David Liu

Principles of Programming Languages

Lecture Notes for CSC324 (Version 2.1)

Department of Computer Science

University of Toronto

Many thanks to Alexander Biggs, Peter Chen, Rohan Das, Ozan Erdem, Itai David Hass, Hengwei Guo, Kasra Kyanzadeh, Jasmin Lantos, Jason Mai, Sina Rezaeizadeh, Ian Stewart-Binks, Ivan Topolcic, Anthony Vandikas, Lisa Zhang, and many anonymous students for their helpful comments and error-spotting in earlier versions of these notes.

Dan Zingaro made substantial contributions to this version of the notes.

Contents

Prelude: The Study of Programming Languages

> It seems to me that there have been two really clean, consistent models of programming so far: the C model and the Lisp model. These two seem points of high ground, with swampy lowlands between them.
>
> Paul Graham

As this is a "programming languages" course, you might be wondering: are we going to study new programming languages, much in the way that we studied Python in CSC108 or even Java in CSC207? Yes and no.

You will be introduced to new programming languages in this course; most notably, Racket and Haskell. However, unlike more introductory courses like CSC108 and CSC207, in this course we leave learning the basics of these new languages up to you. *How do variable assignments work? What is the function that finds the leftmost occurrence of an element in a list? Why is this a syntax error?* These are the types of questions that we expect you to be able to research and solve on your own.[1] Instead, we focus on the ways in which these languages allow us to *express ourselves*; that is, we'll focus on particular affordances of these languages, considering the design and implementation choices the creators of these languages made, and compare these decisions to more familiar languages you have used to this point.

[1] Of course, we'll provide useful tutorials and links to standard library references to help you along, but it will be up to you to use them.

Programs and programming languages

We start with a simple question: what is a program? We are used to thinking about programs in one of two ways: as an active entity on our computer that does something when run; or as the source code itself, which tells the computer what to do. As we develop more and more sophisticated programs for more targeted domains, we often lose sight of one crucial fact: that code is not itself the goal, but instead a means of *communication* to the computer describing what we want to achieve.

A programming language, then, isn't just the means of writing code, but a true *language* in the common sense of the word. Unlike what linguists call natu-

ral languages, which often carry ambiguity, nuance, and errors, programming languages target machines, and so must be precise, unambiguous, and perfectly understandable by mechanical algorithms alone. This makes the rules governing programming languages quite inflexible, which is often a source of trouble from beginners. Yet once mastered, the clarity afforded by these languages enables humans to harness the awesome computational power of modern technology.

But even this lens of programming languages as communication is incomplete. Unlike natural languages, which have evolved over millennia, often organically without much deliberate thought,[2] programming languages are not even a century old, and were explicitly designed by humans. As programmers, we tend to lose sight of this, taking our programming language for granted—quirks and oddities and all. But programming languages exhibit the same fascinating design questions, trade-offs, and limitations that are inherent in all software design. Indeed, the various software used to *implement* programming languages—that is, to take human-readable source code and enable a computer to understand it—are some of the most complex and sophisticated programs in existence today.

[2] This is not to minimize the work of language deliberative bodies like the Oxford English Dictionary, but to point out that language evolves far beyond what may be prescribed.

The goal of this course, then, is to stop taking programming languages for granted; to go deeper, from users of programming languages to understanding the design and implementation of these languages.

Syntax and grammars

The **syntax** of a programming language is the set of rules governing what the allowed expressions of a programming language can look like; these are the rules governing allowed program structure. The most common way of specifying the syntax of a language is through a *grammar*, which is a formal description of how to generate expressions by substitution. For example, the following is a simple grammar to generate arithmetic expressions:

```
1 <expr> = NUMBER
2        | '(' <expr> <op> <expr> ')'
3
4 <op>   = '+' | '-' | '*' | '/'
```

We say that the left-hand side names `<expr>` and `<op>` are *non-terminal symbols*, meaning that we generate valid expressions by substituting for them using these grammar rules. By convention, we'll put angle brackets around all non-terminal symbols.

The all-caps `NUMBER` is a terminal symbol, representing any numeric literal (e.g., `3` or `-1.5`).[3] The vertical bar `|` indicates alternatives (read it as "or"): for example, `<op>` can be replaced by any one of the strings `'+'`, `'-'`, `'*'`, or `'/'`.

[3] We'll typically use `INTEGER` when we need to instead specify any integral literal.

It is important to note that this grammar is recursive, as an `<expr>` can be replaced by more occurrences of `<expr>`. This should match your intuition about

both arithmetic expressions and programs themselves, which can both contain subparts of arbitrary length and nesting!

With these grammars in hand, it is easy to specify the syntax of a programming language: an expression is **syntactically valid** if and only if it can be generated by the language's grammar. For the arithmetic expression language above, the expressions `(3 + 5)` and `((4 - 2) * (5 / 10))` are syntactically valid, but `(3 +)` and `(3 - + * 5)` and even `3 + 5` are not.

Abstract syntax trees

Source code is merely representation of a program; as text, it is useful for communicating ideas among humans, but not at all useful for a machine. To "run" a program requires the computer to operate on the program's source code—but as you have surely experienced before, working purely with strings is often tricky and cumbersome, as that datatype is far less structured than what its contents suggest.

Therefore, one of the key steps for any operation on a program is to **parse** it, which here means to convert the source code—a string—into a more structured representation of the underlying program. Because the source code can only represent a valid program if it is syntactically valid, parsing is always informed by the programming language's grammar, and is the stage at which syntax errors are detected. The problem of parsing text is a rich and deep problem in its own right, but due to time constraints we won't spend time discussing parsing techniques in these notes, instead relying on either the simplicity of the language syntax (Racket) or built-in code parsing tools (Python) to resolve this task for us.

What we will focus on is the output of parsing: this "more structured representation of the underlying program." The most common representation, used by interpreters and compilers for virtually every programming language, is the **abstract syntax tree (AST)**, as trees are a natural way to represent the inherently hierarchical and recursive organization of programs into smaller and smaller components. While different programming languages and even different compilers or interpreters for the same language will differ in their exact AST representation, generally speaking all abstract syntax trees share the following properties:

1. A *leaf* of the tree represents an expression that has no subexpressions, e.g. a literal value (`5`, `"hello"`) or identifier (`person`). We call such expressions *atomic*.
2. Each *internal node* of the tree represents a compound expression, i.e., one that is built out of smaller subexpressions. These encompass most of the programming forms you are familiar with, including control flow structures, definitions of functions/types/classes, arithmetic operations and function calls, etc.
3. Nodes can be categorized by the kind of expression they represent, e.g. through an explicit "tag" string, using an object-oriented approach (a `Node` class hierarchy), or using algebraic data types.[4] So, for example, we can speak of "the literal value nodes" or "the function definition nodes" within an AST.

[4] This last one is likely new to you. We'll study algebraic data types later on in this course!

4. AST node types are often in rough correspondence to the grammar rules of a language.[5] So for example, the language's syntax could contain a grammar rule for what a function definition looks like:

```
<function-def> = 'def' <id> '(' [<id> ','] ... ')' ':' '\n' <body>
```

   And when parsed, a program's AST might include a "function definition" node with children for the name, parameters, and body of the function.

[5] Note that this is not necessarily exact; we are skipping over details of parsing and basic syntax analysis that many compilers often do before producing an AST.

As we'll start to see in the next chapter, abstract syntax trees enable us to avoid the idiosyncracies of program syntax and instead get to interesting operations on programs themselves.

Semantics and evaluation

You may have noticed in the previous section that we used the term *expression* to describe the inputs of our parsing. This may strike you as a little strange, since the term *program* usually connotes much more than this. Most modern programming languages, including Python, Java, and C, are fundamentally *imperative* in nature: inspired by the Turing machine model of computation, programs in these languages are organized around *statements* corresponding to instructions that the computer should execute.[6] In this model, we think of "running" a program as telling the computer to execute the instructions found in our program; the result of running the program is whatever happens when these instructions are executed.

[6] "Statement" here includes larger syntactic "block" structures like loops.

While it is certainly a familiar model of computation, and tracks closely to what computer hardware actually requires, one of the downsides of this model is its inherent complexity. In order to understand what a program means, we need to understand what each kind of statement does, and how it impacts control flow and underlying memory. The **semantics** of a programming language are the rules governing the meaning of programs written in that language; for imperative style programs, we need to describe the meaning not just of individual expressions like `3 + 5`, but also the meaning of `return` (interrupts control flow), `for` (iterates through a specified range), and other keywords.

To simplify matters, we'll stick with the easier task of understanding expression-based programs, in which a program is just an expression. In this model, running a program means telling the computer to *evaluate* the expression; the result of running the program is simply the value of the expression after it has been evaluated.[7]

[7] Of course, all imperative languages have a notion of "evaluating expressions"; it's just that those languages include a bunch of other stuff as well.

So the semantics of an expression-based language govern what the *value* of such programs are. This might seem simple, but it's worth spelling out explicitly, because there are actually multiple ways of studying such semantics.

The **denotational semantics** of a programming language specify the abstract value of a program, drawing on formal definitions (e.g., from mathematics). We

won't go into the details here, but instead rely on your intuitions from mathematics and basic programming. In the space below, we've listed several programs (each consisting of just a single Python expression) that have the same denotational value 10:

```
10

3 + 7

1 + 3 ** 2

ord('\n')

(lambda x: x + 3)(7)

list(range(50000000))[11]
```

That is, each of the above expressions, while written and parsed differently, produces the same result when evaluated; we would say that they have the same *mathematical* meaning, or the same value.

However, your gut probably tells you that this isn't the full story. After all, even though these expressions might evaluate to the same value, *how* they each get to that value is quite different. The **operational semantics** of a programming language specify the *evaluation steps* used to determine the value of a program. In imperative-style languages, it is the operational semantics that are hardest to specify, as they deal with complexities of control flow, mutation, and function calls. As we'll see in the next chapter, specifying the operational semantics of expression evaluation alone—especially in a functional context—is generally straightforward.

While we will focus on denotational and operational semantics in this course, it is worth mentioning one other kind of semantics that comes up in programming languages. This is **axiomatic semantics**, where rather than focus on evaluation, we focus on what is true about each piece of a code segment. For example, we might argue that "this loop maintains the invariant that `sum` is the sum of the first `i` integers in list `L`". Sound familiar? You used some of the axiomatic tools—invariants, variants, pre/postconditions—already in CSC236!

Models of computation

It was in the 1930s, years before the invention of the first electronic computing devices, that a young mathematician named Alan Turing created modern computer science as we know it. Incredibly, this came about almost by accident; he had been trying to solve a problem from mathematical logic: the *Entscheidungsproblem* ("decision problem"), which asks whether an algorithm could decide if a logical statement is provable from a given set of axioms. Turing showed

that no such algorithm exists. To answer this question, Turing developed an abstract model of mechanical, procedural computation: a machine that could read in a string of 0's and 1's, a finite state control[8] that could make decisions and write 0's and 1's to its internal memory, and an output space where the computation's result would be displayed. Though its original incarnation was an abstract mathematical object, the fundamental mechanism of the Turing machine—reading data, executing a sequence of instructions to modify internal memory, and producing output—would soon become the von Neumann architecture lying at the heart of modern computers, and seed the paradigm of imperative programming.

[8] Finite state controls are analogous to the *deterministic finite automata* that you learned about in CSC236.

The story of Alan Turing and his machine is one of great genius, great triumph, and great sadness. It is no exaggeration to say that the fundamentals of computer science owe their genesis to this man. Their legacy is felt by every computer scientist, software engineer, and computer engineer alive today.

But there is another story, too.

Alonzo Church

Shortly before Turing published his paper introducing the Turing machine, the logician Alonzo Church had published a paper resolving the same fundamental problem using entirely different means. At the same time that Turing was developing his model of the Turing machine, Church was drawing inspiration from the mathematical notion of *functions* to model computation. Church would later act as Turing's PhD advisor at Princeton, where they showed that their two radically different notions of computation were in fact equivalent: any problem that could be solved in one could be solved in the other. They went a step further and articulated the **Church-Turing Thesis**, which says that *any* reasonable computational model would be just as powerful as their two models. And incredibly, this bold claim still holds true today. With all of our modern technology, we are still limited by the mathematical barriers erected eighty years ago.[9]

[9] To make this amazing idea a little more concrete: no existing programming language and accompanying hardware will ever solve the Decision Problem. None.

And yet to most computer scientists, Turing is much more familiar than Church; the von Neumann architecture is what drives modern hardware design; the most commonly used programming languages today revolve around *state* and *time*, *instructions* and *memory*, the cornerstones of the Turing machine. What were Church's ideas, and why don't we know more about them?

The lambda calculus

The imperative programming paradigm derived from Turing's model of computation has as its fundamental unit the *statement*, a portion of code representing some instruction or command to the computer.[10] Though such statements are composed of subexpressions, these expressions typically do not appear on their own; consider the following odd-looking, but valid, Python program:

[10] For non-grammar buffs, the *imperative* verb tense is what we use when issuing orders: "*Give* me that" or "*Stop* talking, David!"

```
def f(a):
    12 * a - 1
    a
    'hello' + 'goodbye'
```

Even though all three expressions in the body of `f` are evaluated each time the function is called, they are unable to influence the output of this function. We require sequences of *statements* (including keywords like `return`) to do anything useful at all! Even function calls, which might look like standalone expressions, are only useful if the bodies of those functions contain statements for the computer to execute.

In contrast to this instruction-based approach, Alonzo Church created a model called the **lambda calculus** in which expressions themselves are the fundamental, and in fact only, unit of computation. Rather than a program being a sequence of statements, in the lambda calculus a program is a single expression (possibly containing many subexpressions). And when we say that a computer *runs a program*, we do not mean that it performs operations corresponding to statements, but rather that it *evaluates* that single expression.

Two questions arise from this notion of computation: what do we really mean by the words "expression" and "evaluate"? Or in other words, what are the syntax and semantics of the lambda calculus? This is where Church borrowed functions from mathematics, and why the programming paradigm that this model spawned is called functional programming. In the lambda calculus, an expression is one of three things:

1. An **identifier** (or **variable**): a, x, *yolo*, etc.
2. A **function expression**: $\lambda x.x$, for example. This expression represents a function that takes one parameter x, and returns it—in other words, this is the identity function.
3. A **function application** (or **function call**): f *expr*. This expression applies the function f to the expression *expr*.

Now that we have defined our allowable expressions, what do we mean by evaluating them? To evaluate an expression means performing simplifications to it until it cannot be further simplified; we'll call the resulting fully-simplified expression the *value* of the expression.

This definition meshes well with our intuitive notion of evaluation, but we've really just shifted the question: what do we mean by "simplifications?" In fact, in the lambda calculus, identifiers and function expressions have no simplification rules: in other words, they are themselves values, and are fully simplified. On the other hand, function application expression *can* be simplified, using the idea of substitution from mathematics. For example, suppose we apply the identity function to the variable *hi*:

$$(\lambda x.x)\ hi$$

We evaluate this by *substituting hi* for x in the body of the function, obtaining *hi* as a result.

Pretty simple, eh? As surprising as this may be, function-application-as-substitution is the *only* simplification rule for the lambda calculus! So if you can answer questions like "If $f(x) = x^2$, then what is $f(5)$?" then you'll have no trouble understanding the lambda calculus.

The main takeaway from this model is that function application (via substitution) is the only mechanism we have to induce computation; functions can be created using λ and applied to values and even other functions, and through combining functions we create complex computations. A point we'll return to again and again in this course is that functions in the lambda calculus are far more restrictive than the functions we're used to from previous programming experience. The only thing we can do in the lambda calculus when evaluating a function application is substitute the arguments into the function body, and then evaluate that body, producing a single value. These functions have no concept of *time* to require a certain sequence of *instructions*, nor is there any external or global *state* that can influence their behaviour.

At this point the lambda calculus may seem at best like a mathematical curiosity. What does it mean for everything to be a function? Certainly there are things we care about that *aren't* functions, like numbers, strings, classes and every data structure you've studied up to this point—right? But because the Turing machine and the lambda calculus are equivalent models of computation, anything you can do in one, you can also do in the other! So *yes*, we can use functions to represent numbers, strings, and data structures; we'll see this only a little in this course, but rest assured that it can be done.[11] And though the Turing machine is more widespread, the beating heart of the lambda calculus is still alive and well, and learning about it will make you a better computer scientist.

[11] If you'd like to do some reading on this topic, look up *Church encodings*.

A paradigm shift in you

The influence of Church's lambda calculus is most obvious today in the *functional programming paradigm*, a function-centric approach to programming that has heavily influenced languages such as Lisp (and its dialects), ML, Haskell, and F#. You may look at this list and think "I'm never going to use these in the real world," but support for functional programming styles is being adopted in more "mainstream" languages, such as LINQ in C# and lambdas in Java 8. Other languages like Python and JavaScript have supported the functional programming paradigm since their inception.

The goal of this course is not to convert you into the *Cult of FP*, but to open your mind to different ways of solving problems. After all, the more tools you have at your disposal in "the real world," the better you'll be at picking the best one for the job.

Along the way, you will gain a greater understanding of different programming language properties, which will be useful to you whether you are exploring new

languages or studying how programming languages interact with compilers and interpreters, an incredibly interesting field in its own right.[12]

[12] Those of you who are particularly interested in compilers should take CSC488.

Course overview

Chapter 1. We will begin our study of functional programming with two new languages: Racket, a dialect of Lisp commonly used for both teaching and language research, and Haskell, a pure functional programming language with an elaborate and powerful static type system. It might seem like overkill to use two different languages, and we are certainly very conscious of this! Our pedagogical goal here is to reinforce the idea that we are not learning specialized idiosyncrasies of a particular language. Rather, we want to focus on the guiding high-level principles that exist in many different languages, and we believe that the best way to do this is to explore *how* these principles are expressed in multiple languages at once, to gain a deeper understanding of these ideas. We will explore language design features like scope, function call strategies, and tail recursion, comparing Racket and Haskell with each other and with more familiar languages like Python and Java. We will also use this as an opportunity to gain lots of experience with functional programming idioms: (structural) recursion; the list functions map, filter, and fold; and higher-order functions and closures.

Chapter 2. In the next section of the course, we'll do a deep dive into two substantial programming language features: object-oriented programming and (simulating) non-deterministic choices. Rather than study just the features themselves, we'll take the approach of a programming language designer, and ask the question "How do we implement such a feature?" This lens will give us more than just a greater understanding of these language features. Our quest for user-friendly implementations will lead us to *the* defining feature of Racket: a powerful macro system that allows us to extend the very syntax and semantics of a programming language.[13]

[13] Put another way, to allow us to introduce new *kinds of nodes* into an abstract syntax tree.

Chapter 3. If the preceding chapter is all about macros being used to express new concepts and paradigms in a language, our last section of the course is dual to this: expressing *constraints* in a language, through the creation of types. We are all familiar with types; but as with other aspects of programming languages, in this course we'll study types with more attention to detail and creativity than you likely have in the past. In particular, we'll explore Haskell's powerful static type system, and see how to use it to express not just simple notions like "don't add a number to a string", but more abstract concepts like failing computations, mutable state, and external I/O, all with strong guarantees from our Haskell compiler.[14]

[14] One particularly nifty feature we'll talk about is *type inference*, a Haskell compiler feature that means we get all the benefits of static typing without the verbosity of Java's Kingdom of Nouns.

1 *Functional Programming: Theory and Practice*

> Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.
>
> Greenspun's tenth rule of programming

In 1958, John McCarthy created **Lisp**, a bare-bones programming language based on Church's lambda calculus.[1] Since then, Lisp has spawned many dialects (languages based on Lisp with some deviations from its original specifications), among which are Common Lisp, Clojure (which compiles to the Java Virtual Machine), and Racket (a language used actively in educational and programming language research contexts).

[1] Lisp itself is still used to this day; in fact, it has the honour of being the second-oldest programming language still in use. The oldest? Fortran.

In 1987, it was decided at the conference *Functional Programming Languages and Computer Architecture*[2] to form a committee to consolidate and standardize existing non-strict functional languages, and so Haskell was born (we'll study what the term "non-strict" means in this chapter). Though mainly still used in the academic community for research, Haskell has become more widespread as functional programming has become, well, more mainstream. Like Lisp, Haskell is a functional programming language: its main mode of computation involves defining pure functions and combining them to produce complex computation. However, Haskell has many differences from the Lisp family, both immediately noticeable and profound.

[2] Now part of this one: `http://www.icfpconference.org/`

Our goal in this chapter is to expose you to some of the central concepts in programming language theory, and functional programming in particular, without being constrained to one particular language. So in this chapter, we'll draw on examples from three languages: Racket, Haskell, and Python. Our hope here is that by studying the similarities and differences between these languages, you'll gain more insights into the deep concepts in this chapter than by studying any one of these languages alone.

The baseline: "universal" built-ins

While one of the major strengths of the lambda calculus as a model of computation is its simplicity, in practice we want built-in data types and functions to scaffold our programs. Of course, programming languages vary in how they build in their data types and the operations that they support, and so we'll restrict most of our attention in this course to a fairly conservative set of built-ins, common to all three languages:

- Primitive data types and literals: integers, floats, booleans, strings
- Compound data types: lists and maps
- Built-in functions on these data types[3]
- Boolean operations (and, or, not) and if *expressions* (aka "ternary ifs")

[3] Note that different languages will have different names for these functions; it'll be up to you to consult documentation regularly.

Function expressions

With the built-ins out of the way, we'll now turn to one of the first "new" aspects, which is one of the central ideas of functional programming: functions are *first-class* values, meaning that they can be treated and manipulated in the exact same way as other kinds of values in a program.

This is actually a big idea! Many languages do *not* support functions as first-class values, as there are some things you can do with other kinds of values that you can't do with functions. One of these is simple: what is the equivalent of a standalone "function value" in a language?

