable bindings and function bindings) and `e` is an expression.

The type-checking and semantics of a let-expression is much like the semantics of the top-level bindings in our ML program. We evaluate each binding in turn, creating a larger context/environment for the subsequent bindings. So we can use all the earlier bindings for the later ones, and we can use them all for `e`. We call the *scope* of a binding "where it can be used", so the scope of a binding in a let-expression is the later bindings in that let-expression and the "body" of the let-expressions (the `e`). The value `e` evaluates to is the value for the entire let-expression.

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

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

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).

Let-expressions can bind functions too, since functions are just another kind of binding. If a helper function is needed by only one other function and is unlikely to be useful elsewhere, it's good style to bind it locally. For example, here we use a local helper function to help produce the list `[1,2,...,x]`:

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

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

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

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:

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

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(tl(lst))` 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`.

```
fun good_max (lst : int list) =
    if null lst
    then 0
    else if null (tl(lst))
    then hd(lst)
    else
        (* for style, could also use a let-binding for hd(lst) *)
        let val tl_ans = good_max(tl(lst))
        in
            if hd(lst) > tl_ans
            then hd(lst)
            else tl_ans
        end
```

This 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. So to properly deal with that fact, it would be better to 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 returning a list of integers, using the empty list 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 ML 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`.

Given a value, how do you use it? Just like we have `null` to see if a list is empty, we have `isSome` which evaluates to `false` if its argument is `NONE`. Just like we have `hd` and `tl` to get parts of lists (raising an exception for the empty list), we have `valOf` to get the value carried by `SOME` (raising an exception for `NONE`).

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

```
fun better_max (lst : int list) =
    if null lst
    then NONE
```

```
else
    let val tl_ans = better_max(tl(lst))
    in if isSome tl_ans andalso valOf tl_ans > hd(lst)
       then tl_ans
       else SOME (hd(lst))
    end
```

# 5 Each-Of And One-Of Types (Records and Datatypes)

Programming languages need ways to describe data so that we can write programs that operate on that data. We need *base types* such as `int`, `string`, `bool`, and `unit`. We also need ways to build *compound types* from simpler types, such as `tuples` (pairs are 2-tuples, triples are 3-tuples, etc.), `lists`, and even function types (`->`). This section is about record types, which are much like tuples, and `datatype` bindings, which are probably unlike anything you have seen before. Conceptually, this lecture is about describing data in terms of *and* ("each-of" types), *or* ("one-of" types), and *self-reference* ("recursive" types).

Syntactically, an ML record is constructed via `{f1=e1, ..., f2=en}` where `f1`, ... `fn` are *field names* (sequences of letters) and `e1`, ..., `en` are expressions. Exactly like tuples, the evaluation rules are to evaluate the expressions to values and a record with values in all fields is itself a value. The only difference from tuples is that the order of the fields does not matter; we use field names instead of "position" to determine which field is which. Unlike in many languages, we do not need any sort of type declaration before we use a record; we can just use whatever field names we want. The type of a record describes what fields it has and what types those fields have. Again, field order does not matter; in fact, the read-eval-print loop always alphabetizes field names before it prints a type. To extract a field `foo` from a record, we can use `#foo e` where `e` evaluates to a record.

Tuples are, in fact, so much like records that they actually *are* records. When you write `(7,"hi")`, that is exactly the same as writing `{1=7, 2="hi"}`. The type `int*string` is just another way of writing `{1:int, 2:string}`. Now, using the tuple syntax is cleaner and better style, but it is still an elegant language design to have it really just be another language feature (records). That way, the designer and implementor of the language has less work to do; we can completely define how tuples behave by explaining how they are really just a different syntax for particular records. This is our first example of the idea of *syntactic sugar*, a piece of the language that is just a prettier way of writing something already in the language. It is syntactic because it is just a different way of writing something, and it is sugar because it makes the language sweeter.

Let us now return to building compound types in terms of "each of", "one of", and "self reference". Each-of types are often the simplest for people to understand. They are like records, or Java classes with only fields. Given types `t1`, `t2`, `t3`, we make some new type `t` that "has" each of `t1`, `t2`, *and* `t3`. "One of" types are just as important; it is very common to have a type `t` that "has" either `t1`, `t2`, *or* `t3`. In object-oriented languages like Java, one uses subclassing to implement "one of" types as a later section will demonstrate. (In brief, each subclass is a different possibility of what a value of the subclass might actually have.) Finally, self-reference is necessary to describe recursive data structures like lists and trees.

We have actually seen examples of each kind of compound type in earlier lectures. `int * bool` is an each-of type; each value of this type has an `int` and a `bool`. `int option` is a one-of type; each value of this type either has an `int` or it does not. `int list` is a compound type using all three notions: a value of type `int list` has either no data (the empty list) *or* it has an `int` *and* another `int list` (notice the self-reference).

In ML, you use a `datatype` binding to create a new one-of type that comes with *constructors* and *patterns* for building and accessing values of the new type, respectively. A silly example of such a binding is:

```
datatype mytype = TwoInts of int * int
                | Str of string
                | Pizza
```

This binding creates a new type `mytype` with 3 *variants*. A value of type `mytype` is created in one of 3 ways: (1) By using the constructor `TwoInts` on a pair of ints, (2) By using the constructor `Str` on a string, or (3) By using the constructor `Pizza` on nothing. In this sense, constructors are functions (if they take arguments) or constants (if they do not). In our example, the datatype binding adds to the environment/context: (1) `TwoInts` of type `int * int -> mytype`,

(2) `Str` of type `string -> mytype`, and (3) `Pizza` of type `mytype`. As with any other function, an argument to a constructor can be any expression of the correct type.

So constructors give us a way to make values of a datatype such as `mytype`, but we also need a way to use such values after we make them. Doing so requires two kinds of operations: (1) tests to see which variant we have and (2) operations that extract the values that a particular variant has. We have already seen such operations for options and lists. The functions `null` and `isSome` are variant tests. The functions `valOf`, `hd`, and `tl` are the data extractors. Notice they raise exceptions if applied to a value of an unexpected variant.

A `datatype` binding does not directly create variant tests and data extractors. Instead, ML uses *pattern matching*, an elegant and convenient feature that combines the variant test and data extraction. By combining them, the type-checker can check that you do not forget any cases or duplicate any cases.

Pattern-matching is done with a case-expression. The syntax is

```
case e0 of
  p1 => e1
| ...
| pn => en
```

where `e0`, `e1`, ..., `en` are expressions and `p1`, ..., `pn` are *patterns*. We have not seen patterns before. They look a lot like expressions syntactically, but they are more restrictive and their semantics is very different.

Here is a function using pattern matching and our previous datatype example:

```
fun f2 x =  (* f2 has type mytype -> int *)
    case x of
        Pizza => 3
      | TwoInts(i1,i2) => i1 + i2
      | Str s => 8
```

The semantics of a case-expression is to evaluate the expression between the `case` and `of` to some value `v` and then proceed through the patterns in order, finding the first one that *matches*. For now, a pattern-matches if the constructor in it is the same as the constructor for the value `v`. For example, if the value is `TwoInts(7,9)`, then the first pattern would not match and the second one would. We evaluate the expression to the right of the matching pattern (on the other side of the `=>`) and that result is the result of the whole case-expression. The other expressions are not evaluated.

We have not yet explained the data-extraction part of pattern-matching. Since `TwoInts` has two values it "carries", a pattern for it can (and, for now, must) use two variables (the `(i1,i2)`). As part of matching, the corresponding parts of the value `v` (continuing our example, the 7 and the 9) are bound to `i1` and `i2` in the environment used to evaluate the corresponding right-hand side (the `i1+i2`). In this sense, pattern-matching is like a let-expression: It binds variables in a local scope.

It turns out case-expressions are a more general and powerful form of conditional expressions. Like `if e1 then e2 else e3`, they use their first expression to decide which other expression is used to produce the answer. Type-checking is also similar: the different branches must all have the same type and that is the type of the whole expression. The key additional power is that patterns can bind variables, extending the context/environment for the corresponding branch.

In fact, conditional expressions are really just syntactic sugar for case-expressions and a predefined datatype. Among the bindings evaluated before your program starts is:

```
datatype bool = true | false
```

We can then treat `if e1 then e2 else e3` as a syntactic shortcut for
`case e1 of true => e2 | false => e3`.

As we will see in the next two lectures, pattern-matching can be used for much more than just datatype values. However, its basic semantics of comparing a pattern against a value and binding variables for the corresponding branch will not change.

Let us consider two less silly examples of datatype bindings. This first one defines identities as either a social-security number or (presumably for when someone does not have a number), a first name, optional middle name, and last name:

```
datatype id = SSN of int
            | Name of string * (string option) * string
```

Whenever a data definition has a clear "or" in it, datatypes are the way to go. Novice programmers often abuse "each of" types when they want "one of types". For example you might see bad code like this (in any language; we just use ML for the example):

```
(* If ssn is -1, then the other fields are the name, otherwise ssn is
   the identity and the other fields should be ignored *)
type bad_id = {ssn: int, first : string, middle : string option, last : string}
```

On the other hand, if a name record is supposed to have an optional social-security number *and* a (non-optional) name, then an "each of" type is the right thing:

```
type name_record = {ssn: int option, first : string, middle : string option, last : string}
```

Our second example is a data definition for arithmetic expressions containing constants, negations, and additions.

```
datatype exp = Constant of int
             | Negate of exp
             | Add of exp * exp
```

Thanks to the self-reference, what this data definition really describes is a *tree* where the leaves are integers and the internal nodes are either negations with one child or additions with two children. We can write a function that takes an `exp` and evaluates it:

```
fun eval e =
    case e of
        Constant i => i
      | Negate e2  => ~ (eval e2)
      | Add(e1,e2) => (eval e1) + (eval e2)
```

So this function call evaluates to 15:

```
eval (Add (Constant 19, Negate (Constant 4)))
```

Notice how constructors are just functions that we call with other expressions (often other values built from constructors).

