Programming Language
Part A

Coursera - University of Washington

August 8, 2016

Programming language on Coursera, Part A Summary.

Coursera Programming Languages Course
Section 1 Summary

Standard Description: This summary covers **roughly** *the same material as the lecture videos and the materials (slides, code) posted with each video. It can help to read about the material in a narrative style and to have the material for an entire section of the course in a single document, especially when reviewing the material later. Please report errors in these notes on the discussion board.*

Contents

Welcome to Programming Languages

(See also introductory material on the course webpage about course structure, homeworks, grading, software installation, etc. That material is not repeated here.)

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

Many people would say this course "teaches" the 3 languages ML (in Part A), Racket (in Part B), and Ruby (Part C), but that is a poor description. We will use these languages to learn various paradigms and concepts because they are well-suited to do so. If our goal were just to make you as productive as possible in these three languages, the course material would be very different. That said, being able to learn new languages and recognize the similarities and differences across languages is an important goal.

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

The conventional thing to do at the very beginning of a course is to motivate the course, which in this case would explain why you should learn functional programming and more generally why it is worth learning

different languages, paradigms, and language concepts. We will largely *delay* this discussion until after the third homework. It is simply too important to cover when most students are more concerned with getting a sense of what the work in the course will be like, and, more importantly, it is a much easier discussion to have after we have built up shared terminology and experience. Motivation does matter; let's take a "rain-check" with the promise that it will be well worth it.

ML Expressions and Variable Bindings

So let's just start "learning ML" but in a way that teaches core programming-languages concepts rather than just "getting down some code that works." Therefore, pay extremely careful attention to the words used to describe the very, very simple code we start with. We are building a foundation that we will expand very quickly over this week and next week. Do not *yet* try to relate what you see back to what you already know in other languages as that is likely to lead to struggle.

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 *static environment*,[1] which is roughly the types of the preceding bindings in the file. How a binding is evaluated depends on a *dynamic environment*, which is roughly the values of the preceding bindings in the file. When we just say *environment*, we usually mean *dynamic environment*. Sometimes *context* is used as a synonym for *static environment*.

There are several kinds of bindings, but for now let's consider only a *variable binding*, which in 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* to let the *interpreter* know that you are done typing the binding.

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

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

This whole description of what ML programs mean (bindings, expressions, types, values, environments) may seem awfully theoretical or esoteric, but it is exactly the foundation we need to give precise and concise definitions for several different kinds of expressions. Here are several such definitions:

- Integer constants:
  - Syntax: a sequence of digits
  - Type-checking: type `int` in any static environment
  - Evaluation: to itself in any dynamic environment (it is a value)

[1] The word *static* here has a tenuous connection to its use in Java/C/C++, but too tenuous to explain at this point.

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

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

Using `use`

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

```
use "foo.sml";
```

does just that. Its type is `unit` and its result is `()` (the only value of type `unit`), but its effect is to include all the bindings in the file `"foo.sml"`.

Variables are Immutable

Bindings are *immutable*. Given `val x = 8+9;` we produce a dynamic environment where `x` maps to `17`. In this environment, `x` will *always* map to `17`; there is no "assignment statement" in 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.

Function Bindings

Recall that an ML program is a sequence of bindings. Each binding adds to the static environment (for type-checking subsequent bindings) and to the dynamic environment (for evaluating subsequent bindings). We already introduced variable bindings; we now introduce *function bindings*, i.e., how to define and use functions. We will then learn how to build up and use larger pieces of data from smaller ones using *pairs* and *lists*.

A function is sort of like a method in languages like Java — it is something that is called with arguments and has a body that produces a result. Unlike a method, there is no notion of a class, `this`, etc. We also do not have things like return statements. A simple example is this function that computes x^y assuming $y \geq 0$:

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

Syntax:

The syntax for a function binding looks like this (we will generalize this definition a little later in the course):

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

This is a binding for a function named `x0`. It takes n arguments `x1`, ... `xn` of types `t1`, ..., `tn` and has an expression `e` for its body. As always, syntax is just syntax — we must define the typing rules and evaluation rules for function bindings. But roughly speaking, in `e`, the arguments are bound to `x1`, ... `xn` and the result of calling `x0` is the result of evaluating `e`.

Type-checking:

To type-check a function binding, we type-check the body `e` in a static environment that (in addition to all the earlier bindings) maps `x1` to `t1`, ... `xn` to `tn` and `x0` to `t1 * ... * tn -> t`. Because `x0` is in the environment, 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 is up to the type-checker (part of the language implementation) to figure out what `t` should be such that using it for the result type of `x0` makes, "everything work out." For now, we will take it as magical, but *type inference* (figuring out types not written down) is a very cool feature of ML discussed later in the course. 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 a little later.[2]

After a function binding, `x0` is added to the static environment with its type. The arguments are not added to the top-level static environment — they can be used only in the function body.

Evaluation:

The evaluation rule for a function binding is trivial: *A function is a value* — we simply add `x0` to the environment as a function that can be *called* later. As expected for recursion, `x0` is in the dynamic environment in the function body and for subsequent bindings (but not, unlike in say Java, for preceding bindings, so the order you define functions is very important).

Function calls:

Function bindings are useful only with function calls, a new kind of expression. The ***syntax*** is `e0 (e1,...,en)` with the parentheses optional if there is exactly one argument. The ***typing rules*** require that `e0` has a type that looks like `t1*...*tn->t` and for $1 \leq i \leq n$, `ei` has type `ti`. Then the whole call has type `t`. Hopefully, this is not too surprising. For the ***evaluation rules***, we use the environment at the point of the call to evaluate `e0` to `v0`, `e1` to `v1`, ..., `en` to `vn`. Then `v0` must be a function (it will be assuming the call type-checked) and we evaluate the function's body in an environment extended such that the function arguments map to `v1`, ..., `vn`.

Exactly which environment is it we extend with the arguments? The environment that "was current" when the function was *defined*, not the one where it is being called. This distinction will not arise right now, but we will discuss it in great detail later.

Putting all this together, we can determine that this code will produce an environment where `ans` is `64`:

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

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

val ans = cube(4)
```

Pairs and Other Tuples

Programming languages need ways to build compound data out of simpler data. The first way we will learn about in ML is *pairs*. The ***syntax*** to build a pair is `(e1,e2)` which ***evaluates*** `e1` to `v1` and `e2` to `v2` and makes the pair of values `(v1,v2)`, which is itself a value. Since `v1` and/or `v2` could themselves be pairs (possibly holding other pairs, etc.), we can build data with several "basic" values, not just two, say, integers. The ***type*** of a pair is `t1*t2` where `t1` is the type of the first part and `t2` is the type of the second part.

Just like making functions is useful only if we can call them, making pairs is useful only if we can later retrieve the pieces. Until we learn pattern-matching, we will use `#1` and `#2` to access 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

[2]The way we are using pair-reading constructs like `#1` in this lecture and Homework 1 requires these explicit types.

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, and writing a return statement.

```
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)

fun div_mod (x : int, y : int) =  (* note: returning a pair is a real pain in Java *)
    (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.

Pairs and tuples can be nested however you want. For example, `(7,(true,9))` is a value of type `int * (bool * int)`, which is different from `((7,true),9)` which has type `(int * bool) * int` or `(7,true,9)` which has type `int * bool * int`.

Lists

Though we can nest pairs of pairs (or tuples) as deep as we want, for any variable that has a pair, any function that returns a pair, etc. there has to be a type for a pair and that type will determine the amount of "real data." Even with tuples the type specifies how many parts it has. That is often too restrictive; we may need a list of data (say integers) and the length of the list is not yet known when we are type-checking (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 must have the same type.

The empty list, with syntax `[]`, has 0 elements. It is a value, so like all values it evaluates to itself immediately. It can have type `t list` for *any* type `t`, which ML writes as `'a list` (pronounced "quote a list" or "alpha list"). In general, the type `t list` describes lists where all the elements in the list have type `t`. That holds for `[]` no matter what `t` is.

A non-empty list with n values is written `[v1,v2,...,vn]`. You can make a list with `[e1,...,en]` where each expression is evaluated to a value. It is more common to make a list with `e1 :: e2`, pronounced "`e1` consed onto `e2`." Here `e1` evaluates to an "item of type `t`" and `e2` evaluates to a "list of `t` values" and the result is a new list that starts with the result of `e1` and then is all the elements in `e2`.

As with functions and pairs, making lists is useful only if we can then do something with them. As with pairs, we will change how we use lists after we learn pattern-matching, but for now we will use three functions provided by ML. Each takes a list as an argument.

- `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.

Here are some simple examples of functions that take or return lists:

```
fun sum_list (xs : int list) =
    if null xs
    then 0
    else hd(xs) + sum_list(tl xs)

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

fun append (xs : int list, ys : int list) =
    if null xs
    then ys
    else (hd xs) :: append(tl xs, ys)
```

Functions that make and use lists are almost always recursive because a list has an unknown length. To write a recursive function, the thought process involves thinking about the *base case* — for example, what should the answer be for an empty list — and the *recursive case* — how can the answer be expressed in terms of the answer for the rest of the list.

When you think this way, many problems become much simpler in a way that surprises people who are used to thinking about while loops and assignment statements. A great example is the append function above that takes two lists and produces a list that is one list appended to the other. This code implements an elegant recursive algorithm: If the first list is empty, then we can append by just evaluating to the second list. Otherwise, we can append the tail of the first list to the second list. That is almost the right answer, but we need to "cons on" (using `::` has been called "consing" for decades) the first element of the first list. There is nothing magical here — we keep making recursive calls with shorter and shorter first lists and then as the recursive calls complete we add back on the list elements removed for the recursive calls.

Finally, we can combine pairs and lists however we want without having to add any new features to our language. For example, here are several functions that take a list of pairs of integers. Notice how the last function reuses earlier functions to allow for a very short solution. This is very common in functional programming. In fact, it should bother us that `firsts` and `seconds` are so similar but we do not have them share any code. We will learn how to fix that later.

```
fun sum_pair_list (xs : (int * int) list) =
    if null xs
    then 0
    else #1 (hd xs) + #2 (hd xs) + sum_pair_list(tl xs)

fun firsts (xs : (int * int) list) =
    if null xs
    then []
    else (#1 (hd xs))::(firsts(tl xs))

fun seconds (xs : (int * int) list) =
```

```
    if null xs
    then []
    else (#2 (hd xs))::(seconds(tl xs))

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

Let Expressions

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

Syntactically, a let-expression is:

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

where each `bi` is a binding and `e` is an expression.

The type-checking and semantics of a let-expression are much like the semantics of the top-level bindings in our ML program. We evaluate each binding in turn, creating a larger 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-expression (the `e`). The value `e` evaluates to is the value for the entire let-expression, and, unsurprisingly, the type of `e` is the type for the entire let-expression.

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

```
let 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 is 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 evaluate `count`'s body in a dynamic environment that is the environment where `count` was defined, extended with bindings for `count`'s arguments.

The code above does not really utilize this: `count`'s body uses only `from`, `to`, and `count` (for recursion). It could also use `x`, since that is in the environment when `count` is defined. Then we do not need `to` at all, since in the code above it always has the same value as `x`. So this is better style:

```
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 (xs : int list) =
    if null xs
    then 0 (* note: bad style; see below *)
    else if null (tl xs)
    then hd xs
    else if hd xs > bad_max(tl xs)
    then hd xs
    else bad_max(tl xs)
```

If you call `bad_max` with `countup_from1 30`, it will make approximately 2^{30} (over one billion) recursive calls to itself. The reason is an "exponential blowup" — the code calls `bad_max(tl xs)` twice and each of those calls call `bad_max` two more times (so four total) and so on. This sort of programming "error" can be difficult to detect because it can depend on your test data (if the list counts down, the algorithm makes only 30 recursive calls instead of 2^{30}).

We can use let-expressions to avoid repeated computations. This version computes the max of the tail of the list once and stores the resulting value in `tl_ans`.

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

Options

The previous example does not properly handle the empty list — it returns 0. This is bad style because 0 is really not the maximum value of 0 numbers. There is no good answer, but we should deal with this case reasonably. One possibility is to raise an exception; you can learn about ML exceptions on your own if you are interested before we discuss them later in the course. Instead, let's change the return type to either return the maximum number or indicate the input list was empty so there is no maximum. Given the constructs we have, we could "code this up" by return an `int list`, using `[]` if the input was the empty list and a list with one integer (the maximum) if the input list was not empty.

While that works, lists are "overkill" — we will always return a list with 0 or 1 elements. So a list is not really a precise description of what we are returning. The 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`. The type of `NONE` is `'a option` and the type of `SOME e` is `t option` if `e` has type `t`.

Given a value, how do you use it? Just like we have `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 (xs : int list) =
    if null xs
    then NONE
    else
        let val tl_ans = better_max(tl xs)
        in if isSome tl_ans andalso valOf tl_ans > hd xs
           then tl_ans
           else SOME (hd xs)
        end
```

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

```
fun better_max2 (xs : int list) =
    if null xs
    then NONE
    else let (* fine to assume argument nonempty because it is local *)
            fun max_nonempty (xs : int list) =
                if null (tl xs) (* xs must not be [] *)
                then hd xs
                else let val tl_ans = max_nonempty(tl xs)
                     in
                         if hd xs > tl_ans
                         then hd xs
                         else tl_ans
                     end
```

```
in
    SOME (max_nonempty xs)
end
```

Some Other Expressions and Operators

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

- `e1 andalso e2` is logical-and: It evaluates `e2` only if `e1` evaluates to `true`. The result is `true` if `e1` and `e2` evaluate to true. Naturally, `e1` and `e2` must both have type `bool` and the entire expression also has type `bool`. In many languages, such expressions are written `e1 && e2`, but that is not the ML syntax, nor is `e1 and e2` (but `and` is a keyword we will encounter later for a different purpose). Using `e1 andalso e2` is generally better style than the equivalent `if e1 then e2 else false`.
- `e1 orelse e2` is logical-or: It evaluates `e2` only if `e1` evaluates to `false`. The result is `true` if `e1` or `e2` evaluates to true. Naturally, `e1` and `e2` must both have type `bool` and the entire expression also has type `bool`. In many languages, such expressions are written `e1 || e2`, but that is not the ML syntax, nor is `e1 or e2`. Using `e1 orelse e2` is generally better style than the equivalent `if e1 then true else e2`.
- `not e` is logical-negation. `not` is just a provided function of type `bool->bool` that we could have defined ourselves as `fun not x = if x then false else true`. In many languages, such expressions are written `!e`, but in ML the `!` operator means something else (related to mutable variables, which we will not use).
- You can compare many values, including integers, for equality using `e1 = e2`.
- Instead of writing `not (e1 = e2)` to see if two numbers are different, better style is `e1 <> e2`. In many languages, the syntax is `e1 != e2`, whereas ML's `<>` can be remembered as, "less than or greater than."
- The other arithmetic comparisons have the same syntax as in most languages: `>`, `<`, `>=`, `<=`.
- Subtraction is written `e1 - e2`, but it must take two operands, so you *cannot* just write `- e` for negation. For negation, the correct syntax is `~ e`, in particular negative numbers are written like `~7`, *not* `-7`. Using `~e` is better style than `0 - e`, but equivalent for integers.

Lack of Mutation and Benefits Thereof

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

This is a really powerful feature! That may surprise you: how can a language *not* having something be a feature? Because if there is no such feature, then when you are writing *your code* you can rely on *no other code* doing something that would make your code wrong, incomplete, or difficult to use. Having

immutable data is probably the most important "non-feature" a language can have, and it is one of the main contributions of functional programming.

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

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

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

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

would now have `x` and `y` be aliases for the *same* pair? The answer is *you cannot tell* — there is no construct in ML that can figure out whether or not `x` and `y` are aliases, and no reason to worry that they might be. *If* we had mutation, life would be different. Suppose we could say, "change the second part of the pair `x` is bound to so that it holds 5 instead of 4." Then we would have to wonder if `#2 y` would be 4 or 5.

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

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

Making the new pair `(#1 pr, #2 pr)` is bad style, since `pr` is simpler and will do just as well. Yet in languages with mutation, programmers make copies like this all the time, exactly to prevent aliasing where doing an assignment using one variable like `x` causes unexpected changes to using another variable like `y`. In ML, no users of `sort_pair` can ever tell whether we return a new pair or not.

Our second example is our elegant function for list append:

```
fun append (xs : int list, ys : int list) =
    if null xs
    then ys
    else (hd xs) :: append(tl xs, ys)
```

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

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

The `append` example is very similar to the `sort_pair` example, but it is even more compelling because it is hard to keep track of potential aliasing if you have many lists of potentially large lengths. If I append `[1,2]` to `[3,4,5]`, I will get *some* list `[1,2,3,4,5]` but if later someone can *change* the `[3,4,5]` list to be `[3,7,5]` is the appended list still `[1,2,3,4,5]` or is it now `[1,2,3,7,5]`?

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

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

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

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

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

The Pieces of a Programming Language

Now that we have learned enough ML to write some simple functions and programs with it, we can list the essential "pieces" necessary for defining and learning *any* 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 could not do without library support (like access files)?
- Tools: What is available for manipulating programs in the language (compilers, read-eval-print loops, debuggers, ...)

While libraries and tools are essential for being an effective programmer (to avoid reinventing available solutions or unnecessarily doing things manually), this course does not focus on them much. That can leave the wrong impression that we are using "silly" or "impractical" languages, but libraries and tools are just less relevant in a course on the conceptual similarities and differences of programming languages.

Programming Languages (Coursera / University of Washington) Assignment 1

You will write 11 SML functions (and tests for them) related to calendar dates. In all problems, a "date" is an SML value of type `int*int*int`, where the first part is the year, the second part is the month, and the third part is the day. A "reasonable" date has a positive year, a month between 1 and 12, and a day no greater than 31 (or less depending on the month). Your solutions need to work correctly only for reasonable dates, but do not check for reasonable dates (that is a challenge problem) and many of your functions will naturally work correctly for some/all non-reasonable dates. A "day of year" is a number from 1 to 365 where, for example, 33 represents February 2. (We ignore leap years except in one challenge problem.)

1. Write a function `is_older` that takes two dates and evaluates to true or false. It evaluates to true if the first argument is a date that comes before the second argument. (If the two dates are the same, the result is false.)
2. Write a function `number_in_month` that takes a list of dates and a month (i.e., an `int`) and returns how many dates in the list are in the given month.
3. Write a function `number_in_months` that takes a list of dates and a list of months (i.e., an `int list`) and returns the number of dates in the list of dates that are in any of the months in the list of months. *Assume the list of months has no number repeated.* Hint: Use your answer to the previous problem.
4. Write a function `dates_in_month` that takes a list of dates and a month (i.e., an `int`) and returns a list holding the dates from the argument list of dates that are in the month. The returned list should contain dates in the order they were originally given.
5. Write a function `dates_in_months` that takes a list of dates and a list of months (i.e., an `int list`) and returns a list holding the dates from the argument list of dates that are in any of the months in the list of months. *Assume the list of months has no number repeated.* Hint: Use your answer to the previous problem and SML's list-append operator (`@`).
6. Write a function `get_nth` that takes a list of strings and an `int` n and returns the n^{th} element of the list where the head of the list is 1^{st}. Do not worry about the case where the list has too few elements: your function may apply `hd` or `tl` to the empty list in this case, which is okay.
7. Write a function `date_to_string` that takes a date and returns a `string` of the form `January 20, 2013` (for example). Use the operator `^` for concatenating strings and the library function `Int.toString` for converting an `int` to a `string`. For producing the month part, do *not* use a bunch of conditionals. Instead, use a list holding 12 strings and your answer to the previous problem. For consistency, put a comma following the day and use capitalized English month names: January, February, March, April, May, June, July, August, September, October, November, December.
8. Write a function `number_before_reaching_sum` that takes an `int` called *sum*, which you can assume is positive, and an `int list`, which you can assume contains all positive numbers, and returns an `int`. You should return an int n such that the first n elements of the list add to less than sum, but the first $n+1$ elements of the list add to *sum* or more. Assume the entire list sums to more than the passed in value; it is okay for an exception to occur if this is not the case.
9. Write a function `what_month` that takes a day of year (i.e., an `int` between 1 and 365) and returns what month that day is in (1 for January, 2 for February, etc.). Use a list holding 12 integers and your answer to the previous problem.
10. Write a function `month_range` that takes two days of the year `day1` and `day2` and returns an `int list` `[m1,m2,...,mn]` where `m1` is the month of `day1`, `m2` is the month of `day1+1`, ..., and `mn` is the month of day `day2`. Note the result will have length `day2 - day1 + 1` or length 0 if `day1>day2`.
11. Write a function `oldest` that takes a list of dates and evaluates to an `(int*int*int) option`. It evaluates to `NONE` if the list has no dates and `SOME` d if the date d is the oldest date in the list.

12. **Challenge Problem:** Write functions `number_in_months_challenge` and `dates_in_months_challenge` that are like your solutions to problems 3 and 5 except having a month in the second argument multiple times has no more effect than having it once. (Hint: Remove duplicates, then use previous work.)

13. **Challenge Problem:** Write a function `reasonable_date` that takes a date and determines if it describes a real date in the common era. A "real date" has a positive year (year 0 did not exist), a month between 1 and 12, and a day appropriate for the month. Solutions should properly handle leap years. Leap years are years that are either divisible by 400 or divisible by 4 but not divisible by 100. (Do not worry about days possibly lost in the conversion to the Gregorian calendar in the Late 1500s.)

Note: Remember challenge problems are not required for a high grade and will be worth (only) a few points.

Note: The sample solution contains *roughly* 75–80 lines of code, not including challenge problems.

Syntax Hints

Small syntax errors can lead to strange error messages. Here are 3 examples for function definitions:

1. `int * int * int list` means `int * int * (int list)`, not `(int * int * int) list`.
2. `fun f x : t` means the *result type* of `f` is `t`, whereas `fun f (x:t)` means the *argument type* of `f` is `t`. There is no need to write result types (and in later assignments, no need to write argument types).
3. `fun (x t)`, `fun (t x)`, or `fun (t : x)` are all wrong, but the error message suggests you are trying to do something much more advanced than you actually are (which is trying to write `fun (x : t)`).

Summary

Evaluating a correct homework solution should generate these bindings:

```
val is_older = fn : (int * int * int) * (int * int * int) -> bool
val number_in_month = fn : (int * int * int) list * int -> int
val number_in_months = fn : (int * int * int) list * int list -> int
val dates_in_month = fn : (int * int * int) list * int -> (int * int * int) list
val dates_in_months = fn : (int * int * int) list * int list -> (int * int * int) list
val get_nth = fn : string list * int -> string
val date_to_string = fn : int * int * int -> string
val number_before_reaching_sum = fn : int * int list -> int
val what_month = fn : int -> int
val month_range = fn : int * int -> int list
val oldest = fn : (int * int * int) list -> (int * int * int) option
```

Of course, generating these bindings does not guarantee that your solutions are correct. *Test your functions: Put your testing code in a separate file. We will not grade the testing file, nor will you turn it in, but surely you want to run your functions and record your test inputs in a file.*

Assessment

We will automatically test your functions on a variety of inputs, including edge cases. We will also ask peers to evaluate your code for simplicity, conciseness, elegance, and good formatting including indentation and line breaks. Your solution will also be checked for using only features discussed so far in class. In particular, you must not use SML's mutable references or arrays. Do not use pattern matching (until the next assignment where we will require it).

Turn-in Instructions

First, follow the instructions on the course website to submit your solution file (not your testing file) for auto-grading. Do not proceed to the peer-assessment submission until you receive a high-enough grade from the auto-grader: Doing peer assessment requires instructions that include a sample solution, so these instructions will be "locked" until you receive high-enough auto-grader score. Then submit your same solution file again for peer assessment and follow the peer-assessment instructions.

Coursera Programming Languages Course
Section 2 Summary

Standard Description: This summary covers **roughly** *the same material as the lecture videos and the materials (slides, code) posted with each video. It can help to read about the material in a narrative style and to have the material for an entire section of the course in a single document, especially when reviewing the material later. Please report errors in these notes on the discussion board.*

Contents

Conceptual Ways to Build New Types

Programming languages have *base types*, like `int`, `bool`, and `unit` and *compound types*, which are types that contain other types in their definition. We have already seen ways to make compound types in ML, namely by using tuple types, list types, and option types. We will soon learn new ways to make even more flexible compound types and to give names to our new types. To create a compound type, there are really only three essential building blocks. Any decent programming language provides these building blocks in some way:[1]

[1] As a matter of jargon you do not need to know, the terms "each-of types," "one-of types," and "self-reference types" are not standard – they are just good ways to think about the concepts. Usually people just use constructs from a particular language like "tuples" when they are talking about the ideas. Programming-language researchers use the terms "product types," "sum types," and "recursive types." Why product and sum? It is related to the fact that in Boolean algebra where 0 is false and 1 is true, *and* works like multiply and *or* works like addition.

- "Each-of": A compound type `t` describes values that contain *each of* values of type `t1`, `t2`, ..., ***and*** `tn`.
- "One-of": A compound type `t` describes values that contain a value of *one of* the types `t1`, `t2`, ..., ***or*** `tn`.
- "Self-reference": A compound type `t` may refer to itself in its definition in order to describe recursive data structures like lists and trees.

Each-of types are the most familiar to most programmers. Tuples are an example: `int * bool` describes values that contain an `int` *and* a `bool`. A Java class with fields is also an each-of sort of thing.

One-of types are also very common but unfortunately are not emphasized as much in many introductory programming courses. `int option` is a simple example: A value of this type contains an `int` *or* it does not. For a type that contains an `int` *or* a `bool` in ML, we need datatype bindings, which are the main focus of this section of the course. In object-oriented languages with classes like Java, one-of types are achieved with subclassing, but that is a topic for much later in the course.

Self-reference allows types to describe recursive data structures. This is useful in combination with each-of and one-of types. For example, `int list` describes values that either contain nothing *or* contain an `int` *and* another `int list`. A list of integers in any programming language would be described in terms of *or*, *and*, and *self-reference* because that is what it means to be a list of integers.

Naturally, since compound types can nest, we can have any nesting of each-of, one-of, and self-reference. For example, consider the type `(int * bool) list list * (int option) list * bool`.

Records: Another Approach to "Each-of" Types

Record types are "each-of" types where each component is a *named field*. For example, the type `{foo : int, bar : int*bool, baz : bool*int}` describes records with three fields named `foo`, `bar`, and `baz`. This is just a new sort of type, just like tuple types were new when we learned them.

A *record expression* builds a *record value*. For example, the expression `{bar = (1+2,true andalso true), foo = 3+4, baz = (false,9) }` would evaluate to the record value `{bar = (3,true), foo = 7, baz = (false,9)}`, which can have type `{foo : int, bar : int*bool, baz : bool*int}` because the order of fields never matters (we use the field names instead). In general the syntax for a record expression is `{f1 = e1, ..., fn = en}` where, as always, each `ei` can be any expression. Here each `f` can be any field name (though each must be different). A field name is basically any sequence of letters or numbers.

In ML, we do not have to declare that we want a record type with particular field names and field types — we just write down a record expression and the type-checker gives it the right type. The type-checking rules for record expressions are not surprising: Type-check each expression to get some type `ti` and then build the record type that has all the right fields with the right types. *Because the order of field names never matters, the REPL always alphabetizes them when printing just for consistency.*

The evaluation rules for record expressions are analogous: Evaluate each expression to a value and create the corresponding record value.

Now that we know how to build record values, we need a way to access their pieces. For now, we will use `#foo e` where `foo` is a field name. Type-checking requires `e` has a record type with a field named `foo`, and if this field has type `t`, then that is the type of `#foo e`. Evaluation evaluates `e` to a record value and then produces the contents of the `foo` field.

By Name vs. By Position, Syntactic Sugar, and The Truth About Tuples

Records and tuples are *very* similar. They are both "each-of" constructs that allow any number of components. The only real difference is that records are "by name" and tuples are "by position." This means with records we build them and access their pieces by using field names, so the order we write the fields in a record expression does not matter. But tuples do not have field names, so we use the position (first, second, third, ...) to distinguish the components.

By name versus by position is a classic decision when designing a language construct or choosing which one to use, with each being more convenient in certain situations. As a rough guide, by position is simpler for a small number of components, but for larger compound types it becomes too difficult to remember which position is which.

Java method arguments (and ML function arguments as we have described them so far) actually take a hybrid approach: The method body uses variable *names* to refer to the different arguments, but the caller passes arguments by *position*. There are other languages where callers pass arguments by name.[2]

Despite "by name vs. by position," records and tuples are still so similar that we can define tuples entirely in terms of records. Here is how:

- When you write `(e1,...,en)`, it is another way of writing `{1=e1,...,n=en}`, i.e., a tuple expression is a record expression with field names 1, 2, ..., n.
- The type `t1 * ... * tn` is just another way of writing `{1:t1, ..., n:tn}`.
- Notice that `#1 e`, `#2 e`, etc. now already mean the right thing: get the contents of the field named 1, 2, etc.

In fact, this is how ML actually defines tuples: A tuple *is* a record. That is, all the syntax for tuples is just a convenient way to write down and use records. The REPL just always uses the tuple syntax where possible, so if you evaluate `{2=1+2, 1=3+4}` it will print the result as `(7,3)`. Using the tuple *syntax* is better style, but we did not need to give tuples their own *semantics*: we can instead use the "another way of writing" rules above and then reuse the semantics for records.

This is the first of many examples we will see of ***syntactic sugar***. We say, "tuples are just syntactic sugar for records with fields named 1, 2, ..., n." It is *syntactic* because we can describe everything about tuples in terms of equivalent record syntax. It is *sugar* because it makes the language sweeter. The term *syntactic sugar* is widely used. Syntactic sugar is a great way to keep the key ideas in a programming-language small (making it easier to implement) while giving programmers convenient ways to write things. Indeed, in Homework 1 we used tuples without knowing records existed even though tuples are records.

Datatype Bindings: Our Own "One-of" Types

We now introduce *datatype bindings*, our third kind of binding after variable bindings and function bindings. We start with a silly but simple example because it will help us see the many different aspects of a datatype binding. We can write:

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

[2]The phrase "call by name" actually means something else in relation to function arguments. It is a different topic.

Roughly, this defines a new type where values have an `int * int` *or* a `string` *or* nothing. Any value will also be "tagged" with information that lets us know which *variant* it is: These "tags," which we will call *constructors*, are `TwoInts`, `Str`, and `Pizza`. Two constructors could be used to tag the same type of underlying data; in fact this is common even though our example uses different types for each variant.

More precisely, the example above adds four things to the environment:

- A new type `mytype` that we can now use just like any other type
- Three *constructors* `TwoInts`, `Str`, and `Pizza`

A *constructor* is two different things. First, it is either a function for creating values of the new type (if the variant has `of t` for some type `t`) or it is actually a value of the new type (otherwise). In our example, `TwoInts` is a function of type `int*int -> mytype`, `Str` is a function of type `string->mytype`, and `Pizza` is a value of type `mytype`. Second, we use constructors in case-expressions as described further below.

So we know how to build values of type `mytype`: call the constructors (they are functions) with expressions of the right types (or just use the `Pizza` value). The result of these function calls are values that "know which variant they are" (they store a "tag") and have the underlying data passed to the constructor. The REPL represents these values like `TwoInts(3,4)` or `Str "hi"`.

What remains is a way to retrieve the pieces...

How ML Does *Not* Provide Access to Datatype Values

Given a value of type `mytype`, how can we access the data stored in it? First, *we need to find out which variant it is* since a value of type `mytype` might have been made from `TwoInts`, `Str`, or `Pizza` and this affects what data is available. Once we know what variant we have, then we can access the pieces, if any, that variant carries.

Recall how we have done this so for lists and options, which are also one-of types: We had functions for testing which variant we had (`null` or `isSome`) and functions for getting the pieces (`hd`, `tl`, or `valOf`), which raised exceptions if given arguments of the wrong variant.

ML could have taken the same approach for datatype bindings. For example, it could have taken our datatype definition above and added to the environment functions `isTwoInts`, `isStr`, and `isPizza` all of type `mytype -> bool`. And it could have added functions like `getTwoInts` of type `mytype -> int*int` and `getStr` of type `mytype -> string`, which might raise exceptions.

But ML does not take this approach. Instead it does something better. You could write these functions yourself using the better thing, though it is usually poor style to do so. In fact, after learning the better thing, we will no longer use the functions for lists and options the way we have been — we just started with these functions so we could learn one thing at a time.

How ML Provides Access to Datatype Values: Case Expressions

The better thing is a *case expression*. Here is a basic example for our example datatype binding:

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

In one sense, a case-expression is like a more powerful if-then-else expression: Like a conditional expression, it evaluates two of its subexpressions: first the expression between the `case` and `of` keywords and second the expression in the *first branch that matches*. But instead of having two branches (one for `true` and one for `false`), we can have one branch for each variant of our datatype (and we will generalize this further below). Like conditional expressions, each branch's expression must have the same type (`int` in the example above) because the type-checker cannot know what branch will be used.

Each branch has the form `p => e` where `p` is a *pattern* and `e` is an expression, and we separate the branches with the `|` character. Patterns look like expressions, but do not think of them as expressions. Instead they are used to *match* against the result of evaluating the case's first expression (the part after `case`). This is why evaluating a case-expression is called *pattern-matching*.

For this lecture, we keep pattern-matching simple: Each pattern uses a different constructor and pattern-matching picks the branch with the "right one" given the expression after the word `case`. The result of evaluating that branch is the overall answer; no other branches are evaluated. For example, if `TwoInts(7,9)` is passed to `f`, then the second branch will be chosen.

That takes care of the "check the variant" part of using the one-of type, but pattern matching *also* takes care of the "get out the underlying data" part. 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 (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. The type-checker knows what types these variables have because they were specified in the datatype binding that created the constructor used in the pattern.

Why are case-expressions better than functions for testing variants and extracting pieces?

- We can never "mess up" and try to extract something from the wrong variant. That is, we will not get exceptions like we get with `hd []`.
- If a case expression forgets a variant, then the type-checker will give a warning message. This indicates that evaluating the case-expression could find no matching branch, in which case it will raise an exception. If you have no such warnings, then you know this does not occur.
- If a case expression uses a variant twice, then the type-checker will give an error message since one of the branches could never possibly be used.
- If you still want functions like `null` and `hd`, you can easily write them yourself (but do not do so for your homework).
- Pattern-matching is much more general and powerful than we have indicated so far. We give the "whole truth" about pattern-matching below.

Useful Examples of "One-of" Types

Let us now consider several examples where "one-of" types are useful, since so far we considered only a silly example.

First, they are good for enumerating a fixed set of options – and much better style than using, say, small integers. For example:

```
datatype suit = Club | Diamond | Heart | Spade
```

Many languages have support for this sort of *enumeration* including Java and C, but ML takes the next step of letting variants carry data, so we can do things like this:

```
datatype rank = Jack | Queen | King | Ace | Num of int
```

We can then combine the two pieces with an each-of type: `suit * rank`

One-of types are also useful when you have different data in different situations. For example, suppose you want to identify students by their id-numbers, but in case there are students that do not have one (perhaps they are new to the university), then you will use their full name instead (with first name, optional middle name, and last name). This datatype binding captures the idea directly:

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

Unfortunately, this sort of example is one where programmers often show a profound lack of understanding of one-of types and insist on using each-of types, which is like using a saw as a hammer (it works, but you are doing the wrong thing). Consider BAD code like this:

```
(* If student_num is -1, then use the other fields, otherwise ignore other fields *)
 {student_num : int, first : string, middle : string option, last : string}
```

This approach requires all the code to follow the rules in the comment, with no help from the type-checker. It also wastes space, having fields in every record that should not be used.

On the other hand, each-of types are exactly the right approach if we want to store for each student their id-number (if they have one) *and* their full name:

```
{ student_num : int option,
  first       : string,
  middle      : string option,
  last        : string }
```

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

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

Thanks to the self-reference, what this data definition really describes is *trees* where the leaves are integers and the internal nodes are either negations with one child, additions with two children or multiplications 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)
      | Multiply(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).

There are many functions we might write over values of type `exp` and most of them will use pattern-matching and recursion in a similar way. Here are other functions you could write that process an `exp` argument:

- The largest constant in an expression
- A list of all the constants in an expression (use list append)
- A `bool` indicating whether there is at least one multiplication in the expression
- The number of addition expressions in an expression

Here is the last one:

```
fun number_of_adds e =
    case e of
        Constant i      => 0
      | Negate e2       => number_of_adds e2
      | Add(e1,e2)      => 1 + number_of_adds e1 + number_of_adds e2
      | Multiply(e1,e2) => number_of_adds e1 + number_of_adds e2
```

Datatype Bindings and Case Expressions So Far, Precisely

We can summarize what we know about datatypes and pattern matching so far 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" and such a constructor just has type `t`. 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* `v`, 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`.

Type Synonyms

Before continuing our discussion of datatypes, let's contrast them with another useful kind of binding that also introduces a new type name. A *type synonym* simply creates another name for an existing type that is entirely interchangeable with the existing type.

For example, if we write:

```
type foo = int
```

then we can write `foo` wherever we write `int` and vice-versa. So given a function of type `foo->foo` we could call the function with 3 and add the result to 4. The REPL will sometimes print `foo` and sometimes print `int` depending on the situation; the details are unimportant and up to the language implementation. For a type like `int`, such a synonym is not very useful (though later when we study ML's module system we will build on this feature).

But for more complicated types, it can be convenient to create type synonyms. Here are some examples for types we created above:

```
type card = suit * rank

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

Just remember these synonyms are fully interchangeable. For example, if a homework question requires a function of type `card -> int` and the REPL reports your solution has type `suit * rank -> int`, this is okay because the types are "the same."

In contrast, datatype bindings introduce a type that is not the same as any existing type. It creates constructors that produces values of this new type. So, for example, the only type that is the same as `suit` is `suit` unless we later introduce a synonym for it.

Lists and Options are Datatypes

Because datatype definitions can be recursive, we can use them to create our own types for lists. For example, this binding works well for a linked list of integers:

```
datatype my_int_list = Empty
                     | Cons of int * my_int_list
```

We can use the constructors `Empty` and `Cons` to make values of `my_int_list` and we can use case expressions to use such values:[3]

```
val one_two_three = Cons(1,Cons(2,Cons(3,Empty)))

fun append_mylist (xs,ys) =
    case xs of
        Empty => ys
      | Cons(x,xs') => Cons(x, append_mylist(xs',ys))
```

It turns out the lists and options "built in" (i.e., predefined with some special syntactic support) are just datatypes. As a matter of style, it is better to use the built-in widely-known feature than to invent your own.

[3] In this example, we use a variable `xs'`. Many languages do not allow the character `'` in variable names, but ML does and it is common in mathematics to use it and pronounce such a variable "exes prime."