Suppose we want to represent the integer value three in a program. We take for granted how easy it is to do so: simply write the *numeric literal* `3`! But suppose we want to represent a function that "takes a number and adds 3 to it"; in Python, you would probably write:

```
1 def f(x):
2     return x + 3
```

But there's one big difference here: in the former case, we had the value `3` by itself, whereas in the latter case, there is both the function *and* a name `f` associated with the function. In some programming languages, it is *only* possible to define function values together with a given identifier to refer to the function. However, if we want to functions to be first-class values, then we had better be able to write them standalone, independent of a name binding! A function value expressed independently of an identifier is called an **anonymous function**.

In the lambda calculus, all function function are anonymous: $(\lambda x.x)$ is a function value, but doesn't have an associated name. All three of Racket, Haskell, and Python support anonymous functions as well, using the following syntax, each inspired in its own way by the lambda calculus.[4]

[4] We'll use background colours to distinguish languages in code blocks, with yellow for Racket, blue for Haskell, and gray for all others.

```racket
; Racket
(lambda (<param> ...) <body>)
```

```haskell
-- Haskell
\<param> ... -> <body>
```

```python
# Python
lambda <param> ... : <body>
```

In each of the above examples, each `<param>` is called a **parameter** of the function, and must be an identifier. The `<body>` is an expression called the **body** of the function.

For programmers that have never seen anonymous functions before, such functions might seem strange: what's the point of writing a function if you don't have a name to refer to it? While we'll see some motivating examples later in this chapter, for now we'll leave you with a different question: does every expression you write have a name?

Function application

Calling functions in each of the three languages is straightforward, but note that both Racket and Haskell use an unfamiliar syntax.

First, in Python a function call looks like most languages you've probably worked with before:

```
<function>(<arg>, ...)
```

In Racket, the function expression goes *inside* the parentheses:[5]

[5] This syntax is known as *Polish prefix notation*.

```racket
(<function> <arg> ...)
```

One thing that trips students up is that in Racket, every parenthesized expression is treated as a function call, except for the ones starting with keywords like `lambda`. This is in stark contrast with most programming languages, in which expressions are often enclosed in (redundant) parentheses to communicate grouping explicitly. Racket doesn't have the concept of "redundant" parentheses!

In Haskell, parentheses are not required at all; instead, any two expressions separated by a space are considered a function call:

```
<function> <arg> ...
```

Operators are functions

Consider common binary arithmetic operations like addition and multiplication, which we normally think of as being written *infix*, i.e., between its two argument expressions: `3 + 4` or `1.5 * 10`.

Again in the theme of the centrality of functions to our programming, it's important to realize that in fact, these operators are just functions, at least from a mathematical point of view.[6] In fact, all three of Racket, Haskell, and Python treat them as functions!

[6] Formally, we might write something like $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ to represent the $+$ operation as taking two real numbers and outputting a real number.

In Racket, operators are just identifiers that refer to built-in functions, and are called the same way any other function would be called:

```
> (+ 10 20)
30
> (* 3 5)
15
```

In Python, operators are implemented under the hood by delegating to various "dunder methods" for built-in classes:

```
>>> 10 + 20
30
>>> int.__add__(10, 20)  # Equivalent to 10 + 20
30
```

Haskell uses a similar approach. Every infix operator (e.g., `+`) is a function whose name is the same as the operator, but enclosed in parentheses (e.g., `(+)`):

```
> 10 + 20
30
> (+) 10 20
30
```

Again, pretty weird! If you've never thought about it before, this might seem overly complex. Racket actually has the cleanest model (no infix operators, uniform prefix syntax for all functions), at the cost of being more unfamiliar; Python

and Haskell both include a level of indirection to support the more familiar infix syntax we learn in mathematics.

Function purity

One thing you might notice about our grammar rules for defining functions is that a function body is just a single expression. This is again a consequence of our model of an expression-based language, rather than the imperative-style "sequence of statements" that you normally see in other languages.

The best way to think about functions in this style of programming is by analogy the mathematical functions, e.g. $f(x) = x + 1$, which are purely determined by input-output pairs. In this definition, we say that the body of f is the expression $x + 1$, and that we evaluate calls to f by substituting values for x into the body, evaluating the body expression, and then returning the result.

In the function definition rules we gave above, all the function values produced behave exactly the same as mathematical functions: their behaviour is entirely determined by what their body expression evaluates to for a set of given arguments. For example, consider the Racket function `(lambda (x) (+ x 1))`. If we call this function on the number 10,

```
((lambda (x) (+ x 1)) 10)
```

we evaluate this function call by taking the 10 and substituting it for `x` in the expression `(+ x 1)`, producing 11 as the returned value. Note that a `return` keyword isn't necessary: given the `lambda` expression, we know precisely that the value of `(+ x 1)` (with something substituted for `x`) will be returned.

In programming, we say that a **(mathematically) pure function** is a function that satisfies the following properties:

1. The function's behaviour is exactly determined by the *value* of its inputs. For example, the function cannot access data from "outside" its inputs, including standard input, the file system, the Internet, etc. This also rules out randomized functions; we say that pure functions must be *deterministic*.
2. The function only *returns* a value, and does nothing else. In parallel to (1), this means that pure functions cannot print to standard output, write to the file system, or send data across the Internet. We call such actions *side effects*, and so we say that "pure functions have no side effects."

This definition might seem overly restrictive: we often want functions to communicate with the "outside world", either for input or output! As is hopefully becoming a common theme in this course, we start with this type of function because such functions are easiest to reason about: if you understand substitution, then you're good to go. In the final chapter of these notes, we'll see how

to incorporate side effects like mutation and external I/O into a pure functional model.

Name bindings

While Alonzo Church and Alan Turing showed that anonymous functions and function application are sufficient to perform all computations that modern programming languages can, even pure functional programming languages like Haskell offer additional conveniences for the programmer. We've previously discussed some of these—primitive values and built-in functions—and the following example illustrates another:

```
(((lambda (x)
    ((lambda (f)
       (lambda (n)
         (if (equal? n 0) 1 (* n (f (- n 1))))))
     (lambda (v) ((x x) v))))
  (lambda (x) ((lambda (f)
                 (lambda (n)
                   (if (equal? n 0) 1 (* n (f (- n 1))))))
               (lambda (v) ((x x) v)))))
 10)
```

This monstrosity of a program evaluates to 3628800, which is 10!, illustrating the power of anonymous functions to successfully implement something of a recursive nature. However, we certainly don't want to get stuck writing code like this! Instead, every programming language gives us the ability to bind identifiers to values, so that *evaluating* an identifier results in the value that the identifier is bound to.

You have seen one kind of identifier already: the formal parameters used in function definitions. These are a special type of identifier, and are bound to values when the function is called. In this subsection we'll look at using identifiers more generally. We'll use identifiers for two purposes:

1. To "save" the value of subexpressions so that we can refer to them later.
2. To refer to a function name within the body of the function, enabling recursive definitions.

The former is clearly just a convenience to the programmer; the latter does pose a problem to us, but it turns out that writing recursive functions in the lambda calculus *is* possible, as we illustrated in the above example.[7]

It is important to keep in mind that all uses of identifiers beyond their use as parameter names are as a convenience, to make our programs easier to understand, but not to truly extend the power of the lambda calculus. Unlike the

[7] The main construct used to implement recursive functions is known as the *Y combinator*.

imperative programming languages you've used so far, identifier bindings in pure functional programming are *immutable*: once bound to a particular value, that identifier cannot be re-bound, and so literally is an alias for a value. This leads us to an extremely powerful concept known as referential transparency. We say that an identifier is **referentially transparent** if it can be substituted with its value *in the source code* without changing the meaning of the program.[8]

This approach to identifiers in functional programming is hugely different than what we are used to in imperative programming, in which re-binding names is not just allowed, but required for some common constructs,[9] or subverted by mutable data structures. Given that re-binding and mutation feels so natural to us, why would we want to give it up? Or put another way, why is referential transparency (which is violated by mutation) so valuable?

Mutation is a powerful tool, but also makes our code harder to reason about: we need to constantly keep track of the "current value" of every identifier throughout the execution of a program. Referential transparency means we can use names and values interchangeably when we reason about our code regardless of where these names appear in the program; a name, once defined, has the same meaning for the rest of the time.[10]

For about 95% of this course we will not use any mutation at all, and even when we do use it, it will be in a very limited way. Remember that the point is to get you thinking about programming in a different way, and we hope in fact that this ban will *simplify* your programming!

[8] This again parallels mathematics. When we write Llet $x = 5$" in a statement or proof, any subsequent statement we make about x should make just as much sense if we replace the x with 5.

[9] e.g., loops

[10] In particular, the whole issue of whether an identifier represents a *value* or a *reference* is rendered completely moot.

Global (aka top-level) name bindings

In Racket, the syntax for a global name binding uses the keyword `define`:

```
(define <id> <expr>)
```

In Haskell and Python, global bindings are written using the familiar = symbol:

```
<id> = <expr>
```

These definitions bind the value of `<expr>` to the identifier `<id>`. Here are some examples of these bindings, including a few that bind a name to a function.

```
# Racket
(define a 3)
(define add-three
  (lambda (x) (+ x 3)))

# Haskell
a = 3
```

```haskell
addThree = \x -> x + 3

# Python
a = 3
add_three = lambda x : x + 3
```

Because function definitions are so common, each language provides a concise way of binding function values to names; you are already familiar with `def` in Python. We're going to use this more convenient notation for the rest of the course, but keep in mind that in Racket and Haskell, these are merely "syntactic sugar" for the lambda expression.[11]

[11] *Syntactic sugar* is a part of a programming language's syntax that doesn't introduce new functionality, but is just another way of writing existing functionality in a simpler, more readable way.

```racket
; Racket
(define (add-three x) (+ x 3))
(define (almost-equal x y) (<= (abs (- x y)) 0.001))
```

```haskell
-- Haskell
addThree x = x + 3
almostEqual x y = abs (x - y) <= 0.001
```

Local bindings

Most programming languages support local scopes as well as global scope; among the most common of these is local scope within functions. For example, function parameters are local to the body of the function.

```racket
(define (f x)
  (+ x 10))  ; can refer to x here in the body of the function

(+ x 10)     ; can't refer to x out here (try it!)
```

In Racket, we can also explicitly create a local scope using the keyword `let*`:

```racket
(let* ([<id> <expr>] ...)
  <body>)
```

A `let*` expression takes pairs `[<id> <expr>]`, binds each expression to the corresponding identifier, and evaluates the `<body>` expression using these bindings. The *value* of the `let*` expression is the value of the `<body>`.

In Haskell, the same local scoping is achieved using the following:[12]

```
let <id> = <expr>
    ...
in
    <body>
```

[12] There's one subtle difference between Racket's `let*` and Haskell's `let`: the latter allows for *recursive* bindings, while the former does not. If you ever need to do this in Racket, you can use `letrec` instead.

What about Python? Python certainly has the concept of local scope, e.g., function and class definitions, but it does not have special syntactic support for let *expressions*, i.e., an expression that involves local bindings and that evaluates to produce a value. This is yet another reminder of the way in which Python is oriented around statements—assignment statements being the most common of these—while Racket and Haskell are oriented around expressions.

Name bindings and code structure

In the previous chapter, we said that we would focus on programs in *expression-based* contexts, in which a program is just a single expression to be evaluated. In practice, this is unwieldy, and this section described the use of identifiers to simplify complex expressions. Therefore the actual program form we'll stick with for this course is:

```
<prog> = <binding> ... <expr>
```

where `<binding>` is a *top-level* binding expression. Moreover, the `<expr>` can now use `let` expressions, which are structured in the same style: an arbitrary number of (distinct) name bindings, followed by a single expression to evaluate. For example, here is a Racket program that computes the distance between three distinct points and evaluates to the minimum distance:[13]

[13] Try writing the equivalent program without using any user-defined names! It's certainly possible, but quite terrible to do so.

```
(define p1 (list 2 3))
(define p2 (list 10 8))
(define p3 (list -1 6))

(define (distance p q)
  (let* ([dx (abs (- (first p) (first q)))]
         [dy (abs (- (second p) (second q)))])
    (sqrt (+ (* dx dx) (* dy dy)))))

(define distance-p1-p2 (distance p1 p2))
(define distance-p1-p3 (distance p1 p3))
(define distance-p2-p3 (distance p2 p3))

(min distance-p1-p2 distance-p1-p3 distance-p2-p3)
```

Written in Python, such a program is simply one where every global or local block of code consists of an arbitrary number of assignments to distinct variables, followed by a single expression to evaluate (or return, if inside a function). The important thing to keep in mind in that context is that such assignments are *non-mutating*: we never bind to the same variable more than once!

Lists and structural recursion

The list data type is one of the most fundamental in programming languages; it is an arbitrary-length compound data type, used to store any number of values.[14] For many of us, lists are the first time we are able to write code that operates on an arbitrary amount of data, without knowing ahead of time how much data there is.

[14] In fact, the name "Lisp" comes from "**l**ist **p**rocessing."

Imperative languages naturally process lists using loops, explicitly relying on notions of time and state to keep track of the "current" list element. This follows a metaphor of a list as a linear sequence of items, in which each item is "processed" one at a time. However, because pure functional programming does not permit the mutation required to re-bind an identifier to each element of the list in turn, we must take another approach to writing programs involving lists.

The key insight here is to identify a recursive structure for the list data type, and use this definition to inform both our representation of and operations on lists. A list is defined recursively as:

- The *empty list* is a list. (Represented in Racket as `'()`, in Haskell and Python as `[]`.)
- If `x` is a value and `lst` is a list, then we can create a new list whose first element is `x` and whose other items are the ones from `lst`. We call this combination the **cons** operation,[15] calling the produced list "`x` cons `lst`." In Racket, this is represented by the `cons` function, and in Haskell by the infix operator `(:)`.[16]

[15] short for "construct"

[16] Python doesn't have a built-in function that does this, perhaps evidence that this recursive representation is not central to Python lists (which it isn't).

For example, we could construct the list `[1, 2, 3]` using this recursive definition in Racket as `(cons 1 (cons 2 (cons 3 '())))`, and in Haskell as `1:2:3:[]`. In Racket and Haskell, we can *deconstruct* non-empty lists into their first and other elements, by using functions `first` and `rest` (Racket), or `head` and `tail` (Haskell). We have the identity `lst = (cons (first lst) (rest lst))` for all non-empty lists `lst`.

Structural recursion on lists

This recursive structure informs not just how we represent lists, but how we operate on lists as well. For example, consider the problem of computing the sum of the elements of a list. Whereas an iterative approach would process each element in turn by adding its value to an accumulator, a recursive approach mimics the recursive structure of the list itself:

- The sum of an empty list is equal to 0.
- The sum of "`x` cons `lst`" is equal to `x` plus the sum of `lst`.

The beauty of this English description is that it translates immediately into a pure function:

```racket
; Racket
(define (sum lst)
  (if (empty? lst)
      0
      (+ (first lst)
         (sum (rest lst)))))
```

```haskell
-- Haskell
sum lst =
  if null lst
  then
    0
  else
    head lst + sum (tail lst)
```

And here is a function that takes a list, and returns a new list containing just the multiples of three in that list. Note that this is a *filtering* operation, something we'll return to later in this chapter.

```racket
(define (multiples-of-3 lst)
  (cond [(empty? lst) '()]
        [(equal? 0 (remainder (first lst) 3))
         (cons (first lst)
               (multiples-of-3 (rest lst)))]
        [else (multiples-of-3 (rest lst))]))
```

This form of recursion is called *structural recursion* because the code follows the structure of the data type on which it operates. What's remarkable about this technique is that because it is based solely on the form of the input data, we can apply it to create a "code template" that works for many list functions:

```racket
(define (f lst)
  (if (empty? lst)
      ...
      (... (first lst)
           (f (rest lst)))))
```

Moreover, this technique can be used for *any* data type with a recursive definition, not just lists. The other fundamental example we'll return to again and again in this course is the recursive definition of trees, and specifically the *abstract syntax trees* that we use to represent programs.

Exercise Break!

The following programming exercises are meant to give you practice working with structural recursion on lists. We recommend completing them in both Racket and Haskell.

1.1 Write a function to determine the length of a list.

1.2 Write a function to determine if a given item appears in a list.

1.3 Write a function to determine the number of duplicates in a list.

1.4 Write a function to remove all duplicates from a list.

1.5 Given two lists, output the items that appear in both lists (intersection). Then, output the items that appear in at least one of the two lists (union).

1.6 Write a function that takes a list of lists, and returns the list that contains the largest item (e.g., given `(list (list 1 2 3) (list 45 10) (list) (list 15))`, return `(list 45 10)`.

1.7 Write a function that takes an item and a list of lists, and inserts the item at the front of every list.

1.8 Write a function that takes a list with no duplicates representing a set (order doesn't matter), and returns a list of lists containing all of its subsets.

1.9 Write a function that takes a list with no duplicates, and a number k, and returns all subsets of size k of that list.

1.10 (Racket only) Modify your function to the previous question so that the parameter k is optional, and if not specified, the function returns all subsets.

1.11 Write a function that takes a list, and returns all *permutations* of that list.

1.12 A *sublist* of a list is a series of *consecutive* items of the list. Given a list of numbers, find the maximum sum of any sublist of that list. (Note: there is a $O(n)$ algorithm that does this, although you should try to get any algorithm that is correct first, as the $O(n)$ algorithm is a little more complex.)

Pattern-matching

We'll now take a short detour away from the pure lambda calculus to introduce one of the most underutilized programming language features: *pattern-matching*, which allows the programmer to specify conditional behaviour depending on the structure of a given value in a concise and understandable syntax. As an example, consider the following value-based branching function definition:

```
f x =
    if x == 0
    then
        10
    else if x == 1
    then
        20
    else
        x + 30
```

While this is fine, we can shorten this substantially in Haskell by using *value pattern-matching* in the function definition.[17] Rather than giving one generic function signature `f x = ...`, we give three pattern-based definitions:

[17] Here, we'll focus on pattern-matching for function definitions only, although both Racket and Haskell support pattern-matching as an arbitrary expression form as well.

```
f 0 = 10
f 1 = 20
f x = x + 30
```

Essentially, Haskell's function definition syntax allows us to eliminate the explicit use of `if-else` tests; instead, we provide the patterns (in this case, `0`, `1`, or `x`—an identifier matches anything[18]) and the Haskell compiler generates the equivalent code that does the testing for us whenever this function is called.

[18] Order matters! The patterns are checked top-down, with the *first* match being selected.

Here is the same idea expressed in Racket. The syntax isn't quite as terse (note the use of the `define/match` keyword), but is certainly more concise than using `if`s or even `cond`:

```
(define/match (f x)
  [(0) 10]
  [(1) 20]
  [(_) (+ x 30)])  ; The underscore matches anything
```

While the above example is cute, you might be skeptical that it generalizes; after all, how often do we write explicit special cases when defining functions? But it turns out that pattern-matching goes far beyond simple value-equality checks. Consider now this recursive list function from the previous section, which we generalized into a template for structural recursion on lists.

```
(define (sum lst)
  (if (empty? lst)
      0
      (+ (first lst)
         (sum (rest lst)))))
```

Both Racket and Haskell support *structural pattern-matching*, a concise syntax for decomposing a value into subparts and binding each part to a new identifier that can be referred to independently. In Racket patterns, we use the expression `(list)` to pattern-match on an empty list, and we use `cons` to perform a structural decomposition pattern match of a list into its "first and rest".[19]

[19] This kind of decomposition is sometimes referred to as *destructuring* in other languages.

```
(define/match (sum lst)
  [((list)) 0]
  [((cons x xs)) (+ x (sum xs))])
```

In Haskell, the syntax is even terser, using the same expressions `[]` and `:` as when we create lists.[20]

[20] As we'll see later when we discuss algebraic data type, this is emphatically *not* a coincidence.

```
sum [] = 0
sum (x:xs) = x + sum xs
```

Exercise Break!

1.13 Redo your solutions to the previous set of list exercises using pattern-matching in both Racket and Haskell.

Tail call elimination

As you have studied in previous courses, function calls are stored on the *call stack*, a part of memory that stores information about the currently active functions as the program runs. In non-recursive programs, the size of the call stack is generally not an issue, but with recursion the call stack quickly fills up with recursive calls. Here is a quick Python example, in which the number of function calls is $\Theta(n)$:

```
def f(n):
  if n == 0:
    return 0
  else:
    return f(n - 1)

# This raises a RuntimeError because the call stack limit is reached.
f(10000)
```

In fact, the same issue of recursion taking up a large amount of memory occurs in Racket and Haskell as well. However, these languages (and many others)

perform **tail call elimination**, which can significantly reduce the space requirements for recursive functions. A **tail call** is a function call that happens as the last instruction of a function before the return; the `f(n-1)` call in the previous example has this property. When a tail call occurs, there is no need to remember where it was called from, because the only thing that's going to happen afterwards is that the value will be returned to the original caller.[21] This property of tail calls is common to all languages; however, some languages take advantage of this, and others do not. Racket is one that does: when it calls a function that it detects is in tail call position, it first removes the calling function's stack frame from the call stack, leading to constant stack height for this:

[21] Simply put: if `f` calls `g` and `g` just calls `h` and returns its value, then when `h` is called there is no need to keep any information about `g`; just return the value to `f` directly!

```
(define (f n)
  (if (equal? n 0)
      0
      (f (- n 1))))
```

Transforming simple recursive functions into tail-recursive ones

Now let's return our our `sum` implementation from the previous section:

```
(define (sum lst)
  (if (empty? lst)
      0
      (+ (first lst)
         (sum (rest lst)))))
```

This implementation is *not* tail-recursive! In the "else" expression, it is the `+`, not `sum`, that is called in tail position. The `sum` call is enclosed inside the outer function call `(+ (first lst) _)`: after the recursive `sum` call returns, its result is added to `(first lst)` before being returned.[22]

[22] The notation `(+ (first lst) _)` captures the idea of a *continuation* that we'll study in more detail in the next chapter.