For practice, other functions you could write that process `exp` values could compute:

- The largest constant in an expression.
- A list of all the constants in an expression (use list append).
- The number of addition expressions in an expression.

# 6 More Pattern Matching, One-Argument Functions, Deep Patterns

We can summarize what we know about datatypes and pattern matching from the previous section as follows: The binding

```
datatype t = C1 of t1 | C2 of t2 | ... | Cn of tn
```

introduces a new type `t` and each constructor `Ci` is a function of type `ti->t`. One omits the "`of ti`" for a variant that "carries nothing." To "get at the pieces" of a `t` we use a case expression:

```
case e of p1 => e1 | p2 => e2 | ... | pn => en
```

A case expression evaluates `e` to a value `v`, finds the first pattern `pi` that *matches* `e`, and evaluates `ei` to produce the result for the whole case expression. So far, patterns have looked like `Ci(x1,...,xn)` where `Ci` is a constructor of type `t1 * ... * tn -> t` (or just `Ci` if `Ci` carries nothing). Such a pattern matches a value of the form `Ci(v1,...,vn)` and binds each `xi` to `vi` for evaluating the corresponding `ei`.

A recursive datatype definition lets us define a kind of tree where the different variants are different kinds of tree nodes (with potentially different numbers of children). Recall this definition defines trees that describe arithmetic expressions.

```
datatype exp = Constant of int
             | Negate of exp
             | Add of exp * exp
```

It turns out that options and lists are essentially just datatypes and stylistically one should generally use pattern-matching for them, *not* the functions `null`, `hd`, `tl`, `isSome`, and `valOf` we saw previously. (We used them because we had not learned pattern-matching yet.)

For options, the constructors are `NONE`, which carries nothing, and `SOME`, which carries one value of any type you want. For example, `SOME 3` has type `int option` and `NONE` can have type `int option` or `(int * int) option` or `'a option`. You can define your own datatypes that work for "any type" like this, but we won't discuss how here.

Lists are similar except the constructor syntax is unusual. The constructor `[]` carries nothing. The constructor `::` carries two things, one of type `t` and one of type `t list` for any `t`, and `::` is written between the two things.

Since `NONE`, `SOME`, `[]`, and `::` are constructors, we use them in patterns just like any other constructors. For example:

```
fun inc_or_zero intoption =
    case intoption of
        NONE => 0
      | SOME i => i+1
fun sum_list intlist =
    case intlist of
        [] => 0
      | hd::tl => hd + sum_list tl
fun append (l1,l2) =
    case l1 of
        [] => l2
      | hd::tl => hd :: append(tl,l2)
```

There are several reasons to prefer pattern-matching over functions that test (like `null`) and extract values (like `hd` and `tl`). With a case expression, it's easier to check for missing and redundant cases; in fact, the type-checker does it for you. The syntax is more concise. If you want to define functions like `null` or `hd`, you can easily do so using pattern-matching. (This is exactly what the standard library does. The functions are useful for passing to other functions, as we will see in the sections on higher-order functions.)

So far we have used pattern-matching for one-of types, but we can use them for each-of types also. Given a record value `{f1=v1,...,fn=vn}`, the pattern `{f1=x1,...,fn=xn}` matches and binds `xi` to `vi`. As you might expect, the order of fields in the pattern does not matter. As before, tuples are syntactic sugar for records, so the tuple value `(v1,...,vn)` matches the pattern `(x1,...,xn)`. So we could write this function for summing the three parts of an `int * int * int`:

```
fun sum_triple triple =
    case triple of
      (x,y,z) => z + y + x
```

However, case expressions with one branch are strange-looking and poor style. Instead, a `val` binding can have a pattern in it. This is just syntactic sugar for a one-branch case-expression; the semantics is to do the match and raise an exception if it fails. So better style would be:

```
fun sum_triple_better triple =
    let val (x,y,z) = triple
    in
        x + y + z
    end
```

But we can do even better. Notice that `sum_triple` and `sum_triple_better` both have type `int * int * int -> int`; they take a triple of ints and produce an int. This function has the same type and is the best style:

```
fun sum_triple_best (x,y,z) =
    x + y + z
```

Previously we said such a function took three arguments. That's how we think and talk about it typically, but it turns out *every function in ML takes one argument*. It is just that we can put a *pattern* where the argument goes and the semantics is to match the value passed to the function with the pattern and use the new bindings in the function body. So `sum_triple_best` is really using syntactic sugar for how `sum_triple_better` is written,

So using a tuple-pattern as a function's one argument (recall every function takes exactly one argument) is "just an idiom" albeit a very, very common one. This is an elegant and useful thing. For example, one might usually have calls like `sum_triple_best(1,2,3)`, but now we know this is just calling `sum_triple_best` with the triple `(1,2,3)`, which matches the pattern `(x,y,z)`, binding `x` to 1, `y` to 2, and `z` to 3. Moreover, we can call `sum_triple_best` with any expression that evaluates to an `int * int * int`. For example, if `g` has type `bool -> int * int * int`, then we can write `sum_triple_best(g(true))`. You cannot do something like that as conveniently in Java or C because functions there actually take multiple arguments and there is no way to write a function like `g` that returns a, "collection of arguments for another function."

As our final useful generalization of pattern-matching, it turns out that where we have been putting variables in our patterns we can also put other patterns. This leads to an elegant recursive definition of pattern-matching. Here are some parts of the recursive definition:

- A variable pattern (`x`) matches any value `v` and introduces one binding (from `x` to `v`).
- A wildcard pattern (`_`) matches any value `v` and introduces no bindings.
- The pattern `C` matches the value `C`, if `C` is a constructor that carries no data.
- The pattern `C p` where `C` is a constructor and `p` is a pattern matches a value of the form `C v` (notice the constructors are the same) if `p` matches `v` (i.e., the nested pattern matches the carried value).
- The pattern `(p1,p2)` matches a pair of values `(v1,v2)` if `p1` matches `v1` and `p2` matches `v2`. It introduces all the bindings introduced by either recursive pattern match.
- ...

This recursive definition extends our previous understanding in two interesting ways. First, for a constructor `C` that carries multiple arguments, we do not have to write patterns like `C(x1,...,xn)` though we often do. We could also write `C x`; this would bind `x` to the tuple that the value `C(v1,...,vn)` carries. What is really going on is that all constructors take 0 or 1 arguments, but the 1 argument can itself be a tuple. So `C(x1,...,xn)` is really a nested pattern where the `(x1,...,xn)` part is just a pattern that matches all tuples with $n$ parts.

Second, and more importantly, we can use nested patterns instead of nested case expressions when we want to match only values that have a certain "shape." For example, the pattern `x::(y::z)` would match all lists that have at least 2 elements, whereas the pattern `x::[]` would match all lists that have exactly one element. Both examples use one constructor pattern inside another one.

The nested patterns can be for different types. For example, the pattern `(_,(_,x),_)::tl` would match a non-empty list where each element was a triple where the second element was a pair. It would bind the list's head's middle element's second element to `x` and the list's tail to `tl`.

Elegant examples of using nested patterns includ "zipping" and "unzipping" lists:

```
exception BadTriple

fun zip3 list_triple =
    case list_triple of
        ([],[],[]) => []
      | (hd1::tl1,hd2::tl2,hd3::tl3) =>
                (hd1,hd2,hd3)::zip3(tl1,tl2,tl3)
      | _ => raise BadTriple

fun unzip3 lst =
```

```
    case lst of
        [] => ([],[],[])
      | (a,b,c)::tl =>
        let val (l1,l2,l3) = unzip3(tl)
        in
            (a::l1,b::l2,c::l3)
        end
```

Here is another example where we check that a list of numbers is sorted:

```
fun nondecreasing intlist =
    case intlist of
        [] => true
      | x::[] => true
      | hd::(next::tl) => (hd <= next andalso nondecreasing (next::tl))
```

Lastly, this silly example of the last idiom computes the sign that would result from multiplying two numbers (without actually doing the multiplication).

```
datatype sign = P | N | Z

fun multsign (x1,x2) =
  let fun sign x = if x=0 then Z else if x>0 then P else N
  in
      case (sign x1,sign x2) of
          (Z,_) => Z
        | (_,Z) => Z
        | (P,P) => P
        | (N,N) => P
        | _     => N
  end
```

Ending a case-expression with the pattern `_`, which matches everything, is somewhat controversial style. The type-checker certainly won't complain about an inexhaustive match, which can be bad if you actually did forget some possibly that you are accidentially covering with your "default case." So more careful coding of the example above would replace the last case with two cases with patterns `(P,N)` and `(N,P)`, but the code as written above is also okay.

