CSE341: Programming Languages Spring 2019
Unit 1 Summary
Dan Grossman, University of Washington

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

Contents

Welcome to Programming Languages **1**
ML Expressions and Variable Bindings **2**
Using `use` **3**
Variables are Immutable **4**
Function Bindings **4**
Pairs and Other Tuples **5**
Lists **6**
Let Expressions **8**
Options **10**
Some Other Expressions and Operators **11**
Lack of Mutation and Benefits Thereof **11**

Welcome to Programming Languages

The top of the course web page has four documents with important information not repeated here. They are the syllabus, the academic-integrity policy, the challenge-program policy, and a description of how this course relates to a course offered on Coursera. *Read these documents thoroughly.*

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

Many people would say this course "teaches" the 3 languages ML, Racket, and Ruby, but that is fairly misleading. We will use these languages to learn various paradigms and concepts because they are well-suited to do so. If our goal were just to make you as productive as possible in these 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 the course, functional programming does some things exactly the opposite of object-oriented programming but also has many similarities. Functional programming is not only a very powerful and elegant approach, but learning it helps you better understand all styles of programming.

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

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

  - Type-checking: type `int` in any static environment
  - Evaluation: to itself in any dynamic environment (it is a value)

- 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 unit 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;
}
```

CSE341: Programming Languages Spring 2019
Unit 2 Summary
Dan Grossman, University of Washington

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

Contents

The Pieces of a Programming Language . . . **1**
Conceptual Ways to Build New Types . . . **2**
Records: Another Approach to "Each-of" Types . . . **3**
By Name vs. By Position, Syntactic Sugar, and The Truth About Tuples . . . **3**
Datatype Bindings: Our Own "One-of" Types . . . **4**
How ML Does *Not* Provide Access to Datatype Values . . . **5**
How ML Provides Access to Datatype Values: Case Expressions . . . **5**
Useful Examples of "One-of" Types . . . **6**
Datatype Bindings and Case Expressions So Far, Precisely . . . **8**
Type Synonyms . . . **8**
Lists and Options are Datatypes . . . **9**
Polymorphic Datatypes . . . **10**
Pattern-Matching for Each-Of Types: The Truth About Val-Bindings . . . **10**
Digression: Type inference . . . **12**
Digression: Polymorphic Types and Equality Types . . . **13**
Nested Patterns . . . **13**
Useful Examples of Nested Patterns . . . **15**
***Optional:* Multiple Cases in a Function Binding** . . . **16**
Exceptions . . . **17**
Tail Recursion and Accumulators . . . **18**
More Examples of Tail Recursion . . . **19**
A Precise Definition of Tail Position . . . **20**

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.

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]

- "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`.

[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.

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.

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

- 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
```

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 now (to be significantly generalized soon), 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 you need to write 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.

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
```

[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."

```
        [] => 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 your homework in the meantime: 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

[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.

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 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
```

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
```

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

```
        | (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
```

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) =
```

[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.

```
    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.
- ...

CSE341: Programming Languages Spring 2019
Unit 3 Summary
Dan Grossman, University of Washington

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

Contents

Introduction and Some Terms **1**
Taking Functions as Arguments **2**
Polymorphic Types and Functions as Arguments **3**
Anonymous functions **4**
Unnecessary Function Wrapping **4**
Maps and filters **5**
Returning functions **6**
Not just for numbers and lists **6**
Lexical Scope **6**
Environments and Closures **7**
(Silly) Examples Including Higher-Order Functions **8**
Why Lexical Scope **8**
Passing Closures to Iterators Like Filter **9**
Fold and More Closure Examples **10**
Another Closure Idiom: Combining Functions **11**
Another Closure Idiom: Currying and Partial Application **12**
The Value Restriction **14**
Mutation via ML References **15**
Another Closure Idiom: Callbacks **15**
Optional: Another Closure Idiom: Abstract Data Types **16**
Optional: Closures in Other Languages **18**
Optional: Closures in Java using Objects and Interfaces **19**
Optional: Closures in C Using Explicit Environments **21**
Standard-Library Documentation **23**

Introduction and Some Terms

This unit 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 unit 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 unit.

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 unit, 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 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 unit, 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 unit. 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.

CSE341: Programming Languages Spring 2019
Unit 4 Summary
Dan Grossman, University of Washington

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

Contents

Modules for Namespace Management 1
Signatures 2
Hiding Things 3
Introducing our extended example 3
Signatures for Our Example 5
A Cute Twist: Expose the `Whole` function 6
Rules for Signature Matching 7
Equivalent Implementations 7
Different modules define different types 10
What is Type Inference? 10
Overview of ML Type Inference 11
More Thorough Examples of ML Type Inference 12
Examples with Polymorphic Types 14
Optional: The Value Restriction 15
Optional: Some Things that Make Type Inference More Difficult 16
Mutual Recursion 16
Motivating and Defining Equivalence 18
Another Benefit of Side-Effect-Free Programming 19
Standard Equivalences 20
Revisiting our Definition of Equivalence 21

Modules for Namespace Management

We start by showing how ML modules can be used to separate bindings into different *namespaces*. We then 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.

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 unit.)
- 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 unit, 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
```

Because of ML's module system, the type-checker does not always know the definition of type synonyms (recall this is a good thing). 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::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
```

[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.

```
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.

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
fun f x = x+y+x
```

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

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

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

CSE341: Programming Languages Spring 2019
Unit 5 Summary
Dan Grossman, University of Washington

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

Contents

Switching from ML to Racket . . . 1
Racket vs. Scheme . . . 2
Getting Started: Definitions, Functions, Lists (and `if`) . . . 2
Syntax and Parentheses . . . 5
Dynamic Typing (and `cond`) . . . 6
Local bindings: `let`, `let*`, `letrec`, local `define` . . . 7
Top-Level Definitions . . . 9
Bindings are Generally Mutable: `set!` Exists . . . 9
The Truth about `cons` . . . 11
Cons cells are immutable, but there is `mcons` . . . 11
Introduction to Delayed Evaluation and Thunks . . . 12
Lazy Evaluation with Delay and Force . . . 13
Streams . . . 14
Memoization . . . 15
Macros: The Key Points . . . 17
Tokenization, Parenthesization, and Scope . . . 18
Defining Macros with `define-syntax` . . . 18
Variables, Macros, and Hygiene . . . 20
More macro examples . . . 22

Switching from ML to Racket

For the next couple weeks, we will use the Racket programming language (instead of ML) and the Dr-Racket programming environment (instead of SML/NJ and Emacs). Notes on installation and basic usage instructions are on the course website in a different document than this one.

Our focus will remain largely on key programming language constructs. We will "switch" to Racket because some of these concepts shine better in Racket. That said, Racket and ML share many similarities: They are both mostly functional languages (i.e., mutation exists but is discouraged) with closures, anonymous functions, convenient support for lists, no return statements, etc. Seeing these features in a second language should help re-enforce the underlying ideas. One moderate difference is that we will not use pattern matching in Racket.

For us, the most important differences between Racket and ML are:

- Racket does not use a static type system. So it accepts more programs and programmers do not need to define new types all the time, but most errors do not occur until run time.
- Racket has a very minimalist and uniform syntax.

Racket has many advanced language features, including macros, a module system quite different from ML, quoting/eval, first-class continuations, contracts, and much more. We will have time to cover only a couple of these topics.

The first few topics cover basic Racket programming since we need to introduce Racket before we start using it to study more advanced concepts. We will do this quickly because (a) we have already seen a similar language and (b) The Racket Guide, `http://docs.racket-lang.org/guide/index.html`, and other documentation at `http://racket-lang.org/` are excellent and free.

Racket vs. Scheme

Racket is derived from Scheme, a well-known programming language that has evolved since 1975. (Scheme in turn is derived from LISP, which has evolved since 1958 or so.) The designers of Racket decided in 2010 that they wanted to make enough changes and additions to Scheme that it made more sense to give the result a new name than just consider it a dialect of Scheme. The two languages remain *very* similar with a short list of key differences (how the empty list is written, whether pairs built by cons are mutable, how modules work), a longer list of minor differences, and a longer list of additions that Racket provides.

Overall, Racket is a modern language under active development that has been used to build several "real" (whatever that means) systems. The improvements over Scheme make it a good choice for this course and for real-world programming. However, it is more of a "moving target" — the designers do not feel as bound to historical precedent as they try to make the language and the accompanying DrRacket system better. So details in the course materials are more likely to become outdated.

Getting Started: Definitions, Functions, Lists (and `if`)

The first line of a Racket file (which is also a Racket module) should be

```
#lang racket
```

This is discussed in the installation/usage instructions for the course. These lecture notes will focus instead on the content of the file after this line. A Racket file contains a collection of definitions.

A definition like

```
(define a 3)
```

extends the top-level environment so that `a` is bound to 3. Racket has very lenient rules on what characters can appear in a variable name, and a common convention is hyphens to separate words like `my-favorite-identifier`.

A subsequent definition like

```
(define b (+ a 2))
```

would bind `b` to 5. In general, if we have `(define x e)` where `x` is a variable and `e` is an expression, we evaluate `e` to a value and change the environment so that `x` is bound to that value. Other than the syntax, this should seem very familiar, although at the end of the lecture we will discuss that, unlike ML, bindings can refer to later bindings in the file. In Racket, *everything* is prefix, such as the addition function used above.

An anonymous function that takes one argument is written `(lambda (x) e)` where the argument is the variable `x` and the body is the expression `e`. So this definition binds a cubing function to `cube1`:

```
(define cube1
  (lambda (x)
    (* x (* x x))))
```

In Racket, different functions really take different numbers of arguments and it is a run-time error to call a function with the wrong number. A three argument function would look like `(lambda (x y z) e)`. However, many functions can take any number of arguments. The multiplication function, `*`, is one of them, so we could have written

```
(define cube2
  (lambda (x)
    (* x x x)))
```

You can consult the Racket documentation to learn how to define your own variable-number-of-arguments functions.

Unlike ML, you can use recursion with anonymous functions because the definition itself is in scope in the function body:

```
(define pow
  (lambda (x y)
    (if (= y 0)
        1
        (* x (pow x (- y 1))))))
```

The above example also used an if-expression, which has the general syntax `(if e1 e2 e3)`. It evaluates `e1`. If the result is `#f` (Racket's constant for false), it evaluates `e3` for the result. If the result is *anything else*, including `#t` (Racket's constant for true), it evaluates `e2` for the result. Notice how this is much more flexible type-wise than anything in ML.

There is a very common form of syntactic sugar you should use for defining functions. It does not use the word `lambda` explicitly:

```
(define (cube3 x)
  (* x x x))
(define (pow x y)
    (if (= y 0)
        1
        (* x (pow x (- y 1)))))
```

This is more like ML's `fun` binding, but in ML `fun` is not just syntactic sugar since it is necessary for recursion.

We can use currying in Racket. After all, Racket's first-class functions are closures like in ML and currying is just a programming idiom.

```
(define pow
  (lambda (x)
    (lambda (y)
      (if (= y 0)
          1
          (* x ((pow x) (- y 1)))))))

(define three-to-the (pow 3))
(define eightyone (three-to-the 4))
(define sixteen ((pow 2) 4))
```

Because Racket's multi-argument functions really are multi-argument functions (not sugar for something else), currying is not as common. There is no syntactic sugar for calling a curried function: we have to write `((pow 2) 4)` because `(pow 2 4)` calls the one-argument function bound to `pow` with two arguments, which is a run-time error. Racket has added sugar for *defining* a curried function. We could have written:

```
(define ((pow x) y)
   (if (= y 0)
       1
       (* x ((pow x) (- y 1)))))
```

This is a fairly new feature and may not be widely known.

Racket has built-in lists, much like ML, and Racket programs probably use lists even more often in practice than ML programs. We will use built-in functions for building lists, extracting parts, and seeing if lists are empty. The function names `car` and `cdr` are a historical accident.

Primitive	Description	Example
`null`	The empty list	`null`
`cons`	Construct a list	`(cons 2 (cons 3 null))`
`car`	Get first element of a list	`(car some-list)`
`cdr`	Get tail of a list	`(cdr some-list)`
`null?`	Return `#t` for the empty-list and `#f` otherwise	`(null? some-value)`

Unlike Scheme, you cannot write `()` for the empty list. You can write `'()`, but we will prefer `null`.

There is also a built-in function `list` for building a list from any number of elements, so you can write `(list 2 3 4)` instead of `(cons 2 (cons 3 (cons 4 null)))`. Lists need not hold elements of the same type, so you can create `(list #t "hi" 14)` without error.

Here are three examples of list-processing functions. `map` and `append` are actually provided by default, so we would not write our own.

```
(define (sum xs)
  (if (null? xs)
      0
      (+ (car xs) (sum (cdr xs)))))

(define (append xs ys)
  (if (null? xs)
      ys
      (cons (car xs) (append (cdr xs) ys))))
```

```
(define (map f xs)
  (if (null? xs)
      null
      (cons (f (car xs)) (map f (cdr xs)))))
```

Syntax and Parentheses

Ignoring a few bells and whistles, Racket has an amazingly simple syntax. Everything in the language is either:

- Some form of *atom*, such as `#t`, `#f`, `34`, `"hi"`, `null`, etc. A particularly important form of atom is an identifier, which can either be a variable (e.g., `x` or `something-like-this!`) or a *special form* such as `define`, `lambda`, `if`, and many more.
- A sequence of things in parentheses `(t1 t2 ... tn)`.

The first thing in a sequence affects what the rest of the sequence means. For example, `(define ...)` means we have a definition and the next thing can be a variable to be defined or a sequence for the sugared version of function definitions.

If the first thing in a sequence is not a special form and the sequence is part of an expression, then we have a function call. Many things in Racket are just functions, such as `+` and `>`.

As a minor note, Racket also allows `[` and `]` in place of `(` and `)` anywhere. As a matter of style, there are a few places we will show where `[...]` is the common preferred option. Racket does *not* allow mismatched parenthesis forms: `(` must be matched by `)` and `[` by `]`. DrRacket makes this easy because if you type `)` to match `[`, it will enter `]` instead.

By "parenthesizing everything" Racket has a syntax that is *unambiguous*. There are never any rules to learn about whether `1+2*3` is `1+(2*3)` or `(1+2)*3` and whether `f x y` is `(f x) y` or `f (x y)`. It makes *parsing*, converting the program text into a tree representing the program structure, trivial. Notice that XML-based languages like HTML take the same approach. In HTML, an "open parenthesis" looks like `<foo>` and the matching close-parenthesis looks like `</foo>`.

For some reason, HTML is only rarely criticized for being littered with parentheses but it is a common complaint leveled against LISP, Scheme, and Racket. If you stop a programmer on the street and ask him or her about these languages, they may well say something about "all those parentheses." This is a bizarre obsession: people who use these languages quickly get used to it and find the uniform syntax pleasant. For example, it makes it very easy for the editor to indent your code properly.

From the standpoint of learning about programming languages and fundamental programming constructs, you should recognize a strong opinion about parentheses (either for or against) as a syntactic prejudice. While everyone is entitled to a personal opinion on syntax, one should not allow it to keep you from learning advanced ideas that Racket does well, like hygienic macros or abstract datatypes in a dynamically typed language or first-class continuations. An analogy would be if a student of European history did not want to learn about the French Revolution because he or she was not attracted to people with french accents.

All that said, practical programming in Racket does require you to get your parentheses correct and Racket differs from ML, Java, C, etc. in an important regard: *Parentheses change the meaning of your program. You cannot add or remove them because you feel like it. They are never optional or meaningless.*

In expressions, `(e)` means evaluate `e` and then call the resulting function with 0 arguments. So `(42)` will be

a run-time error: you are treating the number 42 as a function. Similarly, `((+ 20 22))` is an error for the same reason.

Programmers new to Racket sometimes struggle with remembering that parentheses matter and determining why programs fail, often at run-time, when they are misparenthesized. As an example consider these seven definitions. The first is a correct implementation of factorial and the others are wrong:

```
(define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))   ; 1
(define (fact n) (if (= n 0) (1) (* n (fact (- n 1))))) ; 2
(define (fact n) (if = n 0 1 (* n (fact (- n 1)))))     ; 3
(define fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))   ; 4
(define (fact n) (if (= n 0) 1 (* n fact (- n 1))))     ; 5
(define (fact n) (if (= n 0) 1 (* n ((fact) (- n 1))))) ; 6
(define (fact n) (if (= n 0) 1 (n * (fact (- n 1)))))   ; 7
```

Line	Error
2	calls `1` as a function taking no arguments
3	uses `if` with 5 subexpressions instead of 3
4	bad definition syntax: `(n)` looks like an expression followed by more stuff
5	calls `*` with a function as one of the arguments
6	calls `fact` with 0 arguments
7	treats `n` as a function and calls it with `*`

Dynamic Typing (and `cond`)

Racket does not use a static type system to reject programs before they are run. As an extreme example, the function `(lambda () (1 2))` is a perfectly fine zero-argument function that will cause an error if you ever call it. We will spend significant time in a later lecture comparing dynamic and static typing and their relative benefits, but for now we want to get used to dynamic typing.

As an example, suppose we want to have lists of numbers but where some of the elements can actually be other lists that themselves contain numbers or other lists and so on, any number of levels deep. Racket allows this directly, e.g., `(list 2 (list 4 5) (list (list 1 2) (list 6)) 19 (list 14 0))`. In ML, such an expression would not type-check; we would need to create our own datatype binding and use the correct constructors in the correct places.

Now in Racket suppose we wanted to compute something over such lists. Again this is no problem. For example, here we define a function to sum all the numbers anywhere in such a data structure:

```
(define (sum xs)
  (if (null? xs)
      0
      (if (number? (car xs))
          (+ (car xs) (sum (cdr xs)))
          (+ (sum (car xs)) (sum (cdr xs))))))
```

This code simply uses the built-in *predicates* for empty-lists (`null?`) and numbers (`number?`). The last line assumes `(car xs)` is a list; if it is not, then the function is being misused and we will get a run-time error.

We now digress to introduce the `cond` special form, which is better style for nested conditionals than actually using multiple if-expressions. We can rewrite the previous function as:

```
(define (sum xs)
  (cond [(null? xs) 0]
        [(number? (car xs)) (+ (car xs) (sum (cdr xs)))]
        [#t (+ (sum (car xs)) (sum (cdr xs)))]))
```

A `cond` just has any number of parenthesized pairs of expressions, `[e1 e2]`. The first is a test; if it evaluates to `#f` we skip to the next branch. Otherwise we evaluate `e2` and that is the answer. As a matter of style, your last branch should have the test `#t`, so you do not "fall off the bottom" in which case the result is some sort of "void object" that you do not want to deal with.

As with `if`, the result of a test does not have to be `#t` or `#f`. Anything other than `#f` is interpreted as true for the purpose of the test. It is sometimes bad style to exploit this feature, but it can be useful.

Now let us take dynamic typing one step further and change the specification for our `sum` function. Suppose we even want to allow non-numbers and non-lists in our lists in which case we just want to "ignore" such elements by adding 0 to the sum. If this is what you want (and it may not be — it could silently hide mistakes in your program), then we can do that in Racket. This code will never raise an error:

```
(define (sum xs)
  (cond [(null? xs) 0]
        [(number? xs) xs]
        [(list? xs) (+ (sum (car xs)) (sum (cdr xs)))]
        [#t 0]))
```

Local bindings: `let`, `let*`, `letrec`, local `define`

For all the usual reasons, we need to be able to define local variables inside functions. Like ML, there are expression forms that we can use anywhere to do this. Unlike ML, instead of one construct for local bindings, there are four. This variety is good: Different ones are convenient in different situations and using the most natural one communicates to anyone reading your code something useful about how the local bindings are related to each other. This variety will also help us learn about scope and environments rather than just accepting that there can only be one kind of let-expression with one semantics. How variables are looked up in an environment is a fundamental feature of a programming language.

First, there is the expression of the form

```
(let ([x1 e1]
      [x2 e2]
      ...
      [xn en])
  e)
```

As you might expect, this creates local variables `x1`, `x2`, ... `xn`, bound to the results of evaluating `e1`, `e2`, ..., `en`. and then the body `e` can use these variables (i.e., they are in the environment) and the result of `e` is the overall result. Syntactically, notice the "extra" parentheses around the collection of bindings and the common style of where we use square parentheses.

But the description above left one thing out: What environment do we use to evaluate `e1`, `e2`, ..., `en`? It turns out we use the environment from "*before*" the let-expression. That is, later variables do *not* have earlier ones in their environment. If `e3` uses `x1` or `x2`, that would either be an error or would mean some *outer* variable of the same name. This is *not* how ML let-expressions work. As a silly example, this function doubles its argument:

```
(define (silly-double x)
  (let ([x (+ x 3)]
        [y (+ x 2)])
    (+ x y -5)))
```

This behavior is sometimes useful. For example, to swap the meaning of `x` and `y` in some local scope you can write `(let ([x y][y x]) ...)`. More often, one uses `let` where this semantics versus "each binding has the previous ones in its environment" does not matter: it communicates that the expressions are independent of each other.

If we write `let*` in place of let, then the semantics *does* evaluate each binding's expression in the environment produced from the previous ones. This *is* how ML let-expressions work. It is often convenient: If we only had "regular" `let`, we would have to nest let-expressions inside each other so that each later binding was in the body of the outer let-expressions. (We would have use n nested `let` expressions each with 1 binding instead of 1 `let*` with n bindings.) Here is an example using `let*`:

```
(define (silly-double x)
  (let* ([x (+ x 3)]
         [y (+ x 2)])
    (+ x y -8)))
```

As indicated above, it is common style to use `let` instead of `let*` when this difference in semantics is irrelevant.

Neither `let` nor `let*` allows recursion since the `e1`, `e2`, ..., `en` cannot refer to the binding being defined or any later ones. To do so, we have a third variant `letrec`, which lets us write:

```
(define (triple x)
  (letrec ([y (+ x 2)]
           [f (lambda (z) (+ z y w x))]
           [w (+ x 7)])
    (f -9)))
```

One typically uses `letrec` to define one or more (mutually) recursive functions, such as this very slow method for taking a non-negative number mod 2:

```
(define (mod2 x)
  (letrec
      ([even? (lambda (x) (if (zero? x) #t (odd? (- x 1))))]
       [odd?  (lambda (x) (if (zero? x) #f (even? (- x 1))))])
    (if (even? x) 0 1)))
```

Alternately, you can get the same behavior as `letrec` by using local defines, which is very common in real Racket code and is in fact the preferred style over let-expressions. In this course, you can use it if you like but do not have to. There are restrictions on where local defines can appear; at the beginning of a function body is one common place where they are allowed.

```
(define (mod2_b x)
  (define even? (lambda(x) (if (zero? x) #t (odd? (- x 1)))))
  (define odd?  (lambda(x) (if (zero? x) #f (even? (- x 1)))))
  (if (even? x) 0 1))
```

We need to be careful with `letrec` and local definitions: They allow code to refer to variables that are initialized *later*, but the expressions for each binding are still evaluated in order.

For mutually recursive functions, this is never a problem: In the examples above, the definition of `even?` refers to the definition of `odd?` even though the expression bound to `odd?` has not yet been evaluated. This is okay because the use in `even?` is in a function body, so it will not be *used* until after `odd?` has been initialized. In contrast, this use of `letrec` is bad:

```
(define (bad-letrec x)
  (letrec ([y z]
           [z 13])
    (if x y z)))
```

The semantics for `letrec` requires that the use of `z` for initializing `y` refers to the `z` in the `letrec`, but the expression for `z` (the `13`) has not been evaluated yet. In this situation, Racket will raise an error when `bad-letrec` is called. (Prior to Racket Version 6.1, it would instead bind `y` to a special "undefined" object, which almost always just had the effect of hiding a bug.)

For this class, you can decide whether to use local defines or not. The lecture materials generally will not, choosing instead whichever of `let`, `let*`, or `letrec` is most convenient and communicates best. But you are welcome to use local defines, with those "next to each other" behaving like `letrec` bindings.

Top-Level Definitions

A Racket file is a module with a sequence of definitions. Just as with let-expressions, it matters greatly to the semantics what environment is used for what definitions. In ML, a file was like an implicit `let*`. In Racket, it is basically like an implicit `letrec`. This is convenient because it lets you order your functions however you like in a module. For example, you do not need to place mutually recursive functions next to each other or use special syntax. On the other hand, there are some new "gotchas" to be aware of:

- You cannot have two bindings use the same variable. This makes no sense: which one would a use of the variable use? With `letrec`-like semantics, we do *not* have one variable shadow another one if they are defined in the same collection of mutually-recursive bindings.
- If an earlier binding uses a later one, it needs to do so in a function body so that the later binding is initialized by the time of the use. In Racket, the "bad" situation of using an uninitialized value causes an error when you use the module (e.g., when you click "Run" for the file in DrRacket).
- So *within* a module/file, there is no top-level shadowing (you can still shadow within a definition or let-expressions), but one module can shadow a binding in another file, such as the files implicitly included from Racket's standard library. For example, although it would be bad style, we could shadow the built-in `list` function with our own. Our own function could even be recursive and call itself like any other recursive function. *However*, the behavior in the REPL is different, so do not shadow a function with your own recursive function definition in the REPL. Defining the recursive function in the Definitions Window and using it in the REPL still works as expected.

Bindings are Generally Mutable: `set!` Exists

While Racket encourages a functional-programming style with liberal use of closures and avoiding side effects, the truth is it has assignment statements. If `x` is in your environment, then `(set! x 13)` will *mutate* the

binding so that `x` now maps to the value 13. Doing so affects all code that has this `x` in its environment. Pronounced "set-bang," the exclamation point is a convention to alert readers of your code that side effects are occurring that may affect other code. Here is an example:

```
(define b 3)
(define f (lambda (x) (* 1 (+ x b))))
(define c (+ b 4))
(set! b 5)
(define z (f 4))
(define w c)
```

After evaluating this program, `z` is bound to 9 because the body of the function bound to `f` will, when evaluated, look up `b` and find `5`. However, `w` is bound to `7` because when we evaluated `(define c (+ b 4))`, we found `b` was `3` and, as usual, the result is to bind `c` to `7` regardless of how we got the `7`. So when we evaluate `(define w c)`, we get `7`; it is irrelevant that `b` has changed.

You can also use `set!` for local variables and the same sort of reasoning applies: you have to think about *when* you look up a variable to determine what value you get. But programmers used to languages with assignment statements are all too used to that.

Mutating top-level bindings is particularly worrisome because we may not know all the code that is using the definition. For example, our function `f` above uses `b` and could act strangely, even fail mysteriously, if `b` is mutated to hold an unexpected value. If `f` needed to defend against this possibility it would need to avoid using `b` after `b` might change. There is a general technique in software development you should know: *If something might get mutated and you need the old value, make a copy before the mutation can occur.* In Racket, we could code this up easily enough:

```
(define f
   (let ([b b])
      (lambda (x) (* 1 (+ x b)))))
```

This code makes the `b` in the function body refer to a local `b` that is initialized to the global `b`.

But is this as defensive as we need to be? Since `*` and `+` are just variables bound to functions, we might want to defend against them being mutated later as well:

```
(define f
   (let ([b b]
         [+ +]
         [* *])
      (lambda (x) (* 1 (+ x b)))))
```

Matters would get worse if `f` used other helper functions: Making local copies of variables bound to the functions would not be enough unless those functions made copies of all their helper functions as well.

Fortunately, none of this is necessary in Racket due to a reasonable compromise: A top-level binding is not mutable unless the module that defined it contains a `set!` for it. So if the file containing `(define b 4)` did not have a `set!` that changed it, then we can rest assured that no other file will be allowed to use `set!` on that binding (it will cause an error). And all the predefined functions like `+` and `*` are in a module that does not use `set!` for them, so they also cannot be mutated. (In Scheme, all top-level bindings really are mutable, but programmers typically just assume they won't be mutated since it is too painful to assume otherwise.)

So the previous discussion is *not* something that will affect most of your Racket programming, but it is useful to understand what `set!` means and how to defend against mutation by making a copy. The point is that the possibility of mutation, which Racket often avoids, makes it very difficult to write correct code.

The Truth about `cons`

So far, we have used `cons`, `null`, `car`, `cdr`, and `null?` to create and access lists. For example, `(cons 14 (cons #t null))` makes the list `'(14 #t)` where the quote-character shows this is printing a list value, not indicating an (erroneous) function call to 14.

But the truth is *cons just makes a pair* where you get the first part with `car` and the second part with `cdr`. Such pairs are often called *cons cells* in languages like Racket. So we can write `(cons (+ 7 7) #t)` to produce the pair `'(14 . #t)` where the period shows that this is *not* a list. A list is, by convention and according to the `list?` predefined function, either `null` or a pair where the cdr (i.e., second component) is a list. A cons cell that is not a list is often called an *improper list*, especially if it has nested cons cells in the second position, e.g., `(cons 1 (cons 2 (cons 3 4)))` where the result prints as `'(1 2 3 . 4)`.

Most list functions like `length` will give a run-time error if passed an improper list. On the other hand, the built-in `pair?` primitive returns true for anything built with `cons`, i.e., any improper or proper list *except* the empty list.