In this case, there is a natural way to convert this `sum` implementation into one that is tail-recursive by essentially replacing the `(+ (first lst) _)` with a single recursive call. To do this, we use an *extra accumulating parameter* to store the "add `(first lst)`" part, updating its value at each recursive call. Here is our tail-recursive implementation:

```
(define (sum-tail lst)
  (sum-helper lst 0))

(define (sum-helper lst acc)
  (if (empty? lst)
      acc
      (sum-helper (rest lst) (+ acc (first lst)))))
```

In the above example, the parameter `acc` plays this role, accumulating the sum of the items "processed so far" in the list. We can use substitution to see exactly what happens when we call `(sum-tail '(1 2 3 4))`:

```
(sum-tail '(1 2 3 4))
(sum-helper '(1 2 3 4) 0)  ; Initial call to sum-helper; acc = 0
(sum-helper '(2 3 4) 1)    ; The new acc value is
                           ;   (+ 0 (first '(1 2 3 4))) = = 1
(sum-helper '(3 4) 3)      ; acc = (+ 1 (first '(2 3 4))) = 3
(sum-helper '(4) 6)        ; acc = (+ 3 (first '(3 4))) = 6
(sum-helper '() 10)        ; acc = (+ 6 (first '(4))) = 10
10                         ; Base case: acc is returned
```

As you might expect, this transformation technique generalized beyond this simple example! For example, here is the exact same idea applied to our earlier `multiples-of-3` function:[23]

[23] Question: Why did we use `append` and not `cons` here?

```
(define (multiples-of-3 lst)
  (multiples-of-3-helper lst '()))

(define (multiples-of-3-helper lst acc)
  (if (empty? lst)
      acc
      (multiples-of-3-helper (rest lst)
                             (if (equal? 0 (remainder (first lst) 3))
                                 (append acc (list (first lst)))
                                 acc))))
```

Exercise Break!

1.14 Rewrite your solutions to the previous exercises using tail recursion.

1.15 Consider a generalized version of the standard recursive template:

```
(define (f lst)
  (if (empty? lst)
      x
      (g (first lst) (f (rest lst)))))
```

(Here, `x` and `g` are arbitrary.) Transform `f` into an equivalent tail-recursive function.

From tail recursion to loops

Let's look once more at `sum-tail`:

```
(define (sum-tail lst)
  (sum-helper lst 0))

(define (sum-helper lst acc)
  (if (empty? lst)
      acc
      (sum-helper (rest lst) (+ acc (first lst)))))
```

Our previous trace through the evaluation of `(sum-tail '(1 2 3 4))` produced the function calls `(sum-helper '(1 2 3 4) 0)`, `(sum-helper '(2 3 4) 1)`, etc. In a language without tail call elimination, all of these `sum-helper` calls would occupy separate frames on the stack, but in a language (like Racket) that performs tail call elimination, we know that each of these function calls replaces the stack frame for the one before it. In particular, we can view the sequence

```
(sum-helper '(1 2 3 4) 0)
(sum-helper '(2 3 4) 1)
(sum-helper '(3 4) 3)
(sum-helper '(4) 6)
(sum-helper '() 10)
```

not as five separate function calls, but instead five separate executions of the function body, with the argument values `lst` and `acc` changing at each iteration. And of course, repeated executions of a block of code is naturally represented in an iterative manner using loops.

To see this transformation in action, we first write our existing code in Python:[24]

[24] Note that while Python does not perform tail-call elimination, that's an implementation detail—the algorithm remains unchanged.

```
def sum_tail(main_lst):
  return sum_helper(main_lst, 0)

def sum_helper(lst, acc):
  if lst == []:
    return acc
  else:
    # The lst[1:] implements (rest lst), but is unidiomatic Python.
    return sum_helper(lst[1:], acc + lst[0])
```

We now will transform `sum_helper` into a `while` loop as follows:

- Initialize two variables `lst` and `acc` representing the parameters of `sum_helper`,

using the argument values in the initial call `sum_helper(main_lst, 0)`.
- Place the body of `sum_helper` inside a `while True` loop.
- Replace the recursive `sum_helper` call with statements to update the values of `lst` and `acc`.

Applying these rules gives us the resulting code:

```
def sum_tail2(main_lst):
  lst, acc = main_lst, 0

  while True:
    if lst == []:
      return acc
    else:
      lst, acc = lst[1:], acc + lst[0]
```

Now, this code is certainly unidiomatic Python code, both because of the `while True` and because of the list slicing operation `lst[1:]`. The beauty of this approach is that we obtained this code by applying a mechanical transformation on our tail-recursive version—that is, without taking into account anything about the tail-recursive function did! Further analysis on our part would reveal that the `lst == []` and `lst = lst[1:]` pieces act as boilerplate to iterate through each element of the list `main_lst`, and so we can simplify this to the very idiomatic

```
def sum_tail3(main_lst):
  acc = 0
  for x in main_lst:
    acc = acc + x
  return acc
```

Exercise Break!

1.16 Consider a Python function of the following form:

```
def f(main_lst):
  lst, acc = main_lst, 0

  while True:
    if lst == []:
      return g1(acc)
    else:
      lst, acc = lst[1:], g2(lst[0], acc)
```

Rewrite this function into idiomatic Python, in the same way we did for `sum_tail3`. Make sure you understand exactly what mechanical steps you need to perform!

Higher-order functions

So far, we have kept a strict division between our types representing data values—numbers, booleans, strings, and lists—and the functions that operate on them. However, as we said earlier, functions are values, so it is natural to ask: can functions operate on other functions?

The answer in our course is a most emphatic *yes*, and in fact this is the heart of functional programming: the ability for functions to take in other functions and use them, combine them, and even return new ones. Let's see some simple programming examples.[25]

[25] The *differential operator*, which takes as input a function $f(x)$ and returns its derivative $f'(x)$, is another example of a "higher-order function" (although most calculus courses won't use this terminology). By the way, so is the indefinite integral.

```
; Take an input *function* and apply it to 1
(define (apply-to-1 f) (f 1))
(apply-to-1 even?)                  ; #f
(apply-to-1 list)                   ; '(1)
(apply-to-1 (lambda (x) (+ 15 x)))  ; 16

; Take two functions and apply them to the same argument
(define (apply-two f1 f2 x)
  (list (f1 x) (f2 x)))
(apply-two even? odd? 16)           ; '(#t #f)

; Apply the same function to an argument twice in a row
(define (apply-twice f x)
  (f (f x)))
(apply-twice sqr 3)                 ; 81
```

Higher-order list functions

With the discussion in the previous section, you might get the impression that people who use functional programming spend all of their time using recursion. But in fact this is not the case! Instead of using recursion explicitly, code often uses three critical higher-order functions to compute with lists.[26] The first two, `map` and `filter` are extremely straightforward:

[26] Of course, these higher-order functions themselves are implemented recursively.

```
; (map f lst)
; Returns a new list by applying `f` to each element in `lst`
> (map (lambda (x) (* x 3)) (list 1 2 3 4))
'(3 6 9 12)

```

```
; (filter pred lst)
; Creates a new list whose elements are those in 'lst'
; that make `pred` output #t.
> (filter (lambda (x) (> x 1)) (list 4 -1 0 15))
'(4 15)
```

To illustrate the third common higher-order function, let's first demonstrate how we might write `map` and `filter` using loops and mutation:

```
def map(f, lst):
    acc = []
    for x in lst:
        acc.append(f(x))
    return acc

def filter(pred, lst):
    acc = []
    for x in lst:
        if pred(x):
            acc.append(x)
    return acc
```

Both of these functions use the same accumulator pattern, using an *accumulator variable* to store a value that gets updated at each loop iteration, and is finally returned at the function's end. We can generalize this pattern into a higher-order function accepting two additional arguments: an initial value and a function to update the accumulator inside the loop.

```
def accumulate(combine, init, lst):
    acc = init
    for x in lst:
        acc = combine(acc, x)
    return acc
```

This more general loop pattern is codified in a recursive fashion in both Racket and Haskell in a function called `foldl`:[27]

```
(define (foldl combine init lst)
  (if (empty? lst)
      init
      (foldl combine
             (combine (first lst) init)
             (rest lst))))
```

[27] Note that the order of the arguments of the "combine" functions are different in the two languages.

```haskell
foldl combine init lst =
  if null lst
  then
    init
  else
    foldl combine (combine init (head lst)) (tail lst)
```

Though all three of `map`, `filter`, and `foldl` are extremely useful in performing most computations on lists, both `map` and `filter` are constrained in having to return lists, while `foldl` can return any data type. In fact, given the flexibility of this function illustrated in its corresponding loop version, it should be clear at least conceptually that it is possible to implement `map` and `filter` in terms of `foldl`—doing so is a great exercise at this point in your learning.

Exercise Break!

1.17 You might notice that the Racket implementation of `foldl` is very similar to the implementation of `sum-helper` from the previous section. Use `foldl` to implement `sum` in both Racket and Haskell. Note that you should be able to pass in the "plus" function directly (i.e., not using a lambda), but this may require a bit of research when doing so in Haskell.

1.18 Implement a function that takes a predicate (boolean function) and a list, and returns the number of items in the list that satisfy the predicate.

1.19 Is `foldl` tail-recursive? If so, explain why. If not, rewrite it to be tail-recursive.

1.20 Reimplement all of the previous exercises using `map`, `filter`, and/or `foldl`, *without* using explicit recursion.

1.21 Write a function that takes a list of unary functions, and a value `arg`, and returns a list of the results of applying each function to `arg`.

1.22 Implement `map` and `filter` using `foldl`.

1.23 The "l" in `foldl` stands for "left", because items in the list are combined with the accumulator in order from left to right.

```racket
(foldl f 0 (list 1 2 3 4))
; equivalent to...
(f 4 (f 3 (f 2 (f 1 0))))
```

Write another version of fold called `foldr`, which combines the items in the list from right to left:

```racket
(foldr f 0 '(1 2 3 4))
; equivalent to...
(f 1 (f 2 (f 3 (f 4 0))))
```

Hint: this can be done using basic structural recursion—start by mentally dividing the input list into first and rest.

Currying

Currying is a powerful feature of functional programming languages that allows a function to be applied to only *some* of its arguments. We'll talk more about currying when we discuss Haskell's static type system, but for now our interest is in how currying simplifies functions whose bodies call higher-order functions.

Suppose we want to write a function `big` that takes a list `lst` and returns a list containing only the elements from `lst` that are larger than 5. We can accomplish this using the `filter` higher-order function:

```
1 big lst = filter (\x -> 5 < x) lst
```

But notice that the anonymous function there is simply calling the `<` function with its first argument set to 5. That is, it's the `<` function "partially applied" to one of its two arguments. Currying allows us to dispense with the anonymous function, writing simply:

```
1 big lst = filter ((<) 5) lst
```

This kind of flexibility would be very strange indeed for imperative languages like C and Java. But it is this flexibility that gives functional programming its power: it lets us adapt the arity of a function to contexts for which the function may not have been explicitly designed!

The higher-order function `apply`

Next, we will look at one more fundamental higher-order function. As a warm-up, consider the following mysteriously-named function:

```
1 (define ($ f x) (f x))
```

This function takes two arguments, a function and a value, and then applies the function to that value. This is fine for when `f` is unary, but what happens when it's not? For example, what if we wanted to give `$` a binary function and two more arguments, and apply the function to those two arguments? Of course, we could write another function for this purpose, but then what about a function that takes three arguments, or one that takes ten? What we would like, of course, is a higher-order function that takes a function, then any number of additional

arguments, and applies that function to those extra arguments. In Racket, we have a built-in function called `apply` that does almost this:

```
; (apply f lst)
; Call f with arguments taken from the elements of lst
(apply + (list 1 2 3 4))
; equivalent to...
(+ 1 2 3 4)

; More generally,
(apply f (list x1 x2 x3 ... xn))
; equivalent to...
(f x1 x2 x3 ... xn)
```

Note that `apply` differs from `map`, even though the types of their arguments are very similar (both take a function and a list). Remember that `map` calls its function argument once for *each* value in the list separately, while `apply` calls its function argument just once, on all of the items in the list at once. The return value of `map` is always a list; the return value of `apply` is whatever its function argument returns.

In Haskell, the story is similar but not identical. We have a *binary infix operator* `($)`, which acts as unary function application:

```
f $ x
-- equivalent to...
f x
```

However, due to constraints imposed by Haskell's type system, it does *not* provide an equivalent to Racket's `apply`, which works on functions of arbitrary arity. We'll discuss Haskell's type system in detail later in this course.

Exercise Break!

1.24 Look up *rest arguments* in Racket, which allow you to define functions that take in an arbitrary number of arguments. Then, implement a function `($$ f x1 ... xn)`, which is equivalent to `(f x1 ... xn)`.

Functions returning functions

We have now seen functions that take primitive values and other functions, but so far they have all had primitive values as their output. Now, we'll turn our attention to another type of higher-order function: a function that *returns* a function. This is extremely powerful: in its most general form, it allows us to define

new functions dynamically, that is, during the execution of a program. Here is a simple example of this:

```
(define (make-adder x)
  ; The body of make-adder is a function value.
  (lambda (y) (+ x y)))
(make-adder 10) ; #<procedure>

(define add-10 (make-adder 10))
add-10     ; #<procedure>
(add-10 3) ; 13
```

The function `add-10` certainly seems to be adding ten to its argument; using the substitution model of evaluation for `(make-adder 10)`, we see how this happens:

```
(make-adder 10)
; ==> (substitute 10 for x in the body of make-adder)
(lambda (y) (+ 10 y))
```

Exercise Break!

1.25 Write a function that takes a single argument `x`, and returns a new function that takes a list and checks whether `x` is in that list or not.

1.26 Write a function that takes a unary function and a positive integer n, and returns a new unary function that applies the function to its argument n times.

1.27 Write a function `flip` that takes a binary function `f`, and returns a new binary function `g` such that `(g x y) = (f y x)` for all valid arguments `x` and `y`.

1.28 Write a function that takes two unary functions `f` and `g`, and returns a new unary function that always returns the max of `f` and `g` applied to its argument.

1.29 Write the following Racket function:

```
#|
(fix f n x)
  f: a function taking m arguments
  n: a natural number, 1 <= n <= m
  x: an argument

  Return a new function g that takes m-1 arguments,
  which acts as follows:
  (g a_1 ... a_{n-1} a_{n+1} ... a_m)
  = (f a_1 ... a_{n-1} x a_{n+1} ... a_m)

```

```
  That is, x is inserted as the nth argument in a call to f.

> (define f (lambda (x y z) (+ x (* y (+ z 1)))))
> (define g (fix f 2 100))
> (g 2 4) ; equivalent to (f 2 100 4)
502
|#
```

(Hint: recall *rest arguments* from an earlier exercise.)

1.30 Write a function `curry`, which does the following:

```
#|
(curry f)
  f: a binary function

  Return a new higher-order unary function g that takes an
  argument x, and returns a new unary function h that takes
  an argument y, and returns (f x y).

> (define f (lambda (x y) (- x y)))
> (define g (curry f))
> ((g 10) 14) ; equivalent to (f 10 14)
-4
|#
```

1.31 Generalize your previous function to work on a function with `m` arguments, where `m` is given as a parameter.

Programming with abstract syntax trees

A natural generalization of the list data type is the *tree* data type, in which the "recursive" part contains an arbitrary number of recursive subcomponents rather than just one. In this section, we'll start looking at how abstract syntax trees (ASTs) are represented in real code, and how our discussion of structural recursion allows us to easily operate on such trees.

Racket: Quoted expressions

First, let's see how we can represent ASTs in Racket. This is actually one of the fundamental strengths of all Lisp languages: the parenthesization of the source code immediately creates a *nested list* structure, which is simply another representation of a tree. To make this even more explicit in the language, any Racket expression (no matter how complex or deeply nested) can be turned into a static nested list simply by prefixing it with an apostrophe. We call this **quoting** an expression.

```
> (+ 1 2)  ; A regular Racket expression
3
> '(+ 1 2) ; Quoting the expression: a list of three elements.
'(+ 1 2)
> (first '(+ 1 2))
'+
> (second '(+ 1 2))
1
> (third '(+ 1 2))
2
> '(+ (* 2 3) (* 4 5))  ; This is a nested list
'(+ (* 2 3) (* 4 5))
```

Even though in Racket a quoted expression really is just a list, we will call it a Racket **datum**[28] to distinguish it from a regular list. Formally, here are the types of values that comprise the "tree" in a Racket datum:

[28] plural: "datums", not "data" to avoid confusion

1. Quoted literals (`5`, `#t`, `"hello"`) represent those same values as in the original code: for example, `(equal? '5 5)` returns true.
2. Quoted identifiers and keywords become *symbols*; a symbol is a primitive datatype that functions as a unique identifier. For example, the datum `'(+ 1 2)` has as its first element the symbol `'+`, which can be compared using `equal?` to other values, and used in pattern-matching. Note that keywords like `define` *also* get quoted into symbols, and so the only way to determine which symbols correspond to identifiers and which correspond to keywords is to start with a fixed set of keywords to check.
3. Quoted compound expressions become lists, in which each element is the quoted version of the corresponding subexpression. `'(+ 2 3)` is equivalent

to `(list '+ '2 '3)`, which is in turn equivalent to `(list '+ 2 3)`.

Given this structure, we can use structural pattern-matching to take in a datum and perform recursive computations on it. For example, here is a function that takes a Racket datum consisting of just numeric literals, identifiers, and arithmetic function calls, and returns the number of times + is used:

```
(define/match (num-plus datum)
  ; For a function call (which in this case is *any* parenthesized
  ; expression), recurse on each subexpression and add up the total
  ; number of occurrences. Note the use of (list expr ...) to bind
  ; lists of arbitrary length to expr.
  [((list expr ...))
    (apply + (map num-plus expr))]
  ; Value pattern-match on '+
  [('+) 1]
  ; For any other atomic value (identifier or literal), return 0
  [(_) 0])

> (num-plus '+)
1
> (num-plus 3)
0
> (num-plus '(+ (* 4 5) (+ (- (+ 3 6) 7) 8)))
3
```

Haskell: value constructors

As a consequence of Haskell's type system, we cannot store values of different types in the same list, making it cumbersome to directly represent Racket datums in the same way (note that those Racket lists freely mix nested lists, symbols, and literals of all different types). Instead, Haskell's type system permits the concise declaration of, and pattern-matching on, a type with multiple different constructors. We will study this more formally later in the course, but for now we think you'll find the following representation of an "expression" type to be quite intuitive:

```
data Expr = NumLiteral Int      -- An integer literal
          | Identifier String   -- An identifier
          | Call Expr [Expr]    -- A function call, where the first
                                -- Expr is the function being called,
                                -- and the list [Expr] contains the
                                -- argument expressions.
```

Again, the syntax is not too important here; the main point is that `NumLiteral`, `Identifier`, and `Call` are three *value constructors* that take in, respectively, an integer, a string, and another expression and list of expressions, and return a value of type `Expr`. Note that `NumLiteral` and `Identifier` are atomic expressions (they are leaves in the AST), while `Call` is recursive. As an example, we would represent the Racket expression `(+ 2 3)` by the value:

```
Call (Identifier "+") [NumLiteral 2, NumLiteral 3]
```

This might seem cumbersome at first, but because of structural pattern-matching, it is just as easy to implement the `num-plus` function in Haskell.[29]

```
numPlus (Call f args) = numPlus f + sum (map numPlus args)
numPlus (Identifier "+") = 1
numPlus (Identifier _) = 0
numPlus (NumLiteral _) = 0
```

[29] We explicitly differentiate between the `Identifier` and `NumLiteral` case for illustrative purposes only. As long as we kept the first two lines the same, we could have used a single `_` in the third pattern, in the same way we did for Racket.

Undefined programs and evaluation order

So far in this chapter, we have focused on the *denotational semantics* of pure functional programs: *what* they mean, or what they evaluate to.[30] We have seen how the tools of recursion and higher-order functions can be used to express interesting computations in this setting, but even then have focused on the simple rule of function application as *substitution*.

[30] Remember that we think of programs as consisting of a sequence of definitions followed by a single expression to evaluate.

While the denotational semantics of a language determine what expressions must evaluate to, the operational semantics determine *how* expressions are evaluated. For simple expressions like `5`, `(+ 20 30)`, and `((lambda (x) x) 1)`, this is not very interesting: literals are evaluated simply by using their literal value, while the other two expressions reach a numeric value of 50 and 1, respectively, in just a single simplification step each. However, operational semantics must cover not just these simple cases, but more complex expressions as well. Consider the expression `((lambda (x) (+ x 6)) (+ 4 4))`. We have a choice about how to evaluate this function call. We could evaluate the `(+ 4 4)` first, and then call the `lambda` expression on it:

```
((lambda (x) (+ x 6)) (+ 4 4))
; evaluate (+ 4 4)
((lambda (x) (+ x 6)) 8)
; substitute 8 for x
(+ 8 6)
14
```

Or, we could substitute the entire `(+ 4 4)` expression into the body of the lambda first, and then evaluate the result:

```
((lambda (x) (+ x 6)) (+ 4 4))
; substitute (+ 4 4) for x
(+ (+ 4 4) 6)
; evaluate (+ 4 4)
(+ 8 6)
; evaluate (+ 8 6)
14
```

Perhaps unsurprisingly, it turns out that these two are equivalent. In fact, the **Church-Rosser Theorem** says (informally) that for any program of this form in the lambda calculus, *every* possible order of function application must result in the same final value. Informally, every road leads to the same destination.

However, even this informal version of the theorem only applies to valid function applications, and not ones that result in an error or non-terminating computation. What happens in those cases?

It is the answer to this question that leads to our discussion of the different *evaluation orders* used to deal with function applications, which has significant implications for a programming language.[31]

[31] While we'll stick to the pure functional world here, evaluation order is even more important in imperative-style languages that support functions with side effects.

Consider, for example, the function call `(f 10 (/ 1 0))`. Is this always guaranteed to raise an error for any binary function `f`?