Returning to how general pattern-matching is, it turns out that every val-binding and function argument is really a pattern. So the syntax is actually `val p = e` and `fun f p = e` where `p` is a pattern. In fact, many ML programmers use a form of function-binding where you can have multiple cases by repeating the function name and using a different pattern:

```
fun f p1 = e1
|   f p2 = e2
...
|   f pn = en
```

This is just syntactic sugar for:

```
fun f x =
   case x of
     p1 => e1
   | p2 => e2
...
   | pn => en
```

These notes do not use the repeated function-binding style, but this is a matter of personal taste.

# 7 Tail Recursion

To understand tail recursion and accumulators, consider these two functions for summing the elements of a list:

```
fun sum_list_1 lst =
    case lst of
        [] => 0
      | i::lst1 => i + sum_list_1 lst1

fun sum_list_2 lst =
    let fun f (lst,acc) =
          case lst of
                [] => acc
              | i::lst1 => f(lst1,i+acc)
    in
        f(lst,0)
    end
```

Both functions compute the same results, but `sum_list_2` is more complicated, using a local helper function that takes an extra argument, called `acc` for "accumulator." In the base case of `f` we return `acc` and the value passed for the outermost call is 0, the same value used in the base case of `sum_list_1`. This pattern is common: The base case in the non-accumulator style becomes the initial accumulator and the base case in the accumulator style just returns the accumulator.

Why might `sum_list_2` be preferred when it is clearly more complicated? To answer, we need to understand a little bit about how function calls are implemented. At its simplest, there is a *call stack*, which is a stack (the data structure with push and pop operations) with one element for each function call that has been started but has not yet completed. So when the evaluation of one function body calls another function, a new element is pushed on the call stack and it is popped off when the called function completes.

So for `sum_list_1`, there will be one call-stack element (sometimes just called a "stack frame") for each recursive call to `sum_list_1`, i.e., the stack will be as big as the list. This is necessary because after each stack frame is popped off the caller has to, "do the rest of the body" – namely add `i` to the recursive result and return.

Given the description so far, `sum_list_2` is no better: `sum_list_2` makes a call to `f` which then makes one recursive call for each list element. However, when `f` makes a recursive call to `f`, *there is nothing more for the caller to do after the callee returns except return the callee's result*. This situation is called a *tail call* (let's not try to figure out why it's called this) and functional languages like ML typically include an optimization: When a call is a tail call, the caller's stack-frame is popped before the call – the callee's stack-frame just replaces the caller's. This makes sense: the caller was just going to return the callee's result anyway. For `sum_list_2` that means the call stack needs just 2 elements at any point (one for `sum_list_2` and one for the current call to `f`).

Why do implementations of functional languages include this optimization? It is because this way recursion can sometimes be as efficient as a while-loop, which also does not make the call-stack bigger. The "sometimes" is exactly when calls are tail calls, but tail calls do not need to be to the same function (`f` can call `g`), so they are more flexible than while-loops that always have to "call" the same loop. Using an accumulator is a common way to turn a recursive function into a "tail-recursive function" (one where all recursive calls are tail calls), but not always. For example, functions that process trees (instead of lists) typically have call stacks that grow as big as the depth of a tree, but that's true in Java too.

While most people rely on intuition for "which calls are tail calls," we can be more precise by defining *tail position* recursively and saying a call is a tail call if it is in tail position. The definition has one part for each kind of expression; here are several parts:

- In `fun f(x) = e`, `e` is in tail position.
- If `if e1 then e2 else e3` is in tail position, then `e2` and `e3` are in tail position (not `e1`). (Similar for case).
- If `let b1 ... bn in e end` is in tail position, then `e` is in tail position (not any binding expressions).
- Function-call arguments are not in tail position.

- ...

# 8 Taking/Returning Functions; Function Closures

This section and the next one focus on first-class functions and function closures. By "first-class" we mean that functions can be computed and passed wherever other values can in ML. As examples, we can pass them to functions, return them from functions, put them in pairs, have them be part of the data a datatype constructor carries, etc. Function closures refer to functions that use variables defined outside of them. The term *higher-order function* just refers to a function that takes or returns other functions.

## 8.1 Passing Functions for Code Reuse

Here is a first example of a function that takes another function:

```
fun n_times (f,n,x) =
    if n=0
    then x
    else f (n_times(f,n-1,x))
```

We can tell the argument `f` is a function because the last line calls `f` with an argument. What `n_times` does is compute `f(f(...(f(x))))` where the number of calls to `f` is `n`. That is a geuninely useful helper function to have around. For example, here are 3 different uses of it:

```
fun double x = x+x
val x1 = n_times(double,4,7) (* answer: 112 *)

fun increment x = x+1
val x2 = n_times(increment,4,7) (* answer: 11 *)

val x3 = n_times(tl,2,[4,8,12,16]) (* answer: [12,16] *)
```

Like any helper function `n_times` lets us *abstract* the common parts of multiple computations so we can *reuse* some code in different ways by passing in different arguments. The main novelty is making one of those arguments a function, which is a powerful and flexible programming idiom. It also makes perfect sense — we are not introducing any new language constructs here, just using ones we already know in ways you may not have thought of.

## 8.2 Parametric Polymorphism

Let us now consider the type of `n_times`, which is `('a -> 'a) * int * 'a -> 'a`. It might be simpler at first to consider the type `(int -> int) * int * int -> int`, which is how `n_times` is used for `x1` and `x2` above: It takes 3 arguemnts, the first of which is itself a function that takes and return an `int`. Similarly, for `x3` we use `n_times` as though it has type `(int list -> int list) * int * int list -> int list`. But choosing either one of these types for `n_times` would make it less useful (some of our examples would not type-check). The type `('a -> 'a) * int * 'a -> 'a` says the third argument and result can be any type, but they have to be the *same* type, as does the argument and return type for the first argument. When types can be any type and do not have to be the same as other types, we use different letters (`'b`, `'c`, etc.)

This is called *paramteric polymorphism*, the ability of functions to take arguments of any type. It is technically a separate issue from first-class functions since there are functions that take functions and do not have polymorphic types and there are functions with polymorphic types that do not take functions. However, many of our examples with first-class functions will have polymorphic types. That is a good thing because it makes our code more reusable. It is quite essential to ML. For example, without paramteric polymorphism, we would have to redefine lists for every type of element that a list might have. Instead, we can have functions that work for any kind of list, like `length`, which has type `'a list -> int`.

## 8.3 Anonymous Functions

In our examples above, we defined `double` and `increment` at the top-level, but we already know we can define functions anywhere. So if we were only going to use `double` as the argument to `n_times`, we could instead do this:

```
n_times(let fun f x = x+x in f end, 4, 7)
```

As always, the let-expression returns the result of its body, which in this case is the function bound to `f`, which takes its argument and doubles it. However, this is poor style because ML has a much more concise way to define functions right where you use them:

```
n_times(fn x => x+x, 4, 7)
```

This code is exactly like our previous version: It defines a function that takes an argument `x` and returns `x+x`. It is called an *anonymous function* because we never gave it a name (like `f`). It is common to use anonymous functions as arguments to other functions.

The only thing you cannot do with an anonymous function is recursion, exactly because you have no name to use for the recursive call. In such cases, you need to use a `fun` binding as before. For non-recursive functions, you could use anonymous functions with `val` bindings instead of a `fun` binding. For example, these two bindings are exactly the same thing:

```
fun increment x = x + 1
val increment = fn x => x+1
```

They both bind `increment` to a value that is a function that returns its argument plus 1.

## 8.4 Higher-Order List-Processing Functions

We now consider a very useful higher-order function over lists:

```
fun map (f,lst) =
    case lst of
        [] => []
      | fst::rest => (f fst)::(map(f,rest))
```

The `map` function (provided by the ML standard library as `List.map`, but the implementation really is the 4 lines above) takes a list and a function `f` and produces a new list by applying `f` to each element of the list. Here are two example uses:

```
val x4 = map (increment, [4,8,12,16]) (* answer: [5,9,13,17] *)
val x5 = map (hd, [[1,2],[3,4],[5,6,7]]) (* answer: [1,3,5] *)
```

The type of `map` is illuminating: `('a -> 'b) * 'a list -> 'b list`. You can pass `map` any kind of list you want, but the argument type of `f` must be the element type of the list (they are both `'a`). But the return type of `f` can be a different type `'b`. The resulting list is a `'b list`. For `x4`, both `'a` and `'b` are *instantiated* with `int`. For `x5`, `'a` is `int list` and `'b` is `int`.

The definition and use of map is an incredibly important idiom even though our particular example is simple. We could have easily written a recursive function over lists of integers that incremented all the elements, but instead we divided the work into two parts: The map implementer knew how to traverse a recursive data structure, in this case a list. The map client knew what to do with the data there, in this case increment each number. You could imagine either of these tasks — traversing a complicated piece of data or doing some calculation for each of the pieces — being vastly more complicated and best done by different developers without making assumptions about the other task. That's exactly what writing `map` as a helper function that takes a function lets us do.