What are improper lists good for? The real point is that pairs are a generally useful way to build an each-of type, i.e., something with multiple pieces. And in a dynamically typed language, all you need for lists are pairs and some way to recognize the end of the list, which by convention Racket uses the `null` constant (which prints as `'()`) for. As a matter of style, you should use proper lists and not improper lists for collections that could have any number of elements.

Cons cells are immutable, but there is `mcons`

Cons cells are immutable: When you create a cons cell, its two fields are initialized and will never change. (This is a major difference between Racket and Scheme.) Hence we can continue to enjoy the benefits of knowing that cons cells cannot be mutated by other code in our program. It has another somewhat subtle advantage: The Racket implementation can be clever enough to make `list?` a constant-time operation since it can store with every cons cell whether or not it is a proper list when the cons cell is created. This cannot work if cons cells are mutable because a mutation far down the list could turn it into an improper list.

It is a bit subtle to realize that cons cells really are immutable even though we have `set!`. Consider this code:

```
(define x (cons 14 null))
(define y x)
(set! x (cons 42 null))
(define fourteen (car y))
```

The `set!` of `x` changes the contents of the binding of `x` to be a different pair; it does not alter the contents of the old pair that `x` referred to. You might try to do something like `(set! (car x) 27)`, but this is a syntax error: `set!` requires a variable to assign to, not some other kind of location.

If we want mutable pairs, though, Racket is happy to oblige with a different set of primitives:

- `mcons` makes a mutable pair

- `mcar` returns the first component of a mutable pair
- `mcdr` returns the second component of a mutable pair
- `mpair?` returns `#t` if given a mutable pair
- `set-mcar!` takes a mutable pair and an expression and changes the first component to be the result of the expression
- `set-mcdr!` takes a mutable pair and an expression and changes the second component to be the result of the expression

Since some of the powerful idioms we will study next use mutation to store previously computed results, we will find mutable pairs useful.

Introduction to Delayed Evaluation and Thunks

A key semantic issue for a language construct is *when are its subexpressions evaluated*. For example, in Racket (and similarly in ML and most but not all programming languages), given `(e1 e2 ... en)` we evaluate the function arguments `e2`, ..., `en` once before we execute the function body and given a function `(lambda (...) ...)` we do not evaluate the body until the function is called. We can contrast this rule ("evaluate arguments in advance") with how `(if e1 e2 e3)` works: we do *not* evaluate both `e2` and `e3`. This is why:

```
(define (my-if-bad x y z) (if x y z))
```

is a function that *cannot* be used wherever you use an if-expression; the rules for evaluating subexpressions are fundamentally different. For example, this function would never terminate since every call makes a recursive call:

```
(define (factorial-wrong x)
  (my-if-bad (= x 0)
             1
             (* x (factorial-wrong (- x 1)))))
```

However, we can use the fact that function bodies are not evaluated until the function gets called to make a more useful version of an "if function":

```
(define (my-if x y z) (if x (y) (z)))
```

Now wherever we would write `(if e1 e2 e3)` we could instead write `(my-if e1 (lambda () e2) (lambda () e3))`. The body of `my-if` either calls the zero-argument function bound to `y` or the zero-argument function bound to `z`. So this function is correct (for non-negative arguments):

```
(define (factorial x)
  (my-if (= x 0)
         (lambda () 1)
         (lambda () (* x (factorial (- x 1))))))
```

Though there is certainly no reason to wrap Racket's "if" in this way, the general idiom of using a zero-argument function to *delay evaluation* (do not evaluate the expression now, do it later when/if the zero-argument function is called) is very powerful. As convenient terminology/jargon, when we use a zero-argument function to delay evaluation we call the function a *thunk*. You can even say, "thunk the argument" to mean "use `(lambda () e)` instead of `e`".

Using thunks is a powerful programming idiom. It is not specific to Racket — we could have studied such programming just as well in ML.

Lazy Evaluation with Delay and Force

Suppose we have a large computation that we know how to perform but we do not know if we need to perform it. Other parts of the program know where the result of the computation is needed and there may be 0, 1, or more different places. If we thunk, then we may repeat the large computation many times. But if we do not thunk, then we will perform the large computation even if we do not need to. To get the "best of both worlds," we can use a programming idiom known by a few different (and perhaps technically slightly different) names: lazy-evaluation, call-by-need, promises. The idea is to use mutation to remember the result from the first time we use the thunk so that we do not need to use the thunk again.

One simple implementation in Racket would be:

```
(define (my-delay f)
  (mcons #f f))

(define (my-force th)
  (if (mcar th)
      (mcdr th)
      (begin (set-mcar! th #t)
             (set-mcdr! th ((mcdr th)))
             (mcdr th))))
```

We can create a thunk `f` and pass it to `my-delay`. This returns a pair where the first field indicates we have not used the thunk yet. Then `my-force`, if it sees the thunk has not been used yet, uses it and then uses mutation to change the pair to hold the result of using the thunk. That way, any future calls to `my-force` with the same pair will not repeat the computation. Ironically, while we are using mutation in our *implementation*, this idiom is quite error-prone if the thunk passed to `my-delay` has side effects or relies on mutable data, since those effects will occur at most once and it may be difficult to determine when the first call to `my-force` will occur.

Consider this silly example where we want to multiply the result of two expressions `e1` and `e2` using a recursive algorithm (of course you would really just use `*` and this algorithm does not work if `e1` produces a negative number):

```
(define (my-mult x y)
 (cond [(= x 0) 0]
       [(= x 1) y]
       [#t (+ y (my-mult (- x 1) y))]))
```

Now calling `(my-mult e1 e2)` evaluates `e1` and `e2` once each and then does 0 or more additions. But what if `e1` evaluates to 0 and `e2` takes a long time to compute? Then evaluating `e2` was wasteful. So we could thunk it:

```
(define (my-mult x y-thunk)
 (cond [(= x 0) 0]
       [(= x 1) (y-thunk)]
       [#t (+ (y-thunk) (my-mult (- x 1) y-thunk))]))
```

Now we would call `(my-mult e1 (lambda () e2))`. This works great if `e1` evaluates to 0, fine if `e1` evaluates to 1, and terribly if `e1` evaluates to a large number. After all, now we evaluate `e2` on every recursive call. So let's use `my-delay` and `my-force` to get the best of both worlds:

```
(my-mult e1 (let ([x (my-delay (lambda () e2))]) (lambda () (my-force x))))
```

Notice we create the delayed computation once before calling `my-mult`, then the first time the thunk passed to `my-mult` is called, `my-force` will evaluate `e2` and remember the result for future calls to `my-force x`. An alternate approach that might look simpler is to rewrite `my-mult` to expect a result from `my-delay` rather than an arbitrary thunk:

```
(define (my-mult x y-promise)
 (cond [(= x 0) 0]
       [(= x 1) (my-force y-promise)]
       [#t (+ (my-force y-promise) (my-mult (- x 1) y-promise))]))

(my-mult e1 (my-delay (lambda () e2)))
```

Some languages, most notably Haskell, use this approach for all function calls, i.e., the semantics for function calls is different in these languages: If an argument is never used it is never evaluated, else it is evaluated only once. This is called *call-by-need* whereas all the languages we will use are *call-by-value* (arguments are fully evaluated before the call is made).

Streams

A stream is an infinite sequence of values. We obviously cannot create such a sequence explicitly (it would literally take forever), but we can create code that knows how to produce the infinite sequence and other code that knows how to ask for however much of the sequence it needs.

Streams are very common in computer science. You can view the sequence of bits produced by a synchronous circuit as a stream, one value for each clock cycle. The circuit does not know how long it should run, but it can produce new values forever. The UNIX pipe (`cmd1 | cmd2`) is a stream; it causes `cmd1` to produce only as much output as `cmd2` needs for input. Web programs that react to things users click on web pages can treat the user's activities as a stream — not knowing when the next will arrive or how many there are, but ready to respond appropriately. More generally, streams can be a convenient division of labor: one part of the software knows how to produce successive values in the infinite sequence but does not know how many will be needed and/or what to do with them. Another part can determine how many are needed but does not know how to generate them.

There are many ways to code up streams; we will take the simple approach of representing a stream as a thunk that when called produces a pair of (1) the first element in the sequence and (2) a thunk that represents the stream for the second-through-infinity elements. Defining such thunks typically uses recursion. Here are three examples:

```
(define ones (lambda () (cons 1 ones)))
```

```
(define nats
  (letrec ([f (lambda (x) (cons x (lambda () (f (+ x 1)))))])
    (lambda () (f 1))))
(define powers-of-two
  (letrec ([f (lambda (x) (cons x (lambda () (f (* x 2)))))])
    (lambda () (f 2))))
```

Given this encoding of streams and a stream `s`, we would get the first element via `(car (s))`, the second element via `(car ((cdr (s))))`, the third element via `(car ((cdr ((cdr (s))))))`, etc. Remember parentheses matter: `(e)` calls the thunk `e`.

We could write a higher-order function that takes a stream and a predicate-function and returns how many stream elements are produced before the predicate-function returns true:

```
(define (number-until stream tester)
  (letrec ([f (lambda (stream ans)
                (let ([pr (stream)])
                  (if (tester (car pr))
                      ans
                      (f (cdr pr) (+ ans 1)))))])
    (f stream 1)))
```

As an example, `(number-until powers-of-two (lambda (x) (= x 16)))` evaluates to 4.

As a side-note, all the streams above can produce their next element given at most their previous element. So we could use a higher-order function to abstract out the common aspects of these functions, which lets us put the stream-creation logic in one place and the details for the particular streams in another. This is just another example of using higher-order functions to reuse common functionality:

```
(define (stream-maker fn arg)
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (fn x arg)))))])
    (lambda () (f arg))))
(define ones (stream-maker (lambda (x y) 1) 1))
(define nats  (stream-maker + 1))
(define powers-of-two (stream-maker * 2))
```

Memoization

An idiom related to lazy evaluation that does not actually use thunks is *memoization*. If a function does not have side-effects, then if we call it multiple times with the same argument(s), we do not actually have to do the call more than once. Instead, we can look up what the answer was the first time we called the function with the argument(s).

Whether this is a good idea or not depends on trade-offs. Keeping old answers in a table takes space and table lookups do take some time, but compared to reperforming expensive computations, it can be a big win. Again, for this technique to even be *correct* requires that given the same arguments a function will always return the same result and have no side-effects. So being able to use this *memo table* (i.e., do memoization) is yet another advantage of avoiding mutation.

To implement memoization we do use mutation: Whenever the function is called with an argument we have not seen before, we compute the answer and then add the result to the table (via mutation).

As an example, let's consider 3 versions of a function that takes an `x` and returns fibonacci(x). (A Fibonacci number is a well-known definition that is useful in modeling populations and such.) A simple recursive definition is:

```
(define (fibonacci x)
  (if (or (= x 1) (= x 2))
      1
      (+ (fibonacci (- x 1))
         (fibonacci (- x 2)))))
```

Unfortunately, this function takes exponential time to run. We might start noticing a pause for `(fibonaccci 30)`, and `(fibonacci 40)` takes a thousand times longer than that, and `(fibonacci 10000)` would take more seconds than there are particles in the universe. Now, we could fix this by taking a "count up" approach that remembers previous answers:

```
(define (fibonacci x)
  (letrec ([f (lambda (acc1 acc2 y)
                (if (= y x)
                    (+ acc1 acc2)
                    (f (+ acc1 acc2) acc1 (+ y 1))))])
    (if (or (= x 1) (= x 2))
        1
        (f 1 1 3))))
```

This takes linear time, so `(fibonacci 10000)` returns almost immediately (and with a very large number), but it required a quite different approach to the problem. With memoization we can turn `fibonacci` into an efficient algorithm with a technique that works for lots of algorithms. It is closely related to "dynamic programming," which you often learn about in advanced algorithms courses. Here is the version that does this memoization (the `assoc` library function is described below):

```
(define fibonacci
  (letrec([memo null]
          [f (lambda (x)
               (let ([ans (assoc x memo)])
                 (if ans
                     (cdr ans)
                     (let ([new-ans (if (or (= x 1) (= x 2))
                                        1
                                        (+ (f (- x 1))
                                           (f (- x 2))))])
                       (begin
                         (set! memo (cons (cons x new-ans) memo))
                         new-ans)))))])
    f))
```

It is essential that different calls to `f` use the *same* mutable memo-table: if we create the table inside the call to `f`, then each call will use a new empty table, which is pointless. But we do not put the table at top-level just because that would be bad style since its existence should be known only to the implementation of `fibonacci`.

Why does this technique work to make `(fibonacci 10000)` complete quickly? Because when we evaluate `(f (- x 2))` on any recursive calls, the result is already in the table, so there is no longer an exponential

number of recursive calls. This is much more important than the fact that calling `(fibonacci 10000)` a second time will complete even more quickly since the answer will be in the memo-table.

For a large table, using a list and Racket's `assoc` function may be a poor choice, but it is fine for demonstrating the concept of memoization. `assoc` is just a library function in Racket that takes a value and a list of pairs and returns the first pair in the list where the car of the pair equal to the value. It returns `#f` if no pair has such a car. (The reason `assoc` returns the pair rather than the cdr of the pair is so you can distinguish the case where no pair matches from the case where the pair that matches has `#f` in its cdr. This is the sort of thing we would use an option for in ML.)

Macros: The Key Points

The last topic in this unit is *macros*, which add to the *syntax* of a language by letting programmers define their own syntactic sugar. We start with the key ideas before learning how to define macros in Racket.

A *macro definition* introduces some new syntax into the language. It describes how to transform the new syntax into different syntax in the language itself. A *macro system* is a language (or part of a larger languages) for defining macros. A *macro use* is just using one of the macros previously defined. The semantics of a macro use is to replace the macro use with the appropriate syntax as defined by the macro definition. This process is often called *macro expansion* because it is common but not required that the syntactic transformation produces a larger amount of code.

The key point is that macro expansion happens *before* anything else we have learned about: before type-checking, before compiling, before evaluation. Think of "expanding all the macros" as a pre-pass over your entire program before anything else occurs. So macros get expanded everywhere, such as in function bodies, both branches of conditionals, etc.

Here are 3 examples of macros one might define in Racket:

- A macro so that programmers can write `(my-if e1 then e2 else e3)` where `my-if`, `then`, and `else` are keywords and this macro-expands to `(if e1 e2 e3)`.
- A macro so that programmers can write `(comment-out e1 e2)` and have it transform to `e2`, i.e., it is a convenient way to take an expression `e1` out of the program (replacing it with `e2`) without actually deleting anything.
- A macro so that programmers can write `(my-delay e)` and have it transform to `(mcons #f (lambda () e))`. This is different from the `my-delay` *function* we defined earlier because the function required the caller to pass in a thunk. Here the macro expansion does the thunk creation and the macro user should *not* include an explicit thunk.

Racket has an excellent and sophisticated macro system. For precise, technical reasons, its macro system is superior to many of the better known macro systems, notably the preprocessor in C or C++. So we can use Racket to learn some of the pitfalls of macros in general. Below we discuss:

- How macro systems must handle issues of tokenization, parenthesization, and scope — and how Racket handles parenthesization and scope better than C/C++
- How to define macros in Racket, such as the ones described above
- How macro definitions should be careful about the order expressions are evaluated and how many times they are evaluated
- The key issue of variable bindings in macros and the notion of *hygiene*

Tokenization, Parenthesization, and Scope

The definition of macros and macro expansion is more structured and subtle than "find-and-replace" like one might do in a text editor or with a script you write manually to perform some string substitution in your program. Macro expansion is different in roughly three ways.

First, consider a macro that, "replaces every use of `head` with `car`." In macro systems, that does *not* mean some variable `headt` would be rewritten as `cart`. So the implementation of macros has to at least understand how a programming language's text is broken into *tokens* (i.e., words). This notion of tokens is different in different languages. For example, `a-b` would be three tokens in most languages (a variable, a subtraction, and another variable), but is one token in Racket.

Second, we can ask if macros do or do not understand parenthesization. For example, in C/C++, if you have a macro

```
#define ADD(x,y) x+y
```

then `ADD(1,2/3)*4` gets rewritten as `1 + 2 / 3 * 4`, which is *not* the same thing as `(1 + 2/3)*4`. So in such languages, macro writers generally include lots of explicit parentheses in their macro definitions, e.g.,

```
#define ADD(x,y) ((x)+(y))
```

In Racket, macro expansion preserves the code structure so this issue is not a problem. A Racket macro use always looks like `(x ...)` where `x` is the name of a macro and the result of the expansion "stays in the same parentheses" (e.g., `(my-if x then y else z)` might expand to `(if x y z)`). This is an advantage of Racket's minimal and consistent syntax.

Third, we can ask if macro expansion happens even when creating variable bindings. If not, then local variables can shadow macros, which is probably what you want. For example, suppose we have:

```
(let  ([hd 0] [car 1]) hd) ; evaluates to 0
(let* ([hd 0] [car 1]) hd) ; evaluates to 0
```

If we replace `car` with `hd`, then the first expression is an error (trying to bind `hd` twice) and the second expression now evaluates to 1. In Racket, macro expansion does not apply to variable definitions, i.e., the `car` above is different and shadows any macro for `car` that happens to be in scope.

Defining Macros with `define-syntax`

Let's now walk through the syntax we will use to define macros in Racket. (There have been many variations in Racket's predecessor Scheme over the years; this is one modern approach we will use.) Here is a macro that lets users write `(my-if e1 then e2 else e3)` for any expressions `e1`, `e2`, and `e3` and have it mean exactly `(if e1 e2 e3)`:

```
(define-syntax my-if
  (syntax-rules (then else)
    [(my-if e1 then e2 else e3)
     (if e1 e2 e3)]))
```

- `define-syntax` is the special form for defining a macro.

- `my-if` is the name of our macro. It adds `my-if` to the environment so that expressions of the form `(my-if ...)` will be macro-expanded according to the syntax rules in the rest of the macro definition.
- `syntax-rules` is a keyword.
- The next parenthesized list (in this case `(then else)`) is a list of "keywords" for this macro, i.e., any use of `then` or `else` in the body of `my-if` is just syntax whereas anything not in this list (and not `my-if` itself) represents an arbitrary expression.
- The rest is a list of pairs: how `my-if` might be used and how it should be rewritten if it is used that way.
- In this example, our list has only one option: `my-if` must be used in an expression of the form `(my-if e1 then e2 else e3)` and that becomes `(if e1 e2 e3)`. Otherwise an error results. Note the rewriting occurs *before* any evaluation of the expressions `e1`, `e2`, or `e3`, unlike with functions. This is what we want for a conditional expression like `my-if`.

Here is a second simple example where we use a macro to "comment out" an expression. We use `(comment-out e1 e2)` to be rewritten as `e2`, meaning `e1` will never be evaluated. This might be more convenient when debugging code than actually using comments.

```
(define-syntax comment-out
  (syntax-rules ()
    [(comment-out e1 e2) e2]))
```

Our third example is a macro `my-delay` so that, unlike the `my-delay` function defined earlier, users would write `(my-delay e)` to create a promise such that `my-force` would evaluate `e` and remember the result, rather than users writing `(my-delay (lambda () e))`. Only a macro, not a function, can "delay evaluation by adding a thunk" like this because function calls always evaluate their arguments.

```
(define-syntax my-delay
  (syntax-rules ()
    [(my-delay e)
     (mcons #f (lambda () e))]))
```

We should *not* create a macro version of `my-force` because our function version from earlier is just what we want. Give `(my-force e)` we *do* want to evaluate `e` to a value, which should be an mcons-cell created by `my-delay` and then perform the computation in the `my-force` function. Defining a macro provides no benefit and can be error prone. Consider this awful attempt:

```
(define-syntax my-force
  (syntax-rules ()
     [(my-force e)
      (if (mcar e)
          (mcdr e)
          (begin (set-mcar! e #t)
                 (set-mcdr! e ((mcdr e)))
                 (mcdr e)))]))
```

Due to macro expansion, uses of this macro will end up evaluating their argument *multiple times*, which can have strange behavior if `e` has side effects. Macro users will not expect this. In code like:

```
(let ([t (my-delay some-complicated-expression)])
  (my-force t))
```

this does not matter since `t` is already bound to a value, but in code like:

```
(my-force (begin (print "hi") (my-delay some-complicated-expression)))
```

we end up printing multiple times. Remember that macro expansion copies the entire argument `e` everywhere it appears in the macro definition, but we often want it to be evaluated only once. This version of the macro does the right thing in this regard:

```
(define-syntax my-force
  (syntax-rules ()
    [(my-force e)
     (let ([x e])
       (if (mcar x)
           (mcdr x)
           (begin (set-mcar! x #t)
                  (set-mcdr! x ((mcdr x)))
                  (mcdr x))))]))
```

But, again, there is *no reason* to define a macro like this since a function does exactly what we need. Just stick with:

```
(define (my-force th)
  (if (mcar th)
      (mcdr th)
      (begin (set-mcar! th #t)
             (set-mcdr! th ((mcdr th)))
             (mcdr th))))
```

Variables, Macros, and Hygiene

Let's consider a macro that doubles its argument. Note this is poor style because if you want to double an argument you should just write a function: `(define (double x) (* 2 x))` or `(define (double x) (+ x x))` which are equivalent to each other. But this short example will let us investigate when macro arguments are evaluated and in what environment, so we will use it just as a poor-style example.

These two macros are *not* equivalent:

```
(define-syntax double1
  (syntax-rules ()
         [(double1 e)
          (* 2 e)]))
(define-syntax double2
  (syntax-rules ()
    [(double2 e)
     (+ e e)]))
```

The reason is `double2` will evaluate its argument twice. So `(double1 (begin (print "hi") 17))` prints `"hi"` once but `(double2 (begin (print "hi") 17))` prints `"hi"` twice. The function versions print `"hi"` once, simply because, as always, function arguments are evaluated to values before the function is called.

To fix `double2` without "changing the algorithm" to multiplication instead of addition, we should use a local variable:

```
(define-syntax double3
  (syntax-rules ()
    [(double3 e)
     (let ([x e])
       (+ x x))]))
```

Using local variables in macro definitions to control if/when expressions get evaluated is exactly what you should do, but in less powerful macro languages (again, C/C++ is an easy target for derision here), local variables in macros are typically avoided. The reason has to do with scope and something that is called ***hygiene***. For sake of example, consider this silly variant of `double3`:

```
(define-syntax double4
  (syntax-rules ()
    [(double4 e)
     (let* ([zero 0]
            [x e])
       (+ x x zero))]))
```

In Racket, this macro always works as expected, but that may/should surprise you. After all, suppose I have this use of it:

```
(let ([zero 17])
   (double4 zero))
```

If you do the syntactic rewriting as expected, you will end up with

```
(let ([zero 17])
  (let* ([zero 0]
         [x zero])
       (+ x x zero)))
```

But this expression evaluates to 0, not to 34. The problem is a *free variable* at the macro-use (the `zero` in `(double4 zero)`) ended up in the scope of a local variable in the macro definition. That is why in C/C++, local variables in macro definitions tend to have funny names like `__x_hopefully_no_conflict` in the hope that this sort of thing will not occur. In Racket, the rule for macro expansion is more sophisticated to avoid this problem. Basically, every time a macro is used, all of its local variables are *rewritten* to be fresh new variable names that do not conflict with anything else in the program. This is "one half" of what by definition make Racket macros hygienic.

The other half has to do with free variables in the *macro definition* and making sure they do not wrongly end up in the scope of some local variable where the macro is used. For example, consider this strange code that uses `double3`:

```
(let ([+ *])
   (double3 17))
```

The naive rewriting would produce:

```
(let ([+ *])
  (let ([x 17])
     (+ 17 17)))
```

Yet this produces 17^2, not 34. Again, the naive rewriting is *not* what Racket does. Free variables in a macro definition always refer to what was in the environment where the macro was defined, not where the macro was used. This makes it much easier to write macros that always work as expected. Again macros in C/C++ work like the naive rewriting.

There are situations where you do not want hygiene. For example, suppose you wanted a macro for for-loops where the macro user specified a variable that would hold the loop-index and the macro definer made sure that variable held the correct value on each loop iteration. Racket's macro system has a way to do this, which involves explicitly violating hygiene, but we will not demonstrate it here.

More macro examples

Finally, let's consider a few more useful macro definitions, including ones that use multiple cases for how to do the rewriting. First, here is a macro that lets you write up to two let-bindings using `let*` semantics but with fewer parentheses:

```
(define-syntax let2
  (syntax-rules ()
    [(let2 () body)
     body]
    [(let2 (var val) body)
     (let ([var val]) body)]
    [(let2 (var1 val1 var2 val2) body)
     (let ([var1 val1])
       (let ([var2 val2])
         body))]))
```

As examples, `(let2 () 4)` evaluates to 4, `(let2 (x 5) (+ x 4)` evaluates to 9, and `(let2 (x 5 y 6) (+ x y))` evaluates to 11.

In fact, given support for recursive macros, we could redefine Racket's `let*` entirely in terms of `let`. We need some way to talk about "the rest of a list of syntax" and Racket's `...` gives us this:

```
(define-syntax my-let*
  (syntax-rules ()
    [(my-let* () body)
     body]
    [(my-let* ([var0 val0]
               [var-rest val-rest] ...)
              body)
     (let ([var0 val0])
       (my-let* ([var-rest val-rest] ...)
                body))]))
```

Since macros are recursive, there is nothing to prevent you from generating an infinite loop or an infinite amount of syntax during macro expansion, i.e., before the code runs. The example above does not do this because it recurs on a shorter list of bindings.

Finally, here is a macro for a limited form of for-loop that executes its body $hi - lo$ times. (It is limited because the body is not given the current iteration number.) Notice the use of a let expression to ensure we evaluate `lo` and `hi` exactly once but we evaluate `body` the correct number of times.

```
(define-syntax for
  (syntax-rules (to do)
    [(for lo to hi do body)
     (let ([l lo]
           [h hi])
       (letrec ([loop (lambda (it)
                        (if (> it h)
                            #t
                            (begin body (loop (+ it 1)))))])
         (loop l)))]))
```

CSE341: Programming Languages Spring 2019
Unit 6 Summary
Dan Grossman, University of Washington

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

Contents

Datatype-Programming Without Datatypes 1
Changing How We Evaluate Our Arithmetic Expression Datatype 2
Recursive Datatypes Via Racket Lists 3
Recursive Datatypes Via Racket's `struct` 4
Why the `struct` Approach is Better 6
Implementing a Programming Language in General 7
Implementing a Programming Language Inside Another Language 8
Assumptions and Non-Assumptions About Legal ASTs 8
Interpreters for Languages With Variables Need Environments 9
Implementing Closures 10
Implementing Closures More Efficiently 11
Defining "Macros" Via Functions in the Metalanguage 11
ML versus Racket 12
What is Static Checking? 13
Correctness: Soundness, Completeness, Undecidability 14
Weak Typing 15
More Flexible Primitives is a Related but Different Issue 16
Advantages and Disadvantages of Static Checking 16
1. Is Static or Dynamic Typing More Convenient? 17
2. Does Static Typing Prevent Useful Programs? 17
3. Is Static Typing's Early Bug-Detection Important? 18
4. Does Static or Dynamic Typing Lead to Better Performance? 19
5. Does Static or Dynamic Typing Make Code Reuse Easier? 19
6. Is Static or Dynamic Typing Better for Prototyping? 20
7. Is Static or Dynamic Typing Better for Code Evolution? 20
Optional: `eval` and `quote` 21

Datatype-Programming Without Datatypes

In ML, we used datatype-bindings to define our own one-of types, including recursive datatypes for tree-based data, such as a little language for arithmetic expressions. A datatype-binding introduces a new type into the static environment, along with constructors for creating data of the type and pattern-matching for

using data of the type. Racket, as a dynamically typed language, has nothing directly corresponding to datatype-bindings, but it *does* support the same sort of data definitions and programming.

First, some situations where we need datatypes in ML are simpler in Racket because we can just use dynamic typing to put any kind of data anywhere we want. For example, we know in ML that lists are polymorphic but any particular list must have elements that all have the same type. So we cannot directly build a list that holds "string *or* ints." Instead, we can define a datatype to work around this restriction, as in this example:

```
datatype int_or_string = I of int | S of string

fun funny_sum xs =
    case xs of
        [] => 0
      | (I i)::xs' => i + funny_sum xs'
      | (S s)::xs' => String.size s + funny_sum xs'
```

In Racket, no such work-around is necessary, as we can just write functions that work for lists whose elements are numbers or strings:

```
(define (funny-sum xs)
  (cond [(null? xs) 0]
        [(number? (car xs)) (+ (car xs) (funny-sum (cdr xs)))]
        [(string? (car xs)) (+ (string-length (car xs)) (funny-sum  (cdr xs)))]
        [#t (error "expected number or string")]))
```

Essential to this approach is that Racket has built-in primitives like `null?`, `number?`, and `string?` for testing the type of data at run-time.

But for recursive datatypes like this ML definition for arithmetic expressions:

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

adapting our programming idioms to Racket will prove more interesting.