More importantly, it is better style to use pattern-matching for accessing list and option values, ***not*** the functions `null`, `hd`, `tl`, `isSome`, and `valOf` we saw previously. (We used them because we had not learned pattern-matching yet and we did not want to delay practicing our functional-programming skills.)

For options, all you need to know is `SOME` and `NONE` are constructors, which we use to create values (just like before) and in patterns to access the values. Here is a short example of the latter:

```
fun inc_or_zero intoption =
    case intoption of
        NONE => 0
      | SOME i => i+1
```

The story for lists is similar with a few convenient syntactic peculiarities: `[]` really is a constructor that carries nothing and `::` really is a constructor that carries two things, but `::` is unusual because it is an infix operator (it is placed between its two operands), both when creating things and in patterns:

```
fun sum_list xs =
    case xs of
        [] => 0
      | x::xs' => x + sum_list xs'

fun append (xs,ys) =
    case xs of
        [] => ys
      | x::xs' => x :: append(xs',ys)
```

Notice here `x` and `xs'` are nothing but local variables introduced via pattern-matching. We can use any names for the variables we want. We could even use `hd` and `tl` — doing so would simply shadow the functions predefined in the outer environment.

The reasons why you should usually prefer pattern-matching for accessing lists and options instead of functions like `null` and `hd` is the same as for datatype bindings in general: you cannot forget cases, you cannot apply the wrong function, etc. So why does the ML environment predefine these functions if the approach is inferior? In part, because they are useful for passing as arguments to other functions, a major topic for the next section of the course.

Polymorphic Datatypes

Other than the strange syntax of `[]` and `::`, the only thing that distinguishes the built-in lists and options from our example datatype bindings is that the built-in ones are *polymorphic* – they can be used for carrying values of *any* type, as we have seen with `int list`, `int list list`, `(bool * int) list`, etc. You can do this for your own datatype bindings too, and indeed it is very useful for building "generic" data structures. While we will not focus on using this feature here (i.e., you are not responsible for knowing how to do it), there is nothing very complicated about it. For example, this is *exactly* how options are pre-defined in the environment:

```
datatype 'a option = NONE | SOME of 'a
```

Such a binding does *not* introduce a *type* `option`. Rather, it makes it so that if `t` is a type, then `t option` is type. You can also define polymorphic datatypes that take multiple types. For example, here is a binary tree where internal nodes hold values of type `'a` and leaves hold values of type `'b`

```
datatype ('a,'b) tree = Node of 'a * ('a,'b) tree * ('a,'b) tree
                      | Leaf of 'b
```

We then have types like `(int,int) tree` (in which every node and leaf holds an `int`) and `(string,bool) tree` (in which every node holds a `string` and every leaf holds a `bool`). The way you use constructors and pattern-matching is the same for regular datatypes and polymorphic datatypes.

Pattern-Matching for Each-Of Types: The Truth About Val-Bindings

So far we have used pattern-matching for one-of types, but we can use it 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: the pattern `(x1,...,xn)` is the same as `{1=x1,...,n=xn}` and matches the tuple value `(v1,...,vn)`, which is the same as `{1=v1,...,n=vn}`. So we could write this function for summing the three parts of an `int * int * int`:

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

And a similar example with records (and ML's string-concatenation operator) could look like this:

```
fun full_name (r : {first:string,middle:string,last:string}) =
    case r of
        {first=x,middle=y,last=z} => x ^ " " ^ y ^ " " ^z
```

However, a case-expression with one branch is poor style — it looks strange because the purpose of such expressions is to distinguish *cases*, plural. So how should we use pattern-matching for each-of types, when we know that a single pattern will definitely match so we are using pattern-matching just for the convenient extraction of values? It turns out you can use patterns in val-bindings too! So this approach is better style:

```
fun full_name (r : {first:string,middle:string,last:string}) =
    let val {first=x,middle=y,last=z} = r
    in
        x ^ " " ^ y ^ " " ^z
    end
fun sum_triple (triple : int*int*int) =
    let val (x,y,z) = triple
    in
        x + y + z
    end
```

Actually we can do even better: Just like a pattern can be used in a val-binding to bind variables (e.g., `x`, `y`, and `z`) to the various pieces of the expression (e.g., `triple`), we can use a pattern when defining a function binding and the pattern will be used to introduce bindings by matching against the value passed when the function is called. So here is the third and best approach for our example functions:

```
fun full_name {first=x,middle=y,last=z} =
    x ^ " " ^ y ^ " " ^z
```

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

This version of `sum_triple` should intrigue you: It takes a triple as an argument and uses pattern-matching to bind three variables to the three pieces for use in the function body. But it looks exactly like a function that takes three arguments of type `int`. Indeed, is the type `int*int*int->int` for three-argument functions or for one argument functions that take triples?

It turns out we have been basically lying: There is no such thing as a multi-argument function in ML: ***Every function in ML takes exactly one argument!*** Every time we write a multi-argument function, we are really writing a one-argument function that takes a tuple as an argument and uses pattern-matching to extract the pieces. This is such a common idiom that it is easy to forget about and it is totally fine to talk about "multi-argument functions" when discussing your ML code with friends. But in terms of the actual language definition, it really is a one-argument function: syntactic sugar for expanding out to the first version of `sum_triple` with a one-arm case expression.

This flexibility is sometimes useful. In languages like C and Java, you cannot have one function/method compute the results that are immediately passed to another multi-argument function/method. But with one-argument functions that are tuples, this works fine. Here is a silly example where we "rotate a triple to the right" by "rotating it to the left twice":

```
fun rotate_left (x,y,z) = (y,z,x)
fun rotate_right triple = rotate_left(rotate_left triple)
```

More generally, you can compute tuples and then pass them to functions even if the writer of that function was thinking in terms of multiple arguments.

What about zero-argument functions? They do not exist either. The binding `fun f () = e` is using the unit-pattern `()` to match against calls that pass the unit value `()`, which is the only value of type `unit`. The type unit is just a datatype with only one constructor, which takes no arguments and uses the unusual syntax `()`. Basically, `datatype unit = ()` comes pre-defined.

Digression: Type inference

By using patterns to access values of tuples and records rather than `#foo`, you will find it is no longer necessary to write types on your function arguments. In fact, it is conventional in ML to leave them off — you can always use the REPL to find out a function's type. The reason we needed them before is that `#foo` does not give enough information to type-check the function because the type-checker does not know what other fields the record is supposed to have, but the record/tuple patterns introduced above provide this information. In ML, every variable and function has a type (or your program fails to type-check) — type inference *only* means you do not need to write down the type.

So none of our examples above that used pattern-matching instead of `#middle` or `#2` need argument types. It is often better style to write these less cluttered versions, where again the last one is the best:

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

```
        x + y + z
    end
fun sum_triple (x,y,z) =
    x + y + z
```

This version needs an explicit type on the argument:

```
fun sum_triple (triple : int * int * int) =
    #1 triple + #2 triple + #3 triple
```

The reason is the type-checker cannot take

```
fun sum_triple triple =
    #1 triple + #2 triple + #3 triple
```

and infer that the argument must have type `int*int*int`, since it could also have type `int*int*int*int` or `int*int*int*string` or `int*int*int*bool*string` or an infinite number of other types. If you do not use `#`, ML almost never requires explicit type annotations thanks to the convenience of type inference.

In fact, type inference sometimes reveals that functions are more general than you might have thought. Consider this code, which does use part of a tuple/record:

```
fun partial_sum (x,y,z) = x + z
fun partial_name {first=x, middle=y, last=z} = x ^ " " ^ z
```

In both cases, the inferred function types reveal that the type of `y` can be *any* type, so we can call `partial_sum (3,4,5)` or `partial_sum (3,false,5)`.

We will discuss these *polymorphic functions* as well as how *type inference* works in future sections because they are major course topics in their own right. For now, just stop using `#`, stop writing argument types, and do not be confused if you see the occasional type like `'a` or `'b` due to type inference, as discussed a bit more next...

Digression: Polymorphic Types and Equality Types

We now encourage you to leave explicit type annotations out of your program, but as seen above that can lead to surprisingly general types. Suppose you are asked to write a function of type `int*int*int -> int` that behaves like `partial_sum` above, but the REPL indicates, correctly, that `partial_sum` has type `int*'a*int->int`. *This is okay* because the *polymorphism* indicates that `partial_sum` has a *more general* type. If you can take a type containing `'a`, `'b`, `'c`, etc. and replace each of these *type variables* consistently to get the type you "want," then you have a more general type than the one you want.

As another example, `append` as we have written it has type `'a list * 'a list -> 'a list`, so by consistently replacing `'a` with `string`, we can use `append` as though it has the type `string list * string list -> string list`. We can do this with any type, not just `string`. And we do not actually *do* anything: this is just a mental exercise to check that a type is more general than the one we need. Note that type variables like `'a` must be replaced *consistently*, meaning the type of `append` is *not* more general than `string list * int list -> string list`.

You may also see type variables with two leading apostrophes, like `''a`. These are called *equality types* and they are a fairly strange feature of ML not relevant to our current studies. Basically, the `=` operator in ML

(for comparing things) works for many types, not just `int`, but its two operands must have the same type. For example, it works for `string` as well as tuple types for which all types in the tuple support equality (e.g., `int * (string * bool)`. But it does not work for every type.[4] A type like `''a` can only have an "equality type" substituted for it.

```
fun same_thing(x,y) = if x=y then "yes" else "no" (* has type ''a * ''a -> string *)
fun is_three x = if x=3 then "yes" else "no" (* has type int -> string *)
```

Again, we will discuss polymorphic types and type inference more later, but this digression is helpful for avoiding confusion on Homework 2: if you write a function that the REPL gives a more general type to than you need, that is okay. Also remember, as discussed above, that it is also okay if the REPL uses different type synonyms than you expect.

Nested Patterns

It turns out the definition of patterns is recursive: anywhere we have been putting a variable in our patterns, we can instead put another pattern. Roughly speaking, the semantics of pattern-matching is that the value being matched must have the same "shape" as the pattern and variables are bound to the "right pieces." (This is very hand-wavy explanation which is why a precise definition is described below.) For example, the pattern `a::(b::(c::d))` would match any list with at least 3 elements and it would bind `a` to the first element, `b` to the second, `c` to the third, and `d` to the list holding all the other elements (if any). The pattern `a::(b::(c::[]))` on the other hand, would match only lists with exactly three elements. Another nested patterns is `(a,b,c)::d`, which matches any non-empty list of triples, binding `a` to the first component of the head, `b` to the second component of the head, `c` to the third component of the head, and `d` to the tail of the list.

In general, pattern-matching is about taking a value and a pattern and (1) deciding if the pattern matches the value and (2) if so, binding variables to the right parts of the value. Here are some key parts to the elegant recursive definition of pattern matching:

- A variable pattern (`x`) matches any value `v` and introduces one binding (from `x` to `v`).
- 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). It introduces the bindings that `p` matching `v` introduces.
- The pattern `(p1,p2,...,pn)` matches a tuple value `(v1,v2,...,vn)` if `p1` matches `v1` and `p2` matches `v2`, ..., and `pn` matches `vn`. It introduces all the bindings that the recursive matches introduce.
- (A similar case for record patterns of the form `{f1=p1,...,fn=pn}` ...)

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

[4] It does not work for functions since it is impossible to tell if two functions always do the same thing. It also does not work for type `real` to enforce the rule that, due to rounding of floating-point values, comparing them is almost always wrong algorithmically.

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."

There are additional kinds of patterns as well. Sometimes we do not need to bind a variable to part of a value. For example, consider this function for computing a list's length:

```
fun len xs =
    case xs of
       [] => 0
     | x::xs' => 1 + len xs'
```

We do not use the variable `x`. In such cases, it is better style not to introduce a variable. Instead, the *wildcard pattern* `_` matches everything (just like a variable pattern matches everything), but does not introduce a binding. So we should write:

```
fun len xs =
    case xs of
       [] => 0
      | _::xs' => 1 + len xs'
```

In terms of our general definition, wildcard patterns are straightforward:

- A wildcard pattern (`_`) matches any value `v` and introduces no bindings.

Lastly, you can use integer constants in patterns. For example, the pattern `37` matches the value `37` and introduces no bindings.

Useful Examples of Nested Patterns

An elegant example of using nested patterns rather than an ugly mess of nested case-expressions is "zipping" or "unzipping" lists (three of them in this example):[5]

```
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
```

[5]Exceptions are discussed below but are not the important part of this example.

This example checks that a list of integers is sorted:

```
fun nondecreasing intlist =
    case intlist of
        [] => true
      | _::[] => true
      | head::(neck::rest) => (head <= neck andalso nondecreasing (neck::rest))
```

It is also sometimes elegant to compare two values by matching against a pair of them. This example, for determining the sign that a multiplication would have without performing the multiplication, is a bit silly but demonstrates the idea:

```
datatype sgn = 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 (* many say bad style; I am okay with it *)
  end
```

The style of this last case deserves discussion: When you include a "catch-all" case at the bottom like this, you are giving up any checking that you did not forget any cases: after all, it matches anything the earlier cases did not, so the type-checker will certainly not think you forgot any cases. So you need to be extra careful if using this sort of technique and it is probably less error-prone to enumerate the remaining cases (in this case `(N,P)` and `(P,N)`). That the type-checker will then still determine that no cases are missing is useful and non-trivial since it has to reason about the use `(Z,_)` and `(_,Z)` to figure out that there are no missing possibilities of type `sgn * sgn`.

Optional: Multiple Cases in a Function Binding

So far, we have seen pattern-matching on one-of types in case expressions. We also have seen the good style of pattern-matching each-of types in val or function bindings and that this is what a "multi-argument function" really is. But is there a way to match against one-of types in val/function bindings? This seems like a bad idea since we need multiple possibilities. But it turns out ML has special syntax for doing this in function definitions. Here are two examples, one for our own datatype and one for lists:

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

fun eval (Constant i) = i
  | eval (Negate e2) = ~ (eval e2)
  | eval (Add(e1,e2)) = (eval e1) + (eval e2)
  | eval (Multiply(e1,e2)) = (eval e1) * (eval e2)

fun append ([],ys) = ys
  | append (x::xs',ys) = x :: append(xs',ys)
```

As a matter of *taste*, your instructor has never liked this style very much, and you have to get parentheses in the right places. But it is common among ML programmers, so you are welcome to as well. As a matter of *semantics*, it is just syntactic sugar for a single function body that is a case expression:

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

fun append e =
    case e of
       ([],ys) => ys
     | (x::xs',ys) => x :: append(xs',ys)
```

In general, the syntax

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

is just syntactic sugar for:[6]

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

Notice the `append` example uses nested patterns: each branch matches a pair of lists, by putting patterns (e.g., `[]` or `x::xs'`) inside other patterns.

Exceptions

ML has a built-in notion of exception. You can *raise* (also known as *throw*) an exception with the `raise` primitive. For example, the `hd` function in the standard library raises the `List.Empty` exception when called with `[]`:

```
fun hd xs =
    case xs of
        []   => raise List.Empty
      | x::_ => x
```

[6] As a technicality, `x` must be some variable not already defined in the outer environment and used by one of the expressions in the function.

You can create your own kinds of exceptions with an exception binding. Exceptions can optionally carry values with them, which let the code raising the exception provide more information:

```
exception MyUndesirableCondition
exception MyOtherException of int * int
```

Kinds of exceptions are a *lot* like constructors of a datatype binding. Indeed, they are functions (if they carry values) or values (if they don't) that create values of type `exn` rather than the type of a datatype. So `Empty`, `MyUndesirableCondition`, and `MyOtherException(3,9)` are all values of type `exn`, whereas `MyOtherException` has type `int*int->exn`.

Usually we just use exception constructors as arguments to `raise`, such as `raise MyOtherException(3,9)`, but we can use them more generally to create values of type `exn`. For example, here is a version of a function that returns the maximum element in a list of integers. Rather than return an option or raise a particular exception like `List.Empty` if called with `[]`, it takes an argument of type `exn` and raises it. So the caller can pass in the exception of its choice. (The type-checker can infer that `ex` must have type `exn` because that is the type `raise` expects for its argument.)

```
fun maxlist (xs,ex) =
    case xs of
        [] => raise ex
      | x::[] => x
      | x::xs' => Int.max(x,maxlist(xs',ex))
```

Notice that calling `maxlist([3,4,0],List.Empty)` would not raise an exception; this call passes an exception *value* to the function, which the function then does not *raise*.

The other feature related to exceptions is *handling* (also known as *catching*) them. For this, ML has handle-expressions, which look like `e1 handle p => e2` where `e1` and `e2` are expressions and `p` is a pattern that matches an exception. The semantics is to evaluate `e1` and have the result be the answer. But if an exception matching `p` is raised by `e1`, then `e2` is evaluated and that is the answer for the whole expression. If `e1` raises an exception that does not match `p`, then the entire handle-expression also raises that exception. Similarly, if `e2` raises an exception, then the whole expression also raises an exception.

As with case-expressions, handle-expression can also have multiple branches each with a pattern and expression, syntactically separated by `|`.

Tail Recursion and Accumulators

This topic involves new programming idioms, but no new language constructs. It defines *tail recursion*, describes how it relates to writing *efficient* recursive functions in functional languages like ML, and presents how to use *accumulators* as a technique to make some functions tail recursive.

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

```
fun sum1 xs =
    case xs of
        [] => 0
      | i::xs' => i + sum1 xs'
```

```
fun sum2 xs =
    let fun f (xs,acc) =
            case xs of
                [] => acc
              | i::xs' => f(xs',i+acc)
    in
        f(xs,0)
    end
```

Both functions compute the same results, but `sum2` 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 `sum1`. 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 `sum2` be preferred when it is clearly more complicated? To answer, we need to understand a little bit about how function calls are implemented. Conceptually, 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. Each element stores things like the value of local variables and what part of the function has not been evaluated yet. 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 `sum1`, there will be one call-stack element (sometimes just called a "stack frame") for each recursive call to `sum1`, 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, `sum2` is no better: `sum2` 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 promise an essential 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. Therefore, calls to `sum2` never use more than 1 stack frame.

Why do implementations of functional languages include this optimization? By doing so, 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, something you the programmer can reason about since you can look at the code and identify which calls are tail calls.

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 any language: while-loops are not very useful for processing trees.

More Examples of Tail Recursion

Tail recursion is common for functions that process lists, but the concept is more general. For example, here are two implementations of the factorial function where the second one uses a tail-recursive helper function so that it needs only a small constant amount of call-stack space:

```
fun fact1 n = if n=0 then 1 else n * fact1(n-1)
```

```
fun fact2 n =
    let fun aux(n,acc) = if n=0 then acc else aux(n-1,acc*n)
    in
        aux(n,1)
    end
```

It is worth noticing that `fact1 4` and `fact2 4` produce the same answer even though the former performs $4*(3*(2*(1*1)))$ and the latter performs $(((1*4)*3)*2)*1$. We are relying on the fact that multiplication is associative ($a*(b*c) = (a*b)*c$) and that multiplying by 1 is the identity function ($1*x = x*1 = x$). The earlier `sum` example made analogous assumptions about addition. In general, converting a non-tail-recursive function to a tail-recursive function usually needs associativity, but many functions are associative.

A more interesting example is this inefficient function for reversing a list:

```
fun rev1 lst =
   case lst of
       [] => []
     | x::xs => (rev1 xs) @ [x]
```

We can recognize immediately that it is not tail-recursive since after the recursive call it remains to append the result onto the one-element list that holds the head of the list. Although this is the most natural way to reverse a list recursively, the inefficiency is caused by more than creating a call-stack of depth equal to the argument's length, which we will call n. The worse problem is that the total amount of work performed is proportional to n^2, i.e., this is a quadratic algorithm. The reason is that appending two lists takes time proportional to the length of the first list: it has to traverse the first list — see our own implementations of append discussed previously. Over all the recursive calls to `rev1`, we call `@` with first arguments of length $n-1, n-2, ..., 1$ and the sum of the integers from 1 to $n-1$ is $n*(n-1)/2$.