Strict evaluation semantics

Your intuition for the previous question is probably a resounding *yes*, as this is the behaviour used in almost every modern programming language. First, we say that an expression has an **undefined value** if it represents a computation that does not complete, e.g. due to its corresponding abstract mathematical meaning (e.g., division by 0), or because it represents a non-terminating computation.[32] For a given type of expression, we say that it has **strict evaluation denotational semantics** if and only if whenever an expression of that type contains a subexpression whose value is undefined, the value of the expression itself is also undefined.

[32] This "undefined" value is commonly referred to as *bottom* in the programming languages literature.

Most modern programming languages use strict denotational semantics for function calls. In this context, the function call has two kinds of subexpressions: the function being called, and the arguments to the function; if any of these contains an error, the function call itself generates this error. This is most commonly implemented by (i.e., induces an operational semantics of) a **left-to-right eager evaluation** of function call expressions:

- When evaluating a function call, first evaluate the subexpression representing the function being called (often, but not always, an identifier).

- Then evaluate each argument subexpression, in left-to-right order.
- Finally, "call" the function by substituting the *value* of each subexpression into the body of the function.

Because arguments are evaluated before being substituted into the body of a function, this guarantees that if one of the argument subexpressions (or even the function subexpression itself) is undefined, this is discovered *before* the function is actually called.

Similarly, most languages use strict semantics for *name bindings*; for example, in Racket the top-level binding `(define <id> <expr>)` first evaluates `<expr>`, and then binds the *value* of that expression to the `<id>`.[33]

[33] In introductory classes, we normally learn this as "first evaluate the right-hand side of the =, then assign the value to the left-hand side."

Non-strict syntactic forms

In your programming experience so far, you have probably taken for granted eager evaluation order in function calls. However, you also learned that there are certain types of expressions that do *not* eagerly evaluate all of their subexpressions, for example, boolean operators and conditionals. In Racket, these are `and`, `or`, `if`, and `cond`; even though these may look like ordinary functions, they aren't! Suppose we wrote our own "and" function by wrapping the built-in Racket `and`: `(define (my-and x y) (and x y))`.

Even though `my-and` looks identical to `and`, it's not, all because of evaluation order. Whereas the built-in `and` can stop evaluating its arguments when it reaches a "false" value, the same is not true of `my-and`—the latter is a *function*, and so eagerly evaluates all of its arguments before passing their values to `and`:

```
(and #f (/ 1 0))      ; evaluates to #f
(my-and #f (/ 1 0))   ; raises an error
```

Because of this, we say that `and`, `or`, `if`, and `cond` are *syntactic forms*, rather than identifiers that refer to built-in functions. This point is actually rather subtle, and in fact is not specific to Racket at all! In *any* programming language that uses eager evaluation for functions, short-circuiting boolean operations and conditional are all implemented as syntactic constructs, not simple function calls.

Function bodies and delaying evaluation

Now that we've given a name to eager evaluation, you might look for other places where languages may or may not eagerly evaluate expressions. Consider the following Racket function:

```
(define (f x)
  (length (range 3000000)))
```

Note that the body of this function doesn't use the parameter `x`; when the Racket interpreter *evaluates* this function, does it eagerly evaluate the body of `f`? How about in other languages that you know?

It turns out that Racket doesn't: in general, a function's body is not evaluated until the function is called, even if the body doesn't depend on the function's parameters.[34] This means that we can delay the evaluation of an expression simply by putting it inside a function body. We give the name **thunk** to any nullary function whose purpose is to delay the evaluation of its body expression. For example, `(lambda () (/ 1 0))` is a thunk, wrapping around the error-raising division expression. When we evaluate the `lambda`, this *doesn't* evaluate its body:

[34] This isn't entirely true for modern *compilers*, which may evaluate constant expressions during compilation to speed up program execution time. This optimization, known as *constant folding*, is outside the scope of this course.

```
> (lambda () (/ 1 0))      ; A thunk as an anonymous function
#<procedure>
> (define (bad) (/ 1 0))   ; A thunk bound to a name
> bad                      ; No error when the name is evaluated
#<procedure:bad>
> (bad)                    ; *Calling* the thunk raises an error
/: division by zero
```

Using this idea, we can simulate non-strict semantics for function calls by passing in thunks that wrap the "actual" arguments:

```
; x and y are now thunks
(define (my-and x y)
  (and (x) (y)))

> (my-and (lambda () #f) (lambda () (/ 1 0)))
#f  ; The (/ 1 0) is never evaluated!
```

Exercise Break!

1.32 Given the following nested function call expression, write the order in which the functions are evaluated:

```
(f (a b (c d) (d)) e (e) (f (f g)))
```

1.33 Draw an *abstract syntax tree* associated with the previous expression, where each internal node represents a function call, whose children are the arguments to that function.

1.34 Racket has `lambda` and `define` expressions. Neither of these are functions, but are instead also syntactic forms (like `and`); how do you know?

Non-strict semantics in Haskell

As we mentioned at the start of this chapter, one of the initial design features that set Haskell apart from other languages is that it uses *non-strict semantics* for both function calls and name bindings. In a function call, the argument subexpressions are *not* evaluated right away. Instead, they are only evaluated if and when they are used, and so if an argument is never used, it is never evaluated. This evaluation strategy is known as *lazy* evaluation. Consider the following example:

```
> onlyFirst x y = x
> onlyFirst 15 (head [])
15
> onlyFirst (head []) 15
*** Exception: Prelude.head: empty list
```

The function `onlyFirst` requires only its first argument to evaluate its body, so in the first call, the argument `(head [])` is never evaluated. This means that we can define a *short-circuiting* "and" function just like any other Haskell function:

```
> myAnd x y = if x then y else False
> myAnd False (head [])
False
```

In fact, the `(&&)` and `(||)` boolean operators really are just built-in *functions* for Haskell, unlike in most other programming languages!

Things get even more surprising when it comes to name bindings. In Haskell, the binding `<id> = <expr>` does *not* evaluate `<expr>`, but instead binds the expression to the identifier. It's only when that identifier is required to be evaluated does `<expr>` get evaluated:

```
> x = error "This is an error"    -- No error when x is bound.
> x                               -- Error when x is evaluated.
*** Exception: This is an error
```

Indeed, you may have wondered what the syntax is for defining nullary functions in Haskell, given that the general function definition syntax (`<id> <param> ... = <body>`) seems to coincide with a "simple" name binding (`<id> = <expr>`) when there are no parameters. Now we are able to answer this question: *every* name binding in Haskell defines a function! The binding `<id> = <expr>` defines a *thunk*, analogous to `(define (<id>) <expr>)` in Racket, making it clear that the expression is not evaluated at the time of definition.

The trouble with Haskell's `foldl`

To wrap up this section, we'll revisit one particularly famous wart of Haskell's implementation. Even though we have not discussed efficiency (either time or space) too much in this course, there is one place where this has already come up: the notion of *tail call optimization*, which is central to mitigating call stack growth when using recursion. To start, let's look at how our old friend `foldl` is implemented in Haskell (this mimics the definition in Racket):

```
foldl _ acc [] = acc
foldl f acc (x:xs) =
  let acc' = f acc x
  in
      foldl f acc' xs
```

We use a `let` here to split off an intermediate definition of `acc'` from the recursive call, to make it more obvious that the recursive call to `foldl` is in tail call position. And indeed, Haskell does perform tail call optimization, as you would expect from a pure functional language. However, when we try running a large computation like `foldl max 0 [1..100000000]`, not only does the computation take a long time, but its memory usage increases linearly over time. What gives?

The key to understanding this is in understanding Haskell's lazy evaluation order. Even though laziness might seem like a neat feature, it does have its drawbacks. The evaluation of functions becomes more unpredictable, making it harder to correctly reason about both the time and space efficiency of a program. In the case of our `foldl` implementation, laziness means that the `let` bindings don't actually get evaluated until absolutely necessary—and when is that? Certainly not when we make the recursive call. The expression

```
let acc' = f acc x
in
    foldl f acc' xs
```

reduces to `foldl f (f acc x) xs`. That is, the second argument passed to the recursive call is a *thunk* whose body is `(f acc x)`, not the value produced by actually calling `f`. Let's make that a bit more concrete. Consider calling `foldl max 0 [1, 2, 3]`. In the initial call, the body of `foldl` becomes

```
let acc' = max 0 1
in
    foldl max acc' [2, 3]
```

In a language with strict binding semantics, we would expect `max 0 1` to evaluate to `1` before being bound to `acc'`, leading to the recursive call `foldl max 1`

`[2, 3]`. This is the case in, say, Racket, and combined with tail call optimization this leads to space usage that is constant with respect to the size of the input list, since the space used to store the accumulator is fixed (a single integer). In the case of Haskell, however, `acc'` is bound to a thunk containing `(max 0 1)`. So then even when the recursive call undergoes tail call optimization, the resulting argument data takes a bit more space than before: a thunk instead of the initial integer `0`. This space usage *accumulates*[35] through the recursive calls, with each new binding of `acc'` resulting in a new thunk being stored, leading to the linearly increasing space usage we observe when we call this function. Indeed, these thunks are only called when we reach the end of the list, and `acc` is returned and printed (assuming we call this directly in the Haskell REPL).

[35] pun definitely intended

Because laziness presents its own unique problems, the Haskell language provides ways of explicitly *forcing* the evaluation of an expression, even when that expression's value is not "necessary" for evaluating its containing outer expression. One approach is with the compiler extension `BangPatterns`, whose use is illustrated here:

```haskell
{-# LANGUAGE BangPatterns #-}

foldl _ !acc [] = acc
foldl f !acc (x:xs) =
  let acc' = f acc x
  in
      foldl f acc' xs
```

The only difference, of course, is the first line (telling the compiler to use the `BangPatterns` extension), and the exclamation mark preceding the `acc` parameter in the function definition. This has the effect of making `acc` a *strict parameter*: whenever `foldl` is called, the argument expression corresponding to `acc` must be evaluated before being passed to `foldl`, and of course this includes recursive calls. With this implementation of `foldl` we get the constant space evaluation of `foldl max 0 [1..100000000]`, taking advantage of both tail call optimization and eager evaluation of the calls to `f` to ensure that only an integer accumulator is passed through the recursive calls.

A final note: why did we call `foldl` one of Haskell's "famous warts"? It turns out that even though this is a well-known issue in the Haskell community, for historical reasons the more naturally-named function, `foldl`, behaves in the lazy way and so is generally discouraged. Instead, users wanting to use the more space-efficient version can import `foldl'` from `Data.List`, paying the price of both an extra import and a clumsier name. Go figure.

Lexical closures

To end this chapter, we'll study in more detail one of the most interesting implementation details of a functional programming language: supporting the dy-

namic creation of functions in a space-efficient manner. Recall the `make-adder` function, which takes a number `x` and returns a new "add `x`" function:

```
(define (make-adder x)
  (lambda (y) (+ x y)))
```

Suppose we call `make-adder` multiple times: `(make-adder 10)`, `(make-adder -1)`, `(make-adder 9000)`, etc. A naive (but correct!) implementation of Racket would create and store in memory brand-new functions for each call:

```
; (make-adder 10)
(lambda (y) (+ x 10))
; (make-adder -1)
(lambda (y) (+ x -1))
; (make-adder 9000)
(lambda (y) (+ x 9000))
```

But it seems rather wasteful for Racket to create and store in memory brand-new functions each time, when really all of the function values have essentially the same body, differing only in their value of `x`.

A better implementation factors out the literal `lambda` expression `(lambda (y) (+ x y))`, which is common to all calls to `make-adder`, and instead stores simply the *binding for* `x`, which can vary for each function call. And in fact, this is exactly what Racket does! An abstract, but more accurate, representation of the previous example calls is:[36]

```
; (make-adder 10)
(x: 10) '(lambda (y) (+ x y))
; (make-adder -1)
(x: -1) '(lambda (y) (+ x y))
; (make-adder 9000)
(x: 9000) '(lambda (y) (+ x y))
```

[36] Keep in mind that the `(lambda (y) (+ x y))` is stored *once* in memory, and `make-adder` returns just a reference or pointer to that `lambda`.

The data structure storing the reference to the function code, and the binding for `x`, is called a **closure**. All along, when we've referred to `lambda`s as "function values", each one was actually implemented as a closure! Why has this never come up before? Closures are conceptually necessary only when the function body has a **free identifier**, which is an identifier that is not local to the function. This should make intuitive sense: suppose we have a function where every identifier in its body is either a parameter, or bound in a local `let` expression. Then every time the function is called, all of the data necessary to evaluate that function call is contained in the arguments and the function body—no additional lookup necessary. In the definition of `make-adder`, the inner `lambda` expression has a free identifier `x`, and this is the name whose value will need to be looked up in the

closure. When this closure is itself called, say in the expression `((make-adder 10) 3)`, we take the function `(lambda (y) (+ x y))`, and look up the value of `x` in the closure to retrieve the 10.

In summary, a closure is a data structure containing a pointer to a function, as well as a collection of name-value bindings for all free identifiers for that function. If the body doesn't have any free identifiers, then the closure can be identified with the function body itself, since its name-value binding is empty. Note that in order for the closure to be evaluated, all of the identifiers inside the function body must be in scope *when the function is defined*, just not necessarily local to the function. This was true for the previous example: while `x` was a free identifier for the inner `lambda`, it was still in scope because it was a parameter of the enclosing `make-adder`. In contrast, the following variation would raise a runtime error because of an unbound identifier:

```
(define (make-adder x)
  (lambda (y) (+ z y)))
```

Lexical and dynamic scope

We have just finished saying that closures—and in particular, name bindings—are created when a `lambda` expression is evaluated. But this is actually not the full story; there is yet another consideration to take into account when reasoning about how closures work. Consider the variation of `make-adder`:

```
(define z 10)
(define (make-z-adder) (lambda (x) (+ x z)))
(define add-z (make-z-adder))

> (add-z 5)
15
```

So far, so good. The body of `make-z-adder` is a function with a free identifier `z`. Calling `make-z-adder` evaluates the `lambda`, returning a closure in which the free `z` refers to the global one, and so binds to the value 10. But what happens when we shadow this `z`, and then call `make-z-adder`?

```
(let* ([z 100])
  (let* ([add-z-2 (make-z-adder)])
    (add-z-2 5)))
```

Now the `lambda` expression is evaluated when we call the function in the local scope, but what value of `z` gets bound in the closure? Put more concretely, does this expression output 15 (because `z` is bound to 10) or `105` (because `z` is bound

to 100)? It turns out that this expression evaluates to 15, just like the previous example. Our goal is to understand why.

The question of how to resolve free identifiers when creating closures is very important, even though we usually take it for granted. In Racket, the value used is the one bound to the name that is in scope *where the `lambda` expression appears in the source code*. That is, though closures are created at runtime, how the closures are created (i.e., which bindings are used) is based only on where they are written in the source code. In the previous example, the `z` in the closure returned by `make-z-adder` is bound to the value of the global `z`, which is the one that is in scope where the `lambda` expression is written.

More generally, the binding for *any* identifier in Racket can be determined by taking its location in the source code, and proceeding outwards through enclosing expressions until you find where this identifier is first introduced (either through a binding or a parameter declaration). We say that identifiers in Racket obey **lexical scope** or **static scope**, referring to the fact that this resolution can be done by analysing the source code itself, prior to actually running any code.

In contrast to this is **dynamic scope**, in which identifiers are resolved based on when the identifier is used during program execution, rather than where it appears in the code. If Racket used dynamic scope, the above example would output 105, because the closure created by `make-z-adder` would now bind the enclosing local `z`.

Here is an example of how dynamic scoping works in the `bash` shell scripting language:

```
(let* ([z 100])
  (let* ([add-z-2 (make-z-adder)])
    (add-z-2 5)))
```

In other words, lexical scoping resolves names based on context from the source code, whereas dynamic scoping resolves names based on context from the program state at runtime. Early programming languages used dynamic scope because it was easier to implement, as names could be resolved by looking for the nearest binding on the function call stack, or by recursively accumulating name bindings in an interpreter. However, dynamic scope makes programs very difficult to reason about, as the values of non-parameter names of a function depended on the *myriad* places the function is used, rather than the *single* place where it is defined. Lexical scope was a revolution in how it simplified programming tasks, and is used by every modern programming language today.[37] And it is ideas like that which motivate research in programming languages!

[37] ALGOL was the first language to use lexical scope, in 1958.

A Python puzzle

Closures are often used in the web programming language JavaScript to dynamically create and bind *callbacks*, which are functions used to respond to events

like a user clicking a button or entering some text. A common beginner mistake when creating these functions exposes a very subtle misconception about closures when they are combined with mutation. We'll take a look at an analogous example in Python.

```
def make_functions():
  flist = []
  for i in [1, 2, 3]:
    def print_i():
      print(i)
    print_i()
    flist.append(print_i)
  print('End of flist')
  return flist

def main():
  flist = make_functions()
  for f in flist:
    f()

>>> main()
1
2
3
End of flist
3
3
3
```

The fact that Python also uses lexical scope means that the closure of each of the three `print_i` functions refers to the same `i` variable (in the enclosing scope). That is, the closures here store a *reference*, and not a *value*.[38] After the loop exits, `i` has value 3, and so each of the functions prints the value 3. Note that the closures of the functions store this reference even after `make_functions` exits, and the local variable `i` goes out of scope!

[38] Since we have been avoiding mutation up to this point, there hasn't yet been a distinction between the two.

By the way, if you wanted to fix this behaviour, one way would be to not use the `i` variable directly in the created functions, but instead pass its *value* to another higher-order function. In the code below, each `print_i` function has a closure looking up `x`, which is bound to different values and not changed as the loop iterates.

```
def create_printer(x):
  def print_x():
    print(x)
  return print_x

def make_functions():
```

```
  flist = []
  for i in [1, 2, 3]:
    print_i = create_printer(i)
    print_i()
    flist.append(print_i)
  print('End of loop')

def main():
  flist = make_functions()
  for f in flist:
    f()
```

Secret sharing

Here's one more cute example of using closures to allow "secret" communication between two functions in Python.[39]

[39] The `nonlocal` keyword is used to *prevent* name shadowing, which would happen if a local `secret` variable were created.

```
def make_secret():
  secret = ''

  def alice(s):
    nonlocal secret
    secret = s

  def bob():
    nonlocal secret
    print(secret)
    # Reset secret
    secret = ''

  return alice, bob

>>> alice, bob = make_secret()
>>> alice('Hi bob!')
>>> bob()
Hi bob!
>>> secret
Error ...
```

Exercise Break!

1.35 In the following `lambda` expressions, what are the free identifiers (if any)? (Remember that this is important for understanding what a closure actually stores.)

```
(lambda (x y) (+ x (* y z)))  ; (a)

(lambda (x y) (+ x (w y z)))  ; (b)

(lambda (x y)                 ; (c)
  (let ([z x]
        [y z])
    (+ x y z)))

(lambda (x y)                 ; (d)
  (let* ([z x]
         [y z])
    (+ x y z)))

(let ([z 10])                 ; (e)
  (lambda (x y) (+ x y z)))

(define a 10)                 ; (f)
(lambda (x y) (+ x y a))
```

1.36 Write a snippet of Racket code that contains a function call expression that will evaluate to different values depending on whether Racket uses lexical scope or dynamic scope.

Summary

In this chapter, we looked at the basics of functional programming in two new languages, Racket and Haskell. Discarding mutation and the notion of a "sequence of statements," we framed computation as the evaluation of functions using higher-order functions to build more and more complex programs. However, we did not escape notion of time entirely; in our study of evaluation order, we learned precisely how Racket (and most other languages) eagerly evaluate function call arguments, and how special syntactic forms distinguish themselves from functions precisely because of how their arguments are evaluated. We contrasted this with Haskell, which lazily evaluates its arguments only if and when they are needed.

Our discussion of higher-order functions culminated in our discussion of closures, allowing us to create functions that return new functions, and so achieve an even higher level of abstraction in our program design. Along the way, we discovered the important difference between lexical and dynamic scope, an illustration of one of the big wins that static analysis yields to the programmer. Finally, we saw how closures could be used to share internal state between functions without exposing that data. In fact, this encapsulation of data to internal use in functions should sound familiar from your previous programming experience, and will be explored in the next chapter.

2 *Macros, Objects, and Backtracking*

> In most programming languages, syntax is complex. Macros have to take apart program syntax, analyze it, and reassemble it... A Lisp macro is not handed a string, but a preparsed piece of source code in the form of a list, because the source of a Lisp program is not a string; it is a list. And Lisp programs are really good at taking apart lists and putting them back together. They do this reliably, every day.
>
> Mark Jason Dominus

Now that we have some experience with functional programming, we will briefly study two other programming language paradigms. The first, *object-oriented programming*, is likely very familiar to you, while the second, *logic programming*, is likely not. Because of our limited time in this course, we will not treat either topic with as much detail each deserves, as neither is a true focus of this course. Instead, we will stay in Racket, and *implement* support for these paradigms into the Racket programming language itself, and so achieve two goals at once: first, you will gain deeper understanding of these paradigms by studying the design decisions and trade-offs made in implementing them; and second, you will learn how to use a powerful *macro system* to fundamentally extend the syntax and semantics of a programming language itself.

Object-oriented programming: a re-introduction

> OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.
>
> Alan Kay

Because we have often highlighted the stark differences between imperative and functional programming, you may be surprised to learn that our study of functions has given us all the tools we need to implement a basic object-oriented system like the one you're familiar with from Python (and other imperative languages).

Recall the definition of a *closure*: a data structure containing a reference to function code together with the lexical environment storing the values of the free variables in the function body. Now consider what we mean by an object in a traditional OOP setting: an **object** is a data structure that stores values called the **(instance) attributes** of the object, that is also associated with functions called **(instance) methods** that operate of this data.

This paradigm was developed as a way to organize data that promotes encapsulation, separating private concerns of how this data is organized from the public interface that determine how other code may interact with the data. Unlike the pure functions we have studied so far, a method always takes a special argument,[1] an associated object that we say is calling the method. Though internal attributes of an object are generally not accessible from outside the object, they *are* accessible from within the body of methods that the object calls.

