CSE341: Programming Languages Spring 2016
Unit 6 Summary

Standard Description: This summary covers **roughly** *the same material as class and recitation 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*.