As you learn in a data structures and algorithms course, quadratic algorithms like this are much slower than linear algorithms for large enough n. That said, if you expect n to always be small, it may be be worth valuing the programmer's time and sticking with a simple recursive algorithm. Else, fortunately, using the accumulator idiom leads to an almost-as-simple linear algorithm.

```
fun rev2 lst =
    let fun aux(lst,acc) =
            case lst of
                [] => acc
              | x::xs => aux(xs, x::acc)
    in
        aux(lst,[])
    end
```

The key differences are (1) tail recursion and (2) we do only a constant amount of work for each recursive call because `::` does not have to traverse either of its arguments.

A Precise Definition of Tail Position

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 an expression is not in tail position, then none of its subexpressions are in tail position.
- If `if e1 then e2 else e3` is in tail position, then `e2` and `e3` are in tail position (but not `e1`). (Case-expressions are similar.)
- If `let b1 ... bn in e end` is in tail position, then `e` is in tail position (but no expressions in the bindings are).
- Function-call arguments are not in tail position.
- ...

Programming Languages (Coursera / University of Washington) Assignment 2

You will write 11 SML functions (not counting local helper functions), 4 related to "name substitutions" and 7 related to a made-up solitaire card game.

Your solutions must use pattern-matching. You may not use the functions `null`, `hd`, `tl`, `isSome`, or `valOf`, nor may you use anything containing a `#` character or features not used in class (such as mutation). Note that list order does not matter unless specifically stated in the problem.

Download hw2provided.sml from the course website. The provided code defines several types for you. You will not need to add any additional `datatype` bindings or type synonyms.

The sample solution, not including challenge problems, is *roughly* 130 lines, including the provided code.

Do not miss the "Important Caveat" and "Assessment" after the "Type Summary."

1. This problem involves using first-name substitutions to come up with alternate names. For example, *Fredrick William Smith* could also be *Fred William Smith* or *Freddie William Smith*. Only part (d) is specifically about this, but the other problems are helpful.

(a) Write a function `all_except_option`, which takes a `string` and a `string list`. Return `NONE` if the string is not in the list, else return `SOME` *lst* where *lst* is identical to the argument list except the string is not in it. You may assume the string is in the list at most once. Use `same_string`, provided to you, to compare strings. Sample solution is around 8 lines.

(b) Write a function `get_substitutions1`, which takes a `string list list` (a list of list of strings, the *substitutions*) and a `string` *s* and returns a `string list`. The result has all the strings that are in some list in *substitutions* that also has *s*, but *s* itself should not be in the result. Example:

```
get_substitutions1([["Fred","Fredrick"],["Elizabeth","Betty"],["Freddie","Fred","F"]],
                   "Fred")
(* answer: ["Fredrick","Freddie","F"] *)
```

Assume each list in *substitutions* has no repeats. The result will have repeats if *s* and another string are both in more than one list in *substitutions*. Example:

```
get_substitutions1([["Fred","Fredrick"],["Jeff","Jeffrey"],["Geoff","Jeff","Jeffrey"]],
                   "Jeff")
(* answer: ["Jeffrey","Geoff","Jeffrey"] *)
```

Use part (a) and ML's list-append (`@`) but no other helper functions. Sample solution is around 6 lines.

(c) Write a function `get_substitutions2`, which is like `get_substitutions1` except it uses a tail-recursive local helper function.

(d) Write a function `similar_names`, which takes a `string list list` of substitutions (as in parts (b) and (c)) and a *full name* of type `{first:string,middle:string,last:string}` and returns a list of full names (type `{first:string,middle:string,last:string} list`). The result is all the full names you can produce by substituting for the first name (and *only the first name*) using *substitutions* and parts (b) or (c). The answer should begin with the original name (then have 0 or more other names). Example:

```
similar_names([["Fred","Fredrick"],["Elizabeth","Betty"],["Freddie","Fred","F"]],
              {first="Fred", middle="W", last="Smith"})
(* answer: [{first="Fred", last="Smith", middle="W"},
            {first="Fredrick", last="Smith", middle="W"},
            {first="Freddie", last="Smith", middle="W"},
            {first="F", last="Smith", middle="W"}] *)
```

Do not eliminate duplicates from the answer. Hint: Use a local helper function. Sample solution is around 10 lines.

2. This problem involves a solitaire card game invented just for this question. You will write a program that tracks the progress of a game; writing a game player is a challenge problem. You can do parts (a)–(e) before understanding the game if you wish.

A game is played with a *card-list* and a *goal*. The player has a list of *held-cards*, initially empty. The player makes a move by either *drawing*, which means removing the first card in the card-list from the card-list and adding it to the held-cards, or *discarding*, which means choosing one of the held-cards to remove. The game ends either when the player chooses to make no more moves or when the sum of the values of the held-cards is greater than the goal.

The objective is to end the game with a low score (0 is best). Scoring works as follows: Let *sum* be the sum of the values of the held-cards. If *sum* is greater than *goal*, the *preliminary score* is three times ($sum - goal$), else the preliminary score is ($goal - sum$). The score is the preliminary score unless all the held-cards are the same color, in which case the score is the preliminary score divided by 2 (and rounded down as usual with integer division; use ML's `div` operator).

(a) Write a function `card_color`, which takes a card and returns its color (spades and clubs are black, diamonds and hearts are red). Note: One case-expression is enough.

(b) Write a function `card_value`, which takes a card and returns its value (numbered cards have their number as the value, aces are 11, everything else is 10). Note: One case-expression is enough.

(c) Write a function `remove_card`, which takes a list of cards `cs`, a card `c`, and an exception `e`. It returns a list that has all the elements of `cs` except `c`. If `c` is in the list more than once, remove only the first one. If `c` is not in the list, raise the exception `e`. You can compare cards with `=`.

(d) Write a function `all_same_color`, which takes a list of cards and returns true if all the cards in the list are the same color. Hint: An elegant solution is very similar to one of the functions using nested pattern-matching in the lectures.

(e) Write a function `sum_cards`, which takes a list of cards and returns the sum of their values. *Use a locally defined helper function that is tail recursive. (Take "calls use a constant amount of stack space" as a requirement for this problem.)*

(f) Write a function `score`, which takes a `card list` (the held-cards) and an `int` (the goal) and computes the score as described above.

(g) Write a function `officiate`, which "runs a game." It takes a `card list` (the card-list) a `move list` (what the player "does" at each point), and an `int` (the goal) and returns the score at the end of the game after processing (some or all of) the moves in the move list in order. Use a locally defined recursive helper function that takes several arguments that together represent the current state of the game. As described above:

- The game starts with the held-cards being the empty list.
- The game ends if there are no more moves. (The player chose to stop since the `move list` is empty.)
- If the player discards some card `c`, play continues (i.e., make a recursive call) with the held-cards not having `c` and the card-list unchanged. If `c` is not in the held-cards, raise the `IllegalMove` exception.
- If the player draws and the card-list is (already) empty, the game is over. Else if drawing causes the sum of the held-cards to exceed the goal, the game is over (after drawing). Else play continues with a larger held-cards and a smaller card-list.

Sample solution for (g) is under 20 lines.

3. **Challenge Problems:**

(a) Write `score_challenge` and `officiate_challenge` to be like their non-challenge counterparts except each ace can have a value of 1 or 11 and `score_challenge` should always return the least (i.e., best) possible score. (Note the game-ends-if-sum-exceeds-goal rule should apply only if there is no sum that is less than or equal to the goal.) Hint: This is easier than you might think.

(b) Write `careful_player`, which takes a card-list and a goal and returns a move-list such that calling `officiate` with the card-list, the goal, and the move-list has this behavior:

- The value of the held cards never exceeds the goal.
- A card is drawn whenever the goal is more than 10 greater than the value of the held cards. As a detail, you should (attempt to) draw, even if no cards remain in the card-list.
- If a score of 0 is reached, there must be no more moves.
- If it is possible to reach a score of 0 by discarding a card followed by drawing a card, then this must be done. Note `careful_player` will have to look ahead to the next card, which in many card games is considered "cheating." Also note that the previous requirement takes precedence: There must be no more moves after a score of 0 is reached even if there is another way to get back to 0.

Notes:

- There may be more than one result that meets the requirements above. The autograder should work for any correct strategy — it checks that the result meets the requirements.
- This problem is *not* a continuation of problem 3(a). In this problem, all aces have a value of 11.

Type Summary

Evaluating a correct homework solution should generate these bindings, in addition to the bindings from the code provided to you — *but see the important caveat that follows.*

```
val all_except_option = fn : string * string list -> string list option
val get_substitutions1 = fn : string list list * string -> string list
val get_substitutions2 = fn : string list list * string -> string list
val similar_names = fn : string list list * {first:string, last:string, middle:string}
    -> {first:string, last:string, middle:string} list
val card_color = fn : card -> color
val card_value = fn : card -> int
val remove_card = fn : card list * card * exn -> card list
val all_same_color = fn : card list -> bool
val sum_cards = fn : card list -> int
val score = fn : card list * int -> int
val officiate = fn : card list * move list * int -> int
```

Important Caveat

The REPL may give your functions *equivalent types* or *more general* types. This is fine. In the sample solution, the bindings for problems 1d, 2a, 2b, 2c, and 2d were all more general. For example, `card_color` had type `suit * 'a -> color` and `remove_card` had type `''a list * ''a * exn -> ''a list`. They are more general, which means there is a way to replace the *type variables* (`'a` or `''a`) with types to get the bindings listed above. As for equivalent types, because `type card = suit*rank`, types like `card -> int` and `suit*rank->int` are equivalent. They are the same type, and the REPL simply chooses one way of printing the type. Also, the order of fields in records never matters.

If you write down explicit argument types for functions, you will probably not see equivalent or more-general types, but we encourage the common ML approach of omitting all explicit types.

Of course, generating these bindings does not guarantee that your solutions are correct. *Test your functions: Put your testing code in a second file. We will not grade it, but you must turn it in.*

Assessment

We will automatically test your functions on a variety of inputs, including edge cases. We will also ask peers to evaluate your code for simplicity, conciseness, elegance, and good formatting including indentation and line breaks. Your solution will also be checked for using only features discussed so far in class, and to ensure that it does not use any of the banned features listed at the beginning of the assignment, such as `hd` and `#foo`.

Turn-in Instructions

First, follow the instructions on the course website to submit your solution file (not your testing file) for auto-grading. Do not proceed to the peer-assessment submission until you receive a high-enough grade from the auto-grader: Doing peer assessment requires instructions that include a sample solution, so these instructions will be "locked" until you receive high-enough auto-grader score. Then submit your same solution file again for peer assessment and follow the peer-assessment instructions.

Coursera Programming Languages Course
Section 3 Summary

Standard Description: This summary covers **roughly** *the same material as the lecture videos and the materials (slides, code) posted with each video. It can help to read about the material in a narrative style and to have the material for an entire section of the course in a single document, especially when reviewing the material later. Please report errors in these notes on the discussion board.*

Contents

Introduction and Some Terms

This section focuses on *first-class functions* and *function closures*. By "first-class" we mean that functions can be computed, passed, stored, etc. *wherever* other values can be computed, passed, stored, etc. 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" refers to functions that use variables defined outside of them, which makes first-class functions much more powerful, as we will see after starting with simpler first-class functions that do not use this ability. The term *higher-order function* just refers to a function that takes or returns other functions.

Terms like first-class functions, function closures, and higher-order functions are often confused with each other or considered synonyms. Because so much of the world is not careful with these terms, we will not be too worried about them either. But the idea of first-class functions and the idea of function closures really are distinct *concepts* that we often use together to write elegant, reusable code. For that reason, we will delay the idea of closures, so we can introduce it as a separate concept.

There is an even more general term, *functional programming*. This term also is often used imprecisely to refer to several distinct concepts. The two most important and most common are:

- Not using mutable data in most or all cases: We have avoided mutation throughout the course so far and will mostly continue to do so.
- Using functions as values, which is what this section is all about

There are other things that are also considered related to functional programming:

- A programming style that encourages recursion and recursive data structures
- Programming with a syntax or style that is closer to traditional mathematical definitions of functions
- Anything that is not object-oriented programming (this one is really incorrect)
- Using certain programming idioms related to *laziness*, a technical term for a certain kind of programming construct/idiom we will study, briefly, later in the course

An obvious related question is "what makes a programming language a functional language?" Your instructor has come to the conclusion this is not a question for which there is a precise answer and barely makes sense as a question. But one could say that a functional language is one where writing in a functional style (as described above) is more convenient, more natural, and more common than programming in other styles. At a minimum, you need good support for immutable data, first-class functions, and function closures. More and more we are seeing new languages that provide such support but also provide good support for other styles, like object-oriented programming, which we will study some toward the end of the course.

Taking Functions as Arguments

The most common use of first-class functions is passing them as arguments to other functions, so we motivate this use first.

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 genuinely 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.

Once we define such abstractions, we can find additional uses for them. For example, even if our program today does not need to triple any values n times, maybe tomorrow it will, in which case we can just define the function `triple_n_times` using `n_times`:

```
fun triple x = 3*x

fun triple_n_times (n,x) = n_times(triple,n,x)
```

Polymorphic Types and Functions as Arguments

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 arguments, the first of which is itself a function that takes and returns 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 because only some of our example uses would 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 *parametric polymorphism*, or perhaps more commonly *generic types*. It lets functions take arguments of any type. It is a separate issue from first-class functions:

- There are functions that take functions and do not have polymorphic types
- 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.

Without parametric 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` even though it does not use any function arguments. Conversely, here is a higher-order function that is not polymorphic: it has type `(int->int) * int -> int`:[1]

```
fun times_until_zero (f,x) =
    if x = 0 then 0 else 1 + times_until_zero(f, f x)
```

[1] It would be better to make this function tail-recursive using an accumulator.

Anonymous functions

There is no reason that a function like `triple` that is passed to another function like `n_times` needs to be defined at top-level. As usual, it is better style to define such functions locally if they are needed only locally. So we could write:

```
fun triple_n_times (n,x) =
  let fun triple x = 3*x in n_times(triple,n,x) end
```

In fact, we could give the `triple` function an even smaller scope: we need it only as the first argument to `n_times`, so we could have a let-expression there that evaluates to the triple function:

```
fun triple_n_times (n,x) = n_times((let fun triple y = 3*y in triple end), n, x)
```

Notice that in this example, which is actually poor style, we need to have the let-expression "return" `triple` since, as always, a let-expression produces the result of the expression between `in` and `end`. In this case, we simply look up `triple` in the environment, and the resulting function is the value that we then pass as the first argument to `n_times`.

ML has a much more concise way to define functions right where you use them, as in this final, best version:

```
fun triple_n_times (n,x) = n_times((fn y => 3*y), n, x)
```

This code defines an *anonymous function* `fn y => 3*y`. It is a function that takes an argument `y` and has body `3*y`. The `fn` is a keyword and `=>` (not `=`) is also part of the syntax. We never gave the function a name (it is *anonymous*, see?), which is convenient because we did not need one. We just wanted to pass a function to `n_times`, and in the body of `n_times`, this function is bound to `f`.

It is common to use anonymous functions as arguments to other functions. Moreover, you can put an anonymous function anywhere you can put an expression — it simply is a value, the function itself. 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, and `fun` bindings must be in let-expressions or at top-level.

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. So function-bindings are *almost* syntactic sugar, but they support recursion, which is essential.

Unnecessary Function Wrapping

While anonymous functions are incredibly convenient, there is one poor idiom where they get used for no good reason. Consider:

```
fun nth_tail_poor (n,x) = n_times((fn y => tl y), n, x)
```

What is `fn y => tl y`? It is a function that returns the list-tail of its argument. But there is already a variable bound to a function that does the exact same thing: `tl`! In general, there is no reason to write `fn x => f x` when we can just use `f`. This is analogous to the beginner's habit of writing `if x then true else false` instead of `x`. Just do this:

```
fun nth_tail (n,x) = n_times(tl, n, x)
```

Maps and filters

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

```
fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))
```

The `map` function 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 x1 = map (increment, [4,8,12,16]) (* answer: [5,9,13,17] *)
val x2 = 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 `x1`, both `'a` and `'b` are *instantiated* with `int`. For `x2`, `'a` is `int list` and `'b` is `int`.

The ML standard library provides a very similar function `List.map`, but it is defined in a curried form, a topic we will discuss later in this section.

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 is 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,xs) =
    case xs of
        [] => []
      | x::xs' => if f x
                  then x::(filter (f,xs'))
                  else filter (f,xs')
```

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

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

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

Returning functions

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))`.

Not just for numbers and lists

Because ML programs tend to use lists a lot, you might forget that higher-order functions are useful for more than lists. Some of our first examples just used integers. But higher-order functions also are great for our own data structures. Here we use an `is_even` function to see if all the constants in an arithmetic expression are even. We could easily reuse `true_of_all_constants` for any other property we wanted to check.

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

fun is_even v =
    (v mod 2 = 0)

fun true_of_all_constants(f,e) =
    case e of
        Constant i      => f i
      | Negate e1       => true_of_all_constants(f,e1)
      | Add(e1,e2)      => true_of_all_constants(f,e1) andalso true_of_all_constants(f,e2)
      | Multiply(e1,e2) => true_of_all_constants(f,e1) andalso true_of_all_constants(f,e2)

fun all_even e = true_of_all_constants(is_even,e)
```

Lexical Scope

So far, the functions we have passed to or returned from other functions have been *closed*: the function bodies used only the function's argument(s) and any locally defined variables. But we know that functions can do more than that: they can use any bindings that are in scope. Doing so in combination with higher-order functions is very powerful, so it is crucial to learn effective idioms using this technique. But first it is

even more crucial to get the semantics right. This is probably the most subtle and important concept in the entire course, so go slowly and read carefully.

*The body of a function is evaluated in the environment where the function is **defined**, not the environment where the function is **called**.* Here is a very simple example to demonstrate the difference:

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

In this example, `f` is bound to a function that takes an argument `y`. Its body also looks up `x` in the environment where `f` was defined. Hence this function *always* increments its argument since the environment at the definition maps `x` to 1. Later we have a different environment where `f` maps to this function, `x` maps to 2, `y` maps to 3, and we make the call `f x`. Here is how evaluation proceeds:

- Look up `f` to get the previously described function.
- Evaluate the argument `x+y` in the *current* environment by looking up `x` and `y`, producing 5.
- Call the function with the argument 5, which means evaluating the body `x+y` in the *"old"* environment where `x` maps to 1 extended with `y` mapping to 5. So the result is 6.

Notice the argument was evaluated in the current environment (producing 5), but the function body was evaluated in the "old" environment. We discuss below why this semantics is desirable, but first we define this semantics more precisely and understand the semantics with additional silly examples that use higher-order functions.

This semantics is called *lexical scope*. The alternate, inferior semantics where you use the current environment (which would produce 7 in the above example) is called *dynamic scope*.

Environments and Closures

We have said that functions are values, but we have not been precise about what that value exactly is. We now explain that a function value has *two parts*, the *code* for the function (obviously) and the *environment that was current when we created the function*. These two parts really do form a "pair" but we put "pair" in quotation marks because it is not an ML pair, just something with two parts. You *cannot* access the parts of the "pair" separately; all you can do is call the function. This call uses both parts because it evaluates the code part using the environment part.