[1] This "calling object" is often an implicit argument with a fixed keyword name like `this` or `self`.

Historically, the centrality of the object itself to call methods and access (public) attributes led to the natural metaphor of entities sending and responding to messages for modeling computation. Putting the public/private distinction front and centre, we can view an object not as "data + methods", but rather as an entity that can receive messages from external code, and then respond to that message (say by changing its internal state or returning a value). And of course, "receive a message and return a value" is just another way of describing what a function does, as we illustrate in the following example:[2]

[2] We'll use the convention in these notes of treating messages to objects as symbols naming the attribute or method to access.

```
(define (point msg)
  (cond [(equal? msg 'x) 10]
        [(equal? msg 'y) -5]
        [else "Unrecognized message"]))

> (point 'x)
10
> (point 'y)
-5
```

Classes as higher-order functions

Of course, this "point object" is not very compelling: it only has attributes but no methods, making it more like a C struct, and its attribute values are hard-coded.

One solution to the latter problem is to create a point **class**, a template that specifies both the attributes and methods for a type of object. In class-based object-oriented programming, every object is an *instance* of a class, obtaining their attributes and methods from the class definition.[3] Objects are created by calling a class **constructor**, a function whose purpose is to return a new instance of that class, often initializing all of the new instance's attributes.

To translate this into our language of functions, a *constructor* for `Point` is a function that takes two numbers, and returns a function analogous to the one above, except with the `10` and `5` replaced by the constructor's arguments.[4]

```
(define (Point x y)
  (lambda (msg)
    (cond [(equal? msg 'x) x]
          [(equal? msg 'y) y]
          [else "Unrecognized message"])))

> (define p (Point 2 -100))
> (p 'x)
2
> (p 'y)
-100
```

And now we see explicitly the relationship between closures and objects in this model: in the returned function, `x` and `y` are free identifiers, and so must have values bound in a closure when the `Point` constructor returns. Moreover, because the *identifiers* `x` and `y` are local to the `Point` constructor, even though their values are stored in the closure, once the constructor has returned they can't be accessed without passing a message to the object.[5] This is the property of closures that enables encapsulation: even though both `x` and `y` values are accessible by passing the right messages to the object, it is certainly possible to implement "private" attributes in this model.

One other point: *lexical closures* is absolutely required to maintain proper encapsulation of the attributes. Imagine what would happen if the following code were executed in a dynamically-scoped language, and what implications this would have when we create multiple instances of the same class.

```
(define p (Point 2 3))
(let ([x 10])
  (p 'x))
```

[3] Even though class-based OOP is the most common approach, it is not the only one. JavaScript uses **prototypal inheritance** to enable behaviour reuse; objects are not instances of classes, but instead inherit attributes and methods directly from other objects.

[4] We follow the convention of giving the constructor the same (capitalized) name as the class itself.

[5] This is analogous to the `secret` variable in the example at the end of the last chapter.

Adding methods to our class

Next, let's add two simple methods to our `Point` class. Because Racket functions are first-class values, we can treat attributes and methods in the same way: a method is just an attribute that happens to be a function![6] Of course, one of the characteristics that distinguishes methods from arbitrary functions is that in the body of a method, we expect to be able to access all instance attributes of the calling object. It turns out that this is not an issue; study the example below, and see if you can determine why not!

[6] This is a different view than the one taken by, say, Java, in which methods are completely separate from data attributes.

```
(define (Point x y)
  (lambda (msg)
    (cond [(equal? msg 'x) x]
          [(equal? msg 'y) y]
          [(equal? msg 'to-string)
           (lambda ()
             (format "(~a, ~a)" x y))]
          [(equal? msg 'distance)
           ; Return the distance between this and other-point.
           (lambda (other-point)
             (let ([dx (- x (other-point 'x))]
                   [dy (- y (other-point 'y))])
               (sqrt (+ (* dx dx) (* dy dy)))))]
          [else "Unrecognized message"])))

> (define p (Point 3 4))
> (p 'to-string)
#<procedure>
> ((p 'to-string))
"(3, 4)"
> ; Note that (p 'distance) returns a function, so the expression
> ; below is a *nested function call*.
> ((p 'distance) (Point 0 0))
5
```

The problem of boilerplate code

Cool! We have seen just the tip of the iceberg of implementing class-based objects with pure functions. As intellectually stimulating as this is, however, the current technique is not very practical. Imagine creating a series of new classes—and all of the boilerplate code[7] you would have to write each time.

[7] e.g., the message handling with `cond` and `equal?`, `"Unrecognized message"`, etc.

What we'll study next is a way to *augment the very syntax of Racket* to achieve the exact same behaviour in a much more concise, natural way:

```
(class Person
  ; Expression listing all attributes
  (name age likes-chocolate)

  ; Method
  [(greet other-person)
   (string-append "Hello, "
                  (other-person 'name)
                  "! My name is "
                  name
                  ".")]

  ; Another method
  [(can-vote?) (>= age 18)])
```

Exercise Break!

2.1 First, carefully review the final implementation of the `Point` class we gave above. This first question is meant to reinforce your understanding about function syntax in Racket. Predict the value of each of the following expressions (many of them are erroneous—make sure you understand why).

```
Point
(Point 3 4)
(Point 3)
(Point 3 4 'x)
((Point 3 4))
((Point 3 4) 'x)
((Point 3 4) 'distance)
((Point 3 4) 'distance (Point 3 10))
(((Point 3 4) 'distance) (Point 3 10))
```

2.2 Take a look at the `Person` example given above. Even though it is currently invalid Racket code, its intended semantics should be quite clear. Write a `Person` class in the same style as the `Point` class above. This will ensure that you understand our approach for creating classes, so that you are well prepared for the next section.

Pattern-based macros

As we have previously touched on, Racket's extremely simple syntax—programs are nested lists—not only makes code easy to parse, but also easy to manipulate.[8] One neat consequence of this is that Racket has a very powerful macro

[8] This is one of the most distinctive features that all Lisp dialects share.

system, with which developers can quite easily extend the language by adding new keywords, and even embedding entire domain-specific languages.

We are used to thinking about functions as operating on values: a function is an entity that takes values as inputs and returns a new value. As the building blocks of programs, functions are versatile and powerful, and so it is easy to forget that they have limitations: their syntax and semantics are defined by the programming language itself, and not the programmer who writes them. For example, regardless of what function we write in Racket, we know that calling it will invoke a left-to-right eager evaluation of its arguments, because this is how the Racket interpreter has been implemented to handle function calls.

On the other hand, a **macro** is a function that operates on program *source code*: a macro takes code as input and returns new *code*. Many of the features of Integrated Development Environments (IDEs) that we take for granted are macros: code formatting, identifier renaming, automatic refactoring, getter and setter generation, etc. But IDEs are themselves software that operate on text files containing source code, so it is perhaps not too surprising that they implement these features. What is far more interesting, and the topic of this section, is when programming languages themselves allow the programmer to define their own macros, just as they can define their own functions. In such languages, there is an additional step between the parsing of source code and generation of machine code or runtime evaluation, called **macro expansion**, in which macros (both built-in and user-defined) are applied to the abstract syntax tree to generate a new abstract syntax tree. That is, the AST that is actually used to generate machine code or to be interpreted is not necessarily the one originally produced by the parser, and might actually look very different, depending on the macros applied to it!

This may seem kind of abstract: why might we want to use macros? The main use of macros we'll see is to *introduce new syntax* into a programming language. In this section, we'll build up a nifty syntactic construct: the list comprehension. First, let's see how list comprehensions work in Haskell:[9]

[9] You may have seen list comprehensions in Python—this was borrowed liberally from Haskell's syntax.

```
>>> [x + 2 | x <- [0, 10, -2]]
[2,12,0]
```

The list comprehension consists of three important parts: an output expression, an identifier, and an input list. Expressed as a grammar rule, we have:

```
<list-comp> = "[" <out-expr> "|"  <id> "<-" <list-expr> "]"
```

Unfortunately, Racket does not have this built-in list comprehension syntax; fortunately, its macro system will enable to to implement it ourselves! Here is an example of the kind of Racket expression that is our goal:[10]

[10] For a technical reason, we have to use : instead of | in Racket.

```
> (list-comp (+ x 2) : x <- (list 0 10 -2))
'(2 12 0)
```

Let's first think about how we might implement the high-level functionality in Racket, ignoring the syntactic requirements. Recalling your work in the previous chapter, you might notice that a list comprehension is essentially a `map`:

```
> (map (lambda (x) (+ x 2)) (list 0 10 -2))
'(2 12 0)
```

Now, we do some pattern-matching to generalize:

```
; Putting our examples side by side...
(list-comp (+ x 2) : x <- (list 0 10 -2))
(map (lambda (x) (+ x 2)) '(0 10 -2))

; leads to the following generalization.
(list-comp <out-expr> : <id> <- <list-expr>)
(map (lambda (<id>) <out-expr>) <list-expr>)
```

This step is actually the most important one, because it tells us (the programmers) what syntactic transformation the interpreter will need to perform: every time it sees a list comprehension, it should transform it into a call to `map`. It remains to actually tell the Racket interpreter to do this transformation; we do this by writing a **pattern-based macro**:

```
(define-syntax list-comp
  (syntax-rules (: <-)
    [(list-comp <out-expr> : <id> <- <list-expr>)
     (map (lambda (<id>) <out-expr>) <list-expr>)]))
```

Let's break that down. The top-level `define-syntax` is a syntactic form that defines a name binding for a new macro (analogous to `define` for values). Its first argument is the name of the macro—in our case, `list-comp`— and the second argument is the macro expression. In this course, we'll stick to pattern-based macros, which are created using `syntax-rules`. `syntax-rules` itself takes two or more arguments: a list of the *literal keywords* in the syntax—in our case, `:` and `<-`—followed by one or more *syntax pattern rules*, which are pairs of expressions, similar to `define/match`. In each rule, the first expression is a *pattern* to match (starting with `list-comp`), while the second expression is called a *template* that specifies how the new syntax should be generated from parts of the pattern. We only have one pattern rule right now, but that will change shortly.

With this macro defined, we can now evaluate our Racket `list-comp` expression:

```
> (list-comp (+ x 2) : x <- (list 0 10 -2))
'(2 12 0)
```

While this might look just like a plain function call, it isn't! There are two phases involved to produce the result `'(2 12 0)`:

1. First, the `list-comp` expression is transformed into a `map` function call expression, by applying the syntax pattern rule we defined. This is a *code transformation*; you could achieve the same effect by literally typing in `(map (lambda (x) (+ x 2)) (list 0 10 -2))` instead.
2. Second, the `map` expression is evaluated, using plain old function call semantics.

One way to see the difference between `list-comp` and a regular function is to use the syntax in incorrect ways:

```
> (list-comp 1 2 3)
list-comp: bad syntax in: (list-comp 1 2 3)
> (list-comp (+ x 2) : x <- 10)
map: contract violation
  expected: list?
  given: 10
  argument position: 2nd
  other arguments...:
  #<procedure>
```

The first error is a *syntax* error: Racket is saying that it doesn't have a syntax pattern rule that matches the given expression. The second error really demonstrates that a syntax transformation occurs: `(list-comp (+ x 2) : x <- 10)}` might be syntactically valid, but it expands into `(map (lambda (x) (+ x 2)) 10)`, which raises the runtime error you see above.

The purpose of literal keywords

Our syntax pattern rule makes use of both pattern variables and literal keywords. A *pattern variable* is an identifier that can be bound to an arbitrary expression when the pattern matches; in this course, we'll follow a convention of naming these with enclosing angle brackets, but this is not required by Racket. During macro expansion, when Racket finds a pattern-match, it binds the pattern variable to the corresponding expression, and then substitutes this expression where ever the variable appears in the rule's template. This is similar to function calls, in which argument values are bound to parameter names

and substituted into the body of a function, but there is a crucial difference: in macro expansion, *expressions aren't evaluated* before they are bound! This should make intuitive sense as long as you remember that macros are fundamentally about transforming code, not about evaluation; when we say that an expression is bound to a pattern variable, we literally mean that *entire expression* is bound to the variable, *not* the value of the expression.

On the other hand, *literal keywords* are parts of the syntax rule that must appear literally in the syntax expression. These play the same role as keywords in other programming languages (e.g., `else`, `:` in Python), namely to give structure to the program syntax.[11] If we try to use `list-comp` without the two keywords, we get a syntax error—the Racket interpreter does not recognize the expression, because it no longer pattern-matches our rule:

```
> (list-comp (+ x 2) 3 x "hi" (list 0 10 -2))
list-comp: bad syntax ...
```

[11] This is less important in Racket than other languages, because Racket's parenthesized nature provides enough structure already. However, literal keywords can still promote readability: contrast `(list-comp (+ x 2) : x <- (list 0 10 -2))` and `(list-comp (+ x 2) x (list 0 10 -2))`.

Macros with multiple pattern rules

In Haskell, list comprehensions support optional filtering of the input list:

```
>>> [x + 2 | x <- [0, 10, -2], x >= 0]
[2, 12]
```

To achieve this form of list comprehension in Racket, we simply add an extra syntax rule to our macro definition, much like we can add an extra pattern rule to a `define/match`![12]

[12] You'll notice that we cannot use `,` in our macro, replacing it instead with the Python-style `if`.

```
(define-syntax list-comp
  (syntax-rules (: <-)
    [(list-comp <out-expr> : <id> ... <- <list-expr>)
     (map (lambda (<id> ...) <out-expr>) <list-expr>)]))


(define-syntax list-comp-if
  (syntax-rules (: <- if)
    ; This is the old pattern rule.
    [(list-comp-if <out-expr> : <id> <- <list-expr>)
     (map (lambda (<id>) <out-expr>) <list-expr>)]

    ; This is the new pattern rule.
    [(list-comp-if <out-expr> : <id> <- <list-expr> if <condition>)
     (map (lambda (<id>) <out-expr>)
          (filter (lambda (<id>) <condition>)
                  <list-expr>))]))
```

With these two rules in place, we now can use both our original and extended form of `list-comp`:

```
> (list-comp-if (+ x 2) : x <- (list 0 10 -2))
'(2 12 0)
> (list-comp-if (+ x 2) : x <- (list 0 10 -2) if (>= x 0))
'(2 12)
```

As with functions defined using pattern-matching in both Racket and Haskell, macro pattern rules are checked top-to-bottom, with the first match being the one that is chosen. In this example the rules are mutually exclusive, but in general it's certainly possible to define two syntax pattern rules that overlap, so be careful about this when writing your own macros!

Text-based vs. AST-based macros

If you've heard of macros before learning a Lisp-family language, it was probably from C or C++.[13] The C macro system operates on the source text itself, rather than a parsed AST. In the macro example above we noted that we couldn't use `|` or `,` as literal keywords; this is because they are special characters that affect the parsing of Racket code into an AST, and so have an effect even before the macro expansion phase. This limitation aside, it is far easier and safer to define AST-based macros. In this section, we look at two examples of how text-based C macros can lead to unexpected pitfalls.

[13] The typesetting language LaTeX also uses macros extensively.

First, here is a simple example of a C macro that takes one argument and multiplies its argument by 2:

```
#define double(x) 2 * x
```

When we use this macro on an integer literal or an identifier, things seem to work fine:

```
#include <stdio.h>

#define double(x) 2 * x

int main(void) {
    int a = 10;
    printf("%d\n", double(100)); // Correctly prints 200.
    printf("%d\n", double(a));   // Correctly prints 20.
    return 0;
}
```

In each of the above invocations, the *source text* involving `double` is replaced: `double(100)` by `2 * 100` and `double(a)` by `2 * a`. But consider this usage of the macro instead:

```
printf("%d\n", double(5 + 5));
```

Our intuition tells us that again `20` would be printed, but this is not what happens! To see what goes wrong, we need to understand how the substitution works in C macros. Like Racket, macros in C do not evaluate their arguments before substitution; instead, the entire subexpression `5 + 5` is substituted for `x` into the body of the macro. The problem arises from the fact that this is a text-based substitution; since the body of the macro is (in text) `2 * x`, the resulting macro expansion yields `2 * 5 + 5`, and so `15` is printed!

In other words, because C macros use text-based substitution, their arguments are not necessarily preserved as individual subexpressions when put into the macro body—it depends on how the compiler then parses the resulting text. In the above example, the precedence rules governing the parsing of arithmetic expressions caused the `5 + 5` to be split up.

To prevent this, a common rule of thumb when writing C macros is to always parenthesize macro variables in the macro body:[14]

```
#define double(x) 2 * (x)
```

[14] For a similar reason, another rule of thumb is to surround the entire macro body in parentheses when it is a single expression.

Because Racket macros operate on abstract syntax trees, this problem is avoided entirely. When the substitution occurs, the exact expression structure of the Racket program has already been determined, and macro expansion simply swaps subexpressions matching a pattern rule with the corresponding template. The only new subexpressions that appear in the resulting AST are the ones that were explicitly written in the macro definition, and the macro's arguments.

Now consider the following macro, which uses a temporary variable to swap two integers. It seems to work just fine in this small program.

```
#include <stdio.h>

#define swap(a, b) int temp = a; a = b; b = temp;

int main(void) {
    int x = 0, y = 10;
    swap(x, y)
    printf("x is now %d, y is now %d\n", x, y);
    return 0;
}
```

But we run into a problem when trying to compile the following code:

```c
#include <stdio.h>

#define swap(a, b) int temp = a; a = b; b = temp;

int main(void) {
    int x = 0, y = 10;
    swap(x, y)
    swap(x, y)
    printf("x is now %d, y is now %d\n", x, y);
    return 0;
}
```

We get the message `error: redefinition of 'temp'`; each use of the macro declares a local variable `temp`, which is a compilation error. Now, C allows for local scopes to be introduced through curly braces, and so we can solve this problem by enclosing the macro body in curly braces:

```c
#define swap(a, b) {int temp = a; a = b; b = temp;}
```

However, another problem emerges: what if one of the variables we wanted to swap was named `temp`?

```c
#include <stdio.h>

#define swap(a, b) {int temp = a; a = b; b = temp;}

int main(void) {
    int x = 0, temp = 10;
    swap(x, temp)
    printf("x is now %d, temp is now %d\n", x, temp);
    return 0;
}
```

This program compiles successfully, but when run it prints:

```
x is now 0, temp is now 10
```

The variables weren't swapped! To see why, we again can look at the literal text substitution of `swap(x, temp)`:

```
{int temp = x; x = temp; temp = temp;}
```

In this case, all references to `temp` in the macro are for the block-local `temp`, not the `temp` that was initialized to 10 at the top of `main`. So the local `temp` is initialized to the current value of `x` (`0`), then `x` is assigned that value (`0`), and then `temp` is assigned its current value. After the block executes, the values of `x` and `temp` haven't changed at all.

A third approach is to require declaration of `temp` before using this macro. This does work correctly, at the cost of an additional requirement placed on users of this macro.

```
#include <stdio.h>

#define swap(a, b) {temp = a; a = b; b = temp;}

int main(void) {
    int x = 0, y = 10, temp;
    swap(x, y);
    printf("x is now %d, y is now %d\n", x, y);
    return 0;
}
```

These three examples of `swap` illustrate the ways in which the body of a C macro can interact with the scope in which it is used: first, introducing names into the outer scope; second, by inadvertently shadowing identifiers that are passed as arguments to the macro; and third, by directly accessing (and mutating) values from that outer scope. The ability for a C macro to refer to identifiers defined in an outer scope is known as *name capture*, and is a common source of error when writing C macros. The behaviour of these macros depends not just on how they're defined, but on where they are used. If this sounds familiar, it should! *Identifiers in C macros obey dynamic scope*.

That C macros behave in this ways is a natural consequence of how identifiers are just substituted as text during macro expansion. Even though we know that Racket macros are based on AST transformation, it is still plausible for individual identifiers to be substituted as-is during macro expansion. Let's try out a simple example and see what happens.

```
(define-syntax add-temp
  (syntax-rules ()
    [(add-temp <x>)
     (+ <x> temp)]))
```

It may look like `temp` is undefined in the body of `add-temp`, much like `temp` was undeclared in the macro body of our third example. The question is, what happens with:

```
(let* ([temp 10])
  (add-temp 100))
```

Using a literal textual substitution of identifiers, after macro expansion this expression would become

```
(let* ([temp 10])
  (+ 100 temp))
```

This is certainly a valid Racket expression, and would evaluate to `110`. However, this is not what happens!

```
> (let* ([temp 10])
    (add-temp 100))
ERROR temp: undefined;
 cannot reference an identifier before its definition
```

That is, the `temp` in the `add-temp` macro template obeys *lexical scope*, and cannot be bound dynamically where the macro is used.

This is true for all Racket macros: free identifiers in the macro obey lexical scope, meaning they are bound to whatever identifier is in scope where the macro is originally defined, and cannot accidentally bind names when it is used. Because this was seen as a massive improvement over the previous dynamically-scoped macros, early developers of Lisp termed this form of macro *hygienic macros*.[15]

As a consequence of macro hygiene, identifiers defined within a macro body are *not* accessible outside of the macro body—that is, they're local to the macro—even when a straight textual substitution would suggest otherwise. For example:

```
(define-syntax defs
  (syntax-rules ()
    [(defs)
     (begin  ; We use begin to enclose multiple expressions
       (define x 1)
       (define y (+ x 10))
       ; x and y are both in scope here.
       (+ x y))]))

(defs)  ; This evaluates to 12
x       ; This is an error! x and y are not in scope.
```

[15] This behaviour suggests something interesting about macro expansion in Racket. It isn't just that expansion occurs at the level of ASTs; even individual identifiers are just not substituted literally during expansion. While the exact implementation of hygienic macros are beyond the scope of this course, a simplified mental model you can use here is that all local identifiers in a macro body are renamed during expansion to guarantee that there are no collisions with the scope in which the macro is used.