Here is a second very useful higher-order function for lists. It takes a function of type `'a -> bool` and an `'a list` and returns the `'a list` containing only the elements of the input list for which the function returns true:

```
fun filter (f,lst) =
    case lst of
        [] => []
      | fst::rest => if f fst
                     then fst::(filter (f,rest))
                     else filter (f,rest)
```

Here is an example use that assumes the list elements are pairs with second component of type `int`; it returns the lsit elements where the second component is even:

```
fun get_all_even_snd lst = filter((fn (_,v) => v mod 2 = 0), lst)
```

(Notice how we are using a pattern for the argument to our anonymous function.)

## 8.5 Returning Functions

Finally, functions can also return functions. Here is an example:

```
fun double_or_triple f =
    if f 7
    then fn x => 2*x
    else fn x => 3*x
```

The type of `double_or_triple` is `(int -> bool) -> (int -> int)`: The if-test makes the type of `f` clear and as usual the two branches of the if must have the same type, in this case `int->int`. However, ML will print the type as `(int -> bool) -> int -> int`, which is the same thing. The parentheses are unnecessary because the `->` "associates to the right", i.e., `t1 -> t2 -> t3 -> t4` is `t1 -> (t2 -> (t3 -> t4))`.

## 8.6 Function Closures

So far our uses of higher-order functions (i.e., functions taking or returning other functions) have used *closed functions*, meaning functions that only used variables that were arguments or defined inside the function. It is extremely powerful to let functions use variables that are in scope (i.e., in the environment) where the function was defined.

To understand how this works, realize that functions are values, but they are not just the code. They are really a pair (not ML pairs, but still something with two parts): The code and the environment that was current when the function was defined. We call this pair a *function closure* or just a *closure*. Before seeing why this is a good idea, it is worth going through several useless examples carefully. Consider:

```
val x = 1
fun f y = x + y
val x = 2
val y = 3
val z = f (x+y)
```

When `f` is defined, the environment binds `x` to 1, so `f` is the increment function *no matter where it is used.* In the later expression `f (x+y)`, we evaluate `f` to the closure that has the code `fn y => x+y` and environment mapping `x` to 1. We evaluate the `x+y` at the call-site to 5 (since in the environment at the call-site `x` maps to 2 and `y` maps to 3) then execute the function body `x+y` in the environment with `x` mapping to 1 extended to map `y` to 5. So `z` ends up bound to 6.

Closures are more interesting and useful when they are created inside a function and the *free variables* (variables used but not defined in the function) refer to local bindings. For example:

```
fun f y = let val x = 2 in fn z => x + y + z end
```

This function returns a closure with code `fn x => x + y +z` and an environment mapping `x` to 2 and `y` to whatever the caller to `f` passed for `y`. So `f 6` would produce a closure that (when called later) always add 8 to its argument and `f ~1` would produce a closure that increments its argument.

This definition of closure is also important when passing a function to another function. For example, consider:

```
fun f g = let val x = 3 in g 2 end
val x = 4
fun h y = x + y
val z = f h
```

This code binds 6 to z because the closure passed to f always adds 4 to its argument. That is bedcause we evaluate the body of h in the environment where h is defined (which here maps x to 4), not where the function is later called (in the body of f where there happens to be a different x bound to 3).

The rule that free variables in a function refer to the value they have in the environment where the function is *defined* is called *lexical scope*. The natural alternative — using the environment where the function is called — is called *dynamic scope*. There are several reasons to prefer lexical scope for variables. For example, under dynamic scope this code would try to call "the function 37" which makes no sense:

```
fun f x = x
fun g y = f y
val f = 37
val x = g 14
```

More generally, type-checking relies pretty fundamentally on lexical scope. However, the issue is not just type-checking. The good thing about lexical scope is that functions behave the same way no matter where they are used. So you can reason about and test a function without worrying about the environment being different at some place where it is called.

Having *defined* how closures work, the next section demonstrates that they are *useful*.

# 9 Function-Closure Idioms

Using function closures and avoiding mutation are the essence of programming in a functional style. To recongize when closures are a useful tool, it helps to see common idioms that use them. This section considers six such idioms.

## 9.1 Creating similar functions

If we need multiple functions that differ in some small way, we can write a function that given an argument returns an appropriate function. In this silly but small example, the addn function returns a function that adds n to its argument, which we can then use to make any number of different adding functions. Crucially, the function `fn m => n+m` uses the n in the environment where it was defined:

```
val addn = fn n => fn m => n+m
val increment = addn 1
val add_two = addn 2
fun f n =
   if n=0
   then []
   else (addn n)::(f (n-1))
```

## 9.2 Combining functions

When we program with lots of functions, it is useful to create new functions that are just combinations of other functions. You have probably done similar things in mathematics, such as when you compose two functions. For example, here is a function that does exactly function composition:

```
fun compose (f,g) = fn x => f (g x)
```

It takes two functions f and g and returns a function that applies its argument to g and makes that the argument to f. Crucially, the code `fn x => f (g x)` uses the f and g in the environment where it was defined. Notice the type of compose is `('a -> 'b) * ('c -> 'a) -> 'c -> 'b`. Recall it is common (but not always the case) that higher-order functions have *polymorphic types* (types with 'a etc. in them.

A second similar example uses h as a "back-up" function in case g returns NONE:

```
fun f (g,h) = fn x => case g x of NONE => h x | SOME y => y
```

If you are doing this sort of thing often ("if one thing is `NONE` try another thing"), abstracting it into a helper function that takes other functions can be helpful.

## 9.3 Passing functions with private data to iterators

Perhaps the most common and important use of closures is passing them to functions that recurse over data structures, like the `map` function from the previous section:

```
fun map (f,lst) =
    case lst of
        [] => []
      | fst::rest => (f fst)::(map(f,rest))
```

Arguments to map can be closures that use free variables, which makes `map` much more powerful than it would be if `f` could only use the argument that `map` passes to it. For example, this use of `map` truncates values to be less than some value:

```
fun truncate (lst,hi) = map((fn x => if x > hi then hi else x), lst)
```

Another higher-order function over lists that is even more powerful than `map` is `fold`:

```
fun fold (f,acc,l) =
  case l of
    []     => acc
  | hd::tl => fold (f, f(acc,hd), tl)
```

`fold` takes an "initial answer" `acc` and uses `f` to "combine" `acc` and the first element of the list, using this as the new "initial answer" for "folding" over the rest of the list. We can use `fold` to take care of iterating over a list while we provide some function that expresses how to combine elements. For example, to sum the elements in a list `lst`, we can do:

```
fold ((fn (x,y) => x+y), 0, lst)
```

As with `map`, much of `fold`'s power comes from clients passing closures that can have "private fields" (in the form of free variables) for keeping data they want to consult. Similar to the `truncate` example, we could count how many elements are too large:

```
fun num_too_big(lst,hi) = fold((fn (x,y) => if y > hi then x+1 else x),0,lst)
```

This pattern of splitting the recursive traversal (`fold` or `map`) from the data-processing done on the elements (the closures passed in) is fundamental. In our examples, both parts are so easy we could just do the whole thing together in a few simple lines. More generally, we may have a very complicated set of data structures to traverse or we may have very involved data processing to do. It is good to *separate these concerns* so that the programming problems can be solved separately.

While functional programmers have been doing this for decades, it has recently gained much attention thanks to Google's MapReduce (and similar systems from other companies and open-source projects). This work was first done in about 2004 by people very familiar with languages like Scheme and ML and it is revolutionizing how large-scale data-intensive computations are done.

In a nutshell, a company like Google has so much data that it is spread across thousands or tens of thousands of computers and organized in a very complicated way. So one set of people works on writing functions like `map` and `fold` (a function pretty similar to `fold` is called `reduce`) that does all the communication and data-passing for the huge set of computers. There are also many complicated different things you might want to do with this data — handle search queries, look for patterns in news reporting, do spell-checking, etc. These can all be written just in terms of functions passed to `map` and `reduce` without any concern for how the computation is spread across the computers.

It turns out to be crucial that the functions passed to `map` and `reduce` do not do mutation such as assigning to some global variable. The `MapReduce` system assumes that some function `f` can be called on some argument `x` any number

of times and it will always return the same answer and not have any affect on the state of any other variables. This is essential because when you have ten-thousand computers, one or more of them are likely to fail pretty often. When a computer fails, the system can just repeat any computation it did on a different computer since the lack of mutation ensures that it makes absolutely no difference what order the data is processed or how many times it is processed.

In summary, `MapReduce` boils down to 3 concepts:

- Building a fault-tolerant distributed system
- Using higher-order functions to provide a simple interface for the programmers doing the data-processing
- Avoiding mutation so that computations can be repeated and reordered by the system without affecting the result

Naturally, these notes provide an introduction to 2 of these 3 concepts.

## 9.4 Implementing an Abstract Data Type

The key to an abstract data type is requiring clients to use it via a collection of functions rather than directly accessing its private state. Thanks to this abstraction, we can later change how the data type is implemented without changing how it behaves for clients. In an object-oriented language, you might implement an abstract data type be defining a class with all private fields (inaccessible to clients) and some public methods (the interface with clients). We can do the same thing in ML with a record of closures; the variables that the closures use from the environment correspond to the private fields. Admittedly, this programming idiom can appear very fancy/clever/subtle, but it suggests (correctly) that functional programming and object-oriented programming are more similar than they might first appear.