This "pair" is called a *function closure* or just *closure*. The reason is that while the code itself can have *free variables* (variables that are not *bound* inside the code so they need to be bound by some outer environment), the closure carries with it an environment that provides all these bindings. So the closure overall is "closed" — it has everything it needs to produce a function result given a function argument.

In the example above, the binding `fun f y = x + y` bound `f` to a closure. The code part is the function `fn y => x + y` and the environment part maps `x` to 1. Therefore, any call to this closure will return `y+1`.

(Silly) Examples Including Higher-Order Functions

Lexical scope and closures get more interesting when we have higher-order functions, but the semantics already described will lead us to the right answers.

Example 1:

```
val x = 1
fun f y =
    let
        val x = y+1
    in
        fn z => x + y  + z
    end
val x = 3
val g = f 4
val y = 5
val z = g 6
```

Here, `f` is bound to a closure where the environment part maps `x` to 1. So when we later evaluate `f 4`, we evaluate `let val x = y + 1 in fn z => x + y + z end` in an environment where `x` maps to 1 extended to map `y` to 4. But then due to the let-binding we shadow `x` so we evaluate `fn z => x + y + z` in an environment where `x` maps to `5` and `y` maps to 4. How do we evaluate a function like `fn z => x + y + z`? We create a closure with the current environment. So `f 4` returns a closure that, when called, will always add 9 to its argument, no matter what the environment is at any call-site. Hence, in the last line of the example, `z` will be bound to 15.

Example 2:

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

In this example, `f` is bound to a closure that takes another function `g` as an argument and returns the result of `g 2`. The closure bound to `h` *always* adds 4 to its argument because the argument is `y`, the body is `x+y`, and the function is defined in an environment where `x` maps to 4. So in the last line, `z` will be bound to 6. The binding `val x = 3` is totally irrelevant: the call `g 2` is evaluated by looking up `g` to get the closure that was passed in and then using that closure with *its environment* (in which `x` maps to 4) with 2 for an argument.

Why Lexical Scope

While lexical scope and higher-order functions take some getting used to, decades of experience make clear that this semantics is what we want. Much of the rest of this section will describe various widespread idioms that are powerful and that rely on lexical scope.

But first we can also motivate lexical scope by showing how dynamic scope (where you just have one current environment and use it to evaluate function bodies) leads to some fundamental problems.

First, suppose in Example 1 above the body of `f` was changed to `let val q = y+1 in fn z => q + y + z`. Under lexical scope this is fine: we can always change the name of a local variable and its uses without it affecting anything. Under dynamic scope, now the call to `g 6` will make no sense: we will try to look up `q`, but there is no `q` in the environment at the call-site.

Second, consider again the original version of Example 1 but now change the line `val x = 3` to `val x = "hi"`. Under lexical scope, this is again fine: that binding is never actually used. Under dynamic scope, the call to `g 6` will look-up `x`, get a string, and try to add it, which should not happen in a program that type-checks.

Similar issues arise with Example 2: The body of `f` in this example is awful: we have a local binding we never use. Under lexical scope we can remove it, changing the body to `g 2` and know that this has no effect on the rest of the program. Under dynamic scope it would have an effect. Also, under lexical scope we *know* that any use of the closure bound to `h` will add 4 to its argument regardless of how other functions like `g` are implemented and what variable names they use. This is a key separation-of-concerns that only lexical scope provides.

For "regular" variables in programs, lexical scope is the way to go. There are some compelling uses for dynamic scoping for certain idioms, but few languages have special support for these (Racket does) and very few if any modern languages have dynamic scoping as the default. But you have seen one feature that is more like dynamic scope than lexical scope: exception handling. When an exception is raised, evaluation has to "look up" which handle expression should be evaluated. This "look up" is done using the dynamic call stack, with no regard for the lexical structure of the program.

Passing Closures to Iterators Like Filter

The examples above are silly, so we need to show useful programs that rely on lexical scope. The first idiom we will show is passing functions to iterators like *map* and *filter*. The functions we previously passed did not use their environment (only their arguments and maybe local variables), but being able to pass in closures makes the higher-order functions much more widely useful. Consider:

```
fun filter (f,xs) =
    case xs of
        [] => []
      | x::xs' => if f x then x::(filter(f,xs')) else filter(f,xs')

fun allGreaterThanSeven xs = filter (fn x => x > 7, xs)

fun allGreaterThan (xs,n) = filter (fn x => x > n, xs)
```

Here, `allGreaterThanSeven` is "old news" — we pass in a function that removes from the result any numbers 7 or less in a list. But it is much more likely that you want a function like `allGreaterThan` that takes the "limit" as a parameter `n` and uses the function `fn x => x > n`. Notice this requires a closure and lexical scope! When the implementation of `filter` calls this function, we need to look up `n` in the environment where `fn x => x > n` was defined.

Here are two additional examples:

```
fun allShorterThan1 (xs,s) = filter (fn x => String.size x < String.size s, xs)
```

```
fun allShorterThan2 (xs,s) =
    let
        val i = String.size s
    in
        filter(fn x => String.size x < i, xs)
    end
```

Both these functions take a list of strings `xs` and a string `s` and return a list containing only the strings in `xs` that are shorter than `s`. And they both use closures, to look up `s` or `i` when the anonymous functions get called. The second one is more complicated but a bit more efficient: The first one recomputes `String.size s` once per element in `xs` (because `filter` calls its function argument this many times and the body evaluates `String.size s` each time). The second one "precomputes" `String.size s` and binds it to a variable `i` available to the function `fn x => String.size x < i`.

Fold and More Closure Examples

Beyond map and filter, a third incredibly useful higher-order function is *fold*, which can have several slightly different definitions and is also known by names such as *reduce* and *inject*. Here is one common definition:

```
fun fold (f,acc,xs) =
  case xs of
    []     => acc
  | x::xs' => fold (f, f(acc,x), xs')
```

`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 `foo`, we can do:

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

As with `map` and `filter`, much of `fold`'s power comes from clients passing closures that can have "private fields" (in the form of variable bindings) for keeping data they want to consult. Here are two examples. The first counts how many elements are in some integer range. The second checks if all elements are strings shorter than some other string's length.

```
fun numberInRange (xs,lo,hi) =
  fold ((fn (x,y) =>
            x + (if y >= lo andalso y <= hi then 1 else 0)),
         0, xs)

fun areAllShorter (xs,s) =
    let
        val i = String.size s
    in
        fold((fn (x,y) => x andalso String.size y < i), true, xs)
    end
```

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.

Another Closure Idiom: Combining Functions

Function composition

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 inferred to be `('a -> 'b) * ('c -> 'a) -> 'c -> 'b`, which is equivalent to what you might write: `('b -> 'c) * ('a -> 'b) -> ('a -> 'c)` since the two types simply use different type-variable names consistently.

As a cute and convenient library function, the ML library defines the infix operator `o` as function composition, just like in math. So instead of writing:

```
fun sqrt_of_abs i = Math.sqrt(Real.fromInt (abs i))
```

you could write:

```
fun sqrt_of_abs i = (Math.sqrt o Real.fromInt o abs) i
```

But this second version makes clearer that we can just use function-composition to create a function that we bind to a variable with a val-binding, as in this third version:

```
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```

While all three versions are fairly readable, the first one does not immediately indicate to the reader that `sqrt_of_abs` is just the composition of other functions.

The Pipeline Operator

In functional programming, it is very common to compose other functions to create larger ones, so it makes sense to define convenient syntax for it. While the third version above is concise, it, like function composition in mathematics, has the strange-to-many-programmers property that the computation proceeds from right-to-left: "Take the absolute value, convert it to a real, and compute the square root" may be easier to understand than, "Take the square root of the conversion to real of the absolute value."

We can define convenient syntax for left-to-right as well. Let's first define our own infix operator that lets us put the function to the right of the argument we are calling it with:

```
infix |> (* tells the parser |> is a function that appears between its two arguments *)
fun x |> f = f x
```

Now we can write:

```
fun sqrt_of_abs i = i |> abs |> Real.fromInt |> Math.sqrt
```

This operator, commonly called the *pipeline operator*, is very popular in F# programming. (F# is a dialect of ML that runs on .Net and interacts well with libraries written in other .Net languages.) As we have seen, there is nothing complicated about the semantics of the pipeline operator.

Another Closure Idiom: Currying and Partial Application

The next idiom we consider is very convenient in general, and is often used when defining and using higher-order functions like map, filter, and fold. 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. Our previous approach passed 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. Lexical scope is essential to this technique working correctly.

This technique is called *currying* after a logician named Haskell Curry who studied related ideas (so if you do not know that, then the term currying does not make much sense).

Defining and Using a Curried Function

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

```
val sorted3 = fn x => fn y => fn z => z >= y andalso y >= x
```

If we call `sorted3 4` we will get a closure that has `x` in its environment. If we then call this closure with `5`, we will get a closure 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 `((sorted3 4) 5) 6` computes exactly what we want and feels pretty close to calling `sorted3` with 3 arguments. Even better, the parentheses are optional, so we can write exactly the same thing as `sorted3 4 5 6`, which is actually fewer characters than our old tuple approach where we would have:

```
fun sorted3_tupled (x,y,z) = z >= y andalso y >= x
val someClient = sorted3_tupled(4,5,6)
```

In general, the syntax `e1 e2 e3 e4` is implicitly the nested function calls `(((e1 e2) e3) e4)` and this choice was made because it makes using a curried function so pleasant.

Partial Application

Even though we might expect most clients of our curried `sorted3` 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 conceptual arguments. As a silly example, `sorted3 0 0` returns a function that returns `true` if its argument is nonnegative.

Partial Application and Higher-Order Functions

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 xs =>
  case xs of
    []     => acc
  | x::xs' => fold f (f(acc,x)) xs'
```

Now we could use this fold to define a function that sums a list elements like this:

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

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. For example, the types of all these functions use currying:

```
val List.map = fn : ('a -> 'b) -> 'a list -> 'b list
val List.filter = fn : ('a -> bool) -> 'a list -> 'a list
val List.foldl = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

As an example, `List.foldl((fn (x,y) => x+y), 0, [3,4,5])` does not type-check because `List.foldl` expects a `'a * 'b -> 'b` function, not a triple. The correct call is `List.foldl (fn (x,y) => x+y) 0 [3,4,5]`, which calls `List.foldl` with a function, which returns a closure and so on.

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 xs =
  case xs of
    []     => acc
  | x::xs' => fold f (f(acc,x)) xs'
```

Another useful curried function is `List.exists`, which we use in the callback example below. These library functions are easy to implement ourselves, so we should understand they are not fancy:

```
fun exists predicate xs =
    case xs of
      [] => false
    | x::xs' => predicate x orelse exists predicate xs'
```

Currying in General

While currying and partial application are great for higher-order functions, they are great in general too. They work for any multi-argument function and partial application can also be surprisingly convenient. In this example, both `zip` and `range` are defined with currying and `countup` partially applies `range`. The `add_numbers` function turns the list `[v1,v2,...,vn]` into `[(1,v1),(2,v2),...,(n,vn)]`.

```
fun zip xs ys =
    case (xs,ys) of
        ([],[]) => []
      | (x::xs',y::ys') => (x,y) :: (zip xs' ys')
      | _ => raise Empty
```

```
fun range i j = if i > j then [] else i :: range (i+1) j

val countup  = range 1

fun add_numbers xs = zip (countup (length xs)) xs
```

Combining Functions to Curry and Uncurry Other Functions

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.

Efficiency

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 performance-critical functions in your software, it *might* matter to pick the faster way. In the version of the ML compiler we are using, tupling happens to be faster. In widely used implementations of OCaml, Haskell, and F#, curried functions are faster so they are the standard way to define multi-argument functions in those languages.

The Value Restriction

Once you have learned currying and partial application, you might try to use it to create a polymorphic function. Unfortunately, certain uses, such as these, do not work in ML:

```
val mapSome = List.map SOME (*turn [v1,v2,...,vn] into [SOME v1, SOME v2, ..., SOME vn]*)
val pairIt = List.map (fn x => (x,x)) (*turn [v1,v2,...,vn] into [(v1,v1),(v2,v2),...,(vn,vn)]*)
```

Given what we have learned so far, there is no reason why this should not work, especially since all these functions do work:

```
fun mapSome xs = List.map SOME xs
val mapSome = fn xs => List.map SOME xs
val pairIt : int list -> (int * int) list = List.map (fn x => (x,x))
val incrementIt = List.map (fn x => x+1)
```

The reason is called the *value restriction* and it is sometimes annoying. It is in the language for good reason: without it, the type-checker might allow some code to break the type system. This can happen only with code that is using mutation and the code above is not, but the type-checker does not know that.

The simplest approach is to ignore this issue until you get a warning/error about the value restriction. When you do, turn the val-binding back into a fun-binding like in the first example above of what works.

When we study type inference in the next section, we will discuss the value restriction in a little more detail.

Mutation via ML References

We now finally introduce ML's support for mutation. Mutation is okay in some settings. A key approach in functional programming is to use it only when "updating the state of something so all users of that state can see a change has occurred" is the natural way to model your computation. Moreover, we want to keep features for mutation separate so that we know when mutation is not being used.

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`.

One good way to think about a reference is as a record with one field where that field can be updated with the `:=` operator.

Here is a short example:

```
val x = ref 0
val x2 = x (* x and x2 both refer to the same reference *)
val x3 = ref 0
(* val y = x + 1*) (* wrong: x is not an int *)
val y = (!x) + 1 (* y is 1 *)
val _ = x := (!x) + 7 (* the contents of the reference x refers to is now 7 *)
val z1 = !x  (* z1 is  7 *)
val z2 = !x2 (* z2 is also 7 -- with mutation, aliasing matters*)
val z3 = !x3 (* z3 is 0 *)
```

Another Closure Idiom: Callbacks

The next common idiom we 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. Computer players in a game where the events are "it is your turn" is yet another.

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` does not 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 you do not need to know about for this course, but we will show an example later), which are closer to the convenience of closures.

In ML, we will use mutation to show the callback idiom. This is reasonable because 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.

Our example uses 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 function of type `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 `onEvent` is called and it calls each callback in the list:

```
val cbs : (int -> unit) list ref = ref []
fun onKeyEvent f = cbs := f::(!cbs) (* The only "public" binding *)
fun onEvent i =
   let fun loop fs =
        case fs of
          [] => ()
        | f::fs' => (f i; loop fs')
    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 different clients (calls to `onKeyEvent`) that use different bindings of different types in their environment. (The `val _ = e` idiom is common for executing an expression just for its side-effect, in this case registering a callback.)

```
val timesPressed = ref 0
val _ = onKeyEvent (fn _ => timesPressed := (!timesPressed) + 1)

fun printIfPressed i =
    onKeyEvent (fn j => if i=j
                        then print ("you pressed " ^ Int.toString i ^ "\n")
                        else ())

val _ = printIfPressed 4
val _ = printIfPressed 11
val _ = printIfPressed 23
```

Optional: Another Closure Idiom: Abstract Data Types

This last closure idiom we will consider is the fanciest and most subtle. It is not the sort of thing programmers typically do — there is usually a simpler way to do it in a modern programming language. It is included as an advanced example to demonstrate that a record of closures that have the same environment is a lot like an object in object-oriented programming: the functions are methods and the bindings in the environment are private fields and methods. There are no new language features here, just lexical scope. It suggests (correctly) that functional programming and object-oriented programming are more similar than they might first appear (a topic we will revisit later in the course; there are also important differences).

The key to an abstract data type (ADT) is requiring clients to use it via a collection of functions rather than directly accessing its private implementation. 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 ADT by 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.

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. (We could just as easily define a mutable version using ML's references.) In ML, we could define a type that describes our interface:

```
datatype set = S of { insert : int -> set, member : int -> bool, size : unit -> int }
```

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

```
type set = { insert : int -> set, member : int -> bool, size : unit -> int }
```

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 even though sets are each-of types, not one-of types. Notice we are not using any new types or features; we simply have a type describing a record with fields named `insert`, `member`, and `size`, each of which holds a function.

Once we have an empty set, we can use its `insert` field to create a one-element set, and then use that set's `insert` 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 (many of the parentheses are optional but may help understand the code):

```
fun use_sets () =
    let val S s1 = empty_set
        val S s2 = (#insert s1) 34
        val S s3 = (#insert s2) 34
        val S s4 = #insert s3 19
    in
        if (#member s4) 42
        then 99
        else if (#member s4) 19
        then 17 + (#size s3) ()
        else 0
    end
```

Again we are using no new features. `#insert 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.insert(34)` to do something similar. The `val` bindings use pattern-matching to "strip off" 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 make_set xs = (* xs is a "private field" in result *)
```

```
            let (* contains a "private method" in result *)
                fun contains i = List.exists (fn j => i=j) xs
            in
                S { insert = fn i => if contains i
                                     then make_set xs
                                     else make_set (i::xs),
                    member = contains,
                    size   = fn () => length xs
                  }
            end
    in
        make_set []
    end
```

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 three closures. The closures can use `xs`, the helper function `contains`, and `make_set`. Like all function bodies, they are not executed until they are called.

Optional: Closures in Other Languages

To conclude our study of function closures, we digress from ML to show similar programming patterns in Java (using generics and interfaces) and C (using function pointers taking explicit environment arguments). We will not test you on this material, and you are welcome to skip it. However, it may help you understand closures by seeing similar ideas in other settings, and it should help you see how central ideas in one language can influence how you might approach problems in other languages. That is, it could make you a better programmer in Java or C.

For both Java and C, we will "port" this ML code, which defines our own polymorphic linked-list type constructor and three polymorphic functions (two higher-order) over that type. We will investigate a couple ways we could write similar code in Java or C, which will can help us better understand similarities between closures and objects (for Java) and how environments can be made explicit (for C). In ML, there is no reason to define our own type constructor since `'a list` is already written, but doing so will help us compare to the Java and C versions.

```
datatype 'a mylist = Cons of 'a * ('a mylist) | Empty

fun map f xs =
    case xs of
        Empty => Empty
      | Cons(x,xs) => Cons(f x, map f xs)

fun filter f xs =
    case xs of
        Empty => Empty
      | Cons(x,xs) => if f x then Cons(x,filter f xs) else filter f xs

fun length xs =
    case xs of
        Empty => 0
```

```
      | Cons(_,xs) => 1 + length xs
```

Using this library, here are two client functions. (The latter is not particularly efficient, but shows a simple use of `length` and `filter`.)

```
val doubleAll = map (fn x => x * 2)
fun countNs (xs, n : int) = length (filter (fn x => x=n) xs)
```

Optional: Closures in Java using Objects and Interfaces

Java 8 includes support for closures much like most other mainstream object-oriented languages now do (C#, Scala, Ruby, ...), but it is worth considering how we might write similar code in Java without this support, as has been necessary for almost two decades. While we do not have first-class functions, currying, or type inference, we do have generics (Java did not used to) and we can define interfaces with one method, which we can use like function types. Without further ado, here is a Java analogue of the code, followed by a brief discussion of features you may not have seen before and other ways we could have written the code:

```
interface Func<B,A> {
    B m(A x);
}
interface Pred<A> {
    boolean m(A x);
}
class List<T> {
    T       head;
    List<T> tail;
    List(T x, List<T> xs) {
        head = x;
        tail = xs;
    }
    static <A,B> List<B> map(Func<B,A> f, List<A> xs) {
        if(xs==null)
            return null;
        return new List<B>(f.m(xs.head), map(f,xs.tail));
    }
    static <A> List<A> filter(Pred<A> f, List<A> xs) {
        if(xs==null)
            return null;
        if(f.m(xs.head))
            return new List<A>(xs.head, filter(f,xs.tail));
        return filter(f,xs.tail);
    }
    static <A> int length(List<A> xs) {
        int ans = 0;
        while(xs != null) {
            ++ans;
            xs = xs.tail;
        }
        return ans;
```

```
    }
}

class ExampleClients {
    static List<Integer> doubleAll(List<Integer> xs) {
        return List.map((new Func<Integer,Integer>() {
                             public Integer m(Integer x) { return x * 2; }
                         }),
                        xs);
    }
    static int countNs(List<Integer> xs, final int n) {
        return List.length(List.filter((new Pred<Integer>() {
                                            public boolean m(Integer x) { return x==n; }
                                        }),
                                        xs));
    }
}
```

This code uses several interesting techniques and features:

- In place of the (inferred) function types `'a -> 'b` for `map` and `'a -> bool` for `filter`, we have generic interfaces with one method. A class implementing one of these interfaces can have fields of any types it needs, which will serve the role of a closure's environment.
- The generic class `List` serves the role of the datatype binding. The constructor initializes the `head` and `tail` fields as expected, using the standard Java convention of `null` for the empty list.
- Static methods in Java can be generic provided the type variables are explicitly mentioned to the left of the return type. Other than that and syntax, the `map` and `filter` implementations are similar to their ML counterparts, using the one method in the `Func` or `Pred` interface as the function passed as an argument. For `length`, we could use recursion, but choose instead to follow Java's preference for loops.
- If you have never seen anonymous inner classes, then the methods `doubleAll` and `countNs` will look quite odd. Somewhat like anonymous functions, this language feature lets us crate an object that implements an interface without giving a name to that object's class. Instead, we use `new` with the interface being implemented (instantiating the type variables appropriately) and then provide definitions for the methods. As an inner class, this definition can use fields of the enclosing object or *final* local variables and parameters of the enclosing method, gaining much of the convenience of a closure's environment with more cumbersome syntax. (Anonymous inner classes were added to Java to support callbacks and similar idioms.)