We will first consider an ML function that evaluates things of type `exp`, but this function will have a different return type than similar functions we wrote earlier in the course. We will then consider two different approaches for defining and using this sort of "type" for arithmetic expressions in Racket. We will argue the second approach is better, but the first approach is important for understanding Racket in general and the second approach in particular.

Changing How We Evaluate Our Arithmetic Expression Datatype

The most obvious function to write that takes a value of the ML datatype `exp` defined above is one that evaluates the arithmetic expression and returns the result. Previously we wrote such a function like this:

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

The type of `eval_exp_old` is `exp -> int`. In particular, the return type is `int`, an ML integer that we can then add, multiply, etc. using ML's arithmetic operators.

For the rest of this course unit, we will instead write this sort of function to *return an exp*, so the ML type will become `exp -> exp`. The result of a call (including recursive calls) will have the form `Const i` for some int `i`, e.g., `Const 17`. Callers have to check that the kind of `exp` returned is indeed a Const, extract the underlying data (in ML, using pattern-matching), and then themselves use the `Const` constructor as necessary to return an `exp`. For our little arithmetic language, this approach leads to a moderately more complicated program:

```
exception Error of string
fun eval_exp_new e =
    let
        fun get_int e =
            case e of
                Const i => i
              | _ => raise (Error "expected Const result")
    in
        case e of
            Const _ => e (* notice we return the entire exp here *)
          | Negate e2  => Const (~ (get_int (eval_exp_new e2)))
          | Add(e1,e2) => Const ((get_int (eval_exp_new e1)) + (get_int (eval_exp_new e2)))
          | Multiply(e1,e2) => Const ((get_int (eval_exp_new e1)) * (get_int (eval_exp_new e2)))
    end
```

This extra complication has little benefit for our simple type `exp`, but we are doing it for a very good reason: Soon we will be defining little languages that have *multiple kinds of results*. Suppose the result of a computation did not have to be a number because it could also be a boolean, a string, a pair, a function closure, etc. Then our `eval_exp` function needs to return some sort of one-of type and using a subset of the possibilities defined by the type `exp` will serve our needs well. Then a case of `eval_exp` like addition will need to check that the recursive results are the right kind of value. If this check does not succeed, then the line of `get_int` above that raises an exception gets evaluated (whereas for our simple example so far, the exception will never get raised).

Recursive Datatypes Via Racket Lists

Before we can write a Racket function analogous to the ML `eval_exp_new` function above, we need to define the arithmetic expressions themselves. We need a way to *construct* constants, negations, additions, and multiplications, a way to *test* what kind of expression we have (e.g., "is it an addition?"), and a way to *access* the pieces (e.g., "get the first subexpression of an addition"). In ML, the datatype binding gave us all this.

In Racket, dynamic typing lets us just use lists to represent any kind of data, including arithmetic expressions. One sufficient idiom is to use the first list element to indicate "what kind of thing it is" and subsequent list elements to hold the underlying data. With this approach, we can just define our own Racket functions for constructing, testing, and accessing:

```
; helper functions for constructing
(define (Const i) (list 'Const i))
(define (Negate e) (list 'Negate e))
(define (Add e1 e2) (list 'Add e1 e2))
```

```
(define (Multiply e1 e2) (list 'Multiply e1 e2))
; helper functions for testing
(define (Const? x) (eq? (car x) 'Const))
(define (Negate? x) (eq? (car x) 'Negate))
(define (Add? x) (eq? (car x) 'Add))
(define (Multiply? x) (eq? (car x) 'Multiply))
; helper functions for accessing
(define (Const-int e) (car (cdr e)))
(define (Negate-e e) (car (cdr e)))
(define (Add-e1 e) (car (cdr e)))
(define (Add-e2 e) (car (cdr (cdr e))))
(define (Multiply-e1 e) (car (cdr e)))
(define (Multiply-e2 e) (car (cdr (cdr e))))
```

(As an orthogonal note, we have not seen the syntax `'foo` before. This is a Racket *symbol*. For our purposes here, a symbol `'foo` is a lot like a string `"foo"` in the sense that you can use any sequence of characters, but symbols and strings are different kinds of things. Comparing whether two symbols are equal is a fast operation, faster than string equality. You can compare symbols with `eq?` whereas you should not use `eq?` for strings. We could have done this example with strings instead, using `equal?` instead of `eq?`.)

We can now write a Racket function to "evaluate" an arithmetic expression. It is directly analogous to the ML version defined in `eval_exp_new`, just using our helper functions instead of datatype constructors and pattern-matching:

```
(define (eval-exp e)
  (cond [(Const? e) e] ; note returning an exp, not a number
        [(Negate? e) (Const (- (Const-int (eval-exp (Negate-e e)))))]
        [(Add? e) (let ([v1 (Const-int (eval-exp (Add-e1 e)))]
                        [v2 (Const-int (eval-exp (Add-e2 e)))])
                    (Const (+ v1 v2)))]
        [(Multiply? e) (let ([v1 (Const-int (eval-exp (Multiply-e1 e)))]
                             [v2 (Const-int (eval-exp (Multiply-e2 e)))])
                         (Const (* v1 v2)))]
        [#t (error "eval-exp expected an exp")]))
```

Similarly, we can use our helper functions to define arithmetic expressions:

```
(define test-exp (Multiply (Negate (Add (Const 2) (Const 2))) (Const 7)))
(define test-ans (eval-exp test-exp))
```

Notice that `test-ans` is `'(Const -28)`, not `-28`.

Also notice that with dynamic typing there is nothing in the program that defines "what is an arithmetic expression." Only our documentation and comments would indicate how arithmetic expressions are built in terms of constants, negations, additions, and multiplications.

Recursive Datatypes Via Racket's `struct`

The approach above for defining arithmetic expressions is inferior to a second approach we now introduce using the special `struct` construct in Racket. A `struct` definition looks like:

```
(struct foo (bar baz quux) #:transparent)
```

This defines a new "struct" called `foo` that is like an ML constructor. It adds to the environment functions for constructing a foo, testing if something is a foo, and extracting the fields bar, baz, and quux from a foo. The names of these bindings are formed systematically from the constructor name `foo` as follows:

- `foo` is a function that takes three arguments and returns a value that is a foo with a bar field holding the first argument, a baz field holding the second argument, and a quux field holding the third argument.
- `foo?` is a function that takes one argument and returns `#t` for values created by calling `foo` and `#f` for everything else.
- `foo-bar` is a function that takes a foo and returns the contents of the bar field, raising an error if passed anything other than a foo.
- `foo-baz` is a function that takes a foo and returns the contents of the baz field, raising an error if passed anything other than a foo.
- `foo-quux` is a function that takes a foo and returns the contents of the quux field, raising an error if passed anything other than a foo.

There are some useful *attributes* we can include in `struct` definitions to modify their behavior, two of which we discuss here.

First, the `#:transparent` attribute makes the fields and accessor functions visible even outside the module that defines the struct. From a modularity perspective this is questionable style, but it has one big advantage when using DrRacket: It allows the REPL to print struct values with their contents rather than just as an abstract value. For example, with our definition of struct `foo`, the result of `(foo "hi" (+ 3 7) #f)` prints as `(foo "hi" 10 #f)`. Without the `#:transparent` attribute, it would print as `#<foo>`, and every value produced from a call to the `foo` function would print this same way. This feature becomes even more useful for examining values built from recursive uses of structs.

Second, the `#:mutable` attribute makes all fields mutable by also providing mutator functions like `set-foo-bar!`, `set-foo-baz!`, and `set-foo-quux!`. In short, the programmer decides when defining a struct whether the advantages of having mutable fields outweigh the disadvantages. It is also possible to make some fields mutable and some fields immutable.

We can use structs to define a new way to represent arithmetic expressions and a function that evaluates such arithmetic expressions:

```
(struct const (int) #:transparent)
(struct negate (e) #:transparent)
(struct add (e1 e2) #:transparent)
(struct multiply (e1 e2) #:transparent)

(define (eval-exp e)
  (cond [(const? e) e] ; note returning an exp, not a number
        [(negate? e) (const (- (const-int (eval-exp (negate-e e)))))]
        [(add? e) (let ([v1 (const-int (eval-exp (add-e1 e)))]
                        [v2 (const-int (eval-exp (add-e2 e)))])
                    (const (+ v1 v2)))]
        [(multiply? e) (let ([v1 (const-int (eval-exp (multiply-e1 e)))]
                             [v2 (const-int (eval-exp (multiply-e2 e)))])
                         (const (* v1 v2)))]
        [#t (error "eval-exp expected an exp")]))
```

Like with our previous approach, nothing in the language indicates how arithmetic expressions are defined in terms of constants, negations, additions, and multiplications. The structure of this version of `eval-exp` is almost identical to the previous version, just using the functions provided by the struct definitions instead of our own list-processing functions. Defining expressions using the constructor functions is also similar:

```
(define test-exp (multiply (negate (add (const 2) (const 2))) (const 7)))
(define test-ans (eval-exp test-exp))
```

Why the `struct` Approach is Better

Defining structs is *not* syntactic sugar for the list approach we took first. The key distinction is that a struct definition creates a *new type of value*. Given

```
(struct add (e1 e2) #:transparent)
```

the function `add` returns things that cause `add?` to return `#t` and *every other* type-testing function like `number?`, `pair?`, `null?`, `negate?`, and `multiply?` to return `#f`. Similarly, the *only* way to access the `e1` and `e2` fields of an add value is with `add-e1` and `add-e2` — trying to use `car`, `cdr`, `multiply-e1`, etc. is a run-time error. (Conversely, `add-e1` and `add-e2` raise errors for anything that is not an add.)

Notice that our first approach with lists does not have these properties. Something built from the `Add` function we defined *is* a list, so `pair?` returns `#t` for it and we can, despite it being poor style, access the pieces directly with `car` and `cdr`.

So in addition to being more concise, our struct-based approach is superior because it *catches errors sooner*. Using `cdr` or `Multiply-e2` on an addition expression in our arithmetic language is almost surely an error, but our list-based approach sees it as nothing more or less than accessing a list using the Racket primitives for doing so. Similarly, nothing prevents an ill-advised client of our code from writing `(list 'Add "hello")` and yet our list-based `Add?` function would return `#t` given the result list `'(Add "hello")`.

That said, nothing about the struct definitions *as we are using them here* truly enforces invariants. In particular, we would like to ensure the `e1` and `e2` fields of any add expression hold only other arithmetic expressions. Racket has good ways to do that, but we are not studying them here. First, Racket has a *module system* that we can use to expose to clients only parts of a struct definition, so we could hide the constructor function and expose a different function that enforces invariants (much like we did with ML's module system).[1] Second, Racket has a *contract system* that lets programmers define arbitrary functions to use to check properties of struct fields, such as allowing only certain kinds of values to be in the fields.

Finally, we remark that Racket's `struct` is a powerful primitive that *cannot* be described or defined in terms of other things like function definitions or macro definitions. It really creates a new type of data. The feature that the result from `add` causes `add?` to return `#t` but every other type-test to return `#f` is something that no approach in terms of lists, functions, macros, etc. can do. Unless the language gives you a primitive for making new types like this, any other encoding of arithmetic expressions would have to make values that cause *some* other type test such as `pair?` or `procedure?` to return `#t`.

[1] Many people erroneously believe dynamically typed languages cannot enforce modularity like this. Racket's structs, and similar features in other languages, put the lie to this. You do not need abstract types and static typing to enforce ADTs. It suffices to have a way to make new types and then not directly expose the constructors for these types.

Implementing a Programming Language in General

While this course is mostly about what programming-language features *mean* and not how they are *implemented*, implementing a small programming language is still an invaluable experience. First, one great way to understand the semantics of some features is to have to implement those features, which forces you to think through all possible cases. Second, it dispels the idea that things like higher-order functions or objects are "magic" since we can implement them in terms of simpler features. Third, many programming tasks are analogous to implementing an interpreter for a programming language. For example, processing a structured document like a pdf file and turning it into a rectangle of pixels for displaying is similar to taking an input program and turning it into an answer.

We can describe a typical workflow for a language implementation as follows. First, we take a *string* holding the *concrete syntax* of a program in the language. Typically this string would be the contents of one or more files. The *parser* gives errors if this string is not syntactically well-formed, meaning the string cannot possibly contain a program in the language due to things like misused keywords, misplaced parentheses, etc. If there are no such errors, the parser produces a *tree* that represents the program. This is called the *abstract-syntax tree*, or AST for short. It is a much more convenient representation for the next steps of the language implementation. If our language includes type-checking rules or other reasons that an AST may still not be a legal program, the *type-checker* will use this AST to either produce error messages or not. The AST is then passed to the rest of the implementation.

There are basically two approaches to this rest-of-the-implementation for implementing some programming language B. First, we could write an *interpreter* in another language A that takes programs in B and produces answers. Calling such a program in A an "evaluator for B" or an "executor for B" probably makes more sense, but "interpreter for B" has been standard terminology for decades. Second, we could write a *compiler* in another language A that takes programs in B and produces equivalent programs in some other language C (not *the* language C necessarily) and then uses some pre-existing implementation for C. For compilation, we call B the source language and C the target language. A better term than "compiler" would be "translator" but again the term compiler is ubiquitous. For either the interpreter approach or the compiler approach, we call A, the language in which we are writing the implementation of B, the *metalanguage*.

While there are many "pure" interpreters and compilers, modern systems often combine aspects of each and use multiple levels of interpretation and translation. For example, a typical Java system compiles Java source code into a portable intermediate format. The Java "virtual machine" can then start interpreting code in this format but get better performance by compiling the code further to code that can execute directly on hardware. We can think of the hardware itself as an interpreter written in transistors, yet many modern processors actually have translators in the hardware that convert the binary instructions into smaller simpler instructions right before they are executed. There are many variations and enhancements to even this multi-layered story of running programs, but fundamentally each step is some combination of interpretation or translation.

A one-sentence sermon: *Interpreter versus compiler is a feature of a particular programming-language implementation, not a feature of the programming language.* One of the more annoying and widespread misconceptions in computer science is that there are "compiled languages" such as C and "interpreted languages" such as Racket. This is nonsense: I can write an interpreter for C or a compiler for Racket. (In fact, DrRacket takes a hybrid approach not unlike Java.) There is a long history of C being implemented with compilers and functional languages being implemented with interpreters, but compilers for functional languages have been around for decades. SML/NJ, for example, compiles each module/binding to binary code.

Implementing a Programming Language Inside Another Language

Our `eval-exp` function above for arithmetic expressions is a perfect example of an interpreter for a small programming language. The language here is exactly expressions properly built from the constructors for constant, negation, addition, and multiplication expressions. The definition of "properly" depends on the language; here we mean constants hold numbers and negations/additions/multiplications hold other proper subexpressions. We also need a definition of *values* (i.e., results) for our little language, which again is part of the language definition. Here we mean constants, i.e., the subset of expressions built from the `const` constructor. Then `eval-exp` is an interpreter because it is a function that takes expressions in our language and produces values in our language according to the rules for the semantics to our language. Racket is just the *metalanguage*, the "other" language in which we write our interpreter.

What happened to parsing and type-checking? In short, we skipped them. By using Racket's constructors, we basically wrote our programs directly in terms of abstract-syntax trees, relying on having convenient syntax for writing trees rather than having to make up a syntax and writing a parser. That is, we wrote programs with expressions like:

```
(negate (add (const 2) (const 2)))
```

rather than some sort of string like `"- (2 + 2)"`.

While *embedding* a language like arithmetic-expressions inside another language like Racket might seem inconvenient compared to having special syntax, it has advantages even beyond not needing to write a parser. For example, below we will see how we can use the metalanguage (Racket in this case) to write things that act like macros for our language.

Assumptions and Non-Assumptions About Legal ASTs

There is a subtle distinction between two kinds of "wrong" ASTs in a language like our arithmetic expression language. To make this distinction clearer, let's extend our language with three more kinds of expressions:

```
(struct const (int) #:transparent) ; int should hold a number
(struct negate (e1) #:transparent)  ; e1 should hold an expression
(struct add (e1 e2) #:transparent) ; e1, e2 should hold expressions
(struct multiply (e1 e2) #:transparent) ; e1, e2 should hold expressions
(struct bool (b) #:transparent) ; b should hold #t or #f
(struct if-then-else (e1 e2 e3) #:transparent) ; e1, e2, e3 should hold expressions
(struct eq-num (e1 e2) #:transparent) ; e1, e2 should hold expressions
```

The new features include booleans (either true or false), conditionals, and a construct for comparing two numbers and returning a boolean (true if and only if the numbers are the same). Crucially, the result of evaluating an expression in this language could now be:

- an integer, such as `(const 17)`
- a boolean, such as `(bool true)`
- non-existent because when we try to evaluate the program, we get a "run-time type error" – trying to treat a boolean as a number or vice-versa

In other words, there are now two types of *values* in our language – numbers and booleans – and there are operations that should fail if a subexpression evaluates to the wrong kind of value.

This last possibility is something an interpreter should check for and give an appropriate error message. If evaluating some kind of expression (e.g., addition) requires the result of evaluating subexpressions to have a certain type (e.g., a number like `(const 4)` and not a boolean like `(bool #t)`), then the interpreter should check for this result (e.g., using `const?`) rather than assuming the recursive result has the right type. That way, the error message is appropriate (e.g., "argument to addition is not a number") rather than something in terms of the implementation of the interpreter.

The code posted with the course materials corresponding to these notes has two full interpreters for this language. The first does not include any of this checking while the second, better one does. Calling the first interpreter `eval-exp-wrong` and the second one `eval-exp`, here is just the addition case for both:

```
; eval-exp-wrong
[(add? e)
 (let ([i1 (const-int (eval-exp-wrong (add-e1 e)))]
       [i2 (const-int (eval-exp-wrong (add-e2 e)))])
   (const (+ i1 i2)))]

; eval-exp
[(add? e)
 (let ([v1 (eval-exp (add-e1 e))]
       [v2 (eval-exp (add-e2 e))])
   (if (and (const? v1) (const? v2))
       (const (+ (const-int v1) (const-int v2)))
       (error "add applied to non-number")))]
```

However, `eval-exp` *is* assuming that the expression it is evaluating is a legal AST for the language. It can handle `(add (const 2) (const 2))`, which evaluates to `(const 4)` or `(add (const 2) (bool #f))`, which encounters an error, but it does not gracefully handle `(add #t #f)` or `(add 3 4)`. These are not legal ASTs, according to the rules we have in comments, namely:

- The `int` field of a `const` should hold a Racket number.
- The `b` field of a `bool` should hold a Racket boolean.
- All other fields of expressions should hold other legal ASTs. (Yes, the definition is recursive.)

It is reasonable for an interpreter to *assume* it is given a legal AST, so it is *okay* for it to "just crash" with a strange, implementation-dependent error message if given an illegal AST.

Interpreters for Languages With Variables Need Environments

The biggest thing missing from our arithmetic-expression language is variables. That is why we could just have one recursive function that took an expression and returned a value. As we have known since the very beginning of the course, since expressions can contain variables, evaluating them requires an environment that maps variables to values. So an interpreter for a language with variables needs a recursive helper function that takes an expression and an environment and produces a value.[2]

[2]In fact, for languages with features like mutation or exceptions, the helper function needs even more parameters.

The representation of the environment is part of the interpreter's implementation in the metalanguage, not part of the abstract syntax of the language. Many representations will suffice and fancy data structures that provide fast access for commonly used variables are appropriate. But for our purposes, ignoring efficiency is okay. Therefore, with Racket as our metalanguage, a simple association list holding pairs of strings (variable names) and values (what the variables are bound to) can suffice.

Given an environment, the interpreter uses it differently in different cases:

- To evaluate a variable expression, it looks up the variable's name (i.e., the string) in the environment.
- To evaluate most subexpressions, such as the subexpressions of an addition operation, the interpreter passes to the recursive calls the same environment that was passed for evaluating the outer expression.
- To evaluate things like the body of a let-expression, the interpreter passes to the recursive call a slightly different environment, such as the environment it was passed with one more binding (i.e., pair of string and value) in it.

To evaluate an entire program, we just call our recursive helper function that takes an environment with the program and a suitable initial environment, such as the empty environment, which has no bindings in it.

Implementing Closures

To implement a language with function closures and lexical scope, our interpreter needs to "remember" the environment that "was current" when the function was defined so that it can use this environment *instead of* the caller's environment when the function is called. The "trick" to doing this is rather direct: We can literally create a small data structure called a *closure* that includes the environment along with the function itself. *It is this pair (the closure) that is the result of interpreting a function.* In other words, a function is not a value, a closure is, so the evaluation of a function produces a closure that "remembers" the environment from when we evaluated the function.

We also need to implement function calls. A call has two expressions `e1` and `e2` for what would look like `e1 e2` in ML or `(e1 e2)` in Racket. (We consider here one-argument functions, though the implementation will naturally support currying for simulating multiple argument functions.) We evaluate a call as follows:

- We evaluate `e1` using the current environment. The result should be a closure (else it is a run-time error).
- We evaluate `e2` using the current environment. The result will be the argument to the closure.
- We evaluate the body of the code part of the closure **using the environment part of the closure** extended with the argument of the code part mapping to the argument at the call-site.

In the homework assignment connected to these course materials, there is an additional extension to the environment for a variable that allows the closure to call itself recursively. But the key idea is the same: we extend the environment-stored-with-the-closure to evaluate the closure's function body.

This really is how interpreters implement closures. It is the semantics we learned when we first studied closures, just "coded up" in an interpreter.

Implementing Closures More Efficiently

It may seem expensive that we store the "whole current environment" in every closure. First, it is not that expensive when environments are association lists since different environments are just extensions of each other and we do not copy lists when we make longer lists with `cons`. (Recall this sharing is a big benefit of not mutating lists, and we do not mutate environments.) Second, in practice we can save space by storing only those parts of the environment that the function body might possibly use. We can look at the function body and see what *free variables* it has (variables used in the function body whose definitions are outside the function body) and the environment we store in the closure needs only these variables. After all, no execution of the closure can ever need to look up a variable from the environment if the function body has no use of the variable. Language implementations *precompute* the free variables of each function before beginning evaluation. They can store the result with each function so that this set of variables is quickly available when building a closure.

Finally, you might wonder how compilers implement closures if the target language does not itself have closures. As part of the translation, function definitions still evaluate to closures that have two parts, code and environment. However, we do not have an interpreter with a "current environment" whenever we get to a variable we need to look up. So instead, we change all the functions in the program to take an *extra argument* (the environment) and change all function calls to *explicitly pass in this extra argument.* Now when we have a closure, the code part will have an extra argument and the caller will pass in the environment part for this argument. The compiler then just needs to translate all uses of free variables to code that uses the extra argument to find the right value. In practice, using good data structures for environments (like arrays) can make these variable lookups very fast (as fast as reading a value from an array).

Defining "Macros" Via Functions in the Metalanguage

When implementing an interpreter or compiler, it is essential to keep separate what is in *the language being implemented* and what is in *the language used for doing the implementation (the metalanguage).* For example, `eval-exp` is a Racket function that takes an arithmetic-expression-language expression (or whatever language we are implementing) and produces an arithmetic-expression-language value. So for example, an arithmetic-expression-language expression would never include a use of `eval-exp` or a Racket addition expression.

But since we are writing our to-be-evaluated programs in Racket, we can use Racket helper functions to help us create these programs. Doing so is basically defining *macros* for our language using Racket functions as the macro language. Here is an example:

```
(define (double e) ; takes language-implemented syntax and produces language-implemented syntax
   (multiply e (const 2)))
```

Here `double` is a Racket function that takes the syntax for an arithmetic expression and produces the syntax for an arithmetic expression. Calling `double` produces abstract syntax in our language, much like macro expansion. For example, `(negate (double (negate (const 4))))` produces `(negate (multiply (negate (const 4)) (const 2)))`. Notice this "macro" `double` does not evaluate the program in any way: we produce abstract syntax that can then be evaluated, put inside a larger program, etc.

Being able to do this is an advantage of "embedding" our little language inside the Racket metalanguage. The same technique works regardless of the choice of metalanguage. However, this approach does not handle issues related to variable shadowing as well as a real macro system that has hygienic macros.

Here is a different "macro" that is interesting in two ways. First the argument is a *Racket* list of *language-being-implemented* expressions (syntax). Second, the "macro" is recursive, calling itself once for each element in the argument list:

```
(define (list-product es)
  (if (null? es)
      (const 1)
      (multiply (car es) (list-product (cdr es)))))
```

ML versus Racket

Before studying the general topic of static typing and the advantages/disadvantages thereof, it is interesting to do a more specific comparison between the two languages we have studied so far, ML and Racket. The languages are similar in many ways, with constructs that encourage a functional style (avoiding mutation, using first-class closures) while allowing mutation where appropriate. There are also many differences, including very different approaches to syntax, ML's support for pattern-matching compared to Racket's accessor functions for structs, Racket's multiple variants of let-expressions, etc.

But the most widespread difference between the two languages is that ML has a static type system that Racket does not.[3]

We study below precisely what a static type system is, what ML's type system guarantees, and what the advantages and disadvantages of static typing are. Anyone who has programmed in ML and Racket probably already has some ideas on these topics, naturally: ML rejects lots of programs before running them by doing type-checking and reporting errors. To do so, ML enforces certain restrictions (e.g., all elements of a list must have the same type). As a result, ML ensures the absence of certain errors (e.g., we will never try to pass a string to the addition operator) "at compile time."

More interestingly, could we describe ML and its type system in terms of ideas more Racket-like and, conversely, could we describe Racket-style programming in terms of ML? It turns out we can and that doing so is both mind-expanding and a good precursor to subsequent topics.

First consider how a Racket programmer might view ML. Ignoring syntax differences and other issues, we can describe ML as roughly defining a *subset* of Racket: Programs that run produce similar answers, but ML rejects many more programs as illegal, i.e., not part of the language. What is the advantage of that? ML is designed to reject programs that are likely bugs. Racket allows programs like `(define (f y) (+ y (car y)))`, but any call to `f` would cause an error, so this is hardly a useful program. So it is helpful that ML rejects this program rather than waiting until a programmer tests `f`. Similarly, the type system catches bugs due to inconsistent assumptions by different parts of the program. The functions `(define (g x) (+ x x))` and `(define (h z) (g (cons z 2)))` are both sensible by themselves, but if the `g` in `h` is bound to this definition of `g`, then any call to `h` fails much like any call to `f`. On the other hand, ML rejects Racket-like programs that are not bugs as well. For example, in this code, both the if-expression and the expression bound to `xs` would not type-check but represent reasonable Racket idioms depending on circumstances:

```
(define (f x) (if (> x 0) #t (list 1 2)))
(define xs (list 1 #t "hi"))
(define y (f (car xs)))
```

So now how might an ML programmer view Racket? One view is just the reverse of the discussion above, that Racket accepts a superset of programs, some of which are errors and some of which are not. A more interesting view is that Racket is just ML where *every expression is part of one big datatype*. In this view, the result of every computation is *implicitly* "wrapped" by a constructor into the one big datatype and

[3]There is a related language Typed Racket also available within the DrRacket system that interacts well with Racket and many other languages — allowing you to mix files written in different languages to build applications. We will not study that in this course, so we refer here only to the language Racket.

primitives like + have implementations that check the "tags" of their arguments (e.g., to see if they are numbers) and raise errors as appropriate. In more detail, it is like Racket has this one datatype binding:

```
datatype theType = Int of int
                 | String of string
                 | Pair of theType * theType
                 | Fun of theType -> theType
                 | ... (* one constructor per built-in type *)
```

Then it is like when programmers write something like 42, it is *implicitly* really `Int 42` so that the result of every expression has type `theType`. Then functions like + raise errors if both arguments do not have the right constructor and their result is also wrapped with the right constructor if necessary. For example, we could think of `car` as being:

```
fun car v = case v of Pair(a,b) => a | _  => raise ... (* give some error *)
```

Since this "secret pattern-matching" is not exposed to programmers, Racket also provides which-constructor functions that programmers can use instead. For example, the primitive `pair?` can be viewed as:

```
fun pair? v = case v of Pair _ => true | _  => false
```

Finally, Racket's struct definitions do one thing you cannot quite do with ML datatype bindings: They dynamically add new constructors to a datatype.[4]

The fact that we can think of Racket in terms of `theType` suggests that anything you can do in Racket can be done, perhaps more awkwardly, in ML: The ML programmer could just program explicitly using something like the `theType` definition above.

What is Static Checking?

What is usually meant by "static checking" is anything done to reject a program *after* it (successfully) parses but *before* it runs. If a program does not parse, we still get an error, but we call such an error a "syntax error" or "parsing error." In contrast, an error from static checking, typically a "type error," would include things like undefined variables or using a number instead of a pair. We do static checking without any input to the program identified — it is "compile-time checking" though it is irrelevant whether the language implementation will use a compiler or an interpreter after static checking succeeds.

What static checking is performed is part of the definition of a programming language. Different languages can do different things; some languages do no static checking at all. Given a language with a particular definition, you could also use other tools that do even more static checking to try to find bugs or ensure their absence even though such tools are not part of the language definition.