As an example, consider an implementation of a set of integers that supports creating a new bigger set and seeing if an integer is in a set. Our sets are mutation-free in the sense that adding an integer to a set produces a new, different set. In ML, we could define a type that describes this interface:

```
datatype set = S of { add : int -> set, member : int -> bool }
```

Roughly speaking, a set is a record with two fields, each of which holds a function. It would be simpler to write:

```
type set = { add : int -> set, member : int -> bool }
```

but this does not work in ML because `type` bindings cannot be recursive. So we have to deal with the mild inconvenience of having a constructor `S` around our record of functions defining a set. Notice we are not using any new types or features; we simply have a type describing a record with fields named `add` and `member`, each of which holds a function.

Once we have an empty set, we can use its `add` field to create a one-element set, and then use that set's `add` field to create a two-element set and so on. So the only other thing our interface needs is a binding like this:

```
val empty_set = ... : set
```

Before implementing this interface, let's see how a client might use it:

```
fun use_a_set () =
    let val S s1 = empty_set
        val S s2 = (#add s1) 34
        val S s3 = (#add s2) 19
    in
        if (#member s3) 42
        then 99
        else if (#member s3) 19
        then 17
        else 0
    end
```

Again we are using no new features. `#add s1` is reading a record field, which in this case produces a function that we can then call with 34. If we were in Java, we might write `s1.add(34)` to do something similar. The `val` bindings use pattern-matching to "get rid of" the `S` constructors on values of type `set`.

There are many ways we could define `empty_set`; they will all use the technique of using a closure to "remember" what elements a set has. Here is one way:

```
val empty_set =
    let fun exists(j,lst) = (* could use currying and/or fold to be fancier *)
            case lst of
                [] => false
              | hd::tl => j=hd orelse exists(j,tl)
        fun make_set lst = (* lst is a "private field" *)
            S { add    = fn i => make_set (i::lst),
                member = fn i => exists (i,lst) }
    in
        make_set []
    end
```

The helper function `exists` just sees if a list has an element; we could also just use `List.exists` in the standard library. All the fanciness is in `make_set` and `empty_set` is just the record returned by `make_set []`. What `make_set` returns is a value of type `set`. It is essentially a record with two functions. The closures produced from `fn i => make_set (i::lst)` and `fn i => exists (i,lst)` are values that *when called* use `lst` — which is the "private field" we need to produce either a `bool` (for `member`) or a new set (for `add`).

## 9.5 Partial application ("currying")

The next idiom we consider is very convenient, especially when defining and using iterators over data structures. We have already seen that in ML every function takes exactly one argument, so you have to use an idiom to get the effect of multiple arguments. Previously we have done this by passing a tuple as the one argument, so each part of the tuple is conceptually one of the multiple arguments. Another more clever and often more convenient way is to have a function take the first conceptual argument and return another function that takes the second conceptual argument and so on.

This technique is called "currying" because someone named Curry was a researcher who studied the idea.

Here is an example of a "3-argument" function that uses currying:

```
val inorder3 = fn x => fn y => fn z =>
                z >= y andalso y >= x
```

If we call `inorder3 4` we will get a closure that has `x` in its environment. If we then call this closure with `5`, we get a closures that has `x` and `y` in its environment. If we then call this closure with `6`, we will get `true` because 6 is greater than 5 and 5 is greater than 4. That is just how closures work.

So `((inorder3 4) 5) 6` computes exactly what we want and feels pretty close to calling `inorder3` with 3 arguments. Even better, the parentheses are optional, so we can write exactly the same thing as `inorder3 4 5 6`, which is actually fewer non-space characters than our old tuple approach where we would have:

```
fun inorder3 (x,y,z) = z >= y andalso y >= x
val someClient = inorder3(4,5,6)
```

Moreover, even though we might expect most clients of our curried `inorder3` to provide all 3 conceptual arguments, they might provide fewer and use the resulting closure later. This is called "partial application" because we are providing a subset (more precisely, a prefix) of the arguments. As a silly example, `inorder3 0 0` returns a function that returns `true` if its argument is nonnegative.

Currying is particularly convenient for creating similar functions with iterators. For example, here is a curried version of a fold function for lists:

```
fun fold f = fn acc => fn l =>
  case l of
    []     => acc
  | hd::tl => fold f (f(acc,hd)) tl
```

Now we could use this fold to define a function that sum's a list elements like this:

```
fun sum1 l = fold ((fn (x,y) => x+y) 0 l
```

But that is unnecessarily complicated compared to just using partial application:

```
val sum2 = fold (fn (x,y) => x+y) 0
```

The convenience of partial application is why many iterators in ML's standard library use currying with the function they take as the first argument.

There is syntactic sugar for defining curried functions; you can just separate the conceptual arguments by spaces rather than using anonymous functions. So the better style for our fold function would be:

```
fun fold f acc l =
  case l of
    []     => acc
  | hd::tl => fold f (f(acc,hd)) tl
```

The ML standard library defines `List.foldl`, which does exactly this, so you should use the library function where appropriate. (The `l` stands for "left" and refers to the order the elements are combined.) There are several such useful higher-order functions defined in the library with currying. Another example is `List.exists`, which we use in the callback example below. However, these library functions are quite easy to implement ourselves, so we should understand they are not fancy:

```
fun exists predicate lst =
    case lst of
      [] => false
    | hd::tl => predicate hd orelse exists predicate tl
```

Sometimes functions are curried but the arguments are not in the order you want for a partial application. Or sometimes a function is curried when you want it to use tuples or vice-versa. Fortunately our earlier idiom of combining functions can take functions using one approach and produce functions using another:

```
fun other_curry1 f = fn x => fn y => f y x
fun other_curry2 f x y = f y x
fun curry f x y = f (x,y)
fun uncurry f (x,y) = f x y
```

Looking at the types of these functions can help you understand what they do. As an aside, the types are also fascinating because if you pronounce `->` as "implies" and `*` as "and", the types of all these functions are logical tautologies.

Finally, you might wonder which is faster, currying or tupling. It almost never matters; they both do work proportional to the number of conceptual arguments, which is typically quite small. For the one or two performance-critical functions in your software, it *might* matter to pick the faster way. In the SML/NJ compiler, tupling happens to be faster. In other implementations of ML, curried functions are faster.

## 9.6 Callbacks

The final common idiom we'll consider is implementing a library that detects when "events" occur and informs clients that have previously "registered" their interest in hearing about events. Clients can register their interest by providing a "callback" — a function that gets called when the event occurs. Examples of events for which you might want this sort of library include things like users moving the mouse or pressing a key. Data arriving from a network interface is another example.

The purpose of these libraries is to allow multiple clients to register callbacks. The library implementer has no idea what clients need to compute when an event occurs, and the clients may need "extra data" to do the computation. So the library implementor should not restrict what "extra data" each client uses. A closure is ideal for this because a function's type `t1 -> t2` doesn't specify the types of any other variables a closure uses, so we can put the "extra data" in the closure's environment.

If you have used "event listeners" in Java's Swing library, then you have used this idiom in an object-oriented setting. In Java, you get "extra data" by defining a subclass with additional fields. This can take an awful lot of keystrokes for a simple listener, which is a (the?) main reason the Java language added anonymous inner classes, which are closer to the convenience of closures.

To see an example in ML, we will finally introduce ML's support for mutation. Mutation is okay in some settings. In this case, we really do want registering a callback to "change the state of the world" — when an event occurs, there are now more callbacks to invoke. In ML, most things really cannot be mutated. Instead you must create a *reference*, which is a container whose contents can be changed. You create a new reference with the expression `ref e` (the initial contents are the result of evaluating `e`). You get a reference `r`'s current contents with `!r` (not to be confused with negation in Java or C), and you change `r`'s contents with `r := e`. The type of a reference that contains values of type `t` is written `t ref`.

Our example will use the idea that callbacks should be called when a key on the keyboard is pressed. We will pass the callbacks an `int` that encodes which key it was. Our interface just needs a way to register callbacks. (In a real library, you might also want a way to unregister them.)

```
val onKeyEvent : (int -> unit) -> unit
```

Clients will pass a `int -> unit` that, when called later with an `int` will do whatever they want. To implement this function, we just use a reference that holds a list of the callbacks. Then when an event actually occurs, we assume the function `on_event` is called and it calls each callback in the list:

```
val cbs : (int -> unit) list ref = ref []
fun onKeyEvent f = cbs := f::(!cbs)
fun on_event i =
   let fun loop l =
        case l of
          [] => ()
        | f::tl => (f i; loop tl)
    in loop (!cbs) end
```

Most importantly, the type of `onKeyEvent` places no restriction on what extra data a callback can access when it is called. Here are three different clients (calls to `onKeyEvent`) that use different variables in functions in their environment:

```
onKeyEvent (fn i => write_to_log(Int.toString i
                                 ^ " got pressed\n"));

val f4_key = 75; (* no idea what integer code is actually f4; pretend it is 75 *)
onKeyEvent (fn i => if i=f4_key then minimizeWindow() else ());

fun prohibit_keys lst =
  onKeyEvent (fn i => if (List.exists (fn j => j=i) lst)
                      then exitProgram()
                      else ());
prohibit_keys [13, 42, 99];
```