There are many different ways we could have written the Java code. Of particular interest:

- Tail recursion is not as efficient as loops in implementations of Java, so it is reasonable to prefer loop-based implementations of `map` and `filter`. Doing so without reversing an intermediate list is more intricate than you might think (you need to keep a pointer to the previous element, with special code for the first element), which is why this sort of program is often asked at programming interviews. The recursive version is easy to understand, but would be unwise for very long lists.
- A more object-oriented approach would be to make `map`, `filter`, and `length` instance methods instead of static methods. The method signatures would change to:

```
<B> List<B> map(Func<B,T> f) {...}
List<T> filter(Pred<T> f) {...}
int length() {...}
```

The disadvantage of this approach is that we have to add special cases in any *use* of these methods if the client may have an empty list. The reason is empty lists are represented as `null` and using `null` as the receiver of a call raises a `NullPointerException`. So methods `doubleAll` and `countNs` would have to check their arguments for `null` to avoid such exceptions.

- Another more object-oriented approach would be to not use `null` for empty lists. Instead we would have an abstract list class with two subclasses, one for empty lists and one for nonempty lists. This approach is a much more faithful object-oriented approach to datatypes with multiple constructors, and using it makes the previous suggestion of instance methods work out without special cases. It does seem more complicated and longer to programmers accustomed to using `null`.
- Anonymous inner classes are just a convenience. We could instead define "normal" classes that implement `Func<Integer,Integer>` and `Pred<Integer>` and create instances to pass to `map` and `filter`. For the `countNs` example, our class would have an `int` field for holding n and we would pass the value for this field to the constructor of the class, which would initialize the field.

Optional: Closures in C Using Explicit Environments

C does have functions, but they are not closures. If you pass a pointer to a function, it is only a code pointer. As we have studied, if a function argument can use only its arguments, higher-order functions are much less useful. So what can we do in a language like C? We can change the higher-order functions as follows:

- Take the environment explicitly as another argument.
- Have the function-argument also take an environment.
- When calling the function-argument, pass it the environment.

So instead of a higher-order function looking something like this:

```
int f(int (*g)(int), list_t xs) { ... g(xs->head) ... }
```

we would have it look like this:

```
int f(int (*g)(void*,int), void* env, list_t xs) { ... g(env,xs->head) ... }
```

We use `void*` because we want `f` to work with functions that use environments of different types, so there is no good choice. Clients will have to cast to and from `void*` from other compatible types. We do not discuss those details here.

While the C code has a lot of other details, this use of explicit environments in the definitions and uses of `map` and `filter` is the key difference from the versions in other languages:

```
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
```

```
typedef struct List list_t;
struct List {
  void * head;
  list_t * tail;
};
list_t * makelist (void * x, list_t * xs) {
  list_t * ans = (list_t *)malloc(sizeof(list_t));
  ans->head = x;
  ans->tail = xs;
  return ans;
}
list_t * map(void* (*f)(void*,void*), void* env, list_t * xs) {
  if(xs==NULL)
    return NULL;
  return makelist(f(env,xs->head), map(f,env,xs->tail));
}
list_t * filter(bool (*f)(void*,void*), void* env, list_t * xs) {
  if(xs==NULL)
    return NULL;
  if(f(env,xs->head))
    return makelist(xs->head, filter(f,env,xs->tail));
  return filter(f,env,xs->tail);
}
int length(list_t* xs) {
  int ans = 0;
  while(xs != NULL) {
    ++ans;
    xs = xs->tail;
  }
  return ans;
}
void* doubleInt(void* ignore, void* i) { // type casts to match what map expects
  return (void*)(((intptr_t)i)*2);
}
list_t * doubleAll(list_t * xs) { // assumes list holds intptr_t fields
  return map(doubleInt, NULL, xs);
}
bool isN(void* n, void* i) { // type casts to match what filter expects
  return ((intptr_t)n)==((intptr_t)i);
}
int countNs(list_t * xs, intptr_t n) { // assumes list hold intptr_t fields
  return length(filter(isN, (void*)n, xs));
}
```

As in Java, using recursion instead of loops is much simpler but likely less efficient. Another alternative would be to define structs that put the code and environment together in one value, but our approach of using is an extra `void*` argument to every higher-order function is more common in C code.

For those interested in C-specification details: Also note the client code above, specifically the code in functions `doubleInt`, `isN`, and `countNs`, is not portable because it is not technically legal to assume that an `intptr_t` can be cast to a `void*` and back unless the value started as a pointer (rather than a number that fits in an `intptr_t`). While the code as written above is a fairly common approach, portable versions

would either need to use a pointer to a number or replace the uses of `void*` in the library with `intptr_t`. The latter approach is still a reusable library because any pointer can be converted to `intptr_t` and back.

Standard-Library Documentation

This topic is not closely related to the rest of the section, but we need it a little for Homework 3, it is useful for any programming language, and it shows some of the useful functions (higher-order or not) predefined in ML.

ML, like many languages, has a *standard library*. This is code that programs in the language can assume is always available. There are two common and distinct reasons for code to be in a standard library:

- We need a standard-library to interface with the "outside world" to provide features that would otherwise be impossible to implement. Examples include opening a file or setting a timer.
- A standard-library can provide functions so common and useful that it is appropriate to define them once so that all programs can use the same function name, order of arguments, etc. Examples include functions to concatenate two strings, map over a list, etc.

Standard libraries are usually so large that it makes no sense to expect to be taught them. You need to get comfortable seeking out documentation and developing a rough intuition on "what is likely provided" and "where it is likely to be." So on Homework 3, we are leaving it to you to find out more about a few simple functions in ML's Standard Library.

The online documentation is very primitive compared to most modern languages, but it is entirely sufficient for our needs. Just go to:

`http://www.standardml.org/Basis/manpages.html`

The functions are organized using ML's *module system*, which we will study the basics of in the next section. For example, useful functions over characters are in the structure `Char`. To use a function `foo` in structure `Bar`, you write `Bar.foo`, which is exactly how we have been using functions like `List.map`. One wrinkle is that functions for the `String` structure are documented under the signature `STRING`. Signatures are basically types for structures, as we will study later. Certain library functions are considered so useful they are not in a structure, like `hd`. These bindings are described at
`http://www.standardml.org/Basis/top-level-chapter.html`.

There is no substitute for precise and complete documentation of code libraries, but sometimes it can be inconvenient to look up the full documentation when you are in the middle of programming and just need a quick reminder. For example, it is easy to forget the order of arguments or whether a function is curried or tupled. Often you can use the REPL to get the information you need quickly. After all, if you enter a function like `List.map`, it evaluates this expression and returns its type. You can even guess the name of a function if you do not remember what it is called. If you are wrong, you will just get an undefined-variable message. Finally, using features just beyond what we will study, you can get the REPL to print out all the bindings provided by a structure. Just do this for example:

```
structure X = List; (* List is the structure we want to know about *)
structure X : LIST  (* This is what the REPL gives back *)
signature X = LIST; (* Write LIST because that is what follows the : on the previous line *)
```

Because looking things up in the REPL is so convenient, some REPLs for other languages have gone further and provided special commands for printing the documentation associated with functions or libraries.

Programming Languages (Coursera / University of Washington) Assignment 3

You will define several SML functions. Many will be very short because they will use other higher-order functions. You may use functions in ML's library; the problems point you toward the useful functions and often ***require*** that you use them. The sample solution is about 120 lines, including the provided code, but not including the challenge problem. This assignment is probably more difficult than Homework 2 even though (perhaps because) many of the problems have 1-line answers.

Download `hw3provided.sml` from the course website.

---

1. Write a function `only_capitals` that takes a `string list` and returns a `string list` that has only the strings in the argument that start with an uppercase letter. Assume all strings have at least 1 character. Use `List.filter`, `Char.isUpper`, and `String.sub` to make a 1-2 line solution.

2. Write a function `longest_string1` that takes a `string list` and returns the longest `string` in the list. If the list is empty, return `""`. In the case of a tie, return the string closest to the beginning of the list. Use `foldl`, `String.size`, and no recursion (other than the implementation of `foldl` is recursive).

3. Write a function `longest_string2` that is exactly like `longest_string1` except in the case of ties it returns the string closest to the end of the list. Your solution should be almost an exact copy of `longest_string1`. Still use `foldl` and `String.size`.

4. Write functions `longest_string_helper`, `longest_string3`, and `longest_string4` such that:

   - `longest_string3` has the same behavior as `longest_string1` and `longest_string4` has the same behavior as `longest_string2`.
   - `longest_string_helper` has type `(int * int -> bool) -> string list -> string` (notice the currying). This function will look a lot like `longest_string1` and `longest_string2` but is more general because it takes a function as an argument.
   - If `longest_string_helper` is passed a function that behaves like `>` (so it returns `true` exactly when its first argument is stricly greater than its second), then the function returned has the same behavior as `longest_string1`.
   - `longest_string3` and `longest_string4` are defined with `val`-bindings and partial applications of `longest_string_helper`.

5. Write a function `longest_capitalized` that takes a `string list` and returns the longest string in the list that begins with an uppercase letter, or `""` if there are no such strings. Assume all strings have at least 1 character. Use a `val`-binding and the ML library's `o` operator for composing functions. Resolve ties like in problem 2.

6. Write a function `rev_string` that takes a `string` and returns the `string` that is the same characters in reverse order. Use ML's `o` operator, the library function `rev` for reversing lists, and two library functions in the `String` module. (Browse the module documentation to find the most useful functions.)

The next two problems involve writing functions over lists that will be useful in later problems.

7. Write a function `first_answer` of type `('a -> 'b option) -> 'a list -> 'b` (notice the 2 arguments are curried). The first argument should be applied to elements of the second argument in order until the first time it returns `SOME v` for some `v` and then `v` is the result of the call to `first_answer`. If the first argument returns `NONE` for all list elements, then `first_answer` should raise the exception `NoAnswer`. Hints: Sample solution is 5 lines and does nothing fancy.

8. Write a function `all_answers` of type `('a -> 'b list option) -> 'a list -> 'b list option` (notice the 2 arguments are curried). The first argument should be applied to elements of the second argument. If it returns `NONE` for any element, then the result for `all_answers` is `NONE`. Else the calls to the first argument will have produced `SOME lst1`, `SOME lst2`, ... `SOME lstn` and the result of `all_answers` is `SOME lst` where `lst` is `lst1`, `lst2`, ..., `lstn` appended together (order doesn't matter). Hints: The sample solution is 8 lines. It uses a helper function with an accumulator and uses `@`. Note `all_answers f []` should evaluate to `SOME []`.

The remaining problems use these type definitions, which are inspired by the type definitions an ML implementation would use to implement pattern matching:

```
datatype pattern = Wildcard | Variable of string | UnitP | ConstP of int
                 | TupleP of pattern list | ConstructorP of string * pattern
datatype valu = Const of int | Unit | Tuple of valu list | Constructor of string * valu
```

Given `valu` v and `pattern` p, either p *matches* v or not. If it does, the match produces a list of `string * valu` pairs; order in the list does not matter. The rules for matching should be unsurprising:

- `Wildcard` matches everything and produces the empty list of bindings.
- `Variable s` matches any value `v` and produces the one-element list holding `(s,v)`.
- `UnitP` matches only `Unit` and produces the empty list of bindings.
- `ConstP 17` matches only `Const 17` and produces the empty list of bindings (and similarly for other integers).
- `TupleP ps` matches a value of the form `Tuple vs` if `ps` and `vs` have the same length and for all i, the i^{th} element of `ps` matches the i^{th} element of `vs`. The list of bindings produced is all the lists from the nested pattern matches appended together.
- `ConstructorP(s1,p)` matches `Constructor(s2,v)` if `s1` and `s2` are the same string (you can compare them with `=`) and `p` matches `v`. The list of bindings produced is the list from the nested pattern match. We call the strings `s1` and `s2` the *constructor name*.
- Nothing else matches.

9. (This problem uses the `pattern` datatype but is not really about pattern-matching.) A function `g` has been provided to you.
   (a) Use `g` to define a function `count_wildcards` that takes a pattern and returns how many `Wildcard` patterns it contains.
   (b) Use `g` to define a function `count_wild_and_variable_lengths` that takes a pattern and returns the number of `Wildcard` patterns it contains plus the sum of the string lengths of all the variables in the variable patterns it contains. (Use `String.size`. We care only about variable names; the constructor names are not relevant.)
   (c) Use `g` to define a function `count_some_var` that takes a string and a pattern (as a pair) and returns the number of times the string appears as a variable in the pattern. We care only about variable names; the constructor names are not relevant.

10. Write a function `check_pat` that takes a pattern and returns true if and only if all the variables appearing in the pattern are distinct from each other (i.e., use different strings). The constructor names are not relevant. Hints: The sample solution uses two helper functions. The first takes a pattern and returns a list of all the strings it uses for variables. Using `foldl` with a function that uses `@` is useful in one case. The second takes a list of strings and decides if it has repeats. `List.exists` may be useful. Sample solution is 15 lines. These are hints: We are not requiring `foldl` and `List.exists` here, but they make it easier.

11. Write a function `match` that takes a `valu * pattern` and returns a `(string * valu) list option`, namely `NONE` if the pattern does not match and `SOME lst` where `lst` is the list of bindings if it does. Note that if the value matches but the pattern has no patterns of the form `Variable s`, then the result is `SOME []`. Hints: Sample solution has one case expression with 7 branches. The branch for tuples uses `all_answers` and `ListPair.zip`. Sample solution is 13 lines. Remember to look above for the rules for what patterns match what values, and what bindings they produce. These are hints: We are not requiring `all_answers` and `ListPair.zip` here, but they make it easier.

12. Write a function `first_match` that takes a value and a list of patterns and returns a `(string * valu) list option`, namely `NONE` if no pattern in the list matches or `SOME lst` where `lst` is the list of bindings for the first pattern in the list that matches. Use `first_answer` and a `handle`-expression. Hints: Sample solution is 3 lines.

(Challenge Problem) Write a function `typecheck_patterns` that "type-checks" a `pattern list`. Types for our made-up pattern language are defined by:

```
datatype typ = Anything (* any type of value is okay *)
             | UnitT (* type for Unit *)
             | IntT (* type for integers *)
             | TupleT of typ list (* tuple types *)
             | Datatype of string (* some named datatype *)
```

`typecheck_patterns` should have type `((string * string * typ) list) * (pattern list) -> typ option`. The first argument contains elements that look like `("foo","bar",IntT)`, which means constructor `foo` makes a value of type `Datatype "bar"` given a value of type `IntT`. Assume list elements all have different first fields (the constructor name), but there are probably elements with the same second field (the datatype name). Under the assumptions this list provides, you "type-check" the `pattern list` to see if there exists some `typ` (call it t) that *all* the patterns in the list can have. If so, return `SOME t`, else return `NONE`.

You must return the "most lenient" type that all the patterns can have. For example, given patterns `TupleP[Variable("x"),Variable("y")]` and `TupleP[Wildcard,Wildcard]`, return `TupleT[Anything,Anything]` even though they could both have type `TupleT[IntT,IntT]`. As another example, if the only patterns are `TupleP[Wildcard,Wildcard]` and `TupleP[Wildcard,TupleP[Wildcard,Wildcard]]`, you must return `TupleT[Anything,TupleT[Anything,Anything]]`.

Type Summary: Evaluating a correct homework solution should generate these bindings, in addition to the bindings for datatype and exception definitions:

```
val g = fn : (unit -> int) -> (string -> int) -> pattern -> int
val only_capitals = fn : string list -> string list
val longest_string1 = fn : string list -> string
val longest_string2 = fn : string list -> string
```

```
val longest_string_helper = fn : (int * int -> bool) -> string list -> string
val longest_string3 = fn : string list -> string
val longest_string4 = fn : string list -> string
val longest_capitalized = fn : string list -> string
val rev_string = fn : string -> string
val first_answer = fn : ('a -> 'b option) -> 'a list -> 'b
val all_answers = fn : ('a -> 'b list option) -> 'a list -> 'b list option
val count_wildcards = fn : pattern -> int
val count_wild_and_variable_lengths = fn : pattern -> int
val count_some_var = fn : string * pattern -> int
val check_pat = fn : pattern -> bool
val match = fn : valu * pattern -> (string * valu) list option
val first_match = fn : valu -> pattern list -> (string * valu) list option
```

Of course, generating these bindings does not guarantee that your solutions are correct. *Test your functions: Put your testing code in a second file. We will not grade it, but you must turn it in.*

Assessment: We will automatically test your functions on a variety of inputs, including edge cases. We will also ask peers to evaluate your code for simplicity, conciseness, elegance, and good formatting including indentation and line breaks. Do not use SML's mutable references or arrays.

Turn-in Instructions

First, follow the instructions on the course website to submit your solution file (not your testing file) for auto-grading. Do not proceed to the peer-assessment submission until you receive a high-enough grade from the auto-grader: Doing peer assessment requires instructions that include a sample solution, so these instructions will be "locked" until you receive high-enough auto-grader score. Then submit your same solution file again for peer assessment and follow the peer-assessment instructions.

Coursera Programming Languages Course
Section 4 Summary

Standard Description: This summary covers **roughly** *the same material as the lecture videos and the materials (slides, code) posted with each video. It can help to read about the material in a narrative style and to have the material for an entire section of the course in a single document, especially when reviewing the material later. Please report errors in these notes on the discussion board.*

Contents

What is Type Inference?

While we have been using ML type inference for a while now, we have not studied it carefully. We will first carefully define what type inference *is* and then see via several examples how ML type inference works.