The most common way to define a language's static checking is via a *type system*. When we studied ML (and when you learned Java), we gave typing rules for each language construct: Each variable had a type, the two branches of a conditional must have the same type, etc. ML's static checking is checking that these rules are followed (and in ML's case, inferring types to do so). But this is the language's *approach* to static checking (how it does it), which is different from the *purpose* of static checking (what it accomplishes). The purpose is to reject programs that "make no sense" or "may try to misuse a language feature." There are

[4] You can do this in ML with the `exn` type, but not with datatype bindings. If you could, static checking for missing pattern-matching clauses would not be possible.

errors a type system typically does not prevent (such as array-bounds errors) and others that a type system *cannot* prevent unless given more information about what a program is supposed to do. For example, if a program puts the branches of a conditional in the wrong order or calls `+` instead of `*`, this is still a program just not the one intended.

For example, one purpose of ML's type system is to prevent passing strings to arithmetic primitives like the division operator. In contrast, Racket uses "dynamic checking" (i.e., run-time checking) by tagging each value and having the division operator check that its arguments are numbers. The ML implementation does not have to tag values for this purpose because it can rely on static checking. But as we will discuss below, the trade-off is that the static checker has to reject some programs that would not actually do anything wrong.

As ML and Racket demonstrate, the typical points at which to prevent a "bad thing" are "compile-time" and "run-time." However, it is worth realizing that there is really a continuum of eagerness about when we declare something an error. Consider for sake of example something that most type systems do not prevent statically: division-by-zero. If we have some function containing the expression `(/ 3 0)`, when could we cause an error:

- Keystroke-time: Adjust the editor so that one cannot even write down a division with a denominator of 0. This is approximate because maybe we were about to write 0.33, but we were not allowed to write the 0.
- Compile-time: As soon as we see the expression. This is approximate because maybe the context is `(if #f (/ 3 0) 42)`.
- Link-time: Once we see the function containing `(/ 3 0)` might be called from some "main" function. This is less approximate than compile-time since some code might never be used, but we still have to approximate what code may be called.
- Run-time: As soon as we execute the division.
- Even later: Rather than raise an error, we could just return some sort of value indicating division-by-zero and not raise an error until that value was used for something where we needed an actual number, like indexing into an array.

While the "even later" option might seem too permissive at first, it is exactly what floating-point computations do. `(/ 3.0 0.0)` produces `+inf.0`, which can still be computed with but cannot be converted to an exact number. In scientific computing this is very useful to avoid lots of extra cases: maybe we do something like take the tangent of $\pi/2$ but only when this will end up not being used in the final answer.

Correctness: Soundness, Completeness, Undecidability

Intuitively, a static checker is correct if it prevents what it claims to prevent — otherwise, either the language definition or the implementation of static checking needs to be fixed. But we can give a more precise description of correctness by defining the terms *soundness* and *completeness*. For both, the definition is with respect to some thing X we wish to prevent. For example, X could be "a program looks up a variable that is not in the environment."

A type system is *sound* if it never accepts a program that, when run with some input, does X.

A type system is *complete* if it never rejects a program that, no matter what input it is run with, will not do X.

A good way to understand these definitions is that *soundness prevents false negatives* and *completeness prevents false positives*. The terms *false negatives* and *false positives* come from statistics and medicine: Suppose there is a medical test for a disease, but it is not a perfect test. If the test does not detect the disease but the patient actually has the disease, then this is a false negative (the test was negative, but that is false). If the test detects the disease but the patient actually does not have the disease, then this is a false positive (the test was positive, but that's false). With static checking, the disease is "performs X when run with some input" and the test is "does the program type-check?" The terms *soundness* and *completeness* come from logic and are commonly used in the study of programming languages. A sound logic proves only true things. A complete logic proves all true things. Here, our type system is the logic and the thing we are trying to prove is "X cannot occur."

In modern languages, type systems are sound (they prevent what they claim to) but not complete (they reject programs they need not reject). Soundness is important because it lets language users and language implementers rely on X never happening. Completeness would be nice, but hopefully it is rare in practice that a program is rejected unnecessarily and in those cases, hopefully it is easy for the programmer to modify the program such that it type-checks.

Type systems are not complete because for almost anything you might like to check statically, it is *impossible* to implement a static checker that given any program in your language (a) always terminates, (b) is sound, and (c) is complete. Since we have to give up one, (c) seems like the best option (programmers do not like compilers that may not terminate).

The impossibility result is exactly the idea of *undecidability* at the heart of the study of the theory of computation. It is an essential topic in a required course (CSE 311). Knowing what it means that nontrivial properties of programs are undecidable is fundamental to being an educated computer scientist. The fact that undecidability directly implies the inherent approximation (i.e., incompleteness) of static checking is probably the most important ramification of undecidability. We simply cannot write a program that takes as input another program in ML/Racket/Java/etc. that always correctly answers questions such as, "will this program divide-by-zero?" "will this program treat a string as a function?" "will this program terminate?" etc.

Weak Typing

Now suppose a type system is unsound for some property X. Then to be safe the language implementation should still, at least in some cases, perform dynamic checks to prevent X from happening and the language definition should allow that these checks might fail at run-time.

But an alternative is to say it is the programmer's fault if X happens and the language definition does *not* have to check. In fact, if X happens, then the running program can do *anything*: crash, corrupt data, produce the wrong answer, delete files, launch a virus, or set the computer on fire. If a language has programs where a legal implementation is allowed to set the computer on fire (even though it probably would not), we call the language *weakly typed*. Languages where the behavior of buggy programs is more limited are called *strongly typed*. These terms are a bit unfortunate since the correctness of the type system is only part of the issue. After all, Racket is dynamically typed but nonetheless strongly typed. Moreover, a big source of actual undefined and unpredictable behavior in weakly typed languages is array-bounds errors (they need not check the bound — they can just access some other data by mistake), yet few type systems check array bounds.

C and C++ are the well-known weakly typed languages. Why are they defined this way? In short, because the designers do not want the language definition to force implementations to do all the dynamic checks that would be necessary. While there is a time cost to performing checks, the bigger problem is that the implementation has to keep around extra data (like tags on values) to do the checks and C/C++ are designed

as lower-level languages where the programmer can expect extra "hidden fields" are not added.

An older now-much-rarer perspective in favor of weak typing is embodied by the saying "strong types for weak minds." The idea is that any strongly typed language is either rejecting programs statically or performing unnecessary tests dynamically (see undecidability above), so a human should be able to "overrule" the checks in places where he/she knows they are unnecessary. In reality, humans are extremely error-prone and we should welcome automatic checking even if it has to err on the side of caution for us. Moreover, type systems have gotten much more expressive over time (e.g., polymorphic) and language implementations have gotten better at optimizing away unnecessary checks (they will just never get all of them). Meanwhile, software has gotten very large, very complex, and relied upon by all of society. It is deeply problematic that 1 bug in a 30-million-line operating system written in C can make the entire computer subject to security exploits. While this is still a real problem and C the language provides little support, it is increasingly common to use other tools to do static and/or dynamic checking with C code to try to prevent such errors.

More Flexible Primitives is a Related but Different Issue

Suppose we changed ML so that the type system accepted any expression `e1 + e2` as long as `e1` and `e2` had *some* type and we changed the evaluation rules of addition to return `0` if one of the arguments did not result in a number. Would this make ML a dynamically typed language? It is "more dynamic" in the sense that the language is more lenient and some "likely" bugs are not detected as eagerly, but there is still a type system rejecting programs — we just changed the definition of what an "illegal" operation is to allow more additions. We could have similarly changed Racket to not give errors if `+` is given bad arguments. The Racket designers choose not to do so because it is likely to mask bugs without being very useful.

Other languages make different choices that report fewer errors by extending the definition of primitive operations to *not* be errors in situations like this. In addition to defining arithmetic over any kind of data, some examples are:

- Allowing out-of-bound array accesses. For example, if `arr` has fewer than 10 elements, we can still allow `arr[10]` by just returning a default value or `arr[10]=e` by making the array bigger.
- Allowing function calls with the wrong number of arguments. Extra arguments can be silently ignored. Too few arguments can be filled in with defaults chosen by the language.

These choices are matters of language design. Giving meaning to what are likely errors is often unwise — it masks errors and makes them more difficult to debug because the program runs long after some nonsense-for-the-application computation occurred. On the other hand, such "more dynamic" features are used by programmers when provided, so clearly someone is finding them useful.

For our purposes here, we just consider this a separate issue from static vs. dynamic typing. Instead of preventing some X (e.g., calling a function with too many arguments) either before the program runs or when it runs, we are changing the language semantics so that we do not prevent X at all – we allow it and extend our evaluation rules to give it a semantics.

Advantages and Disadvantages of Static Checking

Now that we know what static and dynamic typing are, let's wade into the decades-old argument about which is better. We know static typing catches many errors for you early, soundness ensures certain kinds of errors do not remain, and incompleteness means some perfectly fine programs are rejected. We will not answer definitively whether static typing is desirable (if nothing else it depends what you are checking), but

we will consider seven specific claims and consider for each valid arguments made both for and against static typing.

1. Is Static or Dynamic Typing More Convenient?

The argument that dynamic typing is more convenient stems from being able to mix-and-match different kinds of data such as numbers, strings, and pairs without having to declare new type definitions or "clutter" code with pattern-matching. For example, if we want a function that returns either a number or string, we can just return a number or a string, and callers can use dynamic type predicates as necessary. In Racket, we can write:

```
(define (f y) (if (> y 0) (+ y y) "hi"))
(let ([ans (f x)]) (if (number? ans) (number->string ans) ans))
```

In contrast, the analogous ML code needs to use a datatype, with constructors in `f` and pattern-matching to use the result:

```
datatype t = Int of int | String of string
fun f y = if y > 0 then Int(y+y) else String "hi"
val _ = case f x of Int i => Int.toString i | String s => s
```

On the other hand, static typing makes it more convenient to assume data has a certain type, knowing that this assumption cannot be violated, which would lead to errors later. For a Racket function to ensure some data is, for example, a number, it has to insert an explicit dynamic check in the code, which is more work and harder to read. The corresponding ML code has no such awkwardness.

```
(define (cube x)
  (if (not (number? x))
      (error "cube expects a number")
      (* x x x)))
(cube 7)

fun cube x = x * x * x
val _ = cube 7
```

Notice that without the check in the Racket code, the actual error would arise in the body of the multiplication, which could confuse callers that did not know `cube` was implemented using multiplication.

2. Does Static Typing Prevent Useful Programs?

Dynamic typing does not reject programs that make perfect sense. For example, the Racket code below binds `'((7 . 7) . (#t . #t))` to `pair_of_pairs` without problem, but the corresponding ML code does not type-check since there is no type the ML type system can give to `f`.[5]

```
(define (f g) (cons (g 7) (g #t)))
```

[5] This is a limitation of ML. There are languages with more expressive forms of polymorphism that can type-check such code. But due to undecidability, there are always limitations.

```
(define pair_of_pairs (f (lambda (x) (cons x x))))

fun f g = (g 7, g true) (* does not type-check *)
val pair_of_pairs = f (fn x => (x,x))
```

Of course we can write an ML program that produces `((7,7),(true,true))`, but we may have to "work around the type-system" rather than do it the way we want.

On the other hand, dynamic typing derives its flexibility from putting a tag on every value. In ML and other statically typed languages, we can do the same thing *when we want to* by using datatypes and explicit tags. In the extreme, if you want to program like Racket in ML, you can use a datatype to represent "The One Racket Type" and insert explicit tags and pattern-matching everywhere. While this programming style would be painful to use everywhere, it proves the point that there is nothing we can do in Racket that we cannot do in ML. (We discussed this briefly already above.)

```
datatype tort = Int of int
              | String of string
              | Pair of tort * tort
              | Fun of tort -> tort
              | Bool of bool
              | ...
fun f g = (case g of Fun g' => Pair(g' (Int 7), g' (Bool true)))
val pair_of_pairs = f (Fun (fn x => Pair(x,x)))
```

Perhaps an even simpler argument in favor of static typing is that modern type systems are expressive enough that they rarely get in your way. How often do you try to write a function like `f` that does not type-check in ML?

3. Is Static Typing's Early Bug-Detection Important?

A clear argument in favor of static typing is that it catches bugs earlier, as soon you statically check (informally, "compile") the code. A well-known truism of software development is that bugs are easier to fix if discovered sooner, while the developer is still thinking about the code. Consider this Racket program:

```
(define (pow x)
   (lambda (y)
      (if (= y 0)
          1
          (* x (pow x (- y 1))))))
```

While the algorithm looks correct, this program has a bug: `pow` expects curried arguments, but the recursive call passes `pow` two arguments, not via currying. This bug is not discovered until testing `pow` with a `y` not equal to `0`. The equivalent ML program simply does not type-check:

```
fun pow x y = (* does not type-check *)
  if y = 0
  then 1
  else x * pow (x,y-1)
```

Because static checkers catch known kinds of errors, expert programmers can use this knowledge to focus attention elsewhere. A programmer might be quite sloppy about tupling versus currying when writing down most code, knowing that the type-checker will later give a list of errors that can be quickly corrected. This could free up mental energy to focus on other tasks, like array-bounds reasoning or higher-level algorithm issues.

A dynamic-typing proponent would argue that static checking usually catches only bugs you would catch with testing anyway. Since you still need to test your program, the additional value of catching some bugs before you run the tests is reduced. After all, the programs below do not work as exponentiation functions (they use the wrong arithmetic), ML's type system will not detect this, and testing catches this bug and would also catch the currying bug above.

```
(define (pow x) ; wrong algorithm
  (lambda (y)
     (if (= y 0)
         1
         (+ x ((pow x) (- y 1))))))

fun pow x y = (* wrong algorithm *)
   if y = 0
   then 1
   else x + pow x (y - 1)
```

4. Does Static or Dynamic Typing Lead to Better Performance?

Static typing can lead to faster code since it does not need to perform type tests at run time. In fact, much of the performance advantage may result from not storing the type tags in the first place, which takes more space and slows down constructors. In ML, there are run-time tags only where the programmer uses datatype constructors rather than everywhere.

Dynamic typing has three reasonable counterarguments. First, this sort of low-level performance does not matter in most software. Second, implementations of dynamically typed languages can and do try to *optimize away* type tests it can tell are unnecessary. For example, in `(let ([x (+ y y)]) (* x 4))`, the multiplication does not need to check that `x` and `4` are numbers and the addition can check `y` only once. While no optimizer can remove all unnecessary tests from every program (undecidability strikes again), it may be easy enough in practice for the parts of programs where performance matters. Third, if programmers in statically typed languages have to work around type-system limitations, then those workarounds can erode the supposed performance advantages. After all, ML programs that use datatypes have tags too.

5. Does Static or Dynamic Typing Make Code Reuse Easier?

Dynamic typing arguably makes it easier to reuse library functions. After all, if you build lots of different kinds of data out of cons cells, you can just keep using `car`, `cdr`, `cadr`, etc. to get the pieces out rather than defining lots of different getter functions for each data structure. On the other hand, this can mask bugs. For example, suppose you accidentally pass a list to a function that expects a tree. If `cdr` works on both of them, you might just get the wrong answer or cause a mysterious error later, whereas using different types for lists and trees could catch the error sooner.

This is really an interesting design issue more general than just static versus dynamic typing. Often it is good to reuse a library or data structure you already have especially since you get to reuse all the functions

available for it. Other times it makes it too difficult to separate things that are really different conceptually so it is better to define a new type. That way the static type-checker or a dynamic type-test can catch when you put the wrong thing in the wrong place.

6. Is Static or Dynamic Typing Better for Prototyping?

Early in a software project, you are developing a prototype, often at the same time you are changing your views on what the software will do and how the implementation will approach doing it.

Dynamic typing is often considered better for prototyping since you do not need to expend energy defining the types of variables, functions, and data structures when those decisions are in flux. Moreover, you may know that part of your program does not yet make sense (it would not type-check in a statically typed language), but you want to run the rest of your program anyway (e.g., to test the parts you just wrote).

Static typing proponents may counter that it is never too early to document the types in your software design even if (perhaps especially if) they are unclear and changing. Moreover, commenting out code or adding stubs like pattern-match branches of the form `_ => raise Unimplemented` is often easy and documents what parts of the program are known not to work.

7. Is Static or Dynamic Typing Better for Code Evolution?

A lot of effort in software engineering is spent maintaining working programs, by fixing bugs, adding new features, and in general evolving the code to make some change.

Dynamic typing is sometimes more convenient for code evolution because we can change code to be more permissive (accept arguments of more types) without having to change any of the pre-existing clients of the code. For example, consider changing this simple function:

```
(define (f x) (* 2 x))
```

to this version, which can process numbers or strings:

```
(define (f x)
  (if (number? x)
      (* 2 x)
      (string-append x x)))
```

No existing caller, which presumably uses `f` with numbers, can tell this change was made, but new callers can pass in strings or even values where they do not know if the value is a number or a string. If we make the analogous change in ML, no existing callers will type-check since they all must wrap their arguments in the `Int` constructor and use pattern-matching on the function result:

```
fun f x = 2 * x

datatype t = Int of int | String of string
fun f x =
  case f x of
    Int i    => Int (2 * i)
  | String s => String (s ^ s)
```

On the other hand, static type-checking is very useful when evolving code to catch bugs that the evolution introduces. When we change the type of a function, all callers no longer type-check, which means the type-checker gives us an invaluable "to-do list" of all the call-sites that need to change. By this argument, the safest way to evolve code is to change the types of any functions whose specification is changing, which is an argument for capturing as much of your specification as you can in the types.

A particularly good example in ML is when you need to add a new constructor to a datatype. If you did not use wildcard patterns, then you will get a warning for all the case-expressions that use the datatype.

As valuable as the "to-do list from the type-checker" is, it can be frustrating that the program will not run until all items on the list are addressed or, as discussed under the previous claim, you use comments or stubs to remove the parts not yet evolved.

Optional: `eval` and `quote`

(This short description barely scratches the surface of programming with `eval`. It really just introduces the concept. Interested students are encouraged to learn more on their own.)

There is one sense where it is slightly fair to say Racket is an interpreted language: it has a primitive `eval` that can take a representation of a program at run-time and evaluate it. For example, this program, which is poor style because there are much simpler ways to achieve its purpose, may or may not print something depending on `x`:

```
(define (make-some-code y)
   (if y
       (list 'begin (list 'print "hi") (list '+ 4 2))
       (list '+ 5 3)))
(define (f x)
  (eval (make-some-code x)))
```

The Racket function `make-some-code` is strange: It does *not* ever print or perform an addition. All it does is return some list containing symbols, strings, and numbers. For example, if called with `#t`, it returns

```
'(begin (print "hi") (+ 4 2))
```

This is nothing more and nothing less than a three element list where the first element is the symbol `begin`. It is just Racket data. But if we look at this data, it looks just like a Racket program we could run. The nested lists together are a perfectly good *representation* of a Racket expression that, if evaluated, would print `"hi"` and have a result of 6.

The `eval` primitive takes such a representation and, at run-time, evaluates it. We can perform whatever computation we want to generate the data we pass to `eval`. As a simple example, we could append together two lists, like `(list '+ 2)` and `(list 3 4)`. If we call `eval` with the result `'(+ 2 3 4)`, i.e., a 4-element list, then `eval` returns 9.

Many languages have `eval`, many do not, and what the appropriate idioms for using it are is a subject of significant dispute. Most would agree it tends to get overused but is also a really powerful construct that is sometimes what you want.

Can a compiler-based language implementation (notice we did not say "compiled language") deal with `eval`? Well, it would need to have the compiler or an interpreter around at run-time since it cannot know in advance what might get passed to `eval`. An interpreter-based language implementation would also need an interpreter or compiler around at run-time, but, of course, it *already* needs that to evaluate the "regular program."

In languages like Javascript and Ruby, we do not have the convenience of Racket syntax where programs and lists are so similar-looking that `eval` can take a list-representation that looks exactly like Racket syntax. Instead, in these languages, `eval` takes a string and interprets it as concrete syntax by first parsing it and then running it. Regardless of language, `eval` will raise an error if given an ill-formed program or a program that raises an error.

In Racket, it is painful and unnecessary to write `make-some-code` the way we did. Instead, there is a special form `quote` that treats everything under it as symbols, numbers, lists, etc., *not* as functions to be called. So we could write:

```
(define (make-some-code y)
  (if y
      (quote (begin (print "hi") (+ 4 2)))
      (quote (+ 5 3))))
```

Interestingly, `eval` and `quote` are inverses: For any expression `e`, we should have `(eval (quote e))` as a terrible-style but equivalent way to write `e`.

Often `quote` is "too strong" — we want to quote *most* things, but it is convenient to evaluate some code inside of what is mostly syntax we are building. Racket has quasiquote and unquote for doing this (see the manual if interested) and Racket's linguistic predecessors have had this functionality for decades. In modern scripting languages, one often sees analogous functionality: the ability to embed expression evaluation inside a string (which one might or might not then call `eval` on, just as one might or might not use a Racket quote expression to build something for `eval`). This feature is sometimes called *interpolation* in scripting languages, but it is just *quasiquoting*.

CSE341: Programming Languages Spring 2019
Unit 7 Summary
Dan Grossman, University of Washington

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

Contents

Ruby Logistics **1**
Ruby Features Most Interesting for a PL Course **2**
The Rules of Class-Based OOP **3**
Objects, Classes, Methods, Variables, Etc. **3**
Visibility and Getters/Setters **6**
Some Syntax, Semantics, and Scoping To Get Used To **7**
Everything is an Object **7**
The Top-Level **8**
Class Definitions are Dynamic **8**
Duck Typing **8**
Arrays **9**
Passing Blocks **11**
Using Blocks **12**
The `Proc` Class **12**
Hashes and Ranges **13**
Subclassing and Inheritance **14**
Why Use Subclassing? **16**
Overriding and Dynamic Dispatch **17**
The Precise Definition of Method Lookup **19**
Dynamic Dispatch Versus Closures **20**
Implementing Dynamic Dispatch Manually in Racket **21**

Ruby Logistics

The course website provides installation and basic usage instructions for Ruby and its REPL (called `irb`), so that information is not repeated here. Note that for consistency we will require Ruby version 2.x.y (for any x and y), although this is for homework purposes – the concepts we will discuss do not depend on an exact version, naturally.

There is a great amount of free documentation for Ruby at `http://ruby-doc.org` and `http://www.ruby-lang.org/en/documentation/`. We also recommend, *Programming Ruby 1.9 & 2.0, The Pragmatic Programmers' Guide* although this book is not free. Because the online documentation is excellent, the other course materials may not describe in detail every language feature used in the lectures and homeworks

although it is also not our goal to make you hunt for things on purpose. In general, learning new language features and libraries is an important skill after some initial background to point you in the right direction.

Ruby Features Most Interesting for a PL Course

Ruby is a large, modern programming language with various features that make it popular. Some of these features are useful for a course on programming-language features and semantics, whereas others are not useful for our purposes even though they may be very useful in day-to-day programming. Our focus will be on object-oriented programming, dynamic typing, blocks (which are almost closures), and mixins. We briefly describe these features and some other things that distinguish Ruby here — if you have not seen an object-oriented programming language, then some of this overview will not make sense until after learning more Ruby.

- Ruby is a *pure object-oriented* language, which means *all* values in the language are objects. In Java, as an example, some values that are not objects are `null`, `13`, `true`, and `4.0`. In Ruby, every expression evaluates to an object.
- Ruby is *class-based*: Every object is an instance of a class. An object's class determines what methods an object has. (All code is in methods, which are like functions in the sense that they take arguments and return results.) You call a method "on" an object, e.g., `obj.m(3,4)` evaluates the variable `obj` to an object and calls its `m` method with arguments 3 and 4. Not all object-oriented languages are class-based; see, for example, JavaScript.
- Ruby has *mixins*: The next course-unit will describe mixins, which strike a reasonable compromise between multiple inheritance (like in C++) and interfaces (like in Java). Every Ruby class has one superclass, but it can include any number of mixins, which, unlike interfaces, can define methods (not just require their existence).
- Ruby is *dynamically typed*: Just as Racket allowed calling any function with any argument, Ruby allows calling any method on any object with any arguments. If the *receiver* (the object on which we call the method) does not define the method, we get a dynamic error.
- Ruby has *many dynamic features*: In addition to dynamic typing, Ruby allows instance variables (called fields in many object-oriented languages) to be added and removed from objects and it allows methods to be added and removed from classes while a program executes.
- Ruby has *convenient reflection*: Various built-in methods make it easy to discover at run-time properties about objects. As examples, every object has a method `class` that returns the object's class, and a method `methods` that returns an array of the object's methods.
- Ruby has *blocks* and *closures*: Blocks are almost like closures and are used throughout Ruby libraries for convenient higher-order programming. Indeed, it is rare in Ruby to use an explicit loop since collection classes like `Array` define so many useful iterators. Ruby also has fully-powerful closures for when you need them.
- Ruby is a *scripting language*: There is no precise definition of a what makes a language a scripting language. It means the language is engineered toward making it easy to write short programs, providing convenient access to manipulating files and strings (topics we will not discuss), and having less concern for performance. Like many scripting languages, Ruby does not require that you declare variables before using them and there are often many ways to say the same thing.
- Ruby is *popular for web applications*: The Ruby on Rails framework is a popular choice for developing the server side of modern web-sites.

Recall that, taken together, ML, Racket, and Ruby cover three of the four combinations of functional vs. object-oriented and statically vs. dynamically typed.

Our focus will be on Ruby's object-oriented nature, not on its benefits as a scripting language. We also will not discuss at all its support for building web applications, which is a main reason it is currently so popular. As an object-oriented language, Ruby shares much with Smalltalk, a language that has basically not changed since 1980. Ruby does have some nice additions, such as mixins.

Ruby is also a large language with a "why not" attitude, especially with regard to syntax. ML and Racket (and Smalltalk) adhere rather strictly to certain traditional programming-language principles, such as defining a small language with powerful features that programmers can then use to build large libraries. Ruby often takes the opposite view. For example, there are many different ways to write an if-expression.

The Rules of Class-Based OOP

Before learning the syntax and semantics of particular Ruby constructs, it is helpful to enumerate the "rules" that describe languages like Ruby and Smalltalk. Everything in Ruby is described in terms of *object-oriented programming*, which we abbreviate OOP, as follows:

1. All values (as usual, the result of evaluating expressions) are references to *objects*.
2. Given an object, code "communicates with it" by calling its *methods*. A synonym for calling a method is *sending a message*. (In processing such a message, an object is likely to send other messages to other objects, leading to arbitrarily sophisticated computations.)
3. Each object has its own private *state*. Only an object's methods can directly access or update this state.
4. Every object is an instance of a *class*.
5. An object's class determines the object's *behavior*. The class contains method definitions that dictate how an object handles method calls it receives.

While these rules are mostly true in other OOP languages like Java or C#, Ruby makes a more complete commitment to them. For example, in Java and C#, some values like numbers are not objects (violating rule 1) and there are ways to make object state publicly visible (violating rule 3).

Objects, Classes, Methods, Variables, Etc.

(See also the example programs posted with the lecture materials, not all of which are repeated here.)

Class and method definitions

Since every *object* has a *class*, we need to define classes and then create *instances* of them (an object of class `C` is an instance of `C`). (Ruby also predefines many classes in its language and standard library.) The basic syntax (we will add features as we go) for creating a class `Foo` with *methods* `m1`, `m2`, ... `mn` can be:

```
class Foo
   def m1
     ...
   end
```

```
  def m2 (x,y)
    ...
  end

  ...

  def mn z
    ...
  end
end
```

Class names must be capitalized. They include method definitions. A method can take any number of arguments, including 0, and we have a variable for each argument. In the example above, `m1` takes 0 arguments, `m2` takes two arguments, and `mn` takes 1 argument. Not shown here are method bodies. Like ML and Racket functions, a method implicitly returns its last expression. Like Java/C#/C++, you can use an explicit `return` statement to return immediately when helpful. (It is bad style to have a return at the end of your method since it can be implicit there.)

Method arguments can have defaults in which case a caller can pass fewer actual arguments and the remaining ones are filled in with defaults. If a method argument has a default, then all arguments to its right must also have a default. An example is:

```
  def myMethod (x,y,z=0,w="hi")
     ...
  end
```

Calling methods

The method call `e0.m(e1, ..., en)` evaluates `e0`, `e1`, ..., `en` to objects. It then calls the method `m` in the result of `e0` (as determined by the class of the result of `e0`), passing the results of `e1`, ..., `en` as arguments. As for syntax, the parentheses are optional. In particular, a zero-argument call is usually written `e0.m`, though `e0.m()` also works.

To call another method on the same object as the currently executing method, you can write `self.m(...)` or just `m(...)`. (Java/C#/C++ work the same way except they use the keyword `this` instead of `self`.)