## 9.7 A Final Example: Counting Zeros

Higher-order functions provide lots of conciseness, flexibility, and code reuse when used well. There are also many different ways to do the same thing, and which is "best" is somewhat a matter of taste (which approach is most readable, easiest to get correct, etc.) and efficiency. Consider the simple example of counting how many 0s are in an `int list`. Before learning higher-order functions, we might have written:

```
fun count_zeroes_1 lst =
    case lst of
        [] => 0
      | first::rest => (if first=0 then 1 else 0) + count_zeroes_1 rest
```

Using a couple higher-order functions defined in the `List` library we can instead think of counting 0s as having two parts: get rid of non-zeros and see how many elements remain:

```
fun count_zeroes_2 lst =
    List.length (List.filter (fn x => x=0) lst)
```

We could then recognize that this approach is essentially function composition with partial application of the curried filter function:

```
val count_zeroes_3 = List.length o (List.filter (fn x => x=0))
```

And if we *really* wanted to use lots of currying, we could recognize that if `=` were itself a curried function, we could partially apply it to 0 to get a function that compares numbers to 0. Now, `=` is not a curried function, but we can make it one like this:

- Use the ML keyword `op` for turning an infix function into a regular function. For example, `op+` is a function of type `int*int->int` and `op=` is a function of type `''a*''a->bool`.
- Use a helper function to convert a function that takes a pair to a function that takes two curried arguments.

```
fun curry f x y = f (x, y)
val count_zeroes_3' = List.length o (List.filter ((curry (op=)) 0))
```

These "filter then count" approaches seem fine especially when you are trying to be a productive programmer and counting 0s is hardly where you should spend a lot of human time. But they are inefficient since they create a whole list (the list holding the 0s) just to see how long the list is. Our initial approach did not have this inefficiency, nor does a solution using `foldl`:

```
fun count_zeroes_4 lst =
    List.foldl (fn (x,y) => (if x=0 then 1 else 0) + y) 0 lst
```

We can recognize that this can be simplified using partial application of the curried `foldl` function (much like `if e then true else false` can be simplified to `e`):

```
val count_zeroes_5 =
    List.foldl (fn (hd,acc) => (if hd=0 then 1 else 0) + acc) 0
```

While these uses of `foldl` work great, if we plan to write lots of functions that count list elements, we might instead write our own higher-order function that makes such counting easier:

```
fun count_list1 f lst =
    case lst of
        [] => 0
      | hd::tl => (if f hd then 1 else 0) + count_list1 f tl
```

Here we have abstracted out whether an element should be counted, letting callers pass in a function `f` that takes a list element and returns a `bool`. Here are two ways to pass in functions that see if an element is 0:

```
val count_zeroes_6 = count_list1 (fn x => x=0)
val count_zeroes_6' = count_list1 (curry (op=) 0)
```

While `count_list1` is a bit more convenient for callers than `foldl` — assuming that callers want to do counting — we could *implment* the count-list function in terms of fold:

```
fun count_list2 f =
    List.foldl (fn (hd,acc) => (if f hd then 1 else 0) + acc) 0
```

This implementation decision is irrelevant to callers; they use `count_list2` exactly like `count_list1`:

```
val count_zeroes_7 = count_list2 (fn x => x=0)
val count_zeroes_7' = count_list2 (curry (op=) 0)
```

# 10 Modules and Abstract Types

## 10.1 Basic Namespace Management, Structures, and Signatures

To learn the basics of ML, pattern-matching, and functional programming, we have written fairly small programs that are just a sequence of bindings. For larger programs, it definitely helps to have more structure. In ML, we can use *structures* to define *modules* that together are a collection of bindings. At its simplest, you can just write `structure Name = struct bindings end` where `Name` is the name of your structure (you can pick anything; capitalization is a convention) and `bindings` is any list of bindings, containing values, functions, exceptions, datatypes, and types. Inside the structure you can use earlier bindings just like we have been doing "at top-level" (i.e., outside of any module). Outside of the structure, you can refer to a binding $b$ in `Name` by writing `Name.`$b$. This is exactly what we have been doing to use functions like `List.foldl`; now you know how to define your own structures.

Used like this, structures are providing just *namespace management*, a way to avoid different bindings in different parts of the program from shadowing each other. That is very useful, but not particularly interesting. Much more interesting is giving structures *signatures*, which are types for modules and let us provide strict *interfaces* that code outside the module must obey. This first example just uses a signature to describe *everything* in the structure. This example helps us lear the syntax and basic meaning of structures and signatures before we learn how and why to write signatures that do not expose everything.

```
signature MATHLIB =
sig
val fact : int -> int
val half_pi : real
val doubler : int -> int
end

structure MyMathLib :> MATHLIB =
struct
fun fact x =
    if x=0
    then 1
    else x * fact (x - 1)

val half_pi = Math.pi / 2.0

fun doubler y = y + y
end
```

Because of the `:> MATHLIB`, the structure `MyMathLib` will typecheck only if it actually provides everything the signature `MATHLIB` claims it does and with the right types. Signatures can also contain datatype, exception, and type bindings. Because we check the signature when we compile `MyMathLib`, we can use this information when we check any code that uses `MyMathLib`. In other words, we can just check clients *assuming* that the signature is correct.

## 10.2 Hiding Things

We now consider signatures for structures that *hide* things about the implementation. This abstraction is an essential tool in software engineering because it lets us change the implementation while *knowing*, thanks to the signature, that clients will not be able to tell.

Before discussing how to do this using ML modules, let's notice that hiding implementation details is one of the most important concepts when developing software. We can already do this with functions. For example, all 3 of these functions double their argument, and clients (i.e., callers) would have no way to tell if we replaced one of the functions with a different one:

```
fun double1 x = x + x
fun double2 x = x * 2
```

```
val y = 2
fun double3 x = x * y
```

From an engineering perspective, this is a crucial separation of concerns. I can work on improving the implementation of a function and know that I am not breaking any clients. Conversely, nothing clients can do can break how the functions above work.

We would like to have these advantages for whole structures. The key is using signatures to do one or both of the following:

- Make certain bindings inaccessible
- Make types abstract (reveal that a type exists, but do not reveal its implementation)

These techniques help us replace a module implementation with another one without clients being able to tell. They *also* let us enforce invariants about arguments to our functions since abstract types can ensure that values are only created by functions defined within the module.

## 10.3 An Extended Example

To see how to hid things with ML modules, we will consider an extended example. Here is a small library for positive, rational numbers. A positive rational number is a fraction where the numerator and denominator are both greater than 0. We have functions for creating rationals, adding two rationals, and converting a rational to a string. Of course, a real library would have many more functions, but this will suffice for us:

```
structure PosRat1 =
struct
  datatype rational = Whole of int | Frac of int*int
  exception BadFrac

  fun gcd (x,y) =
       if x=y
       then x
       else if x < y
       then gcd(x,y-x)
       else gcd(y,x)

   fun reduce r =
       case r of
           Whole _ => r
         | Frac(x,y) =>
           let val d = gcd(x,y) in
               if d=y
               then Whole(x div d)
               else Frac(x div d, y div d)
           end

   fun make_frac (x,y) =
       if x <= 0 orelse y <= 0
       then raise BadFrac
       else reduce (Frac(x,y))

   fun add (r1,r2) =
       case (r1,r2) of
           (Whole(i),Whole(j))   => Whole(i+j)
         | (Whole(i),Frac(j,k))  => Frac(j+k*i,k)
```

```
        | (Frac(j,k),Whole(i))  => Frac(j+k*i,k)
        | (Frac(a,b),Frac(c,d)) => reduce (Frac(a*d + b*c, b*d))

    fun toString r =
        case r of
            Whole i => Int.toString i
          | Frac(a,b) => (Int.toString a) ^ "/" ^ (Int.toString b)

end
```

The code as written makes several *assumptions*, which should *at least* be documented in comments:

- `gcd` and `reduce` are really local helper functions for putting rationals into reduced form (e.g., `Frac(1,2)` instead of `Frac(2,4)` and `Whole 7` instead of `Frac(21,3)`). We do not want clients calling them since we do not want to be responsible for maintaining their behavior. Moreover, they do not work for negative numbers, so we definitely don't want clients thinking they do.
- `make_frac` will reject non-positive numbers. Therefore, if all rationals are created via `make_frac` and `add`, then all rationals will always be positive. Since `reduce` assumes this, that is an important *invariant*. To ensure it holds, clients should not use the `Whole` and `Frac` constructors directly; they should always call `make_frac`.
- Similarly, we keep all rationals in reduced form. Thanks to this invariant, `toString` will never return `"4/2"` or `"21/3"`, but again a client that creates its own rationals could break this property, e.g., with `PosRat1.toString(Frac(21,3))`.

A signature that simply described all the bindings in this structure would look like this:

```
signature RATIONAL_ALL =
sig
datatype rational = Frac of int * int | Whole of int
exception BadFrac
val gcd : int * int -> int
val reduce : rational -> rational
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
```

If we give this signature to the structure (by starting it with `structure PosRat1 :> RATIONAL_ALL`, ML will check that all the types are right, but clients will still be able to do whatever they want with the constructors, exception, and functions we defined.

A more restrictive signature could hide the bindings that we do not want clients to know about. This effectively makes them private. Unlike in Java, we do this just via the signature, we do not have to change the structure body at all. We define:

```
signature RATIONAL_A =
sig
datatype rational = Frac of int * int | Whole of int
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
```

and change the first line of the structure to `structure PosRat1 :> RATIONAL_A`.

