Coursera Programming Languages Course
Section 7 Summary

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

Contents

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.[1]

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."

[1]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.

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 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.[2]

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.

[2]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.

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, 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 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."

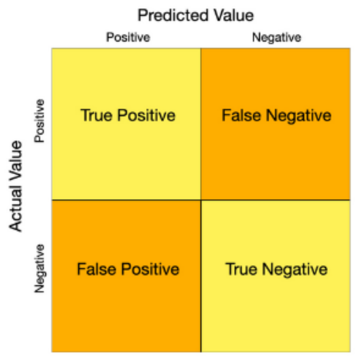
In modern languages, type systems are sound (they prevent what they claim to) but not complete (they reject programs they need not reject). Soundness is important because it lets language users and language implementers rely on X never happening. Completeness would be nice, but hopefully it is rare in 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. 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.

A type system is considered "sound" if it **never accepts/prevent** a program that, when run with some input, **produces behavior or errors** described as "X". This means that no errors are caused by a program that is accepted by the type system.

On the other hand, a type system is considered "complete" if it **never rejects/let** a program that will **not produce behavior or errors** described as "X", regardless of the input used.



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`.[3]

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

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 section 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*.