In OOP, another common name for a method call is a *message send*. So we can say `e0.m e1` sends the result of `e0` the message `m` with the argument that is the result of `e1`. This terminology is "more object-oriented" — as a client, we do not care how the receiver (of the message) is implemented (e.g., with a method named `m`) as long as it can handle the message. As general terminology, in the call `e0.m args`, we call the result of evaluating `e0` the *receiver* (the object receiving the message).

Instance variables

An object has a class, which defines its methods. It also has *instance variables*, which hold values (i.e., objects). Many languages (e.g., Java) use the term *fields* instead of instance variables for the same concept. Unlike Java/C#/C++, our class definition does not indicate what instance variables an instance of the class will have. To add an instance variable to an object, you just assign to it: if the instance variable does not already exist, it is created. All instance variables start with an `@`, e.g., `@foo`, to distinguish them from variables local to a method.

Each object has its own instance variables. Instance variables are mutable. An expression (in a method body) can read an instance variable with an expression like `@foo` and write an instance variable with an expression `@foo = newValue`. Instance variables are private to an object. There is no way to directly access

an instance variable of any other object. So `@foo` refers to the `@foo` instance variable of the current object, i.e., `self.@foo` except `self.@foo` is *not* actually legal syntax.

Ruby also has class variables (which are like Java's static fields). They are written `@@foo`. Class variables are not private to an object. Rather, they are *shared* by all instances of the class, but are still not directly accessible from objects of different classes.

Constructing an object

To create a new instance of class `Foo`, you write `Foo.new(...)` where `(...)` holds some number of arguments (where, as with all method calls, the parentheses are optional and when there are zero or one arguments it is preferred to omit them). The call to `Foo.new` will create a new instance of `Foo` and then, before `Foo.new` returns, call the new object's `initialize` method with all the arguments passed to `Foo.new`. That is, the method `initialize` is special and serves the same role as constructors in other object-oriented languages.

Typical behavior for `initialize` is to create and initialize instance variables. In fact, the normal approach is for `initialize` always to create the same instance variables and for no other methods in the class to create instance variables. But Ruby does not require this and it may be useful on occasion to violate these conventions. Therefore, different instances of a class can have different instance variables.

Expressions and Local Variables

Most expressions in Ruby are actually method calls. Even `e1 + e2` is just syntactic sugar for `e1.+ e2`, i.e., call the `+` method on the result of `e1` with the result of `e2`. Another example is `puts e`, which prints the result of `e` (after calling its `to_s` method to convert it to a string) and then a newline. It turns out `puts` is a method in all objects (it is defined in class `Object` and all classes are subclasses of `Object` — we discuss subclasses later), so `puts e` is just `self.puts e`.

Not every expression is a method call. The most common other expression is some form of conditional. There are various ways to write conditionals; see the example code posted with the lecture materials. As discussed below, loop expressions are rare in Ruby code.

Like instance variables, variables local to a method do not have to be declared: The first time you assign to `x` in a method will create the variable. The scope of the variable is the entire method body. It is a run-time error to use a local variable that has not yet been defined. (In contrast, it is not a run-time error to use an instance variable that has not yet been defined. Instead you get back the `nil` object, which is discussed more below.)

Class Constants and Class Methods

A class constant is a lot like a class variable (see above) except that (1) it starts with a capital letter instead of `@@`, (2) you should not mutate it, and (3) it is publicly visible. Outside of an instance of class `C`, you can access a constant `Foo` of `C` with the syntax `C::Foo`. An example is `Math::PI`.[1]

A class method is like an ordinary method (called an instance method to distinguish from class methods) except (1) it does not have access to any of the instance variables or instance methods of an instance of the class and (2) you can call it from outside the class `C` where it is defined with `C.method_name args`. There are various ways to define a class method; the most common is the somewhat hard-to-justify syntax:

```
def self.method_name args
   ...
end
```

Class methods are called static methods in Java and C#.

[1] Actually, `Math` is a module, not a class, so this is not technically an example, but modules can also have constants.

Visibility and Getters/Setters

As mentioned above, instance variables are private to an object: only method calls with *that object* as the receiver can read or write the fields. As a result, the syntax is `@foo` and the self-object is implied. Notice even other instances of the same class cannot access the instance variables. This is quite object-oriented: you can interact with another object only by sending it messages.

Methods can have different *visibilities*. The default is `public`, which means any object can call the method. There is also `private`, which, like with instance variables, allows only the object itself to call the method (from other methods in the object). In-between is `protected`: A protected method can be called by any object that is an instance of the same class or any subclass of the class.

There are various ways to specify the visibility of a method. Perhaps the simplest is within the class definition you can put `public`, `private`, or `protected` between method definitions. Reading top-down, the most recent visibility specified holds for all methods until the next visibility is specified. There is an implicit `public` before the first method in the class.

To make the contents of an instance variable available and/or mutable, we can easily define getter and setter methods, which by convention we can give the same name as the instance variable. For example:

```
def foo
  @foo
end

def foo= x
  @foo = x
end
```

If these methods are public, now any code can access the instance variable `@foo` indirectly, by calling `foo` or `foo=`. It sometimes makes sense to instead make these methods `protected` if only other objects of the same class (or subclasses) should have access to the instance variables.

As a cute piece of syntactic sugar, when calling a method that ends in a `=` character, you can have spaces before the `=`. Hence you can write `e.foo = bar` instead of `e.foo= bar`.

The advantage of the getter/setter approach is it remains an implementation detail that these methods are implemented as getting and setting an instance variable. We, or a subclass implementer, could change this decision later without clients knowing. We can also omit the setter to ensure an instance variable is not mutated except perhaps by a method of the object.

As an example of a "setter method" that is not *actually* a setter method, a class could define:

```
def celsius_temp= x
  @kelvin_temp = x + 273.15
end
```

A client would likely imagine the class has a `@celsius_temp` instance variable, but in fact it (presumably) does not. This is a good abstraction that allows the implementation to change.

Because getter and setter methods are so common, there is shorter syntax for defining them. For example, to define getters for instance variables `@x`, `@y`, and `@z` and a setter for `@x`, the class definition can just include:

```
attr_reader :y, :z # defines getters
attr_accessor :x # defines getters and setters
```

A final syntactic detail: If a method `m` is private, you can only call it as `m` or `m(args)`. A call like `x.m` or `x.m(args)` would break visibility rules. A call like `self.m` or `self.m(args)` would not break visibility, but still is not allowed.

Some Syntax, Semantics, and Scoping To Get Used To

Ruby has a fair number of quirks that are often convenient for quickly writing useful programs but may take some getting used to. Here are some examples; you will surely discover more.

- There are several forms of conditional expressions, including `e1 if e2` (all on one line), which evaluates `e1` only if `e2` is true (i.e., it reads right-to-left).
- Newlines are often significant. For example, you can write

  ```
  if e1
    e2
  else
    e3
  end
  ```

  But if you want to put this all on one line, then you need to write `if e1 then e2 else e3 end`. Note, however, indentation is never significant (only a matter of style).
- Conditionals can operate on any object and treat every object as "true" with *two* exceptions: `false` and `nil`.
- As discussed above, you can define a method with a name that ends in `=`, for example:

  ```
  def foo= x
    @blah = x * 2
  end
  ```

  As expected, you can write `e.foo=(17)` to change `e`'s `@blah` instance variable to be 34. Better yet, you can adjust the parentheses and spacing to write `e.foo = 17`. This is just syntactic sugar. It "feels" like an assignment statement, but it is really a method call. Stylistically you do this for methods that mutate an object's state in some "simple" way (like setting a field).
- Where you write `this` in Java/C#/C++, you write `self` in Ruby.
- Remember variables (local, instance, or class) get automatically created by assignment, so if you misspell a variable in an assignment, you end up just creating a different variable.

Everything is an Object

Everything is an object, including numbers, booleans, and `nil` (which is often used like `null` in Java). For example, `-42.abs` evaluates to `42` because the `Fixnum` class defines the method `abs` to compute the absolute value and `-42` is an instance of `Fixnum`. (Of course, this is a silly expression, but `x.abs` where `x` currently holds `-42` is reasonable.)

All objects have a `nil?` method, which the class of `nil` defines to return `true` but other classes define to return `false`. Like in ML and Racket, every expression produces a result, but when no particular result

makes sense, `nil` is preferred style (much like ML's `()` and Racket's void-object). That said, it is often convenient for methods to return `self` so that subsequent method calls to the same object can be put together. For example, if the `foo` method returns `self`, then you can write `x.foo(14).bar("hi")` instead of

```
x.foo(14)
x.bar("hi")
```

There are many methods to support *reflection* — learning about objects and their definition during program execution — that are defined for all objects. For example, the method `methods` returns an array of the names of the methods defined on an object and the method `class` returns the class of the object.[2] Such reflection is occasionally useful in writing flexible code. It is also useful in the REPL or for debugging.

The Top-Level

You can define methods, variables, etc. outside of an explicit class definition. The methods are implicitly added to class `Object`, which makes them available from within any object's methods. Hence all methods are really part of some class.[3]

Top-level expressions are evaluated in order when the program runs. So instead of Ruby specifying a main class and method with a special name (like `main`), you can just create an object and call a method on it at top-level.

Class Definitions are Dynamic

A Ruby program (or a user of the REPL) can change class definitions while a Ruby program is running. Naturally this affects all users of the class. Perhaps surprisingly, it even affects instances of the class that have already been created. That is, if you create an instance of `Foo` and then add or delete methods in `Foo`, then the already-created object "sees" the changes to its behavior. After all, every object has a class and the (current) class (definition) defines an object's behavior.

This is usually dubious style because it breaks abstractions, but it leads to a simpler language definition: defining classes and changing their definitions is just a run-time operation like everything else. It can certainly break programs: If I change or delete the `+` method on numbers, I would not expect many programs to keep working correctly. It can be useful to add methods to existing classes, especially if the designer of the class did not think of a useful helper method.

The syntax to add or change methods is particularly simple: Just give a class definition including method definitions for a class that is already defined. The method definitions either replace definitions for methods previously defined (with the same name method name) or are added to the class (if no method with the name previously existed).

Duck Typing

Duck typing refers to the expression, "If it walks like a duck and quacks like a duck, then it's a duck" though a better conclusion might be, "then there is no reason to concern yourself with the possibility that it might

[2] This class is itself just another object. Yes, even classes are objects.
[3] This is not entirely true because modules are not classes.

not be a duck." In Ruby, this refers to the idea that the class of an object (e.g., "Duck") passed to a method is not important so long as the object can respond to all the messages it is expected to (e.g., "walk to x" or "quack now").

For example, consider this method:

```
def mirror_update pt
  pt.x = pt.x * -1
end
```

It is natural to view this as a method that must take an instance of a particular class `Point` (not shown here) since it uses methods `x` and `x=` defined in it. And the `x` getter must return a number since the result of `pt.x` is sent the `*` message with `-1` for multiplication.

But this method is more generally useful. It is not necessary for `pt` to be an instance of `Point` provided it has methods `x` and `x=`.

Moreover, the `x` and `x=` methods need not be a getter and setter for an instance variable `@x`.

Even more generally, we do not need the `x` method to return a number. It just has to return some object that can respond to the `*` message with argument `-1`.

Duck typing can make code more reusable, allowing clients to make "fake ducks" and still use your code. In Ruby, duck typing basically "comes for free" as long you do not explicitly check that arguments are instances of particular classes using methods like `instance_of?` or `is_a?` (discussed below when we introduce subclassing).

Duck typing has disadvantages. The most lenient specification of how to use a method ends up describing the whole implementation of a method, in particular what messages it sends to what objects. If our specification reveals all that, then almost no variant of the implementation will be equivalent. For example, if we know `i` is a number (and ignoring clients redefining methods in the classes for numbers), then we can replace `i+i` with `i*2` or `2*i`. But if we just assume `i` can receive the `+` message with itself as an argument, then we cannot do these replacements since `i` may not have a `*` method (breaking `i*2`) or it may not be the sort of object that `2` expects as an argument to `*` (breaking `2*i`).

Arrays

The `Array` class is *very* commonly used in Ruby programs and there is special syntax that is often used with it. Instances of `Array` have all the uses that arrays in other programming languages have — and much, much more. Compared to arrays in Java/C#/C/etc., they are much more flexible and dynamic with fewer operations being errors. The trade-off is they can be less efficient, but this is usually not a concern for convenient programming in Ruby. In short, all Ruby programmers are familiar with Ruby arrays because they are the standard choice for any sort of collection of objects.

In general, an array is a mapping from numbers (the indices) to objects. The syntax `[e1,e2,e3,e4]` creates a new array with four objects in it: The result of `e1` is in index 0, the result of `e2` is in index 1, and so on. (Notice the indexing starts at 0.) There are other ways to create arrays. For example, `Array.new(x)` creates an array of length `x` with each index initially mapped to `nil`. We can also pass blocks (see below for what blocks actually are) to the `Array.new` method to initialize array elements. For example, `Array.new(x) { 0 }` creates an array of length `x` with all elements initialized to `0` and `Array.new(5) {|i| -i }` creates the array `[0,-1,-2,-3,-4]`.

The syntax for getting and setting array elements is similar to many other programming languages: The expression `a[i]` gets the element in index `i` of the array referred to by `a` and `a[i] = e` sets the same array

index. As you might suspect in Ruby, we are really just calling methods on the `Array` class when we use this syntax.

Here are some simple ways Ruby arrays are more dynamic and less error-causing than you might expect compared to other programming languages:

- As usual in a dynamically typed language, an array can hold objects that are instances of different classes, for example `[14, "hi", false, 34]`.
- Negative array indices are interpreted from the *end* of the array. So `a[-1]` retrieves the last element in the array `a`, `a[-2]` retrieves the second-to-last element, etc.
- There are no array-bounds errors. For the expression `a[i]`, if `a` holds fewer than `i+1` objects, then the result will just be `nil`. Setting such an index is even more interesting: For `a[i]=e`, if `a` holds fewer than `i+1` objects, then the array will *grow dynamically* to hold `i+1` objects, the last of which will be the result of `e`, with the right number of `nil` objects between the old last element and the new last element.
- There are *many* methods and operations defined in the standard library for arrays. If the operation you need to perform on an array is at all general-purpose, peruse the documentation since it is surely already provided. As two examples, the `+` operator is defined on arrays to mean concatenation (a new array where all of the left-operand elements precede all of the right-operand elements), and the `|` operator is like the `+` operator except it removes all duplicate elements from the result.

In addition to all the conventional uses for arrays, Ruby arrays are also often used where in other languages we would use other constructs for tuples, stacks, or queues. Tuples are the most straightforward usage. After all, given dynamic typing and less concern for efficiency, there is little reason to have separate constructs for tuples and arrays. For example, for a triple, just use a 3-element array.

For stacks, the `Array` class defines convenient methods `push` and `pop`. The former takes an argument, grows the array by one index, and places the argument at the new last index. The latter shrinks the array by one index and returns the element that was at the old last index. Together, this is exactly the last-in-first-out behavior that defines the behavior of a stack. (How this is implemented in terms of actually growing and shrinking the underlying storage for the elements is of concern only in the implementation of `Array`.)

For queues, we can use `push` to add elements as just described and use the `shift` method to dequeue elements. The `shift` method returns the object at index 0 of the array, removes it from the array, and shifts all the other elements down one index, i.e., the object (if any) previously at index 1 is now at index 0, etc. Though not needed for simple queues, `Array` also has an `unshift` method that is like `push` except it puts the new object at index 0 and moves all other objects up by 1 index (growing the array size by 1).

Arrays are even more flexible than described here. For example, there are operations to replace any sequence of array elements with the elements of any other array, even if the other array has a different length than the sequence being replaced (hence changing the length of the array).

Overall, this flexible treatment of array sizes (growing and shrinking) is different from arrays in some other programming languages, but it is consistent with treating arrays as maps from numeric indices to objects.

What we have not shown so far are operations that perform some computation using all the contents of an array, such as mapping over the elements to make a new array, or computing a sum of them. That is because the Ruby idioms for such computations use *blocks*, which we introduce next.

Passing Blocks

While Ruby has while loops and for loops not unlike Java, most Ruby code does not use them. Instead, many classes have methods that take *blocks*. These blocks are *almost* closures. For example, integers have a `times` method that takes a block and executes it the number of times you would imagine. For example,

```
x.times { puts "hi" }
```

prints `"hi"` 3 times if `x` is bound to 3 in the environment.

Blocks are closures in the sense that they can refer to variables in scope where the block is defined. For example, after this program executes, `y` is bound to 10:

```
y = 7
[4,6,8].each { y += 1 }
```

Here `[4,6,8]` is an array with with 3 elements. Arrays have a method `each` that takes a block and executes it once for each element. Typically, however, we want the block to be passed each array element. We do that like this, for example to sum an array's elements and print out the running sum at each point:

```
sum = 0
[4,6,8].each { |x|
  sum += x
  puts sum
}
```

Blocks, surprisingly, are not objects. You cannot pass them as "regular" arguments to a method. Rather, any method can be passed either 0 or 1 blocks, separate from the other arguments. As seen in the examples above, the block is just put to the right of the method call. It is also after any other "regular" arguments. For example, the `inject` method is like the `fold` function we studied in ML and we can pass it an initial accumulator as a regular argument:

```
sum = [4,6,8].inject(0) { |acc,elt| acc + elt }
```

(It turns out the initial accumulator is optional. If omitted, the method will use the array element in index 0 as the initial accumulator.)

In addition to the braces syntax shown here, you can write a block using `do` instead of `{` and `end` instead of `}`. This is generally considered better style for blocks more than one line long.

When calling a method that takes a block, you should know how many arguments will be passed to the block when it is called. For the `each` method in `Array`, the answer is 1, but as the first example showed, you can ignore arguments if you have no need for them by omitting the `|...|`.

Many collections, including arrays, have a variety of block-taking methods that look very familiar to functional programmers, including `map`. As another example, the `select` method is like the function we called `filter`. Other useful *iterators* include `any?` (returns true if the block returns true for any element of the collection), `all?` (returns true if the block returns true for every element of the collection), and several more.

Using Blocks

While many uses of blocks involve calling methods in the standard library, you can also define your own methods that take blocks. (The large standard library just makes it somewhat rare to need to do this.)

You can pass a block to *any* method. The method body calls the block using the `yield` keyword. For example, this code prints `"hi"` 3 times:

```
def foo x
  if x
    yield
  else
    yield
    yield
  end
end
foo true { puts "hi" }
foo false { puts "hi" }
```

To pass arguments to a block, you put the arguments after the `yield`, e.g., `yield 7` or `yield(8,"str")`.

Using this approach, the fact that a method may expect a block is implicit; it is just that its body might use `yield`. An error will result if `yield` is used and no block was passed. The behavior when the block and the `yield` disagree on the number of arguments is somewhat flexible and not described in full detail here. A method can use the `block_given?` primitive to see if the caller provided a block. You are unlikely to use this method often: If a block is needed, it is conventional just to assume it is given and have `yield` fail if it is not. In situations where a method may or may not expect a block, often other regular arguments determine whether a block should be present. If not, then `block_given?` is appropriate.

Here is a recursive method that counts how many times it calls the block (with increasing numbers) before the block returns a true result.

```
def count i
    if yield i
      1
    else
      1 + (count(i+1) {|x| yield x})
    end
  end
```

The odd thing is that there is no direct way to pass the caller's block as the callee's block argument. But we can create a new block `{|x| yield x}` and the lexical scope of the `yield` in its body will do the right thing. If blocks were actually function closures that we could pass as objects, then this would be unnecessary function wrapping.

The `Proc` Class

Blocks are not quite closures because they are not objects. We cannot store them in a field, pass them as a regular method argument, assign them to a variable, put them in an array, etc. (Notice in ML and Racket, we could do the equivalent things with closures.) Hence we say that blocks are not "first-class values" because a first-class value is something that can be passed and stored like anything else in the language.

However, Ruby has "real" closures too: The class `Proc` has instances that are closures. The method `call` in `Proc` is how you apply the closure to arguments, for example `x.call` (for no arguments) or `x.call(3,4)`.

To make a `Proc` out of a block, you can write `lambda { ... }` where `{ ... }` is any block. Interestingly, `lambda` is not a keyword. It is just a method in class `Object` (and every class is a subclass of `Object`, so `lambda` is available everywhere) that creates a `Proc` out of a block it is passed. You can define your own methods that do this too; consult the documentation for the syntax to do this.

Usually all we need are blocks, such as in these examples that pass blocks to compute something about an array:

```
a = [3,5,7,9]
b = a.map {|x| x + 1}
i = b.count {|x| x >= 6}
```

But suppose we wanted to create an array of blocks, i.e., an array where each element was something we could "call" with a value. You cannot do this in Ruby because arrays hold objects and blocks are not objects. So this is an error:

```
c = a.map {|x| {|y| x >= y} } # wrong, a syntax error
```

But we can use `lambda` to create an array of instances of `Proc`:

```
c = a.map {|x| lambda {|y| x >= y} }
```

Now we can send the `call` message to elements of the `c` array:

```
c[2].call 17
j = c.count {|x| x.call(5) }
```

Ruby's design is an interesting contrast from ML and Racket, which just provide full closures as the natural choice. In Ruby, blocks are more convenient to use than `Proc` objects and suffice in most uses, but programmers still have `Proc` objects when needed. Is it better to distinguish blocks from closures and make the more common case easier with a less powerful construct, or is it better just to have one general fully powerful feature?

Hashes and Ranges

The `Hash` and `Range` classes are two standard-library classes that are also very common but probably a little less common than arrays. Like arrays, there is special built-in syntax for them. They are also similar to arrays and support many of the same iterator methods, which helps us re-enforce the concept that "how to iterate" can be separated from "what to do while iterating."

A hash is like an array except the mapping is not from numeric indices to objects. Instead, the mapping is from *(any) objects* to objects. If a maps to b, we call a a *key* and b a *value*. Hence a hash is a collection that maps a set of keys (all keys in a hash are distinct) to values, where the keys and values are just objects. We can create a hash with syntax like this:

```
{"SML" => 7, "Racket" => 12, "Ruby" => 42}
```

As you might expect, this creates a hash with keys that here are strings. It is also common (and more efficient) to use Ruby's symbols for hash keys as in:

```
{:sml => 7, :racket => 12, :ruby => 42}
```

We can get and set values in a hash using the same syntax as for arrays, where again the key can be anything, such as:

```
h1["a"] = "Found A"
h1[false] = "Found false"
h1["a"]
h1[false]
h1[42]
```

There are many methods defined on hashes. Useful ones include `keys` (return an array of all keys), `values` (similar for values), and `delete` (given a key, remove it and its value from the hash). Hashes also support many of the same iterators as arrays, such as `each` and `inject`, but some take the keys and the values as arguments, so consult the documentation.

A range represents a contiguous sequence of numbers (or other things, but we will focus on numbers). For example `1..100` represents the integers 1, 2, 3, ..., 100. We could use an array like `Array.new(100) {|i| i}`, but ranges are more efficiently represented and, as seen with `1..100`, there is more convenient syntax to create them. Although there are often better iterators available, a method call like `(0..n).each {|i| e}` is a lot like a for-loop from 0 to n in other programming languages.

It is worth emphasizing that duck typing lets us use ranges in many places where we might naturally expect arrays. For example, consider this method, which counts how many elements of `a` have squares less than 50:

```
def foo a
  a.count {|x| x*x < 50}
end
```

We might naturally expect `foo` to take arrays, and calls like `foo [3,5,7,9]` work as expected. But we can pass to `foo` any object with a `count` method that expects a block taking one argument. So we can also do `foo (2..10)`, which evaluates to 6.

Subclassing and Inheritance

Basic Idea and Terminology

Subclassing is an essential feature of class-based OOP. If class `C` is a subclass of `D`, then every instance of `C` is also an instance of `D`. The definition of `C` *inherits* the methods of `D`, i.e., they are part of `C`'s definition too. Moreover, `C` can *extend* by defining new methods that `C` has and `D` does not. And it can *override* methods, by changing their definition from the inherited definition. In Ruby, this is much like in Java. In Java, a subclass also inherits the field definitions of the superclass, but in Ruby fields (i.e., instance variables) are not part of a class definition because each object instance just creates its own instance variables.

Every class in Ruby except `Object` has one superclass.[4] The classes form a tree where each node is a class and the parent is its superclass. The `Object` class is the root of the tree. In class-based languages, this is

[4] Actually, the superclass of `Object` is `BasicObject` and `BasicObject` has no superclass, but this is not an important detail, so we will ignore it.

called the *class hierarchy*. By the definition of subclassing, a class has all the methods of all its ancestors in the tree (i.e., all nodes between it and the root, inclusive), subject to overriding.

Some Ruby Specifics

- A Ruby class definition specifies a superclass with `class C < D ... end` to define a new class `C` with superclass `D`. Omitting the `< D` implies `< Object`, which is what our examples so far have done.
- Ruby's built-in methods for reflection can help you explore the class hierarchy. Every object has a `class` method that returns the class of the object. Consistently, if confusingly at first, a class is itself an object in Ruby (after all, every value is an object). The class of a class is `Class`. This class defines a method `superclass` that returns the superclass.
- Every object also has methods `is_a?` and `instance_of?`. The method `is_a?` takes a class (e.g., `x.is_a? Integer`) and returns true if the receiver is an instance of `Integer` or any (transitive) subclass of `Integer`, i.e., if it is below `Integer` in the class hierarchy. The method `instance_of?` is similar but returns true only if the receiver is an instance of the class exactly, not a subclass. (Note that in Java the primitive `instanceof` is analogous to Ruby's `is_a?`.)

Using methods like `is_a?` and `instanceof` is "less object-oriented" and therefore often not preferred style. They are in conflict with duck typing.

A First Example: Point and ColorPoint

Here are definitions for simple classes that describe simple two-dimensional points and a subclass that adds a color (just represented with a string) to instances.

```
class Point
  attr_accessor :x, :y
  def initialize(x,y)
    @x = x
    @y = y
  end
  def distFromOrigin
    Math.sqrt(@x * @x  + @y * @y)
  end
  def distFromOrigin2
    Math.sqrt(x * x + y * y)
  end
end
class ColorPoint < Point
  attr_accessor :color
  def initialize(x,y,c="clear")
    super(x,y)
    @color = c
  end
end
```

There are many ways we could have defined these classes. Our design choices here include:

- We make the `@x`, `@y`, and `@color` instance variables mutable, with public getter and setter methods.
- The default "color" for a `ColorPoint` is `"clear"`.

- For pedagogical purposes revealed below, we implement the distance-to-the-origin in two different ways. The `distFromOrigin` method accesses instance variables directly whereas `distFromOrigin2` uses the getter methods on `self`. Given the definition of `Point`, both will produce the same result.

The `initialize` method in `ColorPoint` uses the `super` keyword, which allows an overriding method to call the method of the same name in the superclass. This is not required when constructing Ruby objects, but it is often desired.

Why Use Subclassing?

We now consider the style of defining colored-points using a subclass of the class `Point` as shown above. It turns out this is good OOP style in this case. Defining `ColorPoint` is good style because it allows us to reuse much of our work from `Point` and it makes sense to treat any instance of `ColorPoint` as though it "is a" `Point`.

But there are several alternatives worth exploring because subclassing is often overused in object-oriented programs, so it is worth considering at program-design time whether the alternatives are better than subclassing.

First, in Ruby, we can extend and modify classes with new methods. So we could simply change the `Point` class by replacing its `initialize` method and adding getter/setter methods for `@color`. This would be appropriate only if every `Point` object, including instances of all other subclasses of `Point`, should have a color or at least having a color would not mess up anything else in our program. Usually modifying classes is not a modular change — you should do it only if you know it will not negatively affect anything in the program using the class.

Second, we could just define `ColorPoint` "from scratch," copying over (or retyping) the code from `Point`. In a dynamically typed language, the difference in *semantics* (as opposed to style) is small: instances of `ColorPoint` will now return false if sent the message `is_a?` with argument `Point`, but otherwise they will work the same. In languages like Java/C#/C++, superclasses have effects on static typing. One advantage of *not* subclassing `Point` is that any later changes to `Point` will not affect `ColorPoint` — in general in class-based OOP, one has to worry about how changes to a class will affect any subclasses.

Third, we could have `ColorPoint` be a subclass of `Object` but have it contain an instance variable, call it `@pt`, holding an instance of `Point`. Then it would need to define all of the methods defined in `Point` to forward the message to the object in `@pt`. Here are two examples, omitting all the other methods (`x=`, `y`, `y=`, `distFromOrigin`, `distFromOrigin2`):

```
def initialize(x,y,c="clear")
  @pt = Point.new(x,y)
  @color = c
end
def x
  @pt.x # forward the message to the object in @pt
end
```

This approach is bad style since again subclassing is shorter and we want to treat a `ColorPoint` as though it "is a" `Point`. But in general, many programmers in object-oriented languages overuse subclassing. In situations where you are making a new kind of data that includes a pre-existing kind of data *as a separate sub-part of it*, this instance-variable approach is better style.