This signature ensures clients do not call `gcd` or `reduce` (or have any idea they exist), so we could change the structure definition in various ways (such as changing their names) and we can feel better about having them fail with negative numbers. But we still have the problem that clients can create bad rationals directly (bypassing `make_frac`).

The way to fix this is to remove the `datatype` binding as well! But that does not quite work because our signature mentions the type `rational` and if we remove the `datatype` binding that type name will make no sense.

So what we want is a way to say that there is a type `rational` but clients cannot know anything about what the type is other than it exists. This is an *abstract type*, and it is a very powerful concept for letting programmers write libraries that have to be used in particular ways. Here is the appropriate signature:

```
signature RATIONAL_B =
sig
type rational (* type now abstract *)
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
```

(Of course, we also have to change the first line of the structure definition. That is always true, so we will stop mentioning it.)

The syntax is just to give a type binding without a definition. Now, how can clients make rationals? Well, the first one will have to be made with `make_frac`. After that, more rationals can be made with `make_frac` or `add`. There is *no other way*, so thanks to the way we wrote `make_frac` and `add`, all rationals will always be in reduced form.

What `RATIONAL_B` took away from clients compared to `RATIONAL_A` is the constructors `Frac` and `Whole`. So clients cannot be rationals directly and they cannot pattern-match on rationals. They have no idea how they are represented internally.

ML lets us do one more slightly fancy thing if we want. `RATIONAL_B` enforces the invariants we want but it makes it a little unpleasant to make rationals that are whole numbers since clients have to call `make_frac(6,1)`. We could add a function to our structure like this:

```
fun make_whole x = Whole x
```

and a line to our signature like this:

```
val make_whole : int -> rational
```

But the `make_whole` function is an unnecessary wrapper just like `if e then true else false`. The constructor `Whole` is *already* a function that takes an `int` and returns a `rational` just like we want. So instead we can make no change to the structure and instead just add to our signature:

```
val Whole : int -> rational
```

This makes sense because constructors are two things: functions that create values of the datatype and things you can use in patterns. This addition to our signature just exposes one of the two — clients know `Whole` is a function, but they do not know it is a constructor.

## 10.4 The General Rules

We can now consider, in general, what it means for a structure `Name` to be a legal implementation of the signature `BLAH`:

- It must define all bindings mentioned in `BLAH` (and it can define more).
- The types of the bindings in `Name` must match those in `BLAH`, but they can be more general (e.g., the implementation can be polymorphic even if the signature says it is not — an example comes later).
- Types can be made abstract.

## 10.5 Back to Our Example: Changing the Implementation

So far, our more restrictive signatures for `PosRat1` have succeeded in ensuring clients: (1) do not create non-positive rationals, (2) do not create non-reduced rationals, (3) do not pass bad arguments to `gcd` or `reduce`. Now we can consider the other advantage of restrictive signatures: we can change structure implementations and *know* that client behavior cannot change.

As a simple example, we could make `gcd` a local function defined inside of `reduce` and know that no client will fail to work since they could not rely on `gcd`'s existence. More interestingly, let's change one of the invariants of our structure. Let's *not* keep rationals in reduced form. Instead, let's just reduce a rational right before we convert it to a string. This simplifies `make_frac` and `add`, while complicating `toString`, which is now the only function that needs `reduce`. Here is the whole structure:

```
structure PosRat2 :> RATIONAL_A (*or B or C*) =
struct
  datatype rational = Whole of int | Frac of int*int
  exception BadFrac

   fun make_frac (x,y) =
       if x <= 0 orelse y <= 0
       then raise BadFrac
       else Frac(x,y)

   fun add (r1,r2) =
       case (r1,r2) of
           (Whole(i),Whole(j))   => Whole(i+j)
         | (Whole(i),Frac(j,k))  => Frac(j+k*i,k)
         | (Frac(j,k),Whole(i))  => Frac(j+k*i,k)
         | (Frac(a,b),Frac(c,d)) => Frac(a*d + b*c, b*d)

   fun toString r =
       let fun gcd (x,y) =
               if x=y
               then x
               else if x < y
               then gcd(x,y-x)
               else gcd(y,x)

           fun reduce r =
               case r of
                   Whole _ => r
                 | Frac(x,y) =>
                   let val d = gcd(x,y) in
                       if d=y
                       then Whole(x div d)
                       else Frac(x div d, y div d)
                   end
       in
           case reduce r of
               Whole i   => Int.toString i
             | Frac(a,b) => (Int.toString a) ^ "/" ^ (Int.toString b)
       end
end
```

Notice the following:

- PosRat2 does *not* have signature RATIONAL_ALL because it does not define gcd or reduce (they are local to toString).
- While PosRat2 does have signature RATIONAL_A, if we give PosRat1 and PosRat2 these signatures, *clients can tell the difference!*. For example, PosRat1.toString(Frac(21,3) returns "21/3" but PosRat2.toString(Frac(21,3) returns "7". So this signature is *not* restrictive enough to ensure changing PosRat1 to PosRat2 will not change client behavior.
- RATIONAL_B and RATIONAL_C *are* restrictive enough to ensure the two structures are totally equivalent for any possible client.

While our two structures so far maintain different invariants, they do use the same definition for the type rational. This is not necessary with signatures RATIONAL_B or RATIONAL_C; a different structure having these signatures could implement the type differently. For example, suppose we realize that special-casing whole-numbers internally is more trouble than it is worth. We could instead just use int*int and define this structure:

```
structure PosRat3 :> RATIONAL_B =
struct
   type rational = int*int
   exception BadFrac

   fun make_frac z =
       let val (x,y) = z in
           if x <= 0 orelse y <= 0
           then raise BadFrac
           else z
       end

   fun add ((a,b),(c,d)) = (a*d + c*b, b*d)

   fun toString (x,y) =
       let fun gcd (x,y) =
               if x=y
               then x
               else if x < y
               then gcd(x,y-x)
               else gcd(y,x)
           val d = gcd (x,y)
           val num = x div d
           val denom = y div d
       in
           Int.toString num ^ (if denom=1
                               then ""
                               else "/" ^ (Int.toString denom))
       end
end
```

(This structure takes the PosRat2 approach of having toString reduce fractions, but that issue is largely orthogonal from the definition of rational.)

Notice that this structure provides everything RATIONAL_B requires. The function make_frac is interesting in that it is the identity function unless it raises an exception, but *clients do not know that* since they cannot tell that rational is int*int. They *cannot* pass just any int*int to add or toString; they must pass something that they know has type rational. As with our other structures, that means rationals are created only by make_frac and add, ensuring that all rationals are positive. Our structure does *not* match RATIONAL_A since it does not provide rational as a datatype with constructors Frac and Whole.

Also notice that our structure as defined does not have signature `RATIONAL_C` because there is no `Whole` function. But we could easily add one – all we need is a function of type `int->rational` that has the behavior we want:

```
fun Whole i = (i,1)
```

No client can distinguish our "real function" from the previous structures' use of the `Whole` constructor as a function.

Finally, suppose we had written `make_frac` like this:

```
fun make_frac x = x
```

This does not enforce our invariant that forbids non-positive numbers, but it does still match signatures `RATIONAL_B` and `RATIONAL_C`. That is intersting because within the module, `make_frac` has type `'a->'a`, but outside it has type `int*int -> rational`. Our previous version above passed signature matching because internally it had type `int*int -> int*int` and `rational=int*int`. The polymorphic version also matches because we are allowed to *specialize* polymorphic functions when doing signature matching. So the type-checker figures out that to match `int*int -> rational` against `'a->'a` it need to first replace `'a` with `int*int` and then use the definition `rational=int*int`. Less formally, the fact that `make_frac` *could* have a polymorphic type does not mean the signature *has* to give it one.

# 11 Extensibility in OOP and FP

This short section compares functional programming to object-oriented programming. There are actually many similarities. For example, a closure is like an object with one method ("call me") where the environment is like a list of private fields. There are also essential differences, such as OOP's built-in support for dynamic dispatch, where the treatment of `this` in Java is intimately tied to subclasses and method overriding. There are also differences between ML and Java that are more syntactic (e.g., the use of curly braces) or orthogonal to OOP vs. FP (e.g., type inference).

## 11.1 The Important Point

Here we just focus on a key issue of how *stylistically* functional and object-oriented programming often take exactly the opposite approach — and which is "better" often depends on how you expect software to evolve or be extended over time.

The canonical example we will consider is a little "expression language." In this language we have a number of "kinds of expressions" (integer constants, negations, additions, multiplications, etc.), and a number of "things to do with expressions" (evaluate them, convert them to strings, see if they have a 0 in them, etc.). This general set-up is common: we have data variants and we have operations over the data. To write all the code we want to write, we need to fill out this conceptual two-dimensional grid:

| | Int | Negate | Add | Mult |
|---|---|---|---|---|
| eval | | | | |
| toString | | | | |
| hasZero | | | | |

In general, we can think of having one column for each data variant and one row for each operation. Often multiple grid squares can be implemented in the same way with some helper function or method in a superclass, but fundamentally we have to "do something" for each grid square. Now we can contrast the styles encouraged by functional programming and object-oriented programming:

- In functional programming, we tend to put all the code for a row together (in a function that has one branch for each data variant). An ML-style datatype definition declares the data variants and we get an inexhaustive pattern-match error/warning if some function does not support all the columns.
- In object-oriented programming, we tend to put all the code for a column together (in a class that has one method for each operation). A Java-style abstract class (or interface) definition declares the operations and we get an error if some non-abstract subclass does not support all the rows.