Contrast this behaviour against simply using the `begin` instead of our `defs` macro:

```
; This still evaluates to 12
(begin
  (define x 1)
  (define y (+ x 10))
  (+ x y))

; But now x is defined as well! The following evaluates to 1.
x
```

Macro ellipses

It is often the case that we want a macro that can be applied to an arbitrary number of expressions.[16] This poses a challenge when writing macro pattern rules, as we can't write an explicit pattern variable for each expression we expect to match. Instead, we can use the ellipsis `...` in a pattern: `<pat> ...` matches zero or more instances of the pattern `<pat>`, which could be a pattern variable or a complex pattern itself.[17]

[16] e.g., `and`, `or`, `cond`

[17] This is analogous to the $*$ operator in regular expressions, which can match an arbitrary number of any character (`.*`) or a complex pattern `(a[0-9]*)*`.

Here is one example of using the ellipsis in a recursive macro that implements `cond` in terms of `if`. Recall that branching of "else if" expressions can be rewritten in terms of nested if expressions:

```
(cond [c1 x1]
      [c2 x2]
      [c3 x3]
      [else y])

; as one cond inside an if...
(if c1
    x1
    (cond [c2 x2]
          [c3 x3]
          [else y]))

; eventually expanding to...
(if c1
    x1
    (if c2
        x2
        (if c3
            x3
            y)))
```

Here is our macro that achieves this behaviour:

```
(define-syntax my-cond
  (syntax-rules (else)  ; Note that `else` is a literal keyword here
    [(my-cond) (void)]
    [(my-cond [else <val>]) <val>]
    [(my-cond [<test> <val>] <next-pair> ...)
     (if <test> <val> (my-cond <next-pair> ...))]))
```

This example actually illustrates two important concepts with Racket's pattern-based macros. The first is how this macro defines not just a syntax pattern, but a *nested* syntax pattern. The first pattern rule will match the expression `(my-cond [else 5])`, but *not* `(my-cond else 5)`—parentheses matter.[18] The second is the `<next-pair> ...` part of the second pattern, which binds *all arguments after the first one*. For example, here's a use of the macro, and one step of macro expansion, which should give you a sense of how this recursive macro works:

[18] Although note that this rule will also match `(my-cond (else 5))`. Racket treats `()` and `[]` as synonymous.

```
(my-cond [c1 x1]
         [c2 x2]
         [c3 x3]
         [else y]))

; <test> is bound to c1, <val> is bound to x1,
; and <next-pair> ... is bound to ALL of
; [c2 x2] [c3 x3] [else y]
(if c1
    x1
    (my-cond [c2 x2] [c3 x3] [else y]))
```

Warning: don't think of the ellipsis as a separate entity, but instead as a modifier for `<next-pair>`. We say that the ellipsis is "bound" to `<next-pair>`, and must appear together with `<next-pair>` in the rule's template.[19] A beginner mistake is to try to treat the ellipsis as an identifier in its own right, bound to the "rest" of the arguments:

[19] The binding goes both ways: `<next-pair>` can't appear without the ellipsis, either.

```
(define-syntax my-cond-bad
  (syntax-rules (else)
    [(my-cond-bad [else <val>]) <val>]
    [(my-cond-bad [<test> <val>] ...)
     ; This would make sense if ... was a variable representing the
     ; remaining arguments, but it isn't.
     ; Instead, the rule below is a syntax error; the ... cannot be
     ; used independently of the pattern it's bound to.
     (if <test> <val> (my-cond-bad ...))]))
```

To summarize, a pattern variable that is modified with an ellipsis is no longer bound to an individual expression, but instead to the entire sequence of expressions. We'll see how to make powerful use of this sequence in the next section.

Exercise Break!

2.3 Explain how a macro is different from a function. Explain how it is *similar* to a function.

2.4 Below, we define a macro, and then use it in a few expressions. Write the resulting expressions after the macros are expanded. Note: do not evaluate any of the resulting expressions! This question is a good check to make sure you understand the difference between the macro expansion and evaluation phases of the Racket interpreter.

```
(define-syntax my-mac
  (syntax-rules ()
    [(my-mac x) (list x x)]))

(my-mac 3)
(my-mac (+ 4 2))
(my-mac (my-mac 1000))
```

2.5 Write macros to implement `and` and `or` in terms of `if`. Note that both syntactic forms take an arbitrary number of arguments.

2.6 Why could we not accomplish the task in the previous question with functions?

2.7 Consult the official Haskell documentation on list comprehensions. One additional feature we did not cover is list comprehensions on multiple variables.

Modify our existing `list-comp` macro to handle this case. Hint: first convert the above expression into a list comprehension within a list comprehension, and use `apply append`.

2.8 Add support for `if`s in list comprehensions with multiple variables. Are there any syntactic issues you need to think through?

Objects revisited

Now that we have seen how to define Racket macros, let's return to our basic implementation of objects and classes. To start, here is an abbreviated `Point` class with only the two attributes and no methods.

```
(define (Point x y)
  (lambda (msg)
    (cond [(equal? msg 'x) x]
          [(equal? msg 'y) y]
          [else "Unrecognized message!"])))
```

The goal now is to abstract away the details of defining a constructor function and the message handling to enable easy declaration of new classes. Because we are interested in defining new identifiers for class constructors, our previous tool for abstraction, functions, is insufficient for this task.[20] Instead, we turn to macros to introduce a new syntactic form into the language, `my-class`:

[20] Why?

```
; This expression should expand into the definition above.
(my-class Point
  (x y))
```

Performing the same pattern-matching procedure as we did for `list-comp`, we can see how we might write a skeleton of the macro (accepting not just two but an arbitrary number of attribute names).

```
(define-syntax my-class
  (syntax-rules ()
    [(my-class <class-name> (<attr> ...))
     (define (<class-name> <attr> ...)
       (lambda (msg)
         (cond ???
               [else "Unrecognized message!"])))]))
```

Unfortunately, this macro is incomplete; without the `???`, it would generate the following code:

```
(my-class Point (x y))
; => (macro expansion)
(define (Point x y)
  (lambda (msg)
    (cond [else "Unrecognized message!"])))
```

What's missing, of course, are the other expressions in the `cond` that match messages corresponding to the attribute names. Let's consider the first attribute `x`. For this attribute, we want to generate the code

```
[(equal? msg 'x) x]
```

Now, to do this we need some way of converting an identifier into a symbol, which we can do using the syntactic form `quote`:

```
[(equal? msg (quote x) x)]
```

For `(my-class Point (x y))`, we actually want this expression to appear for both `x` and `y`:

```
[(equal? msg (quote x) x)]
[(equal? msg (quote y) y)]
```

So the question is how to generate one of these expressions for *each* of the attributes bound to `<attr> ...`; that is, repeat the above pattern an arbitrary number of times, depending on the number of attributes. It turns out that macro ellipses support precisely this behaviour:[21]

[21] I hope you're suitably impressed. I know I am.

```
(define-syntax my-class
  (syntax-rules ()
    [(my-class <class-name> (<attr> ...))
     (define (<class-name> <attr> ...)
       (lambda (msg)
         (cond [(equal? msg (quote <attr>)) <attr>]
               ; Repeat the previous expression once per expression
               ; in <attr> ..., replacing just occurrences of <attr>
               ...
               [else "Unrecognized message!"])))]))
```

Adding methods

Now, let us augment our macro to allow method definitions. Even though we previously made a stink about methods just being any other type of attribute, there is one important difference: it isn't enough to provide just the name of the method in the class definition; we need to provide the method body as well. We will use the following syntax, mimicking Racket's own macro pattern-matching syntax:

```
(my-class <class-name> (<attr> ...)
  (method (<method-name> <param> ...) <body>) ...)
```

Once again, the easiest way to write the pattern-based macro is to create an instance of both the syntactic form we want to write, and the expression we want to expand it into. To use our ongoing point example:

```
; What we want to write:
(my-class Point (x y)
  (method (distance other-point)
   (let* ([dx (- x (other-point 'x))]
          [dy (- y (other-point 'y))])
     (sqrt (+ (* dx dx) (* dy dy))))))

; What we want it to expand into:
(define (Point x y)
  (lambda (msg)
    (cond [(equal? msg 'x) x]
          [(equal? msg 'y) y]
          [(equal? msg 'distance)
           (lambda (other-point)
             (let* ([dx (- x (other-point 'x))]
                    [dy (- y (other-point 'y))])
               (sqrt (+ (* dx dx) (* dy dy)))))])))
```

After carefully examining these two expressions, the pieces just fall into place:[22]

[22] This showcases the use of *nested ellipses* in a pattern expression. Holy cow, that rocks! Racket's insistence in always pairing an ellipsis with a named identifier and strict use of parentheses really help distinguish the ellipsis pairings.

```
(define-syntax my-class
  (syntax-rules (method)
    [(my-class <class-name>
       ; This ellipsis is paired with <attr>
       (<attr> ...)
       ; This ellipsis is paired with <param>
       (method (<method-name> <param> ...) <body>)
       ; This ellipsis is paired with the whole previous pattern
       ...)
     (define (<class-name> <attr> ...)
       (lambda (msg)
         (cond [(equal? msg (quote <attr>)) <attr>]
               ...
               [(equal? msg (quote <method-name>))
                (lambda (<param> ...) <body>)]
               ...
               [else "Unrecognized message!"]))))]))
```

The object `__dict__`

You might notice that the repeated `(equal? msg <symbol>)` clauses of the `cond` implement a (slow!) dictionary lookup, where the name of an attribute is keyed to its value for that object. Indeed, in Python every object has a special attribute `__dict__` that refers precisely to its dictionary of attributes (but not methods—more on this later):

```
class A:
    def __init__(self, x):
        self.x = x
        self.y = 10
        self.z = 'hello!'

>>> a = A(True)
>>> a.__dict__
{'x': True, 'y': 10, 'z': 'hello!'}
```

We can improve the performance of our `my-class` macro by using a built-in hashing data structure in Racket to store our attributes and methods.

```
(define-syntax my-class
  (syntax-rules (method)
    [(my-class <class-name> (<attr> ...)
       (method (<method-name> <param> ...) <body>) ...)

     (define (<class-name> <attr> ...)
       (lambda (msg)
         (let* ([__dict__ ; Use the same name as Python
                 (make-immutable-hash
                  (list (cons (quote <attr>) <attr>)
                        ...
                        (cons (quote <method-name>)
                              (lambda (<param> ...) <body>))
                        ...
                        ))])

           ; Look up the given attribute in the object's dictionary.
           (hash-ref __dict__ msg
                     ; Raise an error if attribute not found.
                     (attribute-error (quote <class-name>) msg))))))]))

; Return a thunk that raises an attribute error (with an appropriate message).
(define (attribute-error object attr)
  (lambda () (error (format "~a has no attribute ~a." object attr))))
```

Exercise Break!

2.9 Modify the `__dict__`-based class implementation to enable accessing the `__dict__` value from outside the object, as we can in Python. For example:

```
> (define p (Point 2 3))
> (p '__dict__)
'#hash((x . 2) (y . 3))
```

Note that in Python, the `__dict__` attribute itself doesn't appear in the object's dictionary. Can you achieve the same effect in Racket?

The problem of `self`

So far, our implementation of classes may seem to work, but it has one major deficiency: we cannot explicitly reference the calling object in a method. Consider the following example in Python:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def size(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)

    def same_radius(self, other):
        return self.size() == other.size()
```

Our current implementation has no trouble implementing the initializer method `__init__` (it does this automatically), and it can handle simple attribute access of `x` and `y` because those identifiers would be bound in the closure returned by our class constructor. However, `self.same_radius()` poses a problem:

```
(my-class Point (x y)
  (method (size) (sqrt (+ (* x x) (* y y))))
  (method (same-radius other)
    (equal? ??? ((other 'size)))))
```

Unlike `x` and `y`, there is no `size` *identifier* in scope in the closure: instead, `'size` is a key in the dictionary, but we have no way to automatically convert it! Perhaps the most direct way of solving this problem is to modify our macro to introduce identifiers for each method as well, making them exactly parallel to the "data" attributes that are the constructor parameters:[23]

[23] This models the notion of an implicit `this` in, e.g., Java.

```
(method (same-radius other)
  (equal? (size) (other 'size)))
```

However, this has the drawback that we still wouldn't have a way of referencing `self`, e.g., to pass the calling object to a completely different function. So instead, we'll look at an approach to make `self` an accessible identifier, referencing the calling object:

```
(method (same-radius other)
  (equal? ((self 'size)) ((other 'size))))
```

Pythonic inspiration

To guide our design and implementation, let's first review how `self` works in Python. You are already familiar with the most visible part: `self` is an explicit parameter written in every instance method of the class![24] In fact, if we adopt this convention then we can write the following Racket code without making any changes to our macro at all:

[24] In fact, the name `self` is just a convention: technically, Python interprets the *first* parameter as the one that is bound to the calling object, whatever the parameter name actually is.

```
(my-class Point (x y)
  (method (size self) (sqrt (+ (* (self 'x) (self 'x))
                               (* (self 'y) (self 'y)))))
  (method (same-radius self other)
    (equal? ((self 'size) self) ((other 'size) other))))
```

Look at that last line carefully—what's going on there? Our goal is for `(self 'size)` to return a function that takes no arguments, so that we can just call it (by enclosing it in parentheses). However, what we have here is that `(self 'size)` returns a function that takes a single argument, `self`, and so we need to pass that argument. The same is true when we call `(other 'size)`, too. Of course, this is very redundant: since we're calling `(self 'size)` already, we *know* what the calling object is, and shouldn't need to pass it in again!

This is solved in Python: an instance method call `self.size()` automatically binds the value to the left of the period, `self`, to the first parameter of the `size` method; the arguments in the parentheses get bound to the remaining

method parameters. So how do we modify our macro to add this "automatic `self` binding."? We again turn to Python.

Recall that in our initial look at Python `__dict__` values we noted that an object's `__dict__` only stored its data attributes, and not its methods. It turns out that this is because methods are stored in the `__dict__` of the *class itself*:

```
>>> Point.__dict__
# something that's a wrapper around a dictionary
>>> Point.__dict__['size']
<function Point.size at 0x04986300>
>>> Point.__dict__['same_radius']
<function Point.same_radius at 0x04986228>
```

This gives us a path towards an implementation of methods in our Racket macro:

1. Store methods in a separate class-level dictionary shared among all instances.
2. When methods are looked up in this dictionary, don't just return the method as-is; instead, bind the first parameter of the method to the calling object, and return that new method![25]

[25] Put another way, we *partially apply* the method on the first argument.

Here's an implementation of (1), showing only the macro template:

```
(begin
  ; This is a dictionary of the methods associated with a class.
  (define class__dict__
    (make-immutable-hash (list
                          (cons (quote <method-name>)
                                (lambda (<params> ...) <body>))
                          ...)))
  (define (<class-name> <attr> ...)
    (let*
        ([self__dict__
          ; The object's __dict__ now only stores non-function
          ; attributes. Methods are looked up in class__dict__.
          (make-immutable-hash
          (list (cons (quote <attr>) <attr>) ...))])
      (lambda (attr)
        (cond
          ; Note the lookup order (first object, then class).
          ; This is the same as Python.
          [(hash-has-key? self__dict__ attr)
          (hash-ref self__dict__ attr)]
          [(hash-has-key? class__dict__ attr)
          (hash-ref class__dict__ attr)]
          [else (attribute-error (quote <class-name>) msg)])))))
```

This is a little longer, but conceptually the only thing we've done is taken our original object dictionary and split it in to two, putting the data attributes in one and the methods in the other. (Unfortunately, our methods can't access instance variables anymore, because methods are no longer in the `lambda (<params> ...)` closure. We'll fix that in our next iteration.)

However, this alone doesn't solve (2). First, we introduce the function `fix-first`,[26] which is a higher-order function that takes a function `f` that takes $n+1$ arguments, and a value `x`, and returns a function `g` that takes n arguments, such that `g(x1, x2, ..., xn)` is equivalent to `f(x, x1, x2, ... xn)`. So then in the second branch of the `cond`, we can do `(fix-first ??? (hash-ref class_dict__ attr))`. But what goes in the `????` We need a way to refer to the object itself—and remember that the "object" is the entire `lambda` expression itself. In other words, this expression needs to be *recursive*; we can do this by using `letrec`:

[26] We leave the implementation of this function as an exercise.

```
(define-syntax my-class
  (syntax-rules (method)
    [(my-class <class-name> (<attr> ...)
       (method (<method-name> <params> ...) <body>) ...)

     (begin
       ; This is a dictionary of the methods associated with a class.
       (define class__dict__
         (make-immutable-hash (list
                               (cons (quote <method-name>)
                                     (lambda (<params> ...) <body>))
                               ...)))
       (define (<class-name> <attr> ...)
         ; We use letrec to give a name to the object we're returning.
         ; This is so that we can bind the object to the 'self'
         ; parameter in the call to fix-first.
         (letrec
             ([self__dict__
               ; The object's __dict__ now only stores non-function
               ; attributes. Methods are looked up in class__dict__.
               (make-immutable-hash
                (list (cons (quote <attr>) <attr>) ...))]
              [me (lambda (attr)
                    (cond
                      [(hash-has-key? self__dict__ attr)
                       (hash-ref self__dict__ attr)]
                      [(hash-has-key? class__dict__ attr)
                       (fix-first me (hash-ref class__dict__ attr))]
                      [else
                       ((attribute-error (quote <class-name>) attr))]))])
           ; Return the lambda representing the object.
           me))])))
```

The power of chained lookups

In the previous section, we used the idea of a *lookup chain* to separate instance attributes and methods. The idea of using multiple dictionaries to store different kinds of "attributes" is a versatile technique, and is commonly used in interpreters to implement different forms of inheritance. In Python, for example, since each class has its own `__dict__`, it is straightforward to implement method inheritance simply by inspecting the sequence of superclasses until the method is found.[27]

[27] This works even though Python supports multiple inheritance; the built-in `mro` function, standing for *method resolution order*, returns a sequence of the superclasses in the order they are checked for a given method.

JavaScript dispenses with the notion of separate "classes", and instead supports a chain of objects directly. In this language, almost every object keeps a reference to another object, called its *prototype*. When an attribute is accessed on an object, the object's own "dictionary" is checked first; if the attribute isn't found, the object's prototype is checked, and then the object's prototype's prototype, and so on until the root "base object" (with no prototype) is reached. This form of inheritance is known as *prototypal inheritance*, distinguishing it from the class-based inheritance of Python and other languages. But fundamentally, both class-based and prototypal inheritance can be thought of as variations of the same idea of chained lookups!

Exercise Break!

Each of these exercises involves extending our basic class macro in interesting ways. Be warned that these are more challenging than the usual exercises.

2.10 Modify the `my-class` macro to add support for **private attributes**.

2.11 Modify the `my-class` macro to add support for some form of **inheritance**, as described in the previous section.

Manipulating control flow I: streams

In the first half of this chapter, we saw how to use macros to introduce new syntactic forms to reduce boilerplate code and create name bindings for classes. Hopefully this gave you a taste of the flexibility and power of macros: by changing the very syntax of our core language, we enable new paradigms, and hence new idioms and techniques for crafting software. Now, we'll start to look more explicitly at another common usage of macros: manipulating control flow to circumvent the eager evaluation of function calls in Racket.

Our first foray into this area will be to implement streams, a "lazy" analogue of the familiar list data type. First, recall the recursive definition of a list:

- A list is either empty, or
- A value "cons'd" with another list.

In Racket, `cons` is a function, and so all lists built up using `cons` are eagerly evaluated: all list elements are evaluated when the list is constructed.[28] However, this is often not necessary: when we traverse a list, we often only care about accessing one element at a time, and do not need the other elements to have been evaluated at this point. This motivates our implementation of a stream in Racket, which follows the following recursive definition:

[28] The same is true of the Racket built-in `list`, as it too is a function.

- A stream is either empty, or
- A *thunk wrapping a value* "cons'd" with a *thunk wrapping another stream*.

As we discussed in the previous chapter, the thunks here are used to delay the evaluation of the elements of the stream until it is called. While we could implement this stream version of `cons` by explicitly passing thunks into the built-in `cons`, we again use macros to reduce some boilerplate:

```
13 ; Empty stream value, and check for empty stream.
14 (define s-null 's-null)
15 (define (s-null? stream) (equal? stream s-null))
16
17 #|
18 (s-cons <first> <rest>) -> stream?
19   <first>: any/c?
20   <rest>: stream?
21     E.g., s-null or another s-cons expression).
22
23   Creates a stream whose first value is <first>, and whose other
24   items are the ones in <rest>. Unlike a regular list, both <first>
25   and <rest> are wrapped in a thunks, delaying their evaluation.
26
27   Note: s-cons is a MACRO, not a function!
28 |#
29 (define-syntax s-cons
```

```
  (syntax-rules ()
    [(s-cons <first> <rest>)
     (cons (thunk <first>) (thunk <rest>))]))

; These two define the stream-equivalents of "first" and "rest".
; We need to use `car` and `cdr` here for a technical reason that
; isn't important for this course.
(define (s-first stream) ((car stream)))
(define (s-rest stream) ((cdr stream)))


#|
(make-stream <expr> ...) -> stream?
  <expr> ... : any/c?

  Returns a stream containing the given values.
  Note that this is also a macro. (why?)
|#
(define-syntax make-stream
  (syntax-rules ()
    [(make-stream) s-null]
    [(make-stream <first> <rest> ...)
     (s-cons <first> (make-stream <rest> ...))]))
```

The beauty of this macro-based stream implementation is that its public interface is identical to built-in lists:

```
> (define s1 (s-cons 3 (s-cons 4 (s-cons 5 s-null))))
> (s-first s1)
3
> (s-first (s-rest s1))
4
> (s-first (s-rest (s-rest s1)))
5
> (s-null? (s-rest (s-rest (s-rest s1))))
#t
```

The difference, of course, is in the creation of these streams:

```
> (define items1 (list 1 2 (error "error")))
ERROR error
> (define items2 (make-stream 1 2 (error "error")))
> (s-first items2)
1
```

Lazy lists in Haskell

It's been a while since we worked with Haskell, but it's worth pointing out that Haskell uses lazy evaluation for all of its function calls, and so the cons operation `(:)` is already lazy. This means that in Haskell, the list data type we've been using all along are already streams:

```
> x = [1, 2, error "error"]
> head x
1
```

Infinite streams

In this section, we'll take a brief look at one of the mind-bending consequences of using streams to delay evaluation of the elements in a sequence: constructing and manipulating *infinite* sequences. While we could do this in Racket, we'll use Haskell's built-in list data types to underscore the fact that Haskell's lists really are streams.

Here are some simple examples.