Overriding and Dynamic Dispatch

Now let's consider a different subclass of `Point`, which is for three-dimensional points:

```
class ThreeDPoint < Point
  attr_accessor :z
  def initialize(x,y,z)
    super(x,y)
    @z = z
  end
  def distFromOrigin
    d = super
    Math.sqrt(d * d + @z * @z)
  end
  def distFromOrigin2
    d = super
    Math.sqrt(d * d + z * z)
  end
end
```

Here, the code-reuse advantage is limited to inheriting methods `x`, `x=`, `y`, and `y=`, as well as using other methods in `Point` via `super`. Notice that in addition to overriding `initialize`, we used overriding for `distFromOrigin` and `distFromOrigin2`.

Computer scientists have been arguing for decades about whether this subclassing is good style. On the one hand, it does let us reuse quite a bit of code. On the other hand, one could argue that a `ThreeDPoint` is *not* conceptually a (two-dimensional) `Point`, so passing the former when some code expects the latter could be inappropriate. Others say a `ThreeDPoint` is a `Point` because you can "think of it" as its projection onto the plane where `z` equals 0. We will not resolve this legendary argument, but you should appreciate that often subclassing is bad/confusing style even if it lets you reuse some code in a superclass.

The argument against subclassing is made stronger if we have a method in `Point` like `distance` that takes another (object that behaves like a) `Point` and computes the distance between the argument and `self`. If `ThreeDPoint` wants to override this method with one that takes another (object that behaves like a) `ThreeDPoint`, then `ThreeDPoint` instances will *not* act like `Point` instances: their `distance` method will fail when passed an instance of `Point`.

We now consider a *much* more interesting subclass of `Point`. Instances of this class `PolarPoint` behave equivalently to instances of `Point` except for the arguments to `initialize`, but instances use an internal representation in terms of polar coordinates (radius and angle):

```
class PolarPoint < Point
  def initialize(r,theta)
    @r = r
    @theta = theta
  end
  def x
    @r * Math.cos(@theta)
  end
  def y
    @r * Math.sin(@theta)
  end
```

```
  def x= a
    b = y # avoids multiple calls to y method
    @theta = Math.atan(b / a)
    @r = Math.sqrt(a*a + b*b)
    self
  end
  def y= b
    a = y # avoid multiple calls to y method
    @theta = Math.atan(b / a)
    @r = Math.sqrt(a*a + b*b)
    self
  end
  def distFromOrigin
    @r
  end
  # distFromOrigin2 already works!!
end
```

Notice instances of `PolarPoint` do not have instance variables `@x` and `@y`, but the class does override the `x`, `x=`, `y`, and `y=` methods so that clients cannot tell the implementation is different (modulo round-off of floating-point numbers): they can use instances of `Point` and `PolarPoint` interchangeably. A similar example in Java would still have fields from the superclass, but would not use them. The advantage of `PolarPoint` over `Point`, which admittedly is for sake of example, is that `distFromOrigin` is simpler and more efficient.

The key point of this example is that **the subclass does not override `distFromOrigin2`, but the inherited method works correctly**. To see why, consider the definition in the superclass:

```
  def distFromOrigin2
    Math.sqrt(x * x + y * y)
  end
```

Unlike the definition of `distFromOrigin`, this method uses other method calls for the arguments to the multiplications. Recall this is just syntactic sugar for:

```
  def distFromOrigin2
    Math.sqrt(self.x() * self.x() + self.y() * self.y())
  end
```

In the superclass, this can seem like an unnecessary complication since `self.x()` is just a method that returns `@x` and methods of `Point` can access `@x` directly, as `distFromOrigin` does.

However, overriding methods `x` and `y` in a subclass of `Point` changes how `distFromOrigin2` behaves in instances of the subclass. Given a `PolarPoint` instance, its `distFromOrigin2` method is defined with the code above, but when called, `self.x` and `self.y` will call the methods defined in `PolarPoint`, not the methods defined in `Point`.

This semantics goes by many names, including *dynamic dispatch*, *late binding*, and *virtual method calls*. There is nothing quite like it in functional programming, since the way `self` is treated in the environment is special, as we discuss in more detail next.

The Precise Definition of Method Lookup

The purpose of this discussion is to consider the semantics of object-oriented language constructs, particularly calls to methods, as carefully as we have considered the semantics of functional language constructs, particularly calls to closures. As we will see, the key distinguishing feature is what `self` is bound to in the environment when a method is called. The correct definition is what we call *dynamic dispatch*.

The essential question we will build up to is given a call `e0.m(e1,e2,...en)`, what are the rules for "looking up" what method definition `m` we call, which is a non-trivial question in the presence of overriding. But first, let us notice that in general such questions about how we "look up" something are often essential to the semantics of a programming language. For example, in ML and Racket, the rules for looking up variables led to lexical scope and the proper treatment of function closures. And in Racket, we had three different forms of let-expressions exactly because they have different semantics for how to look up variables in certain subexpressions.

In Ruby, the variable-lookup rules for local variables in methods and blocks are not too different from in ML and Racket despite some strangeness from variables not being declared before they are used. But we also have to consider how to "look up" instance variables, class variables, and methods. In all cases, the answer depends on the object bound to `self` — and `self` is treated specially.

In any environment, `self` maps to some object, which we think of as the "current object" — the object currently executing a method. To look up an instance variable `@x`, we use the object bound to `self` – each object has its own state and we use `self`'s state. To look up a class variable `@@x`, we just use the state of the object bound to `self.class` instead. To look up a method `m` for a method call is more sophisticated...

In class-based object-oriented languages like Ruby, the rule for evaluating a method call like `e0.m(e1,...,en)` is:

- Evaluate `e0`, `e1`, ..., `en` to values, i.e., objects `obj0`, `obj1`, ..., `objn`.
- Get the class of `obj0`. Every object "knows its class" at run-time. Think of the class as part of the state of `obj0`.
- Suppose `obj0` has class `A`. If `m` is defined in `A`, call that method. Otherwise recur with the superclass of `A` to see if it defines `m`. Raise a "method missing" error if neither `A` nor any of its superclasses define `m`. (Actually, in Ruby the rule is actually to instead call a method called `method_missing`, which any class can define, so we again start looking in `A` and then its superclass. But most classes do not define `method_missing` and the definition of it in `Object` raises the error we expect.)
- We have now found the method to call. If the method has *formal arguments* (i.e., argument names or parameters) `x1`, `x2`, ..., `xn`, then the environment for evaluating the body will map `x1` to `obj1`, `x2` to `obj2`, etc. But there is one more thing that is the essence of object-oriented programming and has no real analogue in functional programming: We always have `self` in the environment. **While evaluating the method body, `self` is bound to `obj0`, the object that is the "receiver" of the message.**

The binding of `self` in the callee as described above is what is meant by the synonyms "late-binding," "dynamic dispatch," and "virtual method calls." It is central to the semantics of Ruby and other OOP languages. It means that when the body of `m` calls a method on `self` (e.g., `self.someMethod 34` or just `someMethod 34`), we use the class of `obj0` to resolve `someMethod`, *not necessarily* the class of the method we are executing. This is why the `PolarPoint` class described above works as it does.

There are several important comments to make about this semantics:

- Ruby's mixins complicate the lookup rules a bit more, so the rules above are actually simplified by ignoring mixins. When we study mixins, we will revise the method-lookup semantics accordingly.
- This semantics is quite a bit more complicated than ML/Racket function calls. It may not seem that way if you learned it first, which is common because OOP and dynamic dispatch seem to be a focus in many introductory programming courses. But it is truly more complicated: we have to treat the notion of `self` differently from everything else in the language. Complicated does not necessarily mean it is inferior or superior; it just means the language definition has more details that need to be described. This semantics has clearly proved useful to many people.
- Java and C# have significantly more complicated method-lookup rules. They do have dynamic dispatch as described here, so studying Ruby should help understand the semantics of method lookup in those languages. But they *also* have *static overloading*, in which classes can have multiple methods with the same name but taking different types (or numbers) of arguments. So we need to not just find *some* method with the right name, but we have to find one that *matches* the types of the arguments at the call. Moreover, multiple methods might match and the language specifications have a long list of complicated rules for finding the *best* match (or giving a type error if there is no best match). In these languages, one method overrides another only if its arguments have the same type and number. None of this comes up in Ruby where "same method name" always means overriding and we have no static type system. In C++, there are even more possibilities: we have static overloading and different forms of methods that either do or do not support dynamic dispatch.

Dynamic Dispatch Versus Closures

To understand how dynamic dispatch differs from the lexical scope we used for function calls, consider this simple ML code that defines two mutually recursive functions:

```
fun even x = if x=0 then true else odd (x-1)
and odd  x = if x=0 then false else even (x-1)
```

This creates two closures that both have the other closure in their environment. If we later shadow the `even` closure with something else, e.g.,

```
fun even x = false
```

that will *not* change how `odd` behaves. When `odd` looks up `even` in the environment where `odd` was defined, it will get the function on the first line above. That is "good" for understanding how `odd` works *just from looking where is defined*. On the other hand, suppose we wrote a better version of `even` like:

```
fun even x = (x mod 2) = 0
```

Now our `odd` is not "benefiting from" this optimized implementation.

In OOP, we can use (abuse?) subclassing, overriding, and dynamic dispatch to change the behavior of `odd` by overriding `even`:

```
class A
  def even x
     if x==0 then true else odd(x-1) end
  end
```

```
  def odd x
     if x==0 then false else even(x-1) end
  end
end
class B < A
  def even x # changes B's odd too!
     x % 2 == 0
  end
end
```

Now `(B.new.odd 17)` will execute faster because `odd`'s call to `even` will resolve to the method in `B` – all because of what `self` is bound to in the environment. While this is certainly convenient in the short example above, it has real drawbacks. We cannot look at one class (`A`) and know how calls to the code there will behave. In a subclass, what if someone overrode `even` and did not know that it would change the behavior of `odd`? Basically, any calls to methods that might be overridden need to be thought about very carefully. It is likely often better to have private methods that cannot be overridden to avoid problems. Yet overriding and dynamic dispatch is the biggest thing that distinguishes object-oriented programming from functional programming.

Implementing Dynamic Dispatch Manually in Racket

Let's now consider *coding up* objects and dynamic dispatch in Racket using nothing more than pairs and functions.[5] This serves two purposes:

- It demonstrates that one language's *semantics* (how the primitives like message send work in the language) can typically be coded up as an *idiom* (simulating the same behavior via some helper functions) in another language. This can help you be a better programmer in different languages that may not have the features you are used to.
- It gives a lower-level way to understand how dynamic dispatch "works" by seeing how we would do it manually in another language. An interpreter for an object-oriented language would have to do something similar for automatically evaluating programs in the language.

Also notice that we did an analogous exercise to better understand closures earlier in the course: We showed how to get the effect of closures in Java using objects and interfaces or in C using function pointers and explicit environments.

Our approach will be different from what Ruby (or Java for that matter) actually does in these ways:

- Our objects will just contain a list of fields and a list of methods. This is not "class-based," in which an object would have a list of fields and a class-name and then the class would have the list of methods. We could have done it that way instead.
- Real implementations are more efficient. They use better data structures (based on arrays or hashtables) for the fields and methods rather than simple association lists.

Nonetheless, the key ideas behind how you implement dynamic dispatch still come through. By the way, we are wise to do this in Racket rather than ML, where the types would get in our way. In ML, we would likely

[5]Though we did not study it, Racket has classes and objects, so you would not actually want to do this in Racket. The point is to understand dynamic dispatch by manually coding up the same idea.

end up using "one big datatype" to give all objects and all their fields the same type, which is basically awkwardly programming in a Racket-like way in ML. (Conversely, typed OOP languages are often no friendlier to ML-style programming unless they add separate constructs for generic types and closures.)

Our objects will just have fields and methods:

```
(struct obj (fields methods))
```

We will have `fields` hold an immutable list of *mutable* pairs where each element pair is a symbol (the field name) and a value (the current field contents). With that, we can define helper functions `get` and `set` that given an object and a field-name, return or mutate the field appropriately. Notice these are just plain Racket functions, with no special features or language additions. We do need to define our own function, called `assoc-m` below, because Racket's `assoc` expects an immutable list of immutable pairs.

```
(define (assoc-m v xs)
  (cond [(null? xs) #f]
        [(equal? v (mcar (car xs))) (car xs)]
        [#t (assoc-m v (cdr xs))]))

(define (get obj fld)
  (let ([pr (assoc-m fld (obj-fields obj))])
    (if pr
        (mcdr pr)
        (error "field not found"))))

(define (set obj fld v)
  (let ([pr (assoc-m fld (obj-fields obj))])
    (if pr
        (set-mcdr! pr v)
        (error "field not found"))))
```

More interesting is calling a method. The `methods` field will also be an association list mapping method names to functions (no mutation needed since we will be less dynamic than Ruby). The key to getting dynamic dispatch to work is that these functions will all take an extra *explicit* argument that is *implicit* in languages with built-in support for dynamic dispatch. This argument will be "self" and our Racket helper function for sending a message will simply pass in the correct object:

```
(define (send obj msg . args)
  (let ([pr (assoc msg (obj-methods obj))])
    (if pr
        ((cdr pr) obj args)
        (error "method not found" msg))))
```

Notice how the function we use for the method gets passed the "whole" object `obj`, which will be used for any sends to the object bound to `self`. (The code above uses Racket's support for variable-argument functions because it is convenient — we could have avoided it if necessary. Here, `send` can take any number of arguments greater than or equal to 2. The first argument is bound to `obj`, the second to `msg`, and all others are put in a list (in order) that is bound to `args`. Hence we expect `(cdr pr)` to be a function that takes two arguments: we pass `obj` for the first argument and the list `args` for the second argument.)

Now we can define `make-point`, which is just a Racket function that produces a point object:

```
(define (make-point _x _y)
  (obj
   (list (mcons 'x _x)
         (mcons 'y _y))
   (list (cons 'get-x (lambda (self args) (get self 'x)))
         (cons 'get-y (lambda (self args) (get self 'y)))
         (cons 'set-x (lambda (self args) (set self 'x (car args))))
         (cons 'set-y (lambda (self args) (set self 'y (car args))))
         (cons 'distToOrigin
               (lambda (self args)
                 (let ([a (send self 'get-x)]
                       [b (send self 'get-y)])
                   (sqrt (+ (* a a) (* b b)))))))))
```

Notice how each of the methods takes a first argument, which we just happen to call `self`, which has no special meaning here in Racket. We then use `self` as an argument to `get`, `set`, and `send`. If we had some other object we wanted to send a message to or access a field of, we would just pass that object to our helper functions by putting it in the `args` list. In general, the second argument to each function is a list of the "real arguments" in our object-oriented thinking.

By using the `get`, `set`, and `send` functions we defined, making and using points "feels" just like OOP:

```
(define p1 (make-point 4 0))
(send p1 'get-x)        ; 4
(send p1 'get-y)        ; 0
(send p1 'distToOrigin) ; 4
(send p1 'set-y 3)
(send p1 'distToOrigin) ; 5
```

Now let's simulate subclassing...

Our encoding of objects does not use classes, but we can still create something that reuses the code used to define points. Here is code to create points with a color field and getter/setter methods for this field. The key idea is to have the constructor create a point object with `make-point` and then extend this object by creating a new object that has the extra field and methods:

```
(define (make-color-point _x _y _c)
  (let ([pt (make-point _x _y)])
    (obj
     (cons (mcons 'color _c)
           (obj-fields pt))
     (append (list
              (cons 'get-color (lambda (self args) (get self 'color)))
              (cons 'set-color (lambda (self args) (set self 'color (car args)))))
             (obj-methods pt)))))
```

We can use "objects" returned from `make-color-point` just like we use "objects" returned from `make-point`, plus we can use the field `color` and the methods `get-color` and `set-color`.

The essential distinguishing feature of OOP is dynamic dispatch. Our encoding of objects "gets dynamic dispatch right" but our examples do not yet demonstrate it. To do so, we need a "method" in a "superclass"

to call a method that is defined/overridden by a "subclass." As we did in Ruby, let's define polar points by adding new fields and overriding the `get-x`, `get-y`, `set-x`, and `set-y` methods. A few details about the code below:

- As with color-points, our "constructor" uses the "superclass" constructor.
- As would happen in Java, our polar-point objects still have `x` and `y` fields, but we never use them.
- For simplicity, we just override methods by putting the replacements earlier in the method list than the overridden methods. This works because `assoc` returns the first matching pair in the list.

Most importantly, the `distToOrigin` "method" still works for a polar point because the method calls in its body will use the procedures listed with `'get-x` and `'get-y` in the definition of `make-polar-point` just like dynamic dispatch requires. The correct behavior results from our `send` function passing the whole object as the first argument.

```
(define (make-polar-point _r _th)
  (let ([pt (make-point #f #f)])
    (obj
     (append (list (mcons 'r _r)
                   (mcons 'theta _th))
             (obj-fields pt))
     (append
      (list
       (cons 'set-r-theta
             (lambda (self args)
               (begin
                 (set self 'r (car args))
                 (set self 'theta (cadr args)))))
       (cons 'get-x (lambda (self args)
                      (let ([r (get self 'r)]
                            [theta (get self 'theta)])
                        (* r (cos theta)))))
       (cons 'get-y (lambda (self args)
                      (let ([r (get self 'r)]
                            [theta (get self 'theta)])
                        (* r (sin theta)))))
       (cons 'set-x (lambda (self args)
                      (let* ([a     (car args)]
                             [b     (send self 'get-y)]
                             [theta (atan (/ b a))]
                             [r     (sqrt (+ (* a a) (* b b)))])
                        (send self 'set-r-theta r theta))))
       (cons 'set-y (lambda (self args)
                      (let* ([b     (car args)]
                             [a     (send self 'get-x)]
                             [theta (atan (/ b a))]
                             [r     (sqrt (+ (* a a) (* b b)))])
                        (send self 'set-r-theta r theta)))))
      (obj-methods pt)))))
```

We can create a polar-point object and send it some messages like this:

```
(define p3 (make-polar-point 4 3.1415926535))
(send p3 'get-x) ; -4 (or a slight rounding error)
(send p3 'get-y) ; 0 (or a slight rounding error)
(send p3 'distToOrigin) ; 4 (or a slight rounding error)
(send p3 'set-y 3)
(send p3 'distToOrigin) ; 5 (or a slight rounding error)
```

CSE341: Programming Languages Spring 2019
Unit 8 Summary
Dan Grossman, University of Washington

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

Contents

OOP Versus Functional Decomposition **1**
Extending the Code With New Operations or Variants **5**
Binary Methods With Functional Decomposition **7**
Binary Methods in OOP: Double Dispatch **8**
Multimethods **11**
Multiple Inheritance **12**
Mixins **13**
Java/C#-Style Interfaces **16**
Abstract Methods **17**
Introduction to Subtyping **17**
A Made-Up Language of Records **18**
Wanting Subtyping **19**
The Subtyping Relation **19**
Depth Subtyping: A Bad Idea With Mutation **20**
The Problem With Java/C# Array Subtyping **22**
Function Subtyping **23**
Subtyping for OOP **25**
Covariant `self`/`this` **26**
Generics Versus Subtyping **27**
Bounded Polymorphism **29**
Optional: Additional Java-Specific Bounded Polymorphism **30**

OOP Versus Functional Decomposition

We can compare procedural (functional) decomposition and object-oriented decomposition using the classic example of implementing operations for a small expression language. In functional programming, we typically break programs down into functions that perform some operation. In OOP, we typically break programs down into classes that give behavior to some kind of data.

We show that the two approaches largely lay out the same ideas in exactly opposite ways, and which way is "better" is either a matter of taste or depends on how software might be changed or *extended* in the future. We then consider how both approaches deal with operations over multiple arguments, which in many object-oriented languages requires a technique called *double (multiple) dispatch* in order to stick with an object-oriented style.

The Basic Set-Up

The following problem is the canonical example of a common programming pattern, and, not coincidentally, is a problem we have already considered a couple times in the course. Suppose we have:

- Expressions for a small "language" such as for arithmetic
- Different *variants* of expressions, such as integer values, negation expressions, and addition expressions
- Different *operations* over expressions, such as evaluating them, converting them to strings, or determining if they contain the constant zero in them

This problem leads to a conceptual *matrix* (two-dimensional grid) with one entry for each combination of variant and operation:

	eval	toString	hasZero
Int			
Add			
Negate			

No matter what programming language you use or how you approach solving this programming problem, you need to indicate what the proper behavior is for each entry in the grid. Certain approaches or languages might make it easier to specify defaults, but you are still deciding something for every entry.

The Functional Approach

In functional languages, the standard style is to do the following:

- Define a *datatype* for expressions, with one *constructor* for each variant. (In a dynamically typed language, we might not give the datatype a name in our program, but we are still thinking in terms of the concept. Similarly, in a language without direct support for constructors, we might use something like lists, but we are still thinking in terms of defining a way to construct each variant of data.)
- Define a *function* for each operation.
- In each function, have a branch (e.g., via pattern-matching) for each variant of data. If there is a default for many variants, we can use something like a wildcard pattern to avoid enumerating all the branches.

Note this approach is really just procedural decomposition: breaking the problem down into procedures corresponding to each operation.

This ML code shows the approach for our example: Notice how we define all the kinds of data in one place and then the nine entries in the table are implemented "by column" with one function for each column:

```
exception BadResult of string

datatype exp =
    Int    of int
  | Negate of exp
  | Add    of exp * exp

fun eval e =
    case e of
```

```
        Int _      => e
      | Negate e1  => (case eval e1 of
                           Int i => Int (~i)
                         | _ => raise BadResult "non-int in negation")
      | Add(e1,e2) => (case (eval e1, eval e2) of
                           (Int i, Int j) => Int (i+j)
                         | _ => raise BadResult "non-ints in addition")

fun toString e =
    case e of
        Int i      => Int.toString i
      | Negate e1  => "-(" ^ (toString e1) ^ ")"
      | Add(e1,e2) => "("  ^ (toString e1) ^ " + " ^ (toString e2) ^ ")"

fun hasZero e =
    case e of
        Int i      => i=0
      | Negate e1  => hasZero e1
      | Add(e1,e2) => (hasZero e1) orelse (hasZero e2)
```

The Object-Oriented Approach

In object-oriented languages, the standard style is to do the following:

- Define a *class* for expressions, with one *abstract method* for each operation. (In a dynamically typed language, we might not actually list the abstract methods in our program, but we are still thinking in terms of the concept. Similarly, in a language with duck typing, we might not actually use a superclass, but we are still thinking in terms of defining what operations we need to support.)
- Define a *subclass* for each variant of data.
- In each subclass, have a method definition for each operation. If there is a default for many variants, we can use a method definition in the superclass so that via inheritance we can avoid enumerating all the branches.

Note this approach is a data-oriented decomposition: breaking the problem down into classes corresponding to each data variant.

Here is the Ruby code, which for clarity has the different kinds of expressions subclass the `Exp` class. In a statically typed language, this would be required and the superclass would have to declare the methods that every subclass of `Exp` defines — listing all the operations in one place. Notice how we define the nine entries in the table "by row" with one class for each row.

```
class Exp
  # could put default implementations or helper methods here
end
class Int < Exp
  attr_reader :i
  def initialize i
    @i = i
  end
  def eval
    self
```

```
  end
  def toString
    @i.to_s
  end
  def hasZero
    i==0
  end
end
class Negate < Exp
  attr_reader :e
  def initialize e
    @e = e
  end
  def eval
    Int.new(-e.eval.i) # error if e.eval has no i method (not an Int)
  end
  def toString
    "-(" + e.toString + ")"
  end
  def hasZero
    e.hasZero
  end
end
class Add < Exp
  attr_reader :e1, :e2
  def initialize(e1,e2)
    @e1 = e1
    @e2 = e2
  end
  def eval
    Int.new(e1.eval.i + e2.eval.i) # error if e1.eval or e2.eval have no i method
  end
  def toString
    "(" + e1.toString + " + " + e2.toString + ")"
  end
  def hasZero
    e1.hasZero || e2.hasZero
  end
end
```

The Punch-Line

So we have seen that functional decomposition breaks programs down into functions that perform some operation and object-oriented decomposition breaks programs down into classes that give behavior to some kind of data. These are so exactly opposite that they are the same — just deciding whether to lay out our program "by column" or "by row." Understanding this symmetry is invaluable in conceptualizing software or deciding how to decompose a problem. Moreover, various software tools and IDEs can help you view a program in a different way than the source code is decomposed. For example, a tool for an OOP language that shows you all methods `foo` that override some superclass' `foo` is showing you a column even though the code is organized by rows.

So, which is better? It is often a matter of personal preference whether it seems "more natural" to lay out

the concepts by row or by column, so you are entitled to your opinion. What opinion is most common can depend on what the software is about. For our expression problem, the functional approach is probably more popular: it is "more natural" to have the cases for `eval` together rather than the operations for `Negate` together. For problems like implementing graphical user interfaces, the object-oriented approach is probably more popular: it is "more natural" to have the operations for a kind of data (like a `MenuBar`) together (such as `backgroundColor`, `height`, and `doIfMouseIsClicked` rather than have the cases for `doIfMouseIsClicked` together (for `MenuBar`, `TextBox`, `SliderBar`, etc.). The choice can also depend on what programming language you are using, how useful libraries are organized, etc.

Extending the Code With New Operations or Variants

The choice between "rows" and "columns" becomes less subjective if we later extend our program by adding new data variants or new operations.

Consider the functional approach. Adding a new operation is easy: we can implement a new function without editing any existing code. For example, this function creates a new expression that evaluates to the same result as its argument but has no negative constants:

```
fun noNegConstants e =
    case e of
        Int i       => if i < 0 then Negate (Int(~i)) else e
      | Negate e1   => Negate(noNegConstants e1)
      | Add(e1,e2)  => Add(noNegConstants e1, noNegConstants e2)
```

On the other hand, adding a new data variant, such as `Mult of exp * exp` is less pleasant. We need to go back and change all our functions to add a new case. In a statically typed language, we do get some help: after adding the `Mult` constructor, *if* our original code did not use wildcard patterns, then the type-checker will give a non-exhaustive pattern-match warning everywhere we need to add a case for `Mult`.

Again the object-oriented approach is exactly the opposite. Adding a new variant is easy: we can implement a new subclass without editing any existing code. For example, this Ruby class adds multiplication expressions to our language:

```
class Mult < Exp
  attr_reader :e1, :e2
  def initialize(e1,e2)
    @e1 = e1
    @e2 = e2
  end
  def eval
    Int.new(e1.eval.i * e2.eval.i) # error if e1.eval or e2.eval has no i method
  end
  def toString
    "(" + e1.toString + " * " + e2.toString + ")"
  end
  def hasZero
    e1.hasZero || e2.hasZero
  end
end
```

On the other hand, adding a new operation, such as `noNegConstants`, is less pleasant. We need to go back

and change all our classes to add a new method. In a statically typed language, we do get some help: after declaring in the `Exp` superclass that all subclasses should have a `noNegConstants` method, the type-checker will give an error for any class that needs to implement the method. (This static typing is using abstract methods and abstract classes, which are discussed later.)

Planning for extensibility

As seen above, functional decomposition allows new operations and object-oriented decomposition allows new variants without modifying existing code and without explicitly planning for it — the programming styles "just work that way." It is possible for functional decomposition to support new variants or object-oriented decomposition to support new operations *if you plan ahead* and use somewhat awkward programming techniques (that seem less awkward over time if you use them often).

We do not consider these techniques in detail here and you are not responsible for learning them. For object-oriented programming, "the visitor pattern" is a common approach. This pattern often is implemented using double dispatch, which is covered for other purposes below. For functional programming, we can define our datatypes to have an "other" possibility and our operations to take in a function that can process the "other data." Here is the idea in SML:

```
datatype 'a ext_exp =
    Int    of int
  | Negate of 'a ext_exp
  | Add    of 'a ext_exp * 'a ext_exp
  | OtherExtExp  of 'a

fun eval_ext (f,e) = (* notice we pass a function to handle extensions *)
  case e of
    Int i       => i
  | Negate e1   => 0 - (eval_ext (f,e1))
  | Add(e1,e2)  => (eval_ext (f,e1)) + (eval_ext (f,e2))
  | OtherExtExp e => f e
```

With this approach, we could create an extension supporting multiplication by instantiating `'a` with `exp * exp`, passing `eval_ext` the function `(fn (x,y) => eval_ext(f,e1) * eval_ext(f,e2))`, and using `OtherExtExp(e1,e2)` for multiplying `e1` and `e2`. This approach can support different extensions, but it does not support well combining two extensions created separately.

Notice that it does *not* work to wrap the original datatype in a new datatype like this:

```
datatype myexp_wrong =
         OldExp of exp
       | MyMult of myexp_wrong * myexp_wrong
```

This approach does not allow, for example, a subexpression of an `Add` to be a `MyMult`.

Thoughts on Extensibility

It seems clear that if you expect new operations, then a functional approach is more natural and if you expect new data variants, then an object-oriented approach is more natural. The problems are (1) the future is often difficult to predict; we may not know what extensions are likely, and (2) both forms of extension may be likely. Newer languages like Scala aim to support both forms of extension well; we are still gaining practical experience on how well it works as it is a fundamentally difficult issue.