Java, C, and ML are all examples of *statically typed languages*, meaning every binding has a type that is determined "at compile-time," i.e., before any part of the program is run. The type-checker is a compile-time procedure that either accepts or rejects a program. By contrast, Racket, Ruby, and Python are *dynamically typed languages*, meaning the type of a binding is not determined ahead of time and computations like binding 42 to `x` and then treating `x` as a string result in run-time errors. After we do some programming with Racket, we will compare the advantages and disadvantages of static versus dynamic typing as a significant course topic.

Unlike Java and C, ML is *implicitly typed*, meaning programmers rarely need to write down the types of bindings. This is often convenient (though some disagree as to whether it makes code easier or harder

to read), but in no way changes the fact that ML is statically typed. Rather, the type-checker has to be more sophisticated because it must *infer* (i.e., figure out) what the *type annotations* "would have been" had programmers written all of them. In principle, type inference and type checking could be separate steps (the inferencer could do its part and the checker could see if the result should type-check), but in practice they are often merged into "the type-checker." Note that a correct type-inferencer must find a solution to what all the types should be whenever such a solution exists, else it must reject the program.

Whether type inference for a particular programming language is easy, difficult, or impossible is often not obvious. It is *not* proportional to how permissive the type system is. For example, the "extreme" type systems that "accept everything" and "accept nothing" are both very easy to do inference for. When we say type inference may be impossible, we mean this in the technical sense of undecidability, like the famous halting problem. We mean there are type systems for which no computer program can implement type inference such that (1) the inference process always terminates, (2) the inference process always succeeds if inference is possible, and (3) the inference process always fails if inference is not possible.

Fortunately, ML was rather cleverly designed so that type inference can be performed by a fairly straightforward and elegant algorithm. While there are programs for which inference is intractably slow, programs people write in practice never cause such behavior. We will demonstrate key aspects of the algorithm for ML type inference with a few examples. This will give you a sense that type inference is not "magic." In order to move on to other course topics, we will not describe the full algorithm or write code to implement it.

ML type inference ends up intertwined with parametric polymorphism — when the inferencer determines a function's argument or result "could be anything" the resulting type uses `'a`, `'b`, etc. But type inference and polymorphism are entirely separate concepts: a language could have one or the other. For example, Java has generics but no inference for method argument/result types.

Overview of ML Type Inference

Here is an overview of how ML type inference works (more examples to follow):

- It determines the types of bindings in order, using the types of earlier bindings to infer the types of later ones. This is why you cannot use later bindings in a file. (When you need to, you use mutual recursion and type inference determines the types of all the mutually recursive bindings together. Mutual recursion is covered later in this section.)
- For each `val` or `fun` binding, it analyzes the binding to determine necessary facts about its type. For example, if we see the expression `x+1`, we conclude that `x` must have type `int`. We gather similar facts for function calls, pattern-matches, etc.
- Afterward, use *type variables* (e.g., `'a`) for any unconstrained types in function arguments or results.
- (Enforce the value restriction — only variables and values can have polymorphic types, as discussed later.)

The amazing fact about the ML type system is that "going in order" this way never causes us to reject a program that could type-check nor do we ever accept a program we should not. So explicit type annotations really are optional unless you use features like `#1`. (The problem with `#1` is that it does not give enough information for type inference to know what other fields the tuple/record should have, and the ML type system requires knowing the exact number of fields and all the fields' names.)

Here is an initial, very simple example:

```
val x = 42
fun f(y,z,w) = if y then z+x else 0
```

Type inference first gives `x` type `int` since `42` has type `int`. Then it moves on to infer the type for `f`. Next we will study, via other examples, a more step-by-step procedure, but here let us just list the key facts:

- `y` must have type `bool` because we test it in a conditional.
- `z` must have type `int` because we add it to something we already determined has type `int`.
- `w` can have any type because it is never used.
- `f` must return an `int` because its body is a conditional where both branches return an `int`. (If they disagreed, type-checking would fail.)

So the type of `f` must be `bool * int * 'a -> int`.

More Thorough Examples of ML Type Inference

We will now work through a few examples step-by-step, generating all the facts that the type-inference algorithm needs. Note that humans doing type inference "in their head" often take shortcuts just like humans doing arithmetic in their head, but the point is there is a general algorithm that methodically goes through the code gathering constraints and putting them together to get the answer.

As a first example, consider inferring the type for this function:

```
fun f x =
   let val (y,z) = x in
       (abs y) + z
   end
```

Here is how we can infer the type:

- Looking at the first line, `f` must have type `T1->T2` for some types `T1` and `T2` and in the function body `f` has this type and `x` has type `T1`.
- Looking at the `val`-binding, `x` must be a pair type (else the pattern-match makes no sense), so in fact `T1=T3*T4` for some `T3` and `T4`, and `y` has type `T3` and `z` has type `T4`.
- Looking at the addition expression, we know from the context that `abs` has type `int->int`, so `y` having type `T3` means `T3=int`. Similarly, since `abs y` has type `int`, the other argument to `+` must have type `int`, so `z` having type `T4` means `T4=int`.
- Since the type of the addition expression is `int`, the type of the let-expression is `int`. And since the type of the let-expression is `int`, the return type of `f` is `int`, i.e., `T2=int`.

Putting all these constraints together, `T1=int*int` (since `T1=T3*T4`) and `T2=int`, so `f` has type `int*int->int`.

Next example:

```
fun sum xs =
   case xs of
     [] => 0
   | x::xs' => x + (sum xs')
```

- From the first line, there exists types `T1` and `T2` such that `sum` has type `T1->T2` and `xs` has type `T1`.
- Looking at the case-expression, `xs` must have a type that is compatible with all of the patterns. Looking at the patterns, both of them match any list, since they are built from list constructors (in the `x::xs'` case the subpatterns match anything of any type). So since `xs` has type `T1`, in fact `T1=T3 list` from some type `T3`.
- Looking at the right-hand sides of the case branches, we know they must have the same type as each other and this type is `T2`. Since 0 has type `int`, `T2=int`.
- Looking at the second case branch, we type-check it in a context where `x` and `xs'` are available. Since we are matching the pattern `x::xs'` against a `T3 list`, it must be that `x` has type `T3` and `xs'` has type `T3 list`.
- Now looking at the right-hand side, we add `x`, so in fact `T3=int`. Moreover, the recursive call type-checks because `xs'` has type `T3 list` and `T3 list=T1` and `sum` has type `T1->T2`. Finally, since `T2=int`, adding `sum xs'` type-checks.

Putting everything together, we get `sum` has type `int list -> int`.

Notice that before we got to `sum xs'` we had already inferred everything, but we still have to check that types are used consistently and reject otherwise. For example, if we had written `sum x`, that cannot type-check — it is *inconsistent* with previous facts. Let us see this more thoroughly to see what happens:

```
fun broken_sum xs =
   case xs of
     [] => 0
   | x::xs' => x + (broken_sum x)
```

- Type inference for `broken_sum` proceeds largely the same as for `sum`. The first four bullets from the previous example all apply, giving `broken_sum` type `T3 list ->int`, `x3` type `T3 list`, `x` type `T3`, and `xs'` type `T3 list`. Moreover, `T3=int`.
- We depart from the correct `sum` implementation with the call `broken_sum x`. For this call to type-check, `x` must have the same type as `broken_sum`'s parameter, or in other words, `T1=T3`. However, we know that `T1=T3 list`, so this new constraint `T1=T3` actually generates a contradiction: `T3=T3 list`. If we want to be more concrete, we can use our knowledge that `T3=int` to rewrite this as `int=int list`. Looking at the definition of `broken_sum` it should be obvious that this is exactly the problem: we tried to use `x` as an `int` and as an `int list`.

When your ML program does not type-check, the type-checker reports the expression where it discovered a contradiction and what types were involved in that contradiction. While sometimes this information is helpful, other times the actual problem is with a different expression, but the type-checker did not reach a contradiction until later.

Examples with Polymorphic Types

Our remaining examples will infer polymorphic types. All we do is follow the same procedure we did above, but when we are done, we will have some parts of the function's type that are still *unconstrained*. For each `Ti` that "can be anything" we use a type variable (`'a`, `'b`, etc.).

```
fun length xs =
   case xs of
     [] => 0
   | x::xs' => 1 + (length xs')
```

Type inference proceeds much like with `sum`. We end up determining:

- `length` has type `T1->T2`.
- `xs` has type `T1`.
- `T1=T3 list` (due to the pattern-match)
- `T2=int` because 0 can be the result of a call to `length`.
- `x` has type `T3` and `xs'` has type `T3 list`.
- The recursive call `length xs'` type-checks because `xs'` has type `T3 list`, which is `T1`, the argument type of `length`. And we can add the result because `T2=int`.

So we have all the same constraints as for `sum`, *except* we do not have `T3=int`. In fact, `T3` can be anything and `length` will type-check. So type inference recognizes that when it is all done, it has `length` with type `T3 list -> int` and `T3` can be anything. So we end up with the type `'a list -> int`, as expected. Again the rule is simple: for each `Ti` in the final result that cannot be constrained, use a type variable.

A second example:

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

- Since the argument to `compose` must be a pair (from the pattern used for its argument), `compose` has type `T1*T2->T3`, `f` has type `T1` and `g` has type `T2`.
- Since `compose` returns a function, `T3` is some `T4->T5` where in that function's body, `x` has type `T4`.
- So `g` must have type `T4->T6` for some `T6`, i.e., `T2=T4->T6`.
- And `f` must have type `T6->T7` for some `T7`, i.e., `T1=T6->T7`.
- But the result of `f` is the result of the function returned by `compose`, so `T7=T5` and so `T1=T6->T5`.

Putting together `T1=T6->T5` and `T2=T4->T6` and `T3=T4->T5` we have a type for `compose` of `(T6->T5)*(T4->T6) -> (T4->T5)`. There is nothing else to constrain the types `T4`, `T5`, and `T6`, so we replace them consistently to end up with `('a->'b)*('c->'a) -> ('c->'b)` as expected (and the last set of parentheses are optional, but that is just syntax).

Here is a simpler example that also has multiple type variables:

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

- The first line requires that `f` has type `T1*T2*T3 -> T4`, `x` has type `T1`, `y` has type `T2`, and `z` has type `T3`.

- The two branches of the conditional must have the same type and this is the return type of the function `T4`. Therefore, `T4=T1*T2*T3` and `T4=T2*T1*T3`. This constraint requires `T1=T2`.

Putting together these constraints (and no others), `f` will type-check with type `T1*T1*T3 -> T1*T1*T3` for any types `T1` and `T3`. So replacing each type consistently with a type variable, we get `'a*'a*'b -> 'a*'a*'b`, which is correct: `x` and `y` must have the same type, but `z` can (but need not) have a different type. Notice that the type-checker always requires both branches of a conditional to type-check with the same type, even though here we know which branch will be evaluated.

Optional: The Value Restriction

As described so far in this section, the ML type system is *unsound*, meaning that it would accept programs that when run could have values of the wrong types, such as putting an `int` where we expect a `string`. The problem results from a combination of polymorphic types and mutable references, and the fix is a special restriction to the type system called *the value restriction*.

This is an example program that demonstrates the problem:

```
val r = ref NONE       (* 'a option ref *)
val _ = r := SOME "hi" (* instantiate 'a with string *)
val i = 1 + valOf(!r)  (* instantiate 'a with int *)
```

Straightforward use of the rules for type checking/inference would accept this program even though we should not – we end up trying to add `1` to `"hi"`. Yet everything seems to type-check given the types for the functions/operators `ref` (`'a -> 'a ref`), `:=` (`'a ref * 'a -> unit`), and `!` (`'a ref -> 'a`).

To restore soundness, we need a stricter type system that does not let this program type-check. The choice ML made is to prevent the first line from having a polymorphic type. Therefore, the second and third lines will not type-check because they will not be able to instantiate an `'a` with `string` or `int`. In general, ML will give a variable in a val-binding a polymorphic type only if the expression in the val-binding is a value or a variable. This is called the value restriction. In our example, `ref NONE` is a call to the function `ref`. Function calls are not variables or values. So we get a warning and `r` is given a type `?X1 option ref` where `?X1` is a "dummy type," not a type variable. This makes `r` not useful and the later lines do not type-check. It is not at all obvious that this restriction suffices to make the type system sound, but in fact it is sufficient.

For `r` above, we can use the expression `ref NONE`, but we have to use a type annotation to give `r` a non-polymorphic type, such as `int option ref`. Whatever we pick, one of the next two lines will not type-check.

As we saw previously when studying partial application, the value restriction is occasionally burdensome even when it is not a problem because we are not using mutation. We saw that this binding falls victim to the value-restriction and is not made polymorphic:

```
val pairWithOne = List.map (fn x => (x,1))
```

We saw multiple workarounds. One is to use a function binding, even though without the value restriction it would be unnecessary function wrapping. This function has the desired type `'a list -> ('a * int) list`:

```
fun pairWithOne xs = List.map (fn x => (x,1)) xs
```

One might wonder why we cannot enforce the value restriction only for references (where we need it) and not for immutable types like lists. The answer is the ML type-checker cannot always know which types are

really references and which are not. In the code below, we need to enforce the value restriction on the last line, because `'a foo` and `'a ref` are the same type.

```
type 'a foo = 'a ref
val f : 'a -> 'a foo = ref
val r = f NONE
```

When we study the module system later in this section, we will see the type-checker does not always know the definition of type synonyms. So to be safe, it enforces the value restriction for all types.

Optional: Some Things that Make Type Inference More Difficult

Now that we have seen how ML type inference works, we can make two interesting observations:

- Inference would be more difficult if ML had subtyping (e.g., if every triple could also be a pair) because we would not be able to conclude things like, "`T3=T1*T2`" since the *equals* would be overly restrictive. We would instead need constraints indicating that `T3` is a tuple with *at least* two fields. Depending on various details, this can be done, but type inference is more difficult and the results are more difficult to understand.
- Inference would be more difficult if ML did *not* have parametric polymorphism since we would have to pick some type for functions like `length` and `compose` and that could depend on how they are used.

Mutual Recursion

We have seen many examples of recursive functions and many examples of functions using other functions as helper functions, but what if we need a function `f` to call `g` and `g` to call `f`? That can certainly be useful, but ML's rule that bindings can only use earlier bindings makes it more difficult — which should come first, `f` or `g`?

It turns out ML has special support for mutual recursion using the keyword `and` and putting the mutually recursive functions next to each other. Similarly, we can have mutually recursive `datatype` bindings. After showing these new constructs, we will show that you can actually work around a lack of support for mutually recursive functions by using higher-order functions, which is a useful trick in general and in particular in ML if you do not want your mutually recursive functions next to each other.

Our first example uses mutual recursion to process an `int list` and return a `bool`. It returns true if the list strictly alternates between `1` and `2` and ends with a `2`. Of course there are many ways to implement such a function, but our approach does a nice job of having for each "state" (such as "a 1 must come next" or "a 2 must come next") a function. In general, many problems in computer science can be modeled by such *finite state machines*, and mutually recursive functions, one for each state, are an elegant way to implement finite state machines.[1]

```
fun match xs =
    let fun s_need_one xs =
            case xs of
                [] => true
```

[1] Because all function calls are tail calls, the code runs in a small amount of space, just as one would expect for an implementation of a finite state machine.

```
            | 1::xs' => s_need_two xs'
            | _ => false
        and s_need_two xs =
            case xs of
                [] => false
            | 2::xs' => s_need_one xs'
            | _ => false
    in
        s_need_one xs
    end
```

(The code uses integer constants in patterns, which is an occasionally convenient ML feature, but not essential to the example.)

In terms of syntax, we define mutually recursive functions by simply *replacing* the keyword `fun` for all functions except the first with `and`. The type-checker will type-check all the functions (two in the example above) together, allowing calls among them regardless of order.

Here is a second (silly) example that also uses two mutually recursive `datatype` bindings. The definition of types `t1` and `t2` refer to each other, which is allowed by using `and` in place of `datatype` for the second one. This defines two new datatypes, `t1` and `t2`.

```
datatype t1 = Foo of int | Bar of t2
and t2 = Baz of string | Quux of t1

fun no_zeros_or_empty_strings_t1 x =
    case x of
        Foo i => i <> 0
      | Bar y => no_zeros_or_empty_strings_t2 y
and no_zeros_or_empty_strings_t2 x =
    case x of
        Baz s => size s > 0
      | Quux y => no_zeros_or_empty_strings_t1 y
```

Now suppose we wanted to implement the "no zeros or empty strings" functionality of the code above but for some reason we did not want to place the functions next to each other or we were in a language with no support for mutually recursive functions. We can write almost the same code by having the "later" function pass itself to a version of the "earlier" function that takes a function as an argument:

```
fun no_zeros_or_empty_strings_t1(f,x) =
    case x of
        Foo i => i <> 0
      | Bar y => f y

fun no_zeros_or_empty_string_t2 x =
    case x of
        Baz s => size s > 0
      | Quux y => no_zeros_or_empty_strings_t1(no_zeros_or_empty_string_t2,y)
```

This is yet-another powerful idiom allowed by functions taking functions.

Modules for Namespace Management

We start by showing how ML modules can be used to separate bindings into different *namespaces*. Later segments build on this material to cover the much more interesting and important topic of using modules to hide bindings and types.

To learn the basics of ML, pattern-matching, and functional programming, we have written small programs that are just a sequence of bindings. For larger programs, we want to organize our code with more structure. In ML, we can use *structures* to define *modules* that contain a collection of bindings. At its simplest, you can 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 type synonyms. Inside the structure you can use earlier bindings just like we have been doing "at top-level" (i.e., outside of any module). Outside the structure, you refer to a binding b in `Name` by writing `Name.`b. We have already been using this notation to use functions like `List.foldl`; now you know how to define your own structures.

Though we will not do so in our examples, you can nest structures inside other structures to create a tree-shaped hierarchy. But in ML, modules are *not* expressions: you cannot define them inside of functions, store them in tuples, pass them as arguments, etc.

If in some scope you are using many bindings from another structure, it can be inconvenient to write `SomeLongStructureName.foo` many times. Of course, you can use a val-binding to avoid this, e.g., `val foo = SomeLongStructureName.foo`, but this technique is ineffective if we are using many different bindings from the structure (we would need a new variable for each) or for using constructor names from the structure in patterns. So ML allows you to write `open SomeLongStructureName`, which provides "direct" access (you can just write `foo`) to any bindings in the module that are mentioned in the module's signature. The scope of an open is the rest of the enclosing structure (or the rest of the program at top-level).

A common use of `open` is to write succinct testing code for a module outside the module itself. Other uses of `open` are often frowned upon because it may introduce unexpected shadowing, especially since different modules may reuse binding names. For example, a list module and a tree module may both have functions named `map`.

Signatures

So far, structures are providing just *namespace management*, a way to avoid different bindings in different parts of the program from shadowing each other. Namespace management is very useful, but not very interesting. Much more interesting is giving structures *signatures*, which are types for modules. They let us provide strict *interfaces* that code outside the module must obey. ML has several ways to do this with subtly different syntax and semantics; we just show one way to write down an explicit signature for a module. Here is an example signature definition and structure definition that says the structure `MyMathLib` must have the signature `MATHLIB`:

```
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 type-check 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.

Hiding Things

Before learning how to use ML modules to hide implementation details from clients, let's remember that separating an interface from an implementation is probably the most important strategy for building correct, robust, reusable programs. Moreover, we can already use functions to hide implementations in various ways. 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
```

Another feature we use for hiding implementations is defining functions locally inside other functions. We can later change, remove, or add locally defined functions knowing the old versions were not relied on by any other code. 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.