```
myRepeat x = x : myRepeat x

> take 3 (myRepeat 6)
[6,6,6]

nats = 0 : map (+1) nats

> take nats 5
[0,1,2,3,4]
```

What's going on with that wild `nats` definition? How does `take nats 5` work? To understand what's going on, let's look at a simpler example: `head nats`. Let's carefully trace it using the central mechanic of pure functional programming: substitution.

1. In Haskell, `head` is defined through pattern-matching as `head (x:_) = x`.
2. `nats` is defined as `0 : map (+1) nats`. When we pattern-match `nats` against the definition for `head`, we get `x = 0`, and `0` is returned. Note that `head` completely ignores the rest of the list, and so we don't need to look at the recursive part at all!

Now something a bit more complicated: `head (tail nats)`. Again, let's substitute, keeping in mind the definition for `tail`, `tail (_:xs) = xs`.

```
head (tail nats)
-- We can't call `head` until we can pattern-match on (x:_).
-- To do this, we'll need to expand `(tail nats)` a bit.
head (tail (0 : map (+1) nats))
-- We can evaluate the inner part using pattern-matching on the
-- implementation of tail:
--   tail (_:xs) = xs
head (map (+1) nats)
-- Now we need the following definition of map:
--   map _ [] = []
--   map f (x:xs) = (f x) : (map f xs)
-- In order to pattern-match, we need to expand nats again.
head (map (+1) (0 : map (+1) nats))
-- This allows us to pattern-match: x = 0, and xs = map plus1 nats.
-- We do the substitution into the second rule, but remember:
-- we aren't evaluating the inner expressions!
head (((+1) 0) : (map (+1) (map (+1) nats)))
-- At this point, we can *finally* pattern-match against `head`.
-- As before, we get to discard the complex "tail" of the argument.
(+1) 0
-- Finally, we evaluate ((+1) 0) directly to display the result.
1
```

So the first two elements of `nats` are 0 and 1. The expansion we did suggests what would happen if we access further elements. The part that we discarded, `(map plus1 (map plus1 nats))`, adds one, twice. In subsequent recursive expansions of `nats`, the `map plus1` function calls accumulate, leading to larger and larger numbers.

Since all of the common higher-order list functions work recursively, they apply equally well to infinite lists:

```
squares = map (\x -> x * x) nats
evens = filter (\x -> x `mod` 2 == 0) nats

> take 4 squares
[0,1,4,9]
```

We'll wrap up with a cute example of the Fibonacci numbers.[29]

[29] Exercise: how does this work?

```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

Exercise Break!

2.12 Define an infinite list containing all negative numbers. Then, define an infinite list `ints` containing all integers such that `elem x ints` halts whenever `x` is an integer.

2.13 Define an infinite list containing all rational numbers.

2.14 (Joke) Define an infinite list containing all real numbers.[30]

2.15 Look up **Pascal's Triangle**. Represent this structure in Haskell.

2.16 Repeat your work in the *Infinite streams* section using the stream interface we developed in Racket (i.e., `s-cons`, `make-stream`, etc.).

---

[30] Bonus: why is this funny?

Manipulating control flow II: the ambiguous operator `-<`

In the previous section, we looked at a rather simple application of macros to create an implementation of *streams*, a lazy data structure used to decouple the creation of data from the consumption of that data. For the remainder of this chapter, we'll explore this idea in a different context, one in which the data we create is specified by expressions using *non-deterministic choice*, and the consumption of that data is exposed explicitly through an impure function `next`. Our implementation of such expressions will introduce one new technical concept in programming language theory: the *continuation*, a representation of the control flow at a given point in the execution of a program.

Defining `-<` and `next`

Because you may not be familiar with this language feature, we will describe the functionality of the two relevant expressions before discussing their implementation. The main one is a macro called `-<` (pronounced "amb", short for "ambiguous"),[31] and it behaves as follows:

[31] This name was coined by John McCarthy, inventor of Lisp!

- `-<` takes an arbitrary positive number of argument subexpressions, representing the possible *choices* for the whole `-<` expression.
- If there is at least one argument, `-<` *evaluates and returns* the value of the first argument.
- In addition, if there is more than one argument, `-<` stores a "choice point" that (somehow) contains the remaining arguments so that they can be accessed, one at a time, by calling a separate function `next!`.
- Once all of the arguments have been evaluated, subsequent calls to `next!` return a special constant `DONE`.

Note that here we're describing the denotational semantics of `-<` and `next!`, but *not* their operational semantics! That is, before we get to any implementation at

all, we want you to understand the expected behaviour of these two forms. Here is an example of how we would like to use them in tandem:

```
> (-< 1 2 (+ 3 4))  ; The expression first evaluates to 1.
1
> (next!)  ; Calls to (next!) evaluate and return the other "choices".
2
> (next!)
7
> (next!)  ; Here our constant DONE is represented as a symbol 'done.
'done
```

-< as a self-modifying stream

To get started, let's write a skeleton definition for both `-<` and `next!`. Our inspiration—which might be yours too, based on the previous example—is a *self-modifying stream*: a sequence of values that is accessed one at a time using `next!`, but that actually uses mutation to keep track of which item is next.

The high-level description of our implementation is the following: we'll use a private "stream-like" variable `choices` that is *initialized* by calling `-<` and is accessed and mutated by calling `next!`:

```
(define choices (void))
(define (set-choices! val) (set! choices val))

; A constant representing the state of having "no more choices".
(define DONE 'done)

(define-syntax -<
  (syntax-rules ()
    ; Given one option, reset `choices` and return the option.
    [(-< <expr1>)
     (begin
       (set-choices! (void))
       <expr1>)]

    ; If there are two or more values, return the first one and
    ; store the others in a thunk.
    [(-< <expr1> <expr2> ...)
     (begin
       ; 1. Update `choices` to store a *thunk* that wraps the
       ;    remaining expressions. None of them are evaluated.
       (set-choices! (thunk (-< <expr2> ...)))

       ; 2. Evaluate and return the first expression.
       <expr1>)]))
```

```
47
48
49  #|
50  (next!) -> any/c?
51
52    Returns the next choice, or DONE if there are no more choices.
53  |#
54  (define (next!)
55    (if (void? choices)
56        DONE
57        (choices)))
```

Because we are (for the first time!) using mutation in our approach, our code looks a little different than all of the code we've written so far. In particular, we use the `set!` syntactic form to bind a new value to an existing name (wrapped in the helper `set-choices!`), and `begin` to enclose a sequence of expressions, evaluating each in turn and returning the value of the final one.[32]

[32] In a pure functional approach, it doesn't make sense to evaluate a sequence of expressions but only return the last one, because all the previous expressions can do nothing but return a value. But now with the introduction of `set!`, previous expressions can have a side-effecting behaviour like variable reassignment. This is also why we use an exclamation mark at the end of the function name `next!`.

We can verify that our macro works as intended on the example above:

```
1  > (-< 1 2 (+ 3 4))
2  1
3  > (next!)
4  2
5  > (next!)
6  7
7  > (next!)
8  'done
```

What's happening is the initial call to `-<` is returning the first argument (`1`) and storing the other expressions in a "stream" in `choices`. Calling `next!` is like calling `s-first` on the "stream" to return the first value, but because of the `-<` macro, this also has the effect of replacing `choices` with something akin to `(s-rest choices)`; i.e., permanently consuming the first element so that only the rest of the stream remains.

One particularly nice feature of this implementation is that because the `set-choices!` expression is evaluated before `<expr1>` in the macro template, `choices` will update even if the next expression raises an error:

```
1  > (-< 1 2 (/ 1 0) 4)
2  1
3  > (next!)
4  2
5  > (next!)
6  ERROR /: division by zero
```

```
> (next!)
4
```

This implementation is more complicated than it needs to be, if this were the limit of what we wanted to achieve. But we aren't done yet! While this implementation is able to store the other choices, that's all it saves.

```
> (+ 3 (-< 1 2 (+ 3 4)))
4
> (next!)
2
> (next!)
7
```

When the first expression is evaluated, `(-< 1 2)` returns 1 to the outer expression, which is why the output is 4. However, our implementation sets `next!` to `(thunk (-< 2 (+ 3 4)))`, and so when it is called, it simply returns 2.

In other words, even though the remaining choices in the choice expression are saved, the computational context[33] in which the expression occurs is not. To fix this problem, we'll need some way of saving this computational context—and this brings us to the realm of continuations.

[33] In the previous example, the "+ 3" is the computational context.

Continuations

In our studies to date, we have largely been passive participants in the operational semantics of our languages, subject to the evaluation orders of function calls and special syntactic forms. Even with macros, which do allow us to manipulate evaluation order, we have done so only by rewriting expressions into a fixed set of built-in macros and function calls. However, Racket has a data structure to directly represent (and hence manipulate) control flow from within a program: the continuation. Consider the following simple expression:

```
(+ (* 3 4) (first (list 1 2 3)))
```

By now, you should have a very clear sense of how this expression is evaluated, and in particular the *order in which* the subexpressions are evaluated.[34] For each subexpression s, we define its **continuation** to be a representation of what remains to be evaluated after s itself has been evaluated. Put another way, the continuation of s is a representation of the control flow from immediately after s has been evaluated, up to the final evaluation of the enclosing top-level expression (or enclosing continuation delimiter, a topic we'll explore later this chapter). For example, the continuation of the subexpression `(* 3 4)` is, in English, "evaluate `(first (list 1 2 3))` and then add it to the result." We represent

[34] For example, the fact that `(* 3 4)` is evaluated before the name `first` is looked up in the global environment.

this symbolically using an underscore to represent the "hole" in the remaining expression: `(+ _ (first (list 1 2 3)))`.[35] While it might seem like we can determine continuations simply by literally replacing a subexpression with an underscore, this isn't exactly true. The continuation of the subexpression `(first (list 1 2 3))` is `(+ 12 _)`; because of left-to-right evaluation order in function calls, the `(* 3 4)` is evaluated *before* the call to `first`, and so the continuation of the latter call is simply "add `12` to the result."

[35] We can also represent the continuation as a unary function `(lambda (x) (+ x (first (list 1 2 3))))`.

Since both identifiers and literal values are themselves expressions, they too have continuations. The continuation of `4` in the above expression is `(+ (* 3 _) (first (list 1 2 3)))`, and the continuation of `first` is `(+ 12 (_ (list 1 2 3)))`. Finally, the continuation of the whole expression `(+ (* 3 4) (first (list 1 2 3)))` is simply "return the value," which we represent simply as `_`.

Warning: students often think that an expression's continuation includes the evaluation of the expression itself, which is incorrect—an expression's continuation represents only the control flow *after* its evaluation. This is why we replace the expression with an underscore in the above representation! An easy way to remember this is that an expression's continuation doesn't depend on the expression itself, but rather the expression's position in the larger program.

`shift`: reifying continuations

So far, continuations sound like a rather abstract representation of control flow. What's quite spectacular about Racket is that it it provides ways to access continuations during the executaion of a program. We say that Racket *reifies* continuations, meaning it exposes continuations as values that a program can access and manipulate as easily as numbers and lists. We as humans can read a whole expression and determine the continuations for each of its subexpressions (like we did in the previous section). How can we write a *program* that does this kind of meta-analysis for us?

Though it turns out to be quite possible to implement reified continuations purely in terms of functions, this is beyond the scope of the course.[36] Instead, we will use Racket's syntactic form `shift` to capture continuations for us.

[36] Those interested should look up *continuation-passing style*.

Note: `shift` is imported from `racket/control`; include `(require racket/control)` for all code examples in this section.

Here is the relevant syntax pattern:

```
(shift <id> <body> ...)
```

A `shift` expression has the following semantics:

1. The current continuation of the `shift` expression is bound to `<id>`. By convention, we often use `k` for the `<id>`.
2. The `<body> ...` is evaluated (with `<id>` in scope).

3. The current continuation is **discarded**, and the value returned is simply the value of the last expression in `<body> ...`.

The first two points are pretty straightforward (the binding of the name and evaluation of an expression); the third point is the most surprising, as it causes the value of the `shift` expression to "escape" any enclosing expressions.[37] Let's start just by illustrating the second and third points, without worrying about the continuation binding.

[37] This is similar to using a `return` deep inside a function body, or `raise/throw` for exceptions.

```
> (shift k (+ 4 5))  ; The shift's body is evaluated and returned.
9
> (* 100 (shift k (+ 4 5)))  ; The shift's continuation, (* 100 _), is discarded!
9
```

Now let's return to point 1 above; the identifier `k` is bound to the `shift` expressions continuation. In the first example, this was simply the identity continuation ("return the value"), and in the second example, this was the continuation `(* 100 _)`. Of course, just binding the continuation to the name `k` is not very useful. Moreover, `k` is local to the `shift` expression, so we cannot refer to it after the expression has been evaluated. We'll use mutation to save the stored continuation.[38] Note that as in the previous section, `shift` can take multiple `<body>` expressions, evaluating each one and returning the value of the last one.

[38] You might have some fun thinking about how to approach this and the rest of the chapter without using any mutation.

```
> (define saved-cont (void))
> (* 100
     (shift k
            ; Store the bound continuation in the global variable.
            (set! saved-cont k)
            ; Evaluate and return this last expression.
            (+ 4 5)))
9
```

As expected, when we evaluate the expression, the `(* 100 _)` is discarded, and the `(+ 4 5)` is evaluated and returns. But now we have a global variable storing the bound continuation—let's check it out.

```
> saved-cont
#<procedure>
```

Racket stored the continuation `(* 100 _)` as the unary function `(lambda (x) (* 100 x))`—and we can now call it, just as we would any other function.

```
> (saved-cont 9)
900
```

Pretty cool! But it seems like `shift` did something we didn't want: discard the current continuation. It would be great if we could both store the continuation for future use, but also *not* interrupt the regular control flow, so that `shift` behaves like other kinds of Racket expressions. This turns out to be quite straightfoward, once we remember that we have access to the current continuation right in the body of the `shift`!

```
> (* 100
     (shift k
            ; Store the bound continuation in the global variable.
            (set! saved-cont k)
            ; Call the current continuation k on the desired argument.
            (k (+ 4 5))))
900
```

Using continuations in `-<`

Recall that our motivation for learning about continuations was that our current `-<` macro implementation stored the choices, but not the computational context around each choice. We now have a name for that computational context: it's just the continuation of the `-<` expression! Let's see how to improve our macro using `shift`. Here's a first attempt, replacing the `begin` with a `shift`:

```
(define-syntax -<
  (syntax-rules ()
    [(-< <expr1>)
     (begin
       (set-choices! (void))
       <expr1>)]

    [(-< <expr1> <expr2> ...)
     ; k is bound to the continuation of the (-< ...) expression
     (shift k
       (set-choices! (thunk (-< <expr2> ...)))
       (k <expr1>))]))
```

For example, in the expression `(+ 1 (-< 1 2 3))`, `k` would be bound to `(+ 1 _)`.

Of course, our macro is incomplete, as it doesn't actually "save" `k`. Doing this is straight-forward: inside our saved thunk, rather than simply returning the next choice (with `(-< <expr2> ...)`), we *call `k` on the result*![39]

[39] Of course, the mechanism that actually "saves" `k` is the *closure* created by the thunk. Hey, remember those?

```
(define-syntax -<
  (syntax-rules ()
    [(-< <expr1>)
     (begin
       (set-choices! (void))
       <expr1>)]

    ; Same as before, except the current continuation is stored in the
    ; choices thunk!
    [(-< <expr1> <expr2> ...)
     (shift k
            (set-choices! (thunk (k (-< <expr2> ...))))
            (k <expr1>))]))
```

And now we get our desired behaviour:

```
> (+ 1 (-< 10 20)) ; `k` is (+ 1 _)
11
> (next!)  ; `choices` is (thunk ((+ 1 _) (-< 20))))
21
> (next!)
'done
```

It is rather amazing that such a small change unlocks a huge number of possibilities for our macro; this truly illustrates the power of continuations and macros.

Using choices as subexpressions

So far, we've been focused on the construction of the choices themselves, and checked our work by calling `(next!)` at the top-level (in the REPL). But because the continuations we're storing behave as unary functions, we can take the value returned and pass them to a larger computation. For example:

```
> (* 100 (-< 1 2))
100
> (+ 3 (next!))
203
```

This is a pretty useful property that we seem to get "for free". However, it turns out that there are two subtle issues with our current implementation that come up when embedding `(next!)` inside larger expressions; exploring these will tighten up our implementation and deepen our understanding of continuations.

Delimiting continuations with `reset`

First, it turns out that `shift`'s ability to dynamically capture continuations is a bit *too* powerful. Suppose we extended our previous example with a third choice:

```
> (* 100 (-< 1 2 3))
100
> (+ 3 (next!))  ; Evaluates to (+ 3 (* 100 2))
203
```

If we call `next!` by itself one final time, something interesting happens:

```
> (next!)
303
```

This returns `303` rather than `300`! It seems that the continuation `(+ 3 _)` was captured in the first call to `next!`—why?

Recall that `(next!)` simply calls the thunk stored in `choices`, so let's take a closer look at the thunk that's stored. Let's take `(* 100 (-< 1 2 3))` and perform one step of the macro expansion:

```
(* 100 (-< 1 2 3))
; ==>
(* 100
   (shift k
          (set-choices! (thunk (k (-< 2 3))))
          (k 1)))
```

This macro is recursive: we see that the `choices` thunk contains another use of `-<`, which expands into another use of `shift`:

```
(* 100
   (shift k
          (set-choices!
           (thunk (k
                   (shift k2 ; Renaming to k2 for clarity
                          (set-choices! (thunk (k2 (-< 3))))
                          (k2 2)))))
          (k 1)))
```

So when we call `(next!)` inside `(+ 3 (next!))`, we call the `choices` thunk, which then evaluates a `shift`—and this new `shift` captures its current continuation, which includes not just the stored `(* 100 _)`, but the enclosing `(+ 3 _)` as well! The continuation bound to `k2` isn't just `(* 100 _)`, but instead `(+ 3 (* 100 _))`—and when we call that continuation on the last choice `3`, we get `303`.

This example is a bit counter-intuitive, and illustrates a pitfall of dynamic behaviour: because our current design of `-<` contains delayed calls to `shift`, the actual choices produced by `-<` depend on just on the initial context in which the `-<` was used, but also the contexts in which `(next!)` is called. As with many dynamic language features, this is not necessarily a bad thing, though it can it harder to predict the behaviour of our program. So in this section, we'll look at how to modify our implementation so that calls to `next!` do not modify the continuation applied to subsequent choices.

So far, we've been using `shift` to capture the continuation represented by the entire enclosing expression. However, Racket provides mechanisms to *delimit* continuations, enabling the programmer to exert control over which parts of a continuation are stored and discarded. The key ingredient we'll use here is `reset`, which you can think of as being a "barrier" that limits the continuation "scope" of a `shift`. When a `shift` occurs as a subexpression of a `reset`, the "current continuation" bound in the shift and discarded after the `shift` is evaluated only extends to the `reset`, and so doesn't include any expressions outside of the reset. Here are some examples:

```
> (* 100 (reset (* 2 (shift k 9)))) ; The (* 2 _) is discarded...
900                                  ; but the (* 100 _) is not.

> (define saved-cont (void))
> (* 100
     (reset (* 2 (shift k
                       (set! saved-cont k)
                       9))))
900
> (saved-cont 9)  ; saved-cont is (* 2 _); doesn't include (* 100 _)!
18
```

So to prevent calls to `(next!)` from capturing new continuations in subsequent choices, we can use `reset` to wrap the `thunk` calls:

```
(define (next!)
  (if (void? choices)
      DONE
      (reset (choices))))

> (* 100 (-< 1 2 3))
100
> (+ 3 (next!))
```

```
203
> (next!)
300
```

Aborting a computation with `shift`

Now consider a dual problem:

```
> (* 100 (-< 1 2))
100
> (+ 3 (next!))  ; Evaluates to (+ 3 (* 100 2))
203
```

If we repeat the call to `next!` inside a larger expression, we get an error:

```
> (+ 3 (next!))
+: contract violation
  expected: number?
  given: 'done
  argument position: 2nd
  other arguments...:
```

On the one hand, we should expect this to happen: calling `(next!)` when there are no more choices returns our special constant `DONE`, which cannot be added to a number. On the other, this is pretty annoying: under our current implementation, whenever we embed `(next!)` inside a larger expression, we either need to guarantee that there is at least one remaining choice, or do an explicit check on whether the value that's returned by `(next!)` is `DONE` or not before proceeding with the rest of the computation.