More generally, making software that is both robust and extensible is valuable but difficult. Extensibility can make the original code more work to develop, harder to reason about locally, and harder to change

(without breaking extensions). In fact, languages often provide constructs exactly to *prevent* extensibility. ML's modules can hide datatypes, which prevents defining new operations over them outside the module. Java's `final` modifier on a class prevents subclasses.

Binary Methods With Functional Decomposition

The operations we have considered so far used only one value of a type with multiple data variants: `eval`, `toString`, `hasZero`, and `noNegConstants` all operated on one expression. When we have operations that take two (binary) or more (n-ary) variants as arguments, we often have many more cases. With functional decomposition all these cases are still covered together in a function. As seen below, the OOP approach is more cumbersome.

For sake of example, suppose we add string values and rational-number values to our expression language. Further suppose we change the meaning of `Add` expressions to the following:

- If the arguments are ints or rationals, do the appropriate arithmetic.
- If either argument is a string, convert the other argument to a string (unless it already is one) and return the concatenation of the strings.

So it is an error to have a subexpression of `Negate` or `Mult` evaluate to a `String` or `Rational`, but the subexpressions of `Add` can be any kind of value in our language: int, string, or rational.

The interesting change to the SML code is in the `Add` case of `eval`. We now have to consider 9 (i.e., $3 * 3$) subcases, one for each combination of values produced by evaluating the subexpressions. To make this explicit and more like the object-oriented version considered below, we can move these cases out into a helper function `add_values` as follows:

```
fun eval e =
   case e of
      ...
     | Add(e1,e2)  => add_values (eval e1, eval e2)
      ...

fun add_values (v1,v2) =
    case (v1,v2) of
        (Int i,  Int j)         => Int (i+j)
      | (Int i,  String s)      => String(Int.toString i ^ s)
      | (Int i,  Rational(j,k)) => Rational(i*k+j,k)
      | (String s,  Int i)      => String(s ^ Int.toString i) (* not commutative *)
      | (String s1, String s2)  => String(s1 ^ s2)
      | (String s,  Rational(i,j)) => String(s ^ Int.toString i ^ "/" ^ Int.toString j)
      | (Rational _,    Int _)    => add_values(v2,v1) (* commutative: avoid duplication *)
      | (Rational(i,j), String s) => String(Int.toString i ^ "/" ^ Int.toString j ^ s)
      | (Rational(a,b), Rational(c,d)) => Rational(a*d+b*c,b*d)
      | _ => raise BadResult "non-values passed to add_values"
```

Notice `add_values` is defining all 9 entries in this 2-D grid for how to add values in our language — a different kind of matrix than we considered previously because the rows and columns are variants.

	Int	String	Rational
Int			
String			
Rational			

While the number of cases may be large, that is inherent to the problem. If many cases work the same way, we can use wildcard patterns and/or helper functions to avoid redundancy. One common source of redundancy is *commutativity*, i.e., the order of values not mattering. In the example above, there is only one such case: adding a rational and an int is the same as adding an int and a rational. Notice how we exploit this redundancy by having one case use the other with the call `add_values(v2,v1)`.

Binary Methods in OOP: Double Dispatch

We now turn to supporting the same enhancement of strings, rationals, and enhanced evaluation rules for `Add` in an OOP style. Because Ruby has built-in classes called `String` and `Rational`, we will extend our code with classes named `MyString` and `MyRational`, but obviously that is not the important point. The first step is to add these classes and have them implement all the existing methods, just like we did when we added `Mult` previously. Then that "just" leaves revising the `eval` method of the `Add` class, which previously assumed the recursive results would be instances of `Int` and therefore have a getter method `i`:

```
def eval
  Int.new(e1.eval.i + e2.eval.i) # error if e1.eval or e2.eval have no i method
end
```

Now we could instead replace this method body with code like our `add_values` helper function in ML, but helper *functions* like this are not OOP style. Instead, we expect `add_values` to be a method in the classes that represent values in our language: An `Int`, `MyRational`, or `MyString` should "know how to add itself to another value." So in `Add`, we write:

```
def eval
  e1.eval.add_values e2.eval
end
```

This is a good start and now obligates us to have `add_values` methods in the classes `Int`, `MyRational`, and `MyString`. By putting `add_values` methods in the `Int`, `MyString`, and `MyRational` classes, we nicely divide our work into three pieces using dynamic dispatch depending on the class of the object that `e1.eval` returns, i.e., the receiver of the `add_values` call in the `eval` method in `Add`. But then each of these three needs to handle three of the nine cases, based on the class of the second argument. One approach would be to, in these methods, abandon object-oriented style (!) and use run-time tests of the classes to include the three cases. The Ruby code would look like this:

```
class Int
   ...
   def add_values v
      if v.is_a? Int
          ...
      elsif v.is_a? MyRational
          ...
      else
```

```
            ...
        end
    end
end
class MyRational
    ...
    def add_values v
        if v.is_a? Int
            ...
        elsif v.is_a? MyRational
            ...
        else
            ...
        end
    end
end
class MyString
    ...
    def add_values v
        if v.is_a? Int
            ...
        elsif v.is_a? MyRational
            ...
        else
            ...
        end
    end
end
```

While this approach works, it is really not object-oriented programming. Rather, it is a mix of object-oriented decomposition (dynamic dispatch on the first argument) and functional decomposition (using `is_a?` to figure out the cases in each method). There is not necessarily anything wrong with that — it is probably simpler to understand than what we are about to demonstrate — but it does give up the extensibility advantages of OOP and really is not "full" OOP.

Here is how to think about a "full" OOP approach: The problem inside our three `add_values` methods is that we need to "know" the class of the argument `v`. In OOP, the strategy is to replace "needing to know the class" with calling a method on `v` instead. So we should "tell `v`" to do the addition, passing `self`. And we can "tell `v`" what class `self` is because the `add_values` methods know that: In `Int`, `self` is an `Int` for example. And the way we "tell `v` the class" is to call different methods on `v` for each kind of argument.

This technique is called *double dispatch*. Here is the code for our example, followed by additional explanation:

```
class Int
  ... # other methods not related to add_values
  def add_values v # first dispatch
    v.addInt self
  end
  def addInt v # second dispatch: v is Int
    Int.new(v.i + i)
  end
  def addString v # second dispatch: v is MyString
    MyString.new(v.s + i.to_s)
```

```
  end
  def addRational v # second dispatch: v is MyRational
    MyRational.new(v.i+v.j*i,v.j)
  end
end
class MyString
  ... # other methods not related to add_values
  def add_values v # first dispatch
    v.addString self
  end
  def addInt v # second dispatch: v is Int
    MyString.new(v.i.to_s + s)
  end
  def addString v # second dispatch: v is MyString
    MyString.new(v.s + s)
  end
  def addRational v # second dispatch: v is MyRational
    MyString.new(v.i.to_s + "/" + v.j.to_s + s)
  end
end
class MyRational
  ... # other methods not related to add_values
  def add_values v # first dispatch
    v.addRational self
  end
  def addInt v # second dispatch
    v.addRational self  # reuse computation of commutative operation
  end
  def addString v # second dispatch: v is MyString
    MyString.new(v.s + i.to_s + "/" + j.to_s)
  end
  def addRational v # second dispatch: v is MyRational
    a,b,c,d = i,j,v.i,v.j
    MyRational.new(a*d+b*c,b*d)
  end
end
```

Before understanding how all the method calls work, notice that we do now have our 9 cases for addition in 9 different methods:

- The `addInt` method in `Int` is for when the left operand to addition is an `Int` (in `v`) and the right operation is an `Int` (in `self`).
- The `addString` method in `Int` is for when the left operand to addition is a `MyString` (in `v`) and the right operation is an `Int` (in `self`).
- The `addRational` method in `Int` is for when the left operand to addition is a `MyRational` (in `v`) and the right operation is an `Int` (in `self`).
- The `addInt` method in `MyString` is for when the left operand to addition is an `Int` (in `v`) and the right operation is a `MyString` (in `self`).
- The `addString` method in `MyString` is for when the left operand to addition is a `MyString` (in `v`) and the right operation is a `MyString` (in `self`).

- The `addRational` method in `MyString` is for when the left operand to addition is a `MyRational` (in `v`) and the right operation is a `MyString` (in `self`).
- The `addInt` method in `MyRational` is for when the left operand to addition is an `Int` (in `v`) and the right operation is a `MyRational` (in `self`).
- The `addString` method in `MyRational` is for when the left operand to addition is a `MyString` (in `v`) and the right operation is a `MyRational` (in `self`).
- The `addRational` method in `MyRational` is for when the left operand to addition is a `MyRational` (in `v`) and the right operation is a `MyRational` (in `self`).

As we might expect in OOP, our 9 cases are "spread out" compared to in the ML code. Now we need to understand how dynamic dispatch is picking the correct code in all 9 cases. Starting with the `eval` method in `Add`, we have `e1.eval.add_values e2.eval`. There are 3 `add_values` methods and dynamic dispatch will pick one based on the class of the value returned by `e1.eval`. This the "first dispatch." Suppose `e1.eval` is an `Int`. Then the next call will be `v.addInt self` where `self` is `e1.eval` and `v` is `e2.eval`. Thanks again to dynamic dispatch, the method looked up for `addInt` will be the right case of the 9. This is the "second dispatch." All the other cases work analogously.

Understanding double dispatch can be a mind-bending exercise that re-enforces how dynamic dispatch works, the key thing that separates OOP from other programming. It is not necessarily intuitive, but it what one must do in Ruby/Java to support binary operations like our addition in an OOP style.

Notes:

- OOP languages with *multimethods*, discussed next, do not require the manual double dispatch we have seen here.
- Statically typed languages like Java do not get in the way of the double-dispatch idiom. In fact, needing to declare method argument and return types as well as indicating in the superclass the methods that all subclasses implement can make it easier to understand what is going on. A full Java implementation of our example is posted with the course materials. (It is common in Java to reuse method names for different methods that take arguments of different types. Hence we could use `add` instead of `addInt`, `addString`, and `addRational`, but this can be more confusing than helpful when first learning double dispatch.)

Multimethods

It is *not* true that all OOP languages require the cumbersome double-dispatch pattern to implement binary operations in a full OOP style. Languages with *multimethods*, also known as *multiple dispatch*, provide more intuitive solutions. In such languages, the classes `Int`, `MyString`, and `MyRational` could each define three methods all named `add_values` (so there would be nine total methods in the program named `add_values`). Each `add_values` method would indicate the class it expects for its argument. Then `e1.eval.add_values e2.eval` would pick the right one of the 9 by, at run-time, considering the class of the result of `e1.eval` *and* the class of the result of `e2.eval`.

This is a powerful and *different* semantics than we have studied for OOP. In our study of Ruby (and Java/C#/C++ work the same way), the method-lookup rules involve the run-time class of the receiver (the object whose method we are calling), not the run-time class of the argument(s). Multiple dispatch is "even more dynamic dispatch" by considering the class of multiple objects and using all that information to choose what method to call.

Ruby does not support multimethods because Ruby is committed to having only one method with a particular name in each class. Any object can be passed to this one method. So there is no way to have 3 `add_values` methods in the same class and no way to indicate which method should be used based on the argument.

Java and C++ also do not have multimethods. In these languages you *can* have multiple methods in a class with the same name and the method-call semantics does use the types of the arguments to choose what method to call. But it uses the *types* of the arguments, which are determined at *compile-time* and *not* the run-time class of the result of evaluating the arguments. This semantics is called *static overloading*. It is considered useful and convenient, but it is not multimethods and does not avoid needing double dispatch in our example.

C# has the same static overloading as Java and C++, but as of version 4.0 of the language one can achieve the effect of multimethods by using the type "dynamic" in the right places. We do not discuss the details here, but it is a nice example of combining language features to achieve a useful end.

Many OOP languages have had multimethods for many years — they are not a new idea. Perhaps the most well-known modern language with multimethods is Clojure.

Multiple Inheritance

We have seen that the essence of object-oriented programming is inheritance, overriding, and dynamic dispatch. All our examples have been classes with 1 (immediate) superclass. But if inheritance is so useful and important, why not allow ways to use more code defined in other places such as another class. We now begin discussing 3 related but distinct ideas:

- *Multiple inheritance:* Languages with multiple inheritance let one class extend multiple other classes. It is the most powerful option, but there are some semantic problems that arise that the other ideas avoid. Java and Ruby do not have multiple inheritance; C++ does.
- *Mixins:* Ruby allows a class to have one immediate superclass but any number of mixins. Because a mixin is "just a pile of methods," many of the semantic problems go away. Mixins do not help with all situations where you want multiple inheritance, but they have some excellent uses. In particular, elegant uses of mixins typically involve mixin methods calling methods that they assume are defined in all classes that include the mixin. Ruby's standard libraries make good use of this technique and your code can too.
- *Java/C#-style interfaces:* Java/C# classes have one immediate superclass but can "implement" any number of interfaces. Because interfaces do not provide behavior — they only require that certain methods exist — most of the semantic problems go away. Interfaces are fundamentally about type-checking, which we will study more later in this unit, so there very little reason for them in a language like Ruby. C++ does not have interfaces because inheriting a class with all "abstract" methods (or "pure virtual" methods in C++-speak) accomplishes the same thing as described more below.

To understand why multiple inheritance is potentially useful, consider two classic examples:

- Consider a `Point2D` class with subclasses `Point3D` (adding a z-dimension) and `ColorPoint` (adding a `color` attribute). To create a `ColorPoint3D` class, it would seem natural to have two immediate superclasses, `Point3D` and `ColorPoint` so we inherit from both.
- Consider a `Person` class with subclasses `Artist` and `Cowboy`. To create an `ArtistCowboy` (someone who is both), it would seem natural again to have two immediate superclasses. Note, however, that both the `Artist` class and the `Cowboy` class have a method "draw" that have very different behaviors (creating a picture versus producing a gun).

Without multiple inheritance, you end up copying code in these examples. For example, `ColorPoint3D` can subclass `Point3D` and copy code from `ColorPoint` or it can subclass `ColorPoint` and copy code from `Point3D`.

If we have multiple inheritance, we have to decide what it means. Naively we might say that the new class has all the methods of all the superclasses (and fields too in languages where fields are part of class definitions). However, if two of the immediate superclasses have the *same* fields or methods, what does that mean? Does it matter if the fields or methods are inherited from the same *common ancestor*? Let us explain these issues in more detail before returning to our examples.

With single inheritance, the *class hierarchy* — all the classes in a program and what extends what — forms a tree, where `A` extends `B` means `A` is a child of `B` in the tree. With multiple inheritance, the class hierarchy may not be a tree. Hence it can have "diamonds" — four classes where one is a (not necessarily immediate) subclass of two others that have a common (not necessarily immediate) superclass. By "immediate" we mean directly extends (child-parent relationship) whereas we could say "transitive" for the more general ancestor-descendant relationship.

With multiple superclasses, we may have conflicts for the fields / methods inherited from the different classes. The `draw` method for `ArtistCowboy` objects is an obvious example where we would like somehow to have both methods in the subclass, or potentially to override one or both of them. At the very least we need expressions using `super` to indicate which superclass is intended. But this is not necessarily the only conflict. Suppose the `Person` class has a pocket field that artists and cowboys use for different things. Then perhaps an `ArtistCowboy` should have two pockets, even though the creation of the notion of pocket was in the common ancestor `Person`.

But if you look at our `ColorPoint3D` example, you would reach the opposite conclusion. Here both `Point3D` and `ColorPoint` inherit the notion of x and y from a common ancestor, but we certainly do not want a `ColorPoint3D` to have two `x` methods or two `@x` fields.

These issues are some of the reasons language with multiple inheritance (most well-known is C++) need fairly complicated rules for how subclassing, method lookup, and field access work. For example, C++ has (at least) two different forms of creating a subclass. One always makes copies of all fields from all superclasses. The other makes only one copy of fields that were initially declared by the same common ancestor. (This solution would not work well in Ruby because instance variables are not part of class definitions.)

Mixins

Ruby has *mixins*, which are somewhere between multiple inheritance (see above) and interfaces (see below). They provide actual code to classes that *include* them, but they are not classes themselves, so you cannot create instances of them. Ruby did not invent mixins. Its standard-library makes good use of them, though. A near-synonym for mixins is *traits*, but we will stick with what Ruby calls mixins.

To define a Ruby mixin, we use the keyword `module` instead of `class`. (Modules do a bit more than just serve as mixins, hence the strange word choice.) For example, here is a mixin for adding color methods to a class:

```
module Color
  attr_accessor :color
  def darken
    self.color = "dark " + self.color
  end
end
```

This mixin defines three methods, `color`, `color=`, and `darken`. A class definition can include these methods by using the `include` keyword and the name of the mixin. For example:

```
class ColorPt < Pt
  include Color
end
```

This defines a subclass of `Pt` that also has the three methods defined by `Color`. Such classes can have other methods defined/overridden too; here we just chose not to add anything additional. This is not necessarily good style for a couple reasons. First, our `initialize` (inherited from `Pt`) does not create the `@color` field, so we are relying on clients to call `color=` before they call `color` or they will get `nil` back. So overriding `initialize` is probably a good idea. Second, mixins that use instance variables are stylistically questionable. As you might expect in Ruby, the instance variables they use will be part of the object the mixin is included in. So if there is a name conflict with some intended-to-be separate instance variable defined by the class (or another mixin), the two separate pieces of code will mutate the same data. After all, mixins are "very simple" — they just define a collection of methods that can be included in a class.

Now that we have mixins, we also have to reconsider our method lookup rules. We have to choose something and this is what Ruby chooses: If `obj` is an instance of class `C` and we send message `m` to `obj`,

- First look in the class `C` for a definition of `m`.
- Next look in mixins included in `C`. Later includes shadow earlier ones.
- Next look in `C`'s superclass.
- Next look in `C`'s superclass' mixins.
- Next look in `C`'s super-superclass.
- Etc.

Many of the elegant uses of mixins do the following strange-sounding thing: They define methods that call other methods on `self` that are *not* defined by the mixin. Instead the mixin *assumes* that all classes that include the mixin define this method. For example, consider this mixin that lets us "double" instances of any class that has `+` defined:

```
module Doubler
  def double
    self + self # uses self's + message, not defined in Doubler
  end
end
```

If we include `Doubler` in some class `C` and call `double` on an instance of the class, we will call the `+` method on the instance, getting an error if it is not defined. But if `+` *is* defined, everything works out. So now we can easily get the convenience of doubling just by defining `+` and including the `Doubler` mixin. For example:

```
class AnotherPt
  attr_accessor :x, :y
  include Doubler
  def + other  # add two points
    ans = AnotherPt.new
    ans.x = self.x + other.x
```

```
    ans.y = self.y + other.y
    ans
  end
end
```

Now instances of `AnotherPt` have `double` methods that do what we want. We could even add `double` to classes that already exist:

```
class String
  include Doubler
end
```

Of course, this example is a little silly since the `double` method is so simple that copying it over and over again would not be so burdensome.

The same idea is used a lot in Ruby with two mixins named `Enumerable` and `Comparable`. What `Comparable` does is provide methods `=`, `!=`, `>`, `>=`, `<`, and `<=`, all of which assume the class defines `<=>`. What `<=>` needs to do is return a negative number if its left argument is less than its right, 0 if they are equal, and a positive number if the left argument is greater than the right. So now a class does not have to define all these comparisons — it just defines `<=>` and includes `Comparable`. Consider this example for comparing names:

```
class Name
  attr_accessor :first, :middle, :last
  include Comparable
  def initialize(first,last,middle="")
    @first = first
    @last = last
    @middle = middle
  end
  def <=> other
    l = @last <=> other.last # <=> defined on strings
    return l if l != 0
    f = @first <=> other.first
    return f if f != 0
    @middle <=> other.middle
  end
end
```

Defining methods in `Comparable` is easy, but we certainly would not want to repeat the work for every class that wants comparisons. For example, the `>` method is just:

```
def > other
  (self <=> other) > 0
end
```

The `Enumerable` module is where many of the useful block-taking methods that iterate over some data structure are defined. Examples are `any?`, `map`, `count`, and `inject`. They are all written assuming the class has the method `each` defined. So a class can define `each`, include the `Enumerable` mixin, and have all these convenient methods. So the `Array` class for example can just define `each` and include `Enumerable`. Here is another example for a range class we might define:[1]

[1] We wouldn't actually define this because Ruby already has very powerful range classes.

```
class MyRange
  include Enumerable
  def initialize(low,high)
    @low = low
    @high = high
  end
  def each
    i=@low
    while i <= @high
      yield i
      i=i+1
    end
  end
end
```

Now we can write code like `MyRange.new(4,8).inject {|x,y| x+y}` or `MyRange.new(5,12).count {|i| i.odd?}`. Note that the `map` method in `Enumerable` always returns an instance of `Array`. After all, it does not "know how" to produce an instance of any class, but it does know how to produce an array containing one element for everything produced by `each`. We could define it in the `Enumerable` mixin like this:

```
 def map
   arr = []
   each {|x| arr.push x }
   arr
 end
```

Mixins are not as powerful as multiple inheritance because we have to decide upfront what to make a class and what to make a mixin. Given `Artist` and `Cowboy` classes, we still have no natural way to make an `ArtistCowboy`. And it is unclear which of `Artist` or `Cowboy` or both we might want to define in terms of a mixin.

Java/C#-Style Interfaces

In Java or C#, a class can have only one immediate superclass but it can implement any number of *interfaces*. An interface is just a list of methods and each method's argument types and return type. A class type-checks only if it actually provides (directly or via inheritance) all the methods of all the interfaces it claims to implement. An interface is a type, so if a class `C` implements interface `I`, then we can pass an instance of `C` to a method expecting an argument of type `I`, for example. Interfaces are closer to the idea of "duck typing" than just using classes as types (in Java and C# every class is also a type), but a class has some interface type only if the class definition *explicitly* says it implements the interface. We discuss more about OOP type-checking later in this unit.

Because interfaces do not actually *define* methods — they only name them and give them types — none of the problems discussed above about multiple inheritance arise. If two interfaces have a method-name conflict, it does not matter — a class can still implement them both. If two interfaces disagree on a method's type, then no class can possibly implement them both but the type-checker will catch that. Because interfaces do not define methods, they cannot be used like mixins.

In a dynamically typed language, there is really little reason to have interfaces.[2] We can *already* pass any

[2]Probably the only use would be to change the meaning of Ruby's `is_a?` to incorporate interfaces, but we can more directly just use reflection to find out an object's methods.

object to any method and call any method on any object. It is up to us to keep track "in our head" (preferably in comments as necessary) what objects can respond to what messages. The essence of dynamic typing is not writing down this stuff.

Bottom line: Implementing interfaces does not inherit code; it is purely related to type-checking in statically typed languages like Java and C#. It makes the type systems in these languages more flexible. So Ruby does not need interfaces.

Abstract Methods

Often a class definition has methods that call other methods that are not actually defined in the class. It would be an error to create instances of such a class and use the methods such that "method missing" errors occur. So why define such a class? Because the entire point of the class is to be subclassed and have different subclasses define the missing methods in different ways, relying on dynamic dispatch for the code in the superclass to call the code in the subclass. This much works just fine in Ruby — you can have comments indicating that certain classes are there only for the purpose of subclassing.

The situation is more interesting in statically typed languages. In these languages, the purpose of type-checking is to prevent "method missing" errors, so when using this technique we need to indicate that instances of the superclass must not be created. In Java/C# such classes are called "abstract classes." We also need to give the type of any methods that (non-abstract) subclasses must provide. These are "abstract methods." Thanks to subtyping in these languages, we can have expressions with the *type* of the superclass and know that at run-time the object will actually be one of the subclasses. Furthermore, type-checking ensures the object's class has implemented all the abstract methods, so it is safe to call these methods. In C++, abstract methods are called "pure virtual methods" and serve much the same purpose.

There is an interesting parallel between abstract methods and higher-order functions. In both cases, the language supports a programming pattern where some code is passed other code in a flexible and reusable way. In OOP, different subclasses can implement an abstract method in different ways and code in the superclass, via dynamic dispatch, can then uses these different implementations. With higher-order functions, if a function takes another function as an argument, different callers can provide different implementations that are then used in the function body.

Languages with abstract methods and multiple inheritance (e.g., C++) do not need interfaces. Instead we can just use classes that have nothing but abstract (pure virtual) methods in them like they are interfaces and have classes implementing these "interfaces" just subclass the classes. This subclassing is not inheriting any code exactly because abstract methods do not define methods. With multiple inheritance, we are not "wasting" our one superclass with this pattern.

Introduction to Subtyping

We previously studied static types for functional programs, in particular ML's type system. ML uses its type system to prevent errors like treating a number as a function. A key source of expressiveness in ML's type system (not rejecting too many programs that do nothing wrong and programmers are likely to write) is *parametric polymorphism*, also known as *generics*.

So we should also study static types for object-oriented programs, such as those found in Java. If everything is an object (which is less true in Java than in Ruby), then the main thing we would want our type system to prevent is "method missing" errors, i.e., sending a message to an object that has no method for that message. If objects have fields accessible from outside the object (e.g., in Java), then we also want to prevent

"field missing" errors. There are other possible errors as well, like calling a method with the wrong number of arguments.

While languages like Java and C# have generics these days, the source of type-system expressiveness most fundamental to object-oriented style is *subtype polymorphism*, also known as *subtyping*. ML does not have subtyping, though this decision is really one of language design (it would complicate type inference, for example).

It would be natural to study subtyping using Java since it is a well-known object-oriented language with a type system that has subtyping. But it is also fairly complicated, using classes and interfaces for types that describe objects with methods, overriding, static overloading, etc. While these features have pluses and minuses, they can complicate the fundamental ideas that underlie how subtyping should work in any language.

So while we will briefly discuss subtyping in OOP, we will mostly use a small language with *records* (like in ML, things with named fields holding contents — basically objects with public fields, no methods, and no class names) and functions (like in ML or Racket). This will let us see how subtyping should — and should not — work.

This approach has the disadvantage that we cannot use any of the language we have studied: ML does not have subtyping and record fields are immutable, Racket and Ruby are dynamically typed, and Java is too complicated for our starting point. So we are going to make up a language with just records, functions, variables, numbers, strings, etc. and explain the meaning of expressions and types as we go.

A Made-Up Language of Records

To study the basic ideas behind subtyping, we will use records with mutable fields, as well as functions and other expressions. Our syntax will be a mix of ML and Java that keeps examples short and, hopefully, clear. For records, we will have expressions for making records, getting a field, and setting a field as follows:

- In the expression `{f1=e1, f2=e2, ..., fn=en}`, each `fi` is a field name and each `ei` is an expression. The semantics is to evaluate each `ei` to a value `vi` and the result is the record value `{f1=v1, f2=v2, ..., fn=vn}`. So a record value is just a collection of fields, where each field has a name and a contents.
- For the expression `e.f`, we evaluate `e` to a value `v`. If `v` is a record with an `f` field, then the result is the contents of the `f` field. Our type system will ensure `v` has an `f` field.
- For the expression `e1.f = e2`, we evaluate `e1` and `e2` to values `v1` and `v2`. If `v1` is a record with an `f` field, then we update the `f` field to have `v2` for its contents. Our type system will ensure `v1` has an `f` field. Like in Java, we will choose to have the result of `e1.f = e2` be `v2`, though usually we do not use the result of a field-update.

Now we need a type system, with a form of types for records and typing rules for each of our expressions. Like in ML, let's write record types as `{f1:t1, f2:t2, ..., fn:tn}`. For example, `{x : real, y : real}` would describe records with two fields named `x` and `y` that hold contents of type `real`. And `{foo: {x : real, y : real}, bar : string, baz : string}` would describe a record with three fields where the `foo` field holds a (nested) record of type `{x : real, y : real}`. We then type-check expressions as follows:

- If `e1` has type `t1`, `e2` has type `t2`, ..., `en` has type `tn`, then `{f1=e1, f2=e2, ..., fn=en}` has type `{f1:t1, f2:t2, ..., fn:tn}`.

- If `e` has a record type containing `f : t`, then `e.f` has type `t` (else `e.f` does not type-check).
- If `e1` has a record type containing `f : t` and `e2` has type `t`, then `e1.f = e2` has type `t` (else `e1.f = e2` does not type-check).

Assuming the "regular" typing rules for other expressions like variables, functions, arithmetic, and function calls, an example like this will type-check as we would expect:

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val pythag : {x:real,y:real} = {x=3.0, y=4.0}
val five : real = distToOrigin(pythag)
```

In particular, the function `distToOrigin` has type `{x : real, y : real} -> real`, where we write function types with the same syntax as in ML. The call `distToOrigin(pythag)` passes an argument of the right type, so the call type-checks and the result of the call expression is the return type `real`.

This type system does what it is intended to do: No program that type-checks would, when evaluated, try to look up a field in a record that does not have that field.

Wanting Subtyping

With our typing rules so far, this program would not type-check:

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val c : {x:real,y:real,color:string} = {x=3.0, y=4.0, color="green"}
val five : real = distToOrigin(c)
```

In the call `distToOrigin(c)`, the type of the argument is `{x:real,y:real,color:string}` and the type the function expects is `{x:real,y:real}`, breaking the typing rule that functions must be called with the type of argument they expect. Yet the program above is safe: running it would not lead to accessing a field that does not exist.