But what if you wanted to have two top-level functions that code in other modules could use and have both of them use the same hidden functions? There are ways to do this (e.g., create a record of functions), but it would be convenient to have some top-level functions that were "private" to the module. In ML, there is no "private" keyword like in other languages. Instead, you use signatures that simply mention less: anything not explicitly in a signature cannot be used from the outside. For example, if we change the signature above to:

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

then client code cannot call `MyMathLib.doubler`. The binding simply is not in scope, so no use of it will type-check. In general, the idea is that we can implement the module however we like and only bindings that are explicitly listed in the signature can be called directly by clients.

Introducing our extended example

The rest of our module-system study will use as an example a small module that implements rational numbers. While a real library would provide many more features, ours will just support creating fractions, adding two fractions, and converting fractions to strings. Our library intends to (1) prevent denominators of zero and (2) keep fractions in reduced form (3/2 instead of 9/6 and 4 instead of 4/1). While negative fractions are fine, internally the library never has a negative denominator ($-3/2$ instead of $3/-2$ and $3/2$ instead of $-3/-2$). The structure below implements all these ideas, using the helper function `reduce`, which itself uses `gcd`, for reducing a fraction.

Our module maintains *invariants*, as seen in the comments near the top of the code. These are properties of fractions that all the functions both *assume to be true* and *guarantee to keep true*. If one function violates the invariants, other functions might do the wrong thing. For example, the `gcd` function is incorrect for negative arguments, but because denominators are never negative, `gcd` is never called with a negative argument.

```
structure Rational1 =
struct
(* Invariant 1: all denominators > 0
   Invariant 2: rationals kept in reduced form, including that
                a Frac never has a denominator of 1 *)
  datatype rational = Whole of int | Frac of int*int
  exception BadFrac

(* gcd and reduce help keep fractions reduced,
   but clients need not know about them *)
(* they _assume_ their inputs are not negative *)
  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) =>
           if x=0
           then Whole 0
           else let val d = gcd(abs x,y) in (* using invariant 1 *)
                    if d=y
                    then Whole(x div d)
                    else Frac(x div d, y div d)
                end

(* when making a frac, we ban zero denominators *)
   fun make_frac (x,y) =
       if y = 0
       then raise BadFrac
       else if y < 0
       then reduce(Frac(~x,~y))
       else reduce(Frac(x,y))
```

```
(* using math properties, both invariants hold of the result
   assuming they hold of the arguments *)
   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))

(* given invariant, prints in reduced form *)
   fun toString r =
       case r of
           Whole i => Int.toString i
         | Frac(a,b) => (Int.toString a) ^ "/" ^ (Int.toString b)

end
```

Signatures for Our Example

Let us now try to give our example module a signature such that clients can use it but not violate its invariants.

Since `reduce` and `gcd` are helper functions that we do not want clients to rely on or misuse, one natural signature would be as follows:

```
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
```

To use this signature to hide `gcd` and `reduce`, we can just change the first line of the structure definition above to `structure Rational1 :> RATIONAL_A`.

While this approach ensures clients do not call `gcd` or `reduce` directly (since they "do not exist" outside the module), this is *not* enough to ensure the bindings in the module are used correctly. What "correct" means for a module depends on the specification for the module (not the definition of the ML language), so let's be more specific about some of the desired *properties* of our library for rational numbers:

- Property: `toString` always returns a string representation in reduced form
- Property: No code goes into an infinite loop
- Property: No code divides by zero
- Property: There are no fractions with denominators of 0

The properties are *externally visible*; they are what we promise *clients*. In contrast, the invariants are *internal*; they are facts about the *implementation* that help ensure the properties. The code above maintains the invariants and relies on them in certain places to ensure the properties, notably:

- `gcd` will violate the properties if called with an arguments ≤ 0, but since we know denominators are > 0, `reduce` can pass denominators to `gcd` without concern.
- `toString` and most cases of `add` do not need to call `reduce` because they can assume their arguments are already in reduced form.
- `add` uses the property of mathematics that the product of two positive numbers is positive, so we know a non-positive denominator is not introduced.

Unfortunately, under signature `RATIONAL_A`, clients must still be trusted not to break the properties and invariants! Because the signature exposed the definition of the datatype binding, the ML type system will not prevent clients from using the constructors `Frac` and `Whole` directly, bypassing all our work to establish and preserve the invariants. Clients could make "bad" fractions like `Rational.Frac(1,0)`, `Rational.Frac(3,~2)`, or `Rational.Frac(9,6)`, any of which could then end up causing `gcd` or `toString` to misbehave according to our specification. While we may have *intended* for the client only to use `make_frac`, `add`, and `toString`, our signature allows more.

A natural reaction would be to hide the datatype binding by removing the line
`datatype rational = Frac of int * int | Whole of int`. While this is the right intuition, the resulting signature makes no sense and would be rejected: it repeatedly mentions a type `rational` that is not known to exist. What we want to say instead is that there is a type `rational` *but clients cannot know anything about what the type is other than it exists*. In a signature, we can do just that with an *abstract type*, as this signature shows:

```
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 to use this signature instead. That is always true, so we will stop mentioning it.)

This new feature of abstract types, which makes sense only in signatures, is exactly what we want. It lets our module define operations over a type without revealing the implementation of that type. The syntax is just to give a type binding without a definition. The implementation of the module is unchanged; we are simply changing how much information clients have.

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 with a positive denominator.

What `RATIONAL_B` took away from clients compared to `RATIONAL_A` is the constructors `Frac` and `Whole`. So clients cannot create rationals directly and they cannot pattern-match on rationals. They have no idea how they are represented internally. They do not even know `rational` is implemented as a datatype.

Abstract types are a Really Big Deal in programming.

A Cute Twist: Expose the `Whole` function

By making the `rational` type abstract, we took away from clients the `Frac` and `Whole` constructors. While this was crucial for ensuring clients could not create a fraction that was not reduced or had a non-positive denominator, only the `Frac` constructor was problematic. Since allowing clients to create whole numbers directly cannot violate our specification, we could add a function like:

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

to our structure and `val make_whole : int -> rational` to our signature. But this is unnecessary function wrapping; a shorter implementation would be:

```
val make_whole = Whole
```

and of course clients cannot tell which implementation of `make_whole` we are using. But why create a new binding `make_whole` that is just the same thing as `Whole`? Instead, we could just *export the constructor as a function* with this signature and *no changes or additions to our structure*:

```
signature RATIONAL_C =
sig
type rational (* type still abstract *)
exception BadFrac
val Whole : int -> rational (* client knows only that Whole is a function *)
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
```

This signature tells clients there is a function bound to `Whole` that takes an `int` and produces a `rational`. That is correct: this binding is one of the things the datatype binding in the structure creates. So we are exposing *part* of what the datatype binding provides: that `rational` is a type and that `Whole` is bound to a function. We are still hiding the rest of what the datatype binding provides: the `Frac` constructor and pattern-matching with `Frac` and `Whole`.

Rules for Signature Matching

So far, our discussion of whether a structure "should type-check" given a particular signature has been rather informal. Let us now enumerate more precise rules for what it means for a structure to *match* a signature. (This terminology has nothing to do with pattern-matching.) If a structure does not match a signature assigned to it, then the module does not type-check. A structure `Name` matches a signature `BLAH` if:

- For every val-binding in `BLAH`, `Name` must have a binding with that type *or a more general type* (e.g., the implementation can be polymorphic even if the signature says it is not — see below for an example). This binding could be provided via a val-binding, a fun-binding, or a datatype-binding.
- For every non-abstract type-binding in `BLAH`, `Name` must have the same type binding.
- For every abstract type-binding in `BLAH`, `Name` must have some binding that creates that type (either a datatype binding or a type synonym).

Notice that `Name` can have any additional bindings that are not in the signature.

Equivalent Implementations

Given our property- and invariant-preserving signatures `RATIONAL_B` and `RATIONAL_C`, we know clients cannot rely on any helper functions or the actual representation of rationals as defined in the module. So we could replace the *implementation* with any *equivalent implementation* that had the same properties: as long as any call to the `toString` binding in the module produced the same result, clients could never tell. This is another essential software-development task: improving/changing a library in a way that does not break clients. Knowing clients obey an abstraction boundary, as enforced by ML's signatures, is invaluable.

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, which would still match signatures `RATIONAL_A`, `RATIONAL_B`, or `RATIONAL_C`:

```
structure Rational2 :> RATIONAL_A (* or B or C *) =
struct
  datatype rational = Whole of int | Frac of int*int
  exception BadFrac

   fun make_frac (x,y) =
       if y = 0
       then raise BadFrac
       else if y < 0
       then Frac(~x,~y)
       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) =>
                   if x=0
                   then Whole 0
                   else
                       let val d = gcd(abs 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
```

If we give `Rational1` and `Rational2` the signature `RATIONAL_A`, both will type-check, but clients can still distinguish them. For example, `Rational1.toString(Rational1.Frac(21,3))` produces `"21/3"`, but `Rational2.toString(Rational2.Frac(21,3))` produces `"7"`. But if we give `Rational1` and `Rational2` the signature `RATIONAL_B` or `RATIONAL_C`, then the structures are equivalent for any possible client. This is why it is important to use restrictive signatures like `RATIONAL_B` to begin with: so you can change the structure later without checking all the clients.

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 Rational3 :> RATIONAL_B (* or C *)=
struct
   type rational = int*int
   exception BadFrac

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

   fun Whole i = (i,1)

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

   fun toString (x,y) =
       if x=0
       then "0"
       else
           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 (abs 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 `Rational2` 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 takes an `int*int` and return an `int*int`, but clients do not know the actual return type, only the abstract type `rational`. And while giving it an argument type of `rational` in the signature would match, it would make the module useless since clients would not be able to create a value of type `rational`. Nonetheless, clients *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`, which enforces all our invariants.

Our structure *does not* match `RATIONAL_A` since it does not provide `rational` as a datatype with constructors `Frac` and `Whole`.

Our structure *does* match signature `RATIONAL_C` because we explicitly added a function `Whole` of the right type. No client can distinguish our "real function" from the previous structures' use of the `Whole` constructor as a function.

The fact that `fun Whole i = (i,1)` matches `val Whole : int -> rational` is interesting. The type of `Whole` in the module is actually polymorphic: `'a -> 'a * int`. ML signature matching allows `'a -> 'a * int` to match `int -> rational` because `'a -> 'a * int` is more general than `int -> int * int` and `int -> rational` is a correct abstraction of `int -> int * int`. Less formally, the fact that `Whole` has a polymorphic type inside the module does not mean the signature has to give it a polymorphic type outside the module. And in fact it cannot while using the abstract type since `Whole` cannot have the type `'a -> int * int` or `'a -> rational`.

Different modules define different types

While we have defined different structures (e.g., `Rational1`, `Rational2`, and `Rational3`) with the same signature (e.g., `RATIONAL_B`), that does *not* mean that the bindings from the different structures can be used with each other. For example, `Rational1.toString(Rational2.make_frac(2,3))` will not type-check, which is a good thing since it would print an unreduced fraction. The *reason* it does not type-check is that `Rational2.rational` and `Rational1.rational` are *different types*. They were not created by the same datatype binding even though they happen to look identical. Moreover, outside the module we do not *know* they look identical. Indeed, `Rational3.toString(Rational2.make_frac(2,3))` really needs not to type-check since `Rational3.toString` expects an `int*int` but `Rational2.make_frac(2,3))` returns a value made out of the `Rational2.Frac` constructor.

Motivating and Defining Equivalence

The idea that one piece of code is "equivalent" to another piece of code is fundamental to programming and computer science. You are informally thinking about equivalence every time you simplify some code or say, "here's another way to do the same thing." This kind of reasoning comes up in several common scenarios:

- Code maintenance: Can you simplify, clean up, or reorganize code without changing how the rest of the program behaves?
- Backward compatibility: Can you add new features without changing how any of the existing features work?
- Optimization: Can you replace code with a faster or more space-efficient implementation?
- Abstraction: Can an external client tell if I make this change to my code?

Also notice that our use of restrictive signatures in the previous lecture was largely about equivalence: by using a stricter interface, we make more different implementations equivalent because clients cannot tell the difference.

We want a precise definition of equivalence so that we can decide whether certain forms of code maintenance or different implementations of signatures are actually okay. We do not want the definition to be so strict that we cannot make changes to improve code, but we do not want the definition to be so lenient that replacing one function with an "equivalent" one can lead to our program producing a different answer. Hopefully, studying the concepts and theory of equivalence will improve the way you look at software written in any language.

There are many different possible definitions that resolve this strict/lenient tension slightly differently. We will focus on one that is useful and commonly assumed by people who design and implement programming languages. We will also simplify the discussion by assuming that we have two implementations of a function and we want to know if they are equivalent.

The intuition behind our definition is as follows:

- A function `f` is equivalent to a function `g` (or similarly for other pieces of code) if they produce the same answer and have the same side-effects no matter where they are called in any program with any arguments.
- Equivalence does *not* require the same running time, the same use of internal data structures, the same helper functions, etc. All these things are considered "unobservable", i.e., implementation details that do not affect equivalence.

As an example, consider two very different ways of sorting a list. Provided they both produce the same final answer for all inputs, they can still be equivalent no matter how they worked internally or whether one was faster. However, if they behave differently for some lists, perhaps for lists that have repeated elements, then they would not be equivalent.

However, the discussion above was simplified by implicitly assuming the functions always return and have no other effect besides producing their answer. To be more precise, we need that the two functions when given the same argument in the same environment:

1. Produce the same result (if they produce a result)
2. Have the same (non)termination behavior; i.e., if one runs forever the other must run forever
3. Mutate the same (visible-to-clients) memory in the same way.
4. Do the same input/output
5. Raise the same exceptions

These requirements are all important for knowing that if we have two equivalent functions, we could replace one with the other and no use anywhere in the program will behave differently.

Another Benefit of Side-Effect-Free Programming

One easy way to make sure two functions have the same side effects (mutating references, doing input/output, etc.) is to have no side effects at all. This is exactly what functional languages like ML encourage. Yes, in ML you *could* have a function body mutate some global reference or something, but it is generally bad style. Other functional languages are *pure functional languages* meaning there really is no way to do mutation inside (most) functions.

If you "stay functional" by not doing mutation, printing, etc. in function bodies as a matter of policy, then callers can assume lots of equivalences they cannot otherwise. For example, can we replace `(f x)+(f x)` with `(f x)*2`? In general, that can be a wrong thing to do since calling `f` might update some counter or print something. In ML, that's also possible, but far less likely as a matter of style, so we tend to have more things be equivalent. In a purely functional language, we are guaranteed the replacement does not change anything. The general point is that mutation really gets in your way when you try to decide if two pieces of code are equivalent — it is a great reason to avoid mutation.

In addition to being able to remove repeated computations (like `(f x)` above) when maintaining side-effect-free programs, we can also reorder expressions much more freely. For example, in Java, C, etc.:

```
int a = f(x);
int b = g(y);
return b - a;
```

might produce a different result from:

```
return g(y) - f(x);
```

since `f` and `g` can get called in a different order. Again, this is possible in ML too, but if we avoid side-effects, it is much less likely to matter. (We might still have to worry about a different exception getting thrown and other details, however.)

Standard Equivalences

Equivalence is subtle, especially when you are trying to decide if two functions are equivalent without knowing all the places they may be called. Yet this is common, such as when you are writing a library that unknown clients may use. We now consider several situations where equivalence is guaranteed in any situation, so these are good rules of thumb and are good reminders of how functions and closures work.

First, recall the various forms of syntactic sugar we have learned. We can always use or not use syntactic sugar in a function body and get an equivalent function. If we couldn't, then the construct we are using is not actually syntactic sugar. For example, these definitions of `f` are equivalent regardless of what `g` is bound to:

```
fun f x =
    if x
    then g x
    else false
```

```
fun f x =
    x andalso g x
```

Notice though, that we could not necessarily replace `x andalso g x` with `if g x then x else false` if `g` could have side effects or not terminate.

Second, we can change the name of a local variable (or function parameter) provided we change all uses of it consistently. For example, these two definitions of `f` are equivalent:

```
val y = 14                val y = 14
fun f x = x+y+x           fun f z = z+y+z
```

But there is one rule: in choosing a new variable name, you cannot choose a variable that the function body is already using to refer to something else. For example, if we try to replace `x` with `y`, we get `fun y = y+y+y`, which is *not* the same as the function we started with. A previously-unused variable is never a problem.

Third, we can use or not use a helper function. For example, these two definitions of `g` are equivalent:

```
val y = 14                val y = 14
fun g z = (z+y+z)+z       fun f x = x+y+x
                          fun g z = (f z)+z
```

Again, we must take care not to change the meaning of a variable due to `f` and `g` having potentially different environments. For example, here the definitions of `g` are *not* equivalent:

```
val y = 14                val y = 14
val y = 7                 fun f x = x+y+x
fun g z = (z+y+z)+z       val y = 7
                          fun g z = (f z)+z
```

Fourth, as we have explained before with anonymous functions, unnecessary function wrapping is poor style because there is a simpler equivalent way. For example, `fun g y = f y` and `val g = f` are always equivalent. Yet once again, there is a subtle complication. While this works when we have a variable like `f` bound to the function we are calling, in the more general case we might have an *expression* that evaluates to a function that we then call. Are `fun g y = e y` and `val g = e` always the same for any *expression* `e`? No.

As a silly example, consider `fun h() (print "hi" ; fn x => x+x)` and `e` is `h()`. Then `fun g y = (h()) y` is a function that prints every time it is called. But `val g = h()` is a function that does not print — the program will print `"hi"` once when creating the binding for `g`. This should not be mysterious: we know that val-bindings evaluate their right-hand sides "immediately" but function bodies are not evaluated until they are called.

A less silly example might be if `h` might raise an exception rather than returning a function.

Fifth, it is almost the case that `let val p = e1 in e2 end` can be sugar for `(fn p => e2) e1`. After all, for any expressions `e1` and `e2` and pattern `p`, both pieces of code:

- Evaluate `e1` to a value
- Match the value against the pattern `p`
- If it matches, evaluate `e2` to a value in the environment extended by the pattern match
- Return the result of evaluating `e2`

Since the two pieces of code "do" the exact same thing, they must be equivalent. In Racket, this will be the case (with different syntax). In ML, the only difference is the type-checker: The variables in `p` are allowed to have polymorphic types in the let-version, but not in the anonymous-function version.

For example, consider `let val x = (fn y => y) in (x 0, x true) end`. This silly code type-checks and returns `(0,true)` because `x` has type `'a->'a`. But `(fn x => (x 0, x true)) (fn y => y)` does not type-check because there is no non-polymorphic type we can give to `x` and function-arguments cannot have polymorphic types. This is just how type-inference works in ML.

Revisiting our Definition of Equivalence

By design, our definition of equivalence ignores how much time or space a function takes to evaluate. So two functions that always returned the same answer could be equivalent even if one took a nanosecond and another took a million years. In some sense, this is a *good thing* since the definition would allow us to replace the million-year version with the nanosecond version.

But clearly other definitions matter too. Courses in data structures and algorithms study *asymptotic complexity* precisely so that they can distinguish some algorithms as "better" (which clearly implies some "difference") even though the better algorithms are producing the same answers. Moreover, asymptotic complexity, by design, ignores "constant-factor overheads" that might matter in some programs so once again this stricter definition of equivalence may be too lenient: we might actually want to know that two implementations take "about the same amount of time."

None of these definitions are superior. All of them are valuable perspectives computer scientists use all the time. Observable behavior (our definition), asymptotic complexity, and actual performance are all intellectual tools that are used almost every day by someone working on software.