Given this trade-off, we'd like to a way for a call to `(next!)` to immediately return `DONE`, skipping any other computation around the `(next!)`. And as you might recall from earlier in this chapter, this exact behaviour is provided by `shift`! So here is the change to our `next!` implementation:

```
(define (next!)
  (if (void? choices)
      (shift k DONE)
      (reset (choices))))
```

If there are no more choices, rather than simply return `DONE`, we instead evaluate `(shift k DONE)`, which causes any surrounding computation to be discarded.[40]

[40] It's worth repeating here that this idea of "surrounding computation" is fundamentally dynamic. In the body of `next!`, it looks like there's nothing around `shift` other than the

With this change in place, we are able to now call `(next!)` freely within larger expressions, and have a `'done` "interrupt" the remaining computation.

```
> (* 100 (-< 1 2))
100
> (+ 3 (next!))  ; Evaluates to (+ 3 (* 100 2))
203
> (+ 3 (next!))  ; No more error!
'done
```

Branching choices

Even with the improvements of the previous section, we aren't quite done yet. Our overwriting of `choices` means that at any time, our program is only aware of one choice point. So in the example below, evaluating the second choice clobbers the first:

```
> (+ (-< 10 20) (-< 2 3))
12
> (next!)
13
> (next!)
'done
```

Now, as programming language designers we might stop here and say, "Well, too bad! You can only use one `-<` expression per program." But this is rather unsatisfying for our users, and so let's work a little harder to support multiple choice points. Obviously, just adding more global variables (`choices1`, `choices2`, etc.) isn't an option. We'll need to be able to represent an arbitrary number of choice points, and so a collection-type data structure is most appropriate. For simplicity, we'll use a straightforward one: a stack implemented with a list, with the front of the list representing the top of the stack:

```
; choices is now a list of thunks rather than a single thunk.
(define choices null)
(define (add-choice! c) (push! choices c))
(define (get-choice!) (pop! choices))

; Push a value onto a stack (add to front of list).
(define-syntax push!
  (syntax-rules ()
    [(push! <id> obj)
     (set! <id> (cons obj <id>))]))

```

```
; Pop a value from a stack (remove from front of list).
(define-syntax pop!
  (syntax-rules ()
    [(pop! <id>)
     (let* ([obj (first <id>)])
       (set! <id> (rest <id>))
       obj)]))
```

The idea is pretty straight-forward: every time we encounter a choice expression, we *push* the corresponding thunk onto the choices stack, and every time we call `next!`, we *pop* a thunk off the stack and call it.[41]

[41] This approach is the same as a stack-based implementation of depth-first search—this is not a coincidence.

```
(define-syntax -<
  (syntax-rules ()
    [(-< <expr1>) <expr1>]

    [(-< <expr1> <expr2> ...)
     (shift k
            (add-choice! (thunk (k (-< <expr2> ...))))
            (k <expr1>))]))

#|
(next!) -> any/c

  Returns the next choice, or DONE if there are no more choices.
|#
(define (next!)
  (if (null? choices)
      (shift k DONE)
      (reset ((get-choice!)))))
```

Now let's see what happens when we runour previous example.

```
> (+ (-< 1 2) (-< 3 10))
4
> choices ; choices stores two thunks, one for each -< expression
'(#<procedure> #<procedure>)
```

After evaluating the expression containing two choice expressions, we see that `choices` now stores two thunks.

```
> (next!)
11
> (next!)
```

```
5
> (next!)
12
> (next!)
'done
```

An important detail

Even though it is gratifying that the implementation yields the correct result, it should be at least somewhat surprising that this works so well. In particular, let us trace through that previous example again, but inspect the contents of `choices` each time:

```
> choices
'()
> (+ (-< 1 2) (-< 3 10))
4
> choices
'(#<procedure> #<procedure>)
```

As before, we know that the first element contains the choice `(-< 10)`, while the second contains `(-< 2)`.

```
> (next!)
11
> choices
'(#<procedure>)
```

After the first call to `(next!)`, the top choice is popped and evaluated, yielding 10. But why is 11 output? The captured continuation, of course: `(+ 1 _)`—note that by the time the second choice expression has been evaluated, the first was already evaluated to 1.

Only the single choice `(-< 2)` remains on the stack; because it contains just one argument, the natural thing to expect when calling next again is for this choice to be popped from the stack, leaving no choices. However, this is not what happens:

```
> (next!)
5
> choices
'(#<procedure>)
```

Why is a `5` output, and why is there still a function on `choices`? The answer to both of these questions lies in the continuation of the first expression `(-< 1 2): (+ _ (-< 3 10)))`. That's right, the continuation of the first choice expression contains the second one! This is why a 5 is output; the choice `(-< 2)` is evaluated and passed to the continuation, which then evaluates the second choice expression `(-< 3 10)`, which both returns a `3` *and* pushes a new thunk onto the stack.

Note that the new function on the stack has the same choice as earlier, `(-< 10)`, but its continuation is different: "add 2" rather than "add 1". This is what causes the final output:

```
> (next!)
12
> choices
'()
> (next!)
'done
```

Towards declarative programming

Even though we hope you find our work for the choice macro intellectually stimulating in its own right, you should know that this truly is a powerful mechanism that can change the way you write programs. Just as how in our discussion of streams we saw how thunks could be used to decouple the production and consumption of linear data, our choice library allows this decoupling to occur even when how we produce data is non-linear.

So far in this course, we have contrasted functional programming with imperative programming, with one of the main points of contrast being writing code that is more descriptive of the computation you want the computer to perform, rather than describing *how* to do it—that is, writing programs that are more *declarative* in nature.[42]

For example, here is a simple expression that generates all possible binary strings of length 5, following the English expression "take five characters, each of which is a 0 or 1":[43]

```
> (string (-< #\0 #\1)
          (-< #\0 #\1)
          (-< #\0 #\1)
          (-< #\0 #\1)
          (-< #\0 #\1))
"00000"
> (next!)
"00001"
```

[42] In this course, we take the view of declarative as being a spectrum rather than a yes/no binary characteristic. So we won't say that a language or style *is* declarative, but rather that it's more declarative than what we're used to.

[43] Racket uses the `#\` prefix to indicate a character literal.

```
> (next!)
"00010"
> (next!)
"00011"
> (next!)
"00100"
```

This easy composability is quite remarkable, and is arguably as close to a human English description as possible.[44] Though we can achieve analogous behaviour with plain higher-order list functions (or nested loops), the simplicity of this approach is quite beautiful.

[44] Modulo the code repetition—see the exercises below for further discussion on that.

Exercise Break!

2.17 Suppose we tried to generate all binary strings of length 5 by doing the following:

```
(let* ([char (-< #\0 #\1)])
  (string char char char char char))
```

What does this generate? Explain why. Fix this problem using thunks.

Predicates and backtracking

Generating "all possible combinations" of choices is fun and impressive, but not necessarily that useful without some kind of mechanism to filter out unwanted combinations. While we could do this by collecting choices into a stream and applying a traditional `filter` operation, we'll instead take a different approach to combine the *automatic generation* that choice expressions give us with *automatic backtracking*, which is at the core of the technique used by logic programming languages like Prolog. The **logic programming** paradigm is centred on one simple question: "is this true?" Programs are not a sequence of steps to perform, nor the combination of functions; instead, a logic program consists of the definition of a search space (the universe of values known to the program) and the statement of queries for the computer to answer based on these values. Based on this description, it is probably not hard to see that logic programs are generally more declarative than either imperative or functional programs, given that they fundamentally specify *what* is being queried, but leave it up to the underlying implementation of the language to determine *how* to answer the query.[45]

[45] A more familiar instance of this idea is the database query language SQL, in which programmers write queries specifying what data they want, without worrying (too much) about how this data will be retrieved.

Now, we already have a mechanism for defining a search space, using combinations of choices. What about queries? To start, we will define a query operator `?-`, which takes a unary predicate and an expression, and returns the value of the expression if it satisfies the predicate, and `DONE` if it doesn't.

```
(define (?- pred expr)
  (if (pred expr)
      expr
      DONE))

> (?- even? 10)
10
> (?- even? -3)
'done
```

Now, all of the work we did in the previous sections pays off: we can call this function with a choice expression!

```
> (?- even? (-< 1 2 3 4))
'done
> (next!)
2
> (next!)
'done
> (next!)
'done
```

When the `?-` is originally called, it gets passed the first value in the choice expression (i.e., `1`); but the continuation of the choice expression is `(?- even? _)`, and so each time `next!` is called, the next choice gets passed to the query. To use some terminology of artificial intelligence, calling `next!` causes *backtracking*, in which the program goes back to a previous point in its execution, and makes a different choice.

When making queries with choices, it is often the case that we'll only care about the choices that succeed (i.e., satisfy the predicate), and ignore the others. Therefore it would be nice not to see the intermediate `'done` outputs; we can make one simple change to our predicate to have this available:

```
(define (?- pred expr)
  (if (pred expr)
      expr
      ; Rather than giving up, just try the next choice!
      (next!)))

> (?- even? (-< 1 2 3 4))
2
> (next!)
4
> (next!)
'done
```

Now, every time the `(pred expr)` test fails, `(next!)` is called, which calls `?-` on the next choice; in other words, we *automatically backtrack* on failures.

The subtletly of capturing continuations

Not so fast! There's a bug with this implementation of `?-`. Suppose we want to nest `?-` inside a larger expressions; for example, the expression below represents the computation "multiply by 100 a choice of an even number between 1 and 5." Unfortunately, it doesn't work:

```
> (* 100 (?- even? (-< 1 2 3 4 5)))
20000
> (next!)
40000
> (next!)
'done
```

Something appears to be working: it does look like `?-` correctly selects the even numbers from the choice. The problem is there seems to be an extra multiplication by 100—which suggests that the `(* 100 _)` is being applied twice. Why?

The subtlety lies in the fact that `next!` returns `((get-choice!))` without discarding the current continuation (as we saw earlier, this allows `(next!)` to be composed with larger expressions). But when we "backtrack", we really don't want to preserve the current continuation; we want to entirely start over at a previous point in our computation. Once we understand this, the fix is straightforward: we wrap `(next!)` inside a `shift` to discard the current continuation.

```
(define (backtrack!)
  (shift k (next!))

(define (?- pred expr)
  (if (pred expr)
      expr
      (backtrack!)))

> (* 100 (?- even? (-< 1 2 3 4 5)))
200
> (next!)
400
> (next!)
'done
```

Search examples

Probably the easiest way to appreciate the power that this combination of choice and querying yields is to see some examples in action. We have already seen the use of this automated backtracking as a lazy `filter` operation, producing values in a list of arguments that satisfy a predicate one at a time, rather than all at once. We can extend this fairly easily to passing in multiple choice points, with the one downside being that our `?-` function only takes unary predicates, and so we'll need to pass all of the choices into a list of arguments.

Our problem will be the following: given a number `n`, determine all triples of numbers `(a, b, c)` whose product is `n`. Here are the definitions of the input data and predicate we'll use.

```
7  #|
8  (num-between start end) -> integer?
9    start: integer?
10   end: integer?
11     Precondition: start < end
12   Returns a choice of a number in the range [start..end - 1], inclusive.
13 |#
14 (define (num-between start end)
15   (if (equal? start (- end 1))
16       (-< start)
17       (-< start (num-between (+ 1 start) end))))
18
19 #|
20 (triples n) -> (listof integer?)
21   n: integer?
22
23   Returns a choice of a list of three integers between 1 and n, inclusive.
24 |#
25 (define (triples n)
26   (list (num-between 1 (+ n 1))
27         (num-between 1 (+ n 1))
28         (num-between 1 (+ n 1))))
29
30 #|
31 (product? n) -> (-> (listof integer?) boolean?)
32   n: integer?
33
34   Returns a predicate that takes a list of integers and returns whether
35   their product is n.
36 |#
37 (define (product? n)
38   (lambda (triple)
39     (equal? n (apply * triple))))
```

Finally, here is a use of the query for the range 1–12:

```
> (?- (product? 12) (triples 12))
'(1 1 12)
> (next!)
'(1 2 6)
...
> (next!)
'(12 1 1)
> (next!)
'done
```

See that? In just a few lines of code, we expressed a computation for finding factorizations in a purely declarative way: specifying *what* we wanted to find, rather than *how* to find the solutions. This is the essence of pure logic programming: we take a problem and break it down into logical constraints on certain values, give our program these constraints, and let it find the solutions. This should almost feel like cheating—isn't the program doing all of the hard work?

Remember that we are studying not just different programming paradigms, but also trying to understand what it means to design a language. We spent a fair amount of time implementing the language features `-<`, `next!`, and `?-`, expressly for the purpose of making it easier for users of our augmented Racket to write code. So here we're putting on our user hats, and take advantage of the designers' work!

Satisfiability

One of the most famous problems in computer science is a distilled essence of the above approach: the *satisfiability problem*, in which we are given a propositional boolean formula, and asked whether or not we can make that formula true. Here is an example of such a formula, written in the notation of Racket:[46]

[46] x1, x2, x3, and x4 are boolean variables

```
(and (or x1 (not x2) x4)
     (or x2 x3 x4)
     (or (not x2) (not x3) (not x4))
     (or (not x1) x3 (not x4))
     (or x1 x2 x3)
     (or x1 (not x2) (not x4)))
```

With our current system, it is easy to find solutions:

```
(define (sat lst)
  (let ([x1 (first lst)]
        [x2 (second lst)]
        [x3 (third lst)]
        [x4 (fourth lst)])
    (and (or x1 (not x2) x4)
         (or x2 x3 x4)
         (or (not x2) (not x3) (not x4))
         (or (not x1) x3 (not x4))
         (or x1 x2 x3)
         (or x1 (not x2) (not x4)))))

> (?- sat (list (-< #t #f)
                (-< #t #f)
                (-< #t #f)
                (-< #t #f)))
'(#t #t #t #f)
> (next!)
'(#t #t #f #f)
> (next!)
'(#t #f #t #t)
> (next!)
'(#t #f #t #f)
> (next!)
'(#f #f #t #t)
> (next!)
'(#f #f #t #f)
> (next!)
'done
```

3 *Type systems*

> I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.
>
> Tony Hoare

Here is a problematic Racket function.

```
1 (define (f x)
2   (+ x (first x)))
```

Calling this function on any input will fail, due to a *type error*: the parameter `x` cannot simultaneously be a number (to be a valid argument to `+`) and a list (to be a valid argument to `first`). Our goal for this chapter is to study *type systems* implemented by different programming languages. While you certainly have an intuitive notion of types coming from past programming experience, you may not yet have appreciated the complex design decisions that go into how types are represented and used in a programming language, and how differences in these decisions can have major impacts on the programs written in these languages.

We start with a few definitions to make sure we're on the same page. A **type** is a set of values together with a set of behaviours on those values. Note that this is a pretty abstract notion that differs among programming languages: an `integer` in Racket can take on a different set of values than an `int` in C,[1] and the built-in `String` functions in Haskell are different from the `String` methods in Java. But what is common to all of these languages is the notion of what information a type conveys: if we say that an expression `E` has type "integer", this constrains both the possible *values* we expect that expression to have (i.e., what its denotational meaning is), and how we can use this expression in the context of a large program (e.g., `(+ 1 E)` is okay but `E[10]` is not). A **type system** is the set of rules in a programming language governing the semantics of types in the language. This includes how types are defined, the syntax rules governing where and how types can be written, and how types affect the operational and

[1] Try adding two very large integers in each language!

denotational semantics of the language.

Describing type systems

Our plan for this chapter is to focus mainly on Haskell's type system, though we'll make frequent comparisons to other languages. To begin, we'll look at two general spectrums on which a programming language's type system can reside; these are some of the most commonly-cited characteristics of a type system.

Strong and weak typing

One of the more confused notions is the difference between strong/weak typing and static/dynamic typing. Let's start with strong/weak typing. However, please keep in mind that there is some disagreement about the exact meaning of strong/weak typing, and we present only one such interpretation here.

In a **strongly-typed** language, every value has a fixed type during the execution of a program.[2] Even though this might seem fairly obvious, keep in mind that this property isn't strictly necessary: all data is stored in memory as 0's and 1's, and it is possible that the types are not stored at all!

[2] This does not imply that the type of an *identifier* never changes during runtime. More on this later.

Most modern languages are strongly-typed, which is why we get type errors when we try to call a function on arguments of the wrong type. On the other hand, a **weakly-typed** language has no such guarantees: values can be implicitly interpreted as having a completely different type at runtime than what was originally intended, in a phenomenon known as *type coercion*. For example, in many languages the expression `"5" + 6` is semantically valid, raising no runtime errors. Many languages will convert the integer 6 into the string `"6"`, and concatenate the strings to obtain the string `"56"`.[3] A Java feature called **autoboxing** allows primitive values to be coerced into their corresponding wrapper classes (e.g., `int` to `Integer`) during runtime.

[3] Be aware of your biases here! This evaluation to get `"56"` is so familiar we don't bat an eye. But in other languages like PHP, the same expression would be evaluated to `11`.

The previous examples of type coercion might be surprising, but aren't necessarily safety concerns. In C, the situation is quite different:[4]

[4] We don't want to ruin the fun—try running this program yourself!

```c
#include <stdio.h>

int main(void) {
  printf("56" + 1);
  return 0;
}
```

Depending on who you ask, different levels of type coercion might warrant the label of strong or weak typing. For example, basically every programming language successfully adds `3.5 + 4` without raising a type error. So as with many properties of languages, the takeaway message is that strong/weak typing

is not a binary property, but a set of nuanced rules about what types can be coerced and when. It goes without saying that mastering these rules is important for facility in any programming language.

Static and dynamic typing

Even assuming that we are in a very strongly-typed language with little or no type coercion, there is still the fundamental question of "when does the program know about (and check) types?" The answer to this question is one of the key characteristics of a programming language, as it deeply influences both the language's semantics and the implementation of any interpreter or compiler for that language.

In **statically-typed** languages, the type of every expression is determined directly from the source code, *before any code is actually executed*. In languages that permit mutation, static typing requires that even as variables change values, they can only change to values within the same type. Most of these languages require an explicit type to be specified at the declaration of each variable, although as we'll discuss later, this is not an absolute requirement for static typing.

In contrast, **dynamically-typed** languages do not perform any type-checking until the program is run. In such languages, a type error is a *runtime error*—in the same category as division by zero or array out-of-bounds errors.

As we mentioned earlier, types really form a collection of constraints on programs, so it shouldn't come as a surprise that static typing is really a form of program verification (like proving program correctness from CSC236). If a compiler can check that nowhere in the *code* does a variable take on a different type, we are guaranteed that at no point in the program's *runtime* does that variable take on a different type. By checking the types of all expressions, variables, and functions, the compiler is able to detect code errors (such as invalid function calls) before any code is run at all. Moreover, static type analysis can be sometimes used as a compiler optimization: if we have compile-time guarantees about the types of all expressions during the program's run, we can then drop the runtime type checks when functions are called. As we'll discuss in the next section, statically-typed languages perform type checking before programs are run, essentially rejecting certain programs because they have type errors. This sounds great, and it very often is. However, type systems gain their power by enforcing type constraints; the stricter the constraints, the more erroneous programs can be rejected. This can also have the consequence of rejecting perfectly correct programs[5] that do not satisfy these constraints. However, it's also worth pointing out that most statically-typed languages have a way of "escaping" the type system with a universal type like `Object` or `Any`, and in this way allow one to write code without any type constraints.

[5] We'll see a few examples of this when we study Haskell's type system in the next section.

The basics of Haskell's type system

In Haskell, types are denoted by capitalized names like `Bool`, `Char`, and `Integer`. In the Haskell interpreter, you can see the type of any expression with `:type`, or `:t` for short. Some examples follow; since Haskell's numeric types are a bit complex, we're deferring them until later.

```
> :t True
True :: Bool
> :t 'a'
'a' :: Char
> :t "Hello"
"Hello" :: [Char]
```

Note that the square brackets surrounding the `Char` indicates a list: like C, Haskell interprets strings simply as a list of characters.[6] One important difference between Haskell and Racket is that lists must contain values of the same type, so the expression `[True, 'a']` is rejected by Haskell. A similar restriction also applies to `if` expressions: both the `then` and `else` subexpressions must have the same type:

[6] Haskell also provides a type synonym (i.e., alias) for `[Char]` called `String`.

```
> :t (if 3 > 1 then "hi" else "bye")
(if 3 > 1 then "hi" else "bye") :: [Char]
```

The above example illustrates the static nature of Haskell's type-checking: the type of the entire `if` expression is given, without evaluating it! Haskell can do this precisely because of the aforementioned restriction: because the Haskell compiler knows that both branches evaluate to strings, it is able to determine that the type of the overall expression is a string *without* knowing which branch would actually be evaluated!

Function types and currying

In Haskell, all functions have a type that can be inspected in the same way.

```
> :t not
not :: Bool -> Bool
```

The type signature `Bool -> Bool` means that `not` is a function that takes as input a `Bool` value, and then returns a `Bool` value. The very existence of a type signature for a function has huge implications for how functions can be used. For instance, functions in Haskell must have a fixed number of arguments, fixed types for all their arguments, and one fixed return type. Contrast this with Racket's `member` function, which returns either a list or `#f` (a boolean).

What about functions that take in two parameters? For example, what is the type of the "and" function `(&&)`?

```
> :t (&&)
(&&) :: Bool -> Bool -> Bool
```

Huh, that's weird. One probably would have expected something like `(Bool, Bool) -> Bool` to denote that this function takes in two arguments. So what does `Bool -> Bool -> Bool` actually mean? The `->` operator in Haskell is right-associative, which means that the proper grouping of that signature is `Bool -> (Bool -> Bool)`. This is quite suggestive: Haskell interprets `(&&)` as a *unary* higher-order function that returns a function with type `Bool -> Bool`! This is the great insight from the lambda calculus known as **currying**: any multi-parameter function can be broken down into a nested composition of single-valued functions.[7]

[7] An observant reader might have noticed that the syntactic rules in the Prologue for generating lambda calculus expressions did not contain any mention of multi-parameter functions. This is because currying makes it possible to express such functions within the single-parameter syntax.

Consider the following function definition, which you can think of as a *partial application* of `(&&)`:

```
newF y = (&&) True y
```

The type of `newF` is certainly `Bool -> Bool`: if `y` is `True` then the function evaluates to `True`, and if `y` is `False` then the expression is `False`. By fixing the value of the first argument to `(&&)`, we have created a new function. But in fact the `y` is completely unnecessary to defining `newF`, and the following definition is completely equivalent.

```
newF = (&&) True
```

This makes sense when we think in terms of referential transparency: every time we see an expression like `newF True`, we can *substitute* this definition of `