A natural idea is to make our type system more lenient as follows: If some expression has a record type `{f1:t1, ..., fn:tn}`, then let the expression *also* have a type where some of the fields are removed. Then our example will type-check: Since the expression `c` has type `{x:real,y:real,color:string}`, it can also have type `{x:real,y:real}`, which allows the call to type-check. Notice we could also use `c` as an argument to a function of type `{color:string}->int`, for example.

Letting an expression that has one type also have another type that has less information is the idea of *subtyping*. (It may seem backwards that the *subtype* has *more* information, but that is how it works. A less-backwards way of thinking about it is that there are "*fewer*" values of the subtype than of the supertype because values of the subtype have more obligations, e.g., having more fields.)

The Subtyping Relation

We will now add subtyping to our made-up language, in a way that will not require us to change any of our existing typing rules. For example, we will leave the function-call rule the same, still requiring that the type

of the actual argument *equal* the type of the function parameter in the function definition. To do this, we will add two things to our type system:

- The idea of one type being a subtype of another: We will write `t1 <: t2` to mean `t1` is a subtype of `t2`.
- One and only new typing rule: If `e` has type `t1` and `t1 <: t2`, then `e` (also) has type `t2`.

So now we just need to give rules for `t1 <: t2`, i.e., when is one type a subtype of another. This approach is good language engineering — we have separated the idea of subtyping into a single binary relation that we can define separately from the rest of the type system.

A common misconception is that if we are defining our own language, then we can make the typing and subtyping rules whatever we want. That is only true if we forget that our type system is allegedly preventing something from happening when programs run. If our goal is (still) to prevent field-missing errors, then we cannot add any subtyping rules that would cause us to stop meeting our goal. This is what people mean when they say, "Subtyping is not a matter of opinion."

For subtyping, the key guiding principle is *substitutability*: If we allow `t1 <: t2`, then any value of type `t1` must be able to be used in every way a `t2` can be. For records, that means `t1` should have all the fields that `t2` has and with the same types.

Some Good Subtyping Rules

Without further ado, we can now give four subtyping rules that we can add to our language to accept more programs without breaking the type system. The first two are specific to records and the next two, while perhaps seeming unnecessary, do no harm and are common in any language with subtyping because they combine well with other rules:

- "Width" subtyping: A supertype can have a subset of fields with the same types, i.e., a subtype can have "extra" fields
- "Permutation" subtyping: A supertype can have the same set of fields with the same types in a different order.
- Transitivity: If `t1 <: t2` and `t2 <: t3`, then `t1 <: t3`.
- Reflexivity: Every type is a subtype of itself: `t <: t`.

Notice that width subtyping lets us forget fields, permutation subtyping lets us reorder fields (e.g., so we can pass a `{x:real,y:real}` in place of a `{y:real,x:real}`) and transitivity with those rules lets us do both (e.g., so we can pass a `{x:real,foo:string,y:real}` in place of a `{y:real,x:real}`).

Depth Subtyping: A Bad Idea With Mutation

Our subtyping rules so far let us drop fields or reorder them, but there is no way for a supertype to have a field with a different type than in the subtype. For example, consider this example, which passes a "sphere" to a function expecting a "circle." Notice that circles and spheres have a `center` field that itself holds a record.

```
fun circleY (c:{center:{x:real,y:real}, r:real}) =
   c.center.y
```

```
val sphere:{center:{x:real,y:real,z:real}, r:real}) = {center={x=3.0,y=4.0,z=0.0}, r=1.0}
val _ = circleY(sphere)
```

The type of `circleY` is `{center:{x:real,y:real}, r:real}->real` and the type of `sphere` is `{center:{x:real,y:real,z:real}, r:real}`, so the call `circleY(sphere)` can type-check only if

```
{center:{x:real,y:real,z:real}, r:real} <: {center:{x:real,y:real}, r:real}
```

This subtyping does not hold with our rules so far: We can drop the `center` field, drop the `r` field, or reorder those fields, but we cannot "reach into a field type to do subtyping."

Since we might like the program above to type-check since evaluating it does nothing wrong, perhaps we should add another subtyping rule to handle this situation. The natural rule is "depth" subtyping for records:

- "Depth" subtyping: If `ta <: tb`, then `{f1:t1,...,f:ta,...,fn:tn} <: {f1:t1,...,f:tb,...,fn:tn}`.

This rule lets us use width subtyping on the field `center` to show

```
{center:{x:real,y:real,z:real}, r:real} <: {center:{x:real,y:real}, r:real}
```

so the program above now type-checks.

Unfortunately, this rule breaks our type system, allowing programs that we do not want to allow to type-check! This may not be intuitive and programmers make this sort of mistake often — thinking depth subtyping should be allowed. Here is an example:

```
fun setToOrigin (c:{center:{x:real,y:real}, r:real})=
   c.center = {x=0.0, y=0.0}

val sphere:{center:{x:real,y:real,z:real}, r:real}) = {center={x=3.0,y=4.0,z=0.0}, r=1.0}
val _ = setToOrigin(sphere)
val _ = sphere.center.z
```

This program type-checks in much the same way: The call `setToOrigin(sphere)` has an argument of type `{center:{x:real,y:real,z:real}, r:real}` and uses it as a `{center:{x:real,y:real}, r:real}`. But what happens when we run this program? `setToOrigin` mutates its argument so the `center` field holds a record *with no z field!* So the last line, `sphere.center.z` will not work: it tries to read a field that does not exist.

The moral of the story is simple if often forgotten: In a language with records (or objects) with getters and setters for fields, depth subtyping is unsound — you cannot have a different type for a field in the subtype and the supertype.

Note, however, that if a field is not settable (i.e., it is immutable), then the depth subtyping rule is sound and, like we saw with `circleY`, useful. So this is yet another example of how not having mutation makes programming easier. In this case, it allows more subtyping, which lets us reuse code more.

Another way to look at the issue is that given the three features of (1) setting a field, (2) letting depth subtyping change the type of a field, and (3) having a type system actually prevent field-missing errors, you can have any two of the three.

The Problem With Java/C# Array Subtyping

Now that we understand depth subtyping is unsound if record fields are mutable, we can question how Java and C# treat subtyping for arrays. For the purpose of subtyping, arrays are very much like records, just with field names that are numbers and all fields having the same type. (Since `e1[e2]` computes what index to access and the type system does not restrict what index might be the result, we need all fields to have the same type so that the type system knows the type of the result.) So it should very much surprise us that this code type-checks in Java:

```
class Point { ... } // has fields double x, y
class ColorPoint extends Point { ... } // adds field String color
...
void m1(Point[] pt_arr) {
  pt_arr[0] = new Point(3,4);
}
String m2(int x) {
  ColorPoint[] cpt_arr = new ColorPoint[x];
  for(int i=0; i < x; i++)
     cpt_arr[i] = new ColorPoint(0,0,"green");
  m1(cpt_arr);
  return cpt_arr[0].color;
}
```

The call `m1(cpt_arr)` uses subtyping with `ColorPoint[] <: Point[]`, which is essentially depth subtyping even though array indices are mutable. As a result, it appears that `cpt_arr[0].color` will read the `color` field of an object that does not have such a field.

What actually happens in Java and C# is the assignment `pt_arr[0] = new Point(3,4);` will raise an exception if `pt_arr` is actually an array of `ColorPoint`. In Java, this is an `ArrayStoreException`. The advantage of having the store raise an exception is that no other expressions, such as array reads or object-field reads, need run-time checks. The invariant is that an object of type `ColorPoint[]` always holds objects that have type `ColorPoint` or a subtype, not a supertype like `Point`. Since Java allows depth subtyping on arrays, it cannot maintain this invariant statically. Instead, it has a run-time check on all array assignments, using the "actual" type of array elements and the "actual" class of the value being assigned. So even though in the type system `pt_arr[0]` and `new Point(3,4)` both have type `Point`, this assignment can fail at run-time.

As usual, having run-time checks means the type system is preventing fewer errors, requiring more care and testing, plus the run-time cost of performing these checks on array updates. So why were Java and C# designed this way? It seemed important for flexibility before these languages had generics so that, for example, if you wrote a method to sort an array of `Point` objects, you could use your method to sort an array of `ColorPoint` objects. Allowing this makes the type system simpler and less "in your way" at the expense of statically checking less. Better solutions would be to use generics in combination with subtyping (see bounded polymorphism in the next lecture) or to have support for indicating that a method will not update array elements, in which case depth subtyping is sound.

`null` in Java/C#

While we are on the subject of pointing out places where Java/C# choose dynamic checking over the "natural" typing rules, the far more ubiquitous issue is how the constant `null` is handled. Since this value has no fields or methods (in fact, unlike `nil` in Ruby, it is not even an object), its type should naturally reflect that it cannot be used as the receiver for a method or for getting/setting a field. Instead, Java and C# allow `null` to have *any* object type, as though it defines *every* method and has *every* field. From a static

checking perspective, this is exactly backwards. As a result, the language definition has to indicate that *every* field access and method call includes a run-time check for `null`, leading to the `NullPointerException` errors that Java programmers regularly encounter.

So why were Java and C# designed this way? Because there are situations where it is very convenient to have `null`, such as initializing a field of type `Foo` before you can create a `Foo` instance (e.g., if you are building a cyclic list). But it is also very common to have fields and variables that should never hold `null` and you would like to have help from the type-checker to maintain this invariant. Many proposals for incorporating "cannot be `null`" types into programming languages have been made, but none have yet "caught on" for Java or C#. In contrast, notice how ML uses option types for similar purposes: The types `t option` and `t` are not the same type; you have to use `NONE` and `SOME` constructors to build a datatype where values might or might not actually have a `t` value.

Function Subtyping

The rules for when one function type is a subtype of another function type are even less intuitive than the issue of depth subtyping for records, but they are just as important for understanding how to safely override methods in object-oriented languages (see below).

When we talk about function subtyping, we are talking about using a function of one type in place of a function of another type. For example, if `f` takes a function `g` of type `t1->t2`, can we pass a function of type `t3->t4` instead? If `t3->t4` is a subtype of `t1->t2` then this is allowed because, as usual, we can pass the function `f` an argument that is a subtype of the type expected. But this is not "function subtyping" on `f` — it is "regular" subtyping on function arguments. The "function subtyping" is deciding that one function type is a subtype of another.

To understand function subtyping, let's use this example of a higher-order function, which computes the distance between the two-dimensional point `p` and the result of calling `f` with `p`:

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end
```

The type of `distMoved` is

```
(({x:real,y:real}->{x:real,y:real}) * {x:real,y:real}) -> real
```

So a call to `distMoved` requiring no subtyping could look like this:

```
fun flip p = {x=~p.x, y=~p.y}
val d = distMoved(flip, {x=3.0, y=4.0})
```

The call above could also pass in a record with extra fields, such as `{x=3.0,y=4.0,color="green"}`, but this is just ordinary width subtyping on the second argument to `distMoved`. Our interest here is deciding what functions with types other than `{x:real,y:real}->{x:real,y:real}` can be passed for the first argument to `distMoved`.

First, it is safe to pass in a function with a return type that "promises" more, i.e., returns a subtype of the needed return type for the function `{x:real,y:real}`. For example, it is fine for this call to type-check:

```
fun flipGreen p = {x=~p.x, y=~p.y, color="green"}
val d = distMoved(flipGreen, {x=3.0, y=4.0})
```

The type of `flipGreen` is

```
{x:real,y:real} -> {x:real,y:real,color:string}
```

This is safe because `distMoved` expects a `{x:real,y:real}->{x:real,y:real}` and `flipGreen` is substitutable for values of such a type since the fact that `flipGreen` returns a record that also has a `color` field is not a problem.

In general, the rule here is that if `ta <: tb`, then `t -> ta <: t -> tb`, i.e., the subtype can have a return type that is a subtype of the supertype's return type. To introduce a little bit of jargon, we say return types are *covariant* for function subtyping meaning the subtyping for the return types works "the same way" (co) as for the types overall.

Now let us consider passing in a function with a different argument type. It turns out argument types are NOT covariant for function subtyping. Consider this example call to `distMoved`:

```
fun flipIfGreen p = if p.color = "green"
                    then {x=~p.x, y=~p.y}
                    else {x=p.x, y=p.y}
val d = distMoved(flipIfGreen, {x=3.0, y=4.0})
```

The type of `flipIfGreen` is

```
{x:real,y:real,color:string} -> {x:real,y:real}
```

This program should not type-check: If we run it, the expression `p.color` will have a "no such field" error since the point passed to `flipIfGreen` does not have a `color` field. In short, `ta <: tb`, does NOT mean `ta -> t <: tb -> t`. This would amount to using a function that "needs more of its argument" in place of a function that "needs less of its argument." This breaks the type system since the typing rules will not require the "more stuff" to be provided.

But it turns out it works just fine to use a function that "needs less of its argument" in place of a function that "needs more of its argument." Consider this example use of `distMoved`:

```
fun flipX_Y0 p = {x=~p.x, y=0.0}
val d = distMoved(flipX_Y0, {x=3.0, y=4.0})
```

The type of `flipX_Y0` is

```
{x:real} -> {x:real,y:real}
```

since the only field the argument to `flipX_Y0` needs is `x`. And the call to `distMoved` causes no problem: `distMoved` will always call its `f` argument with a record that has an `x` field and a `y` field, which is more than `flipX_Y0` needs.

In general, the treatment of argument types for function subtyping is "backwards" as follows: If `tb <: ta`, then `ta -> t <: tb -> t`. The technical jargon for "backwards" is *contravariance*, meaning the subtyping for argument types is the reverse (contra) of the subtyping for the types overall.

As a final example, function subtyping can allow contravariance of arguments and covariance of results:

```
fun flipXMakeGreen p = {x=~p.x, y=0.0, color="green"}
val d = distMoved(flipXMakeGreen, {x=3.0, y=4.0})
```

Here `flipXMakeGreen` has type

```
{x:real} -> {x:real,y:real,color:string}
```

This is a subtype of

```
{x:real,y:real} -> {x:real,y:real}
```

because `{x:real,y:real} <: {x:real}` (contravariance on arguments) and
`{x:real,y:real,color:string} <: {x:real,y:real}` (covariance on results).

The general rule for function subtyping is: If `t3 <: t1` and `t2 <: t4`, then `t1->t2 <: t3->t4`. This rule, combined with reflexivity (every type is a subtype of itself) lets us use contravariant arguments, covariant results, or both.

Argument contravariance is the least intuitive concept in the course, but it is worth burning into your memory so that you do not forget it. Many very smart people get confused because it is *not* about calls to methods/functions. Rather it is about the methods/functions themselves. We do not need function subtyping for passing non-function arguments to functions: we can just use other subtyping rules (e.g., those for records). Function subtyping is needed for higher-order functions or for storing functions themselves in records. And object types are related to having records with functions (methods) in them.

Subtyping for OOP

As promised, we can now apply our understanding of subtyping to OOP languages like Java or C#.

An object is basically a record holding fields (which we assume here are mutable) and methods. We assume the "slots" for methods are immutable: If an object's method `m` is implemented with some code, then there is no way to mutate `m` to refer to different code. (An instance of a subclass could have different code for `m`, but that is different than mutating a record field.)

With this perspective, sound subtyping for objects follows from sound subtyping for records and functions:

- A subtype can have extra fields.
- Because fields are mutable, a subtype cannot have a different type for a field.
- A subtype can have extra methods.
- Because methods are immutable, a subtype can have a subtype for a method, which means the method in the subtype can have contravariant argument types and a covariant result type.

That said, object types in Java and C# do not look like record types and function types. For example, we cannot write down a type that looks something like:

```
{fields : x:real, y:real, ...
 methods: distToOrigin : () -> real, ...}
```

Instead, we reuse class names as types where if there is a class `Foo`, then the type `Foo` includes in it all fields and methods implied by the class definition (including superclasses). And, as discussed previously, we also have interfaces, which are more like record types except they do not include fields and we use the name of the interface as a type. Subtyping in Java and C# includes only the subtyping explicitly stated via the subclass relationship and the interfaces that classes explicitly indicate they implement (including interfaces implemented by superclasses).

All said, this approach is more restrictive than subtyping requires, but since it does not allow anything it should not, it soundly prevents "field missing" and "method missing" errors. In particular:

- A subclass can add fields but not remove them
- A subclass can add methods but not remove them
- A subclass can override a method with a covariant return type
- A class can implement more methods than an interface requires or implement a required method with a covariant return type

Classes and types are different things! Java and C# purposely confuse them as a matter of convenience, but you should keep the concepts separate. A class defines an object's behavior. Subclassing inherits behavior, modifying behavior via extension and override. A type describes what fields an object has and what messages it can respond to. Subtyping is a question of substitutability and what we want to flag as a type error. So try to avoid saying things like, "overriding the method in the supertype" or, "using subtyping to pass an argument of the superclass." That said, this confusion is understandable in languages where every class declaration introduces a class and a type with the same name.

Covariant `self`/`this`

As a final subtle detail and advanced point, Java's `this` (i.e., Ruby's `self`) is treated specially from a type-checking perspective. When type-checking a class `C`, we know `this` will have type `C` or a subtype, so it is sound to assume it has type `C`. In a subtype, e.g., in a method overriding a method in `C`, we can assume `this` has the subtype. None of this causes any problems, and it is essential for OOP. For example, in class `B` below, the method `m` can type-check only if `this` has type `B`, not just `A`.

```
class A {
  int m(){ return 0; }
}
class B extends A {
  int x;
  int m(){ return x; }
}
```

But if you recall our manual encoding of objects in Racket, the encoding passed `this` as an extra explicit *argument* to a method. That would suggest *contravariant* subtyping, meaning `this` in a subclass could not have a *subtype*, which it needs to have in the example above.

It turns out `this` is special in the sense that while it is like an extra argument, it is an argument that is covariant. How can this be? Because it is not a "normal" argument where callers can choose "anything" of the correct type. Methods are always called with a `this` argument that is a subtype of the type the method expects.

This is the main reason why coding up dynamic dispatch manually works much less well in statically typed languages, even if they have subtyping: You need special support in your type system for `this`.

Generics Versus Subtyping

We have now studied both subtype polymorphism, also known as subtyping, and parametric polymorphism, also known as generic types, or just generics. So let's compare and contrast the two approaches, demonstrating what each is designed for.

What are generics good for?

There are many programming idioms that use generic types. We do not consider all of them here, but let's reconsider probably the two most common idioms that came up when studying higher-order functions.

First, there are functions that combine other functions such as `compose`:

```
val compose : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
```

Second, there are functions that operate over collections/containers where different collections/containers can hold values of different types:

```
val length : 'a list -> int
val map : ('a -> 'b) -> 'a list -> 'b list
val swap : ('a * 'b) -> ('b * 'a)
```

In all these cases, the key point is that if we had to pick non-generic types for these functions, we would end up with significantly less code reuse. For example, we would need one `swap` function for producing an `int * bool` from a `bool * int` and another `swap` function for swapping the positions of an `int * int`.

Generic types are much more useful and precise than just saying that some argument can "be anything." For example, the type of `swap` indicates that the second component of the result has the same type as the first component of the argument and the first component of the result has the same type as the second component of the argument. In general, we reuse a type variable to indicate when multiple things can have any type but must be the same type.

Generics in Java

Java has had subtype polymorphism since its creation in the 1990s and has had parametric polymorphism since 2004. Using generics in Java can be more cumbersome without ML's support for type inference and, as a separate matter, closures, but generics are still useful for the same programming idioms. Here, for example, is a generic `Pair` class, allowing the two fields to have any type:

```
class Pair<T1,T2> {
  T1 x;
  T2 y;
  Pair(T1 _x, T2 _y){ x = _x; y = _y; }
  Pair<T2,T1> swap() {
     return new Pair<T2,T1>(y,x);
  }
  ...
}
```

Notice that, analogous to ML, "`Pair`" is not a type: something like `Pair<String,Integer>` is a type. The `swap` method is, in object-oriented style, an instance method in `Pair<T1,T2>` that returns a `Pair<T2,T1>`. We could also define a static method:

```
static <T1,T2> Pair<T2,T1> swap(Pair<T1,T2> p) {
   return new Pair<T2,T1>(p.y,p.x);
}
```

For reasons of backwards-compatibility, the previous paragraph is not quite true: Java also has a type `Pair` that "forgets" what the types of its fields are. Casting to and from this "raw" type leads to compile-time warnings that you would be wise to heed: Ignoring them can lead to run-time errors in places you would not expect.

Subtyping is a Bad Substitute for Generics

If a language does not have generics or a programmer is not comfortable with them, one often sees generic code written in terms of subtyping instead. Doing so is like painting with a hammer instead of a paintbrush: technically possible, but clearly the wrong tool. Consider this Java example:

```
class LamePair {
  Object x;
  Object y;
  LamePair(Object _x, Object _y){ x=_x; y=_y; }
  LamePair swap() { return new LamePair(y,x); }
  ...
}

String s = (String)(new LamePair("hi",4).y); // error caught only at run-time
```

The code in `LamePair` type-checks without problem: the fields `x` and `y` have type `Object`, which is a supertype of every class and interface. The difficulties arise when clients use this class. Passing arguments to the constructor works as expected with subtyping.[3] But when we retrieve the contents of a field, getting an `Object` is not very useful: we want the type of value we put back in.

Subtyping does not work that way: the type system knows only that the field holds an `Object`. So we have to use a *downcast*, e.g., `(String)e`, which is a run-time check that the result of evaluating `e` is actually of type `String`, or, in general, a subtype thereof. Such run-time checks have the usual dynamic-checking costs in terms of performance, but, more importantly, in terms of the possibility of failure: this is not checked statically. Indeed, in the example above, the downcast would fail: it is the `x` field that holds a `String`, not the `y` field.

In general, when you use `Object` and downcasts, you are essentially taking a dynamic typing approach: any object could be stored in an `Object` field, so it is up to programmers, without help from the type system, to keep straight what kind of data is where.

What is Subtyping Good For?

We do not suggest that subtyping is not useful: It is great for allowing code to be reused with data that has "extra information." For example, geometry code that operates over points should work fine for colored-points. It is certainly inconvenient in such situations that ML code like this simply does not type-check:

```
fun distToOrigin1 {x=x,y=y} =
```

[3] Java will automatically convert a 4 to an `Integer` object holding a 4.

```
   Math.sqrt (x*x + y*y)

(* does not type-check *)
(* val five = distToOrigin1 {x=3.0,y=4.0,color="red"} *)
```

A generally agreed upon example where subtyping works well is graphical user interfaces. Much of the code for graphics libraries works fine for any sort of graphical element ("paint it on the screen," "change the background color," "report if the mouse is clicked on it," etc.) where different elements such as buttons, slider bars, or text boxes can then be subtypes.

Generics are a Bad Substitute for Subtyping

In a language with generics instead of subtyping, you can code up your own code reuse with higher-order functions, but it can be quite a bit of trouble for a simple idea. For example, `distToOrigin2` below uses getters passed in by the caller to access the `x` and `y` fields and then the next two functions have different types but identical bodies, just to appease the type-checker.

```
fun distToOrigin2(getx,gety,v) =
    let
        val x = getx v
        val y = gety v
    in
        Math.sqrt (x*x + y*y)
    end

fun distToOriginPt (p : {x:real,y:real}) =
    distToOrigin2(fn v => #x v,
                  fn v => #y v,
                  p)

fun distToOriginColorPt (p : {x:real,y:real,color:string}) =
    distToOrigin2(fn v => #x v,
                  fn v => #y v,
                  p)
```

Nonetheless, without subtyping, it may sometimes be worth writing code like `distToOrigin2` if you want it to be more reusable.

Bounded Polymorphism

As Java and C# demonstrate, there is no reason why a statically typed programming language cannot have generic types and subtyping. There are some complications from having both that we will not discuss (e.g., static overloading and subtyping are more difficult to define), but there are also benefits. In addition to the obvious benefit of supporting separately the idioms that each feature supports well, we can combine the ideas to get even more code reuse and expressiveness.

The key idea is to have *bounded generic types*, where instead of just saying "a subtype of `T`" or "for all types `'a`," we can say, "for all types `'a` that are a subtype of `T`." Like with generics, we can then use `'a` multiple times to indicate where two things must have the same type. Like with subtyping, we can treat `'a` as a subtype of `T`, accessing whatever fields and methods we know a `T` has.

We will show an example using Java, which hopefully you can follow just by knowing that `List<Foo>` is the syntax for the type of lists holding elements of type `Foo`.

Consider this `Point` class with a `distance` method:

```
class Pt {
    double x, y;
    double distance(Pt pt) { return Math.sqrt((x-pt.x)*(x-pt.x)+(y-pt.y)*(y-pt.y));  }
    Pt(double _x, double _y) { x = _x; y = _y; }
}
```

Now consider this static method that takes a list of points `pts`, a point `center`, and a radius `radius` and returns a new list of points containing all the input points within `radius` of `center`, i.e., within the circle defined by `center` and `radius`:

```
static List<Pt> inCircle(List<Pt> pts, Pt center, double radius) {
    List<Pt> result = new ArrayList<Pt>();
    for(Pt pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      return result;
}
```

(Understanding the code in the method body is not important.)

This code works perfectly fine for a `List<Pt>`, but if `ColorPt` is a subtype of `Pt` (adding a `color` field and associated methods), then we cannot call `inCircle` method above with a `List<ColorPt>` argument. Because depth subtyping is unsound with mutable fields, `List<ColorPt>` is not a subtype of `List<Pt>`. Even if it were, we would like to have a result type of `List<ColorPt>` when the argument type is `List<ColorPt>`.

For the code above, this is true: If the argument is a `List<ColorPt>`, then the result will be too, but we want a way to express that in the type system. Java's bounded polymorphism lets us describe this situation (the syntax details are not important):

```
static <T extends Pt> List<T> inCircle(List<T> pts, Pt center, double radius) {
  List<T> result = new ArrayList<T>();
    for(T pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      return result;
}
```

This method is polymorphic in type `T`, but `T` must be a subtype of `Pt`. This subtyping is necessary so that the method body can call the `distance` method on objects of type `T`. Wonderful!

Optional: Additional Java-Specific Bounded Polymorphism

While the second version of `inCircle` above is ideal, let us now consider a few variations. First, Java does have enough dynamically checked casts that it is possible to use the first version with a `List<ColorPt>` argument and cast the result from `List<Pt>` to `List<ColorPt>`. We have to use the "raw type" `List` to do it, something like this where `cps` has type `List<ColorPt>`.

```
List<ColorPt> out = (List<ColorPt>)(List) inCircle((List<Pt>)(List)cps, new Pt(0.0,0.0), 1.5);
```

In this case, these casts turn out to be okay: if `inCircle` is passed a `List<ColorPt>` the result will be a `List<ColorPt>`. But casts like this are dangerous. Consider this variant of the method that has the same type as the initial non-generic `inCircle` method:

```
static List<Pt> inCircle(List<Pt> pts, Pt center, double radius) {
    List<Pt> result = new ArrayList<Pt>();
    for(Pt pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      else
        result.add(center);
      return result;
}
```

The difference is that any points not within the circle are "replaced" in the output by `center`. Now if we call `inCircle` with a `List<ColorPt> cps` where one of the points is not within the circle, then the result is *not* a `List<ColorPt>` — it contains a `Pt` object! You might expect then that the cast of the result to `List<ColorPt>` would fail, but Java does not work this way for backward-compatibility reasons: even this cast succeeds. So now we have a value of type `List<ColorPt>` that is not a list of `ColorPt` objects. What happens instead in Java is that a cast will fail later when we get a value from this alleged `List<ColorPt>` and try to use it as `ColorPt` when it is in fact a `Pt`. The blame is clearly in the wrong place, which is why using the warning-inducing casts in the first place is so problematic.

Last, we can discuss what type is best for the `center` argument in our bounded-polymorphic version. Above, we chose `Pt`, but we could also choose `T`:

```
static <T extends Pt> List<T> inCircle(List<T> pts, T center, double radius) {
    List<T> result = new ArrayList<T>();
    for(T pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      return result;
}
```

It turns out this version allows *fewer* callers since the previous version allows, for example, a first argument of type `List<ColorPt>` and a second argument of type `Pt` (and, therefore, via subtyping, also a `ColorPt`). With the argument of type `T`, we require a `ColorPt` (or a subtype) when the first argument has type `List<ColorPt>`. On the other hand, our version that sometimes adds `center` to the output requires the argument to have type `T`:

```
static <T extends Pt> List<T> inCircle(List<T> pts, T center, double radius) {
    List<T> result = new ArrayList<T>();
    for(T pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      else
        result.add(center);
      return result;
}
```

In this last version, if `center` has type `Pt`, then the call `result.add(center)` does not type-check since `Pt` may not be a subtype of `T` (what we know is `T` is a subtype of `Pt`). The actual error message may be a bit confusing: It reports there is no `add` method for `List<T>` that takes a `Pt`, which is true: the `add` method we are trying to use takes a `T`.