In this sense, how you represent the grid in your program (by rows or columns) is a matter of taste. Which seems most natural may depend on what kind of program you are writing.

However, if we expect our code to be extended in certain ways over time, then "which way we organize things" becomes more than a matter of taste:

- Suppose we expect to add new data variants (e.g., exponentiation operations) in the future. Then object-oriented programming is more convenient because we can add a "new column" without changing any existing code. To add a new column in our ML approach, we have to change our datatype binding and all our old functions. As a partial antidote, after adding a new constructor, the type-checker's inexhaustive-match warnings provide a nice to-do list (unless we used wildcard patterns in our initial program).
- Suppose we expect to add new operations (e.g., sum all the ints in an expression) in the future. Then functional programming is more convenient because we can add a "new row" without changing any existing code. To add a new row in our OO approach, we have to change our abstract class (or interface) definition and all our old classes. As a partial antidote, after adding a new abstract method, the type-checker provides a nice to-do list of all the classes that need new methods (unless we used run-time exceptions instead of abstract methods).

The problem, alas, is that the future is hard to predict, so we may not know in advance whether we will want new data variants or new operations. Moreover, maybe we will want both.

- If we write our original OO code in the natural way and then try to add a new row without changing existing code, it doesn't really work. You will end up with a lot of downcasts because the existing types expect something without the new operations, but we need to assume we have the new operations.
- If we write our original functional code in the natural way and then try to add a new column without changinge existing code, it also doesn't work. ML's datatype bindings do not let us "cast" something else, so we will end having to define a whole new datatype with new constructors.

### 11.2 Planning for Extensibility: An Advanced Topic

However, if we write our original code in a way that *plans for extensibility* we can do better, although the code becomes considerably more complicated. An OO programming pattern for making it easy to add new operations without changing existing classes is called the "visitor pattern" (a synonym is "double dispatch"). While we won't go over it, it is well-known and definitely worth learning. Making a class hierarchy "visitable" requires some strange-looking code, but the pattern is always the same and then writing new "visitors" (operations over the data) is reasonably straightforward.

Similarly, we could define ML datatypes that allow for "other unknown possibilities." The "trick" is to use type constructors (not discussed in these notes) and a constructor that carries values of type `'a`. However, then our existing operations won't know how to handle this other case. To fix this, clients will have to pass in a function that handles this other case. For programs that don't extend the datatype with new data variants, we can just use a function that raises an exception. This programming pattern is pretty cumbersome and ML programmers do not tend to use it unless extensibility is really important.

There have been some language proposals and (ongoing) research projects to design languages that make both forms of extensibility convenient within the same program. It is probably fair to say that we can do better than today's popular languages but the community has not yet decided on the "best" (or if there is a "best") way.

## 12 Odds and Ends

This section has a few Standard ML features that arise on the homework assignments associated with these notes even though they are not "big ideas" at the essence of functional programming. This material is not covered in the corresponding lecture slides. Naturally, it could be added to the lectures and/or covered in a recitation section.

## 12.1 `andalso` and `orelse`

The expressions `e1 andalso e2` and `e1 orelse e2` are "short-circuiting" and (like Java's `&&`) and or (like Java's `||`) respectively. Both subexpressions must have type `bool`. As a matter of style, you should usually use them instead of an `if e1 then e2 else e3` expression where it makes sense. Note they can be defined as syntactic sugar:

- `e1 andalso e2` is sugar for `if e1 then e2 else false`.
- `e1 orelse e2` is sugar for `if e1 then true else e2`.

## 12.2 Type Synonyms

A `type` binding introduces a type synonym. For example, the second and third lines here are type synonyms:

```
datatype suit = Clubs | Diamonds | Hearts | Spades
type rank = int (* 1-13, arguably poor style not to use a datatype *)
type card = suit * rank
```

A type synonym is largely just a convenience, allowing the programmer to use `card` or `suit * rank` interchangeably. But in the presence of type inference, programmers rarely write down types and the read-eval-print loop is free to use or not use synonyms when printing out types. So if you need to write a funciton of type `card -> card` and the interpeter indicates your function has type `suit * int -> suit * rank`, this is *not* a problem.

In Section 10 on modules, type synonyms become more interesting because while the synonym still exists *within* the module, we can use a signature to make a type abstract. For example, we could make it so that users of a module do not know the concrete type of `card`. The structure `PosRat3` in Section 10 gives an example.

## 12.3 More General Types

There is another situation where the interpreter may give a different type than you expect even though your code is correct. Perhaps your code is more polymorphic than you realized. For example, consider a function like this:

```
fun f (x,y,z) = if x > 0 then (x,y,z) else (x,z,y)
```

If you intended this function to have type `int*int*int -> int*int*int`, you may be surprised that it actually has type `int*'a*'a -> int*'a*'a`. After all, it is safe to call this function with `y` and `z` having any type, but they must have the same type in order to know what type `f` returns. But since the type `int*'a*'a -> int*'a*'a` is *more general* than `int*int*int -> int*int*int`, this is not a problem: You still wrote a function that can take and return a `int*int*int`. The general rule is that you can take a polymorphic function type (a type with `'a`, `'b`, etc. in it) and *instantiate* the type variables (substitute types for them) to get "another" type for the function. So if your function's can be instantiated to produce the type you actually want, then there is not a problem.

## 12.4 Equality Types and Using = With Compound Data

A strange but sometimes convenient feature of ML is that the equality operator `=` can be used for many different types. We can use it to compare integers, booleans, characters, strings, etc. We can even use it to compare tuples, records, and datatypes *if* all the possible components of these compound types are themselves comparable. Things that are *not* comparable include floating-point numbers (`real`), mutable references (`'a ref`), and functions (`t1 -> t2`).

This distinction can show up in types when you write polymorphic functions that use `=`. For example, consider:

```
fun listeq (lst1,lst2) =
   case (lst1,lst2) of
     ([],[]) => true
   | ([],_)  => false
   | (_,[])  => false
   | (x1::tl1, x2::tl2) => x1=x2 andalso listeq (tl1,tl2)
```

What should the type of `listeq` be? It takes two lists and returns a `bool`. The lists can have elements of any type *provided the elements can be compared with each other using* `=`. ML has a special type for describing this situation. "Equality types," written `''a`, `''b`, etc. are like regular type variables except they can only be isntantiated with types that are legal arguments to `=`. Note equality types themselves are legal arguments to `=`. So the type of `listeq` is `''a list * ''a list -> bool`.

## 12.5 Creating and Raising Exceptions

An exception binding declares a new variant of an exception. It has a constructor (the "name" of the exception) and can either carray an argument or not:

```
exception UhOh
exception BadNews of int * string
```

These bindings just define the existence of a new kind of exception. You can then create values of type `exn` (the type of all exceptions) using these new constants/constructors. So `UhOh` and `BadNews(3,"x")` both have type `exn`. In fact, `BadNews` has type `int * string -> exn`. The built-in "function" `raise` actually raises (in Java's terminolgoy throws) an exception, for example —raise BadNews(3,"x")—. To the type-checker, `raise` is jsut a function of type `exn->'a`. It is odd to call it a function because it does not return. Hence saying the return type is `'a` is reasonable: it is safe to assume it has any return type you want because it will not actually return.

We also may need to *handle* (in Java's terminology catch) exceptions. Rest assured ML has a built-in expression for for this. The syntax is `e0 handle p1 => e1 | ... | pn => en` where the patterns match exceptions. Any exceptions raised by `e0` will be matched against these patterns, like with a case expression. If none match, the exception is (re-)raised.

## 12.6 The Value Restriction

For reasons that will not be fully explained here related to type safety and mutation, ML does not allow all `val` bindings to have polymorphic types. If you stumble across this, it is likely to be with a use of a curried higher-order function like `List.map`. For example, this perfectly fine code fails to type-check and gives a strange error message:

```
val pairAll = List.map (fn x => (x,x))
```

(The call `pairAll [3,4,5]` should return `[(3,3),(4,4),(5,5)]`.)

We expect `pairAll` to have type `'a list -> ('a * 'a) list`, but it does not. These type-check however:

```
val pairAndIncAll = List.map (fn x => (x,x+1)) (* int list -> (int*int) list *)
fun pairAll lst = List.map (fn x => (x,x)) lst (* 'a list -> ('a*'a) list *)
```

So why does the first version not type-check? Because a `fun` binding can have a polymorphic type, but a `val` binding can have a polymorphic type *only* if the expression is a *value*, not something like a function call that has to be evaluated. This is a very strange rule, but it usually does not arise. And without or some other restriction, functions that create mutable references could circumvent ML's type system (explanation not given). In our example above, this does not happen — `List.map` does not use mutable references. But the type-checker doesn't know, so it errs on the side of caution. The type-checker could pick some non-polymorphic type for `pairAll`, but since it doesn't know what one to pick, it gives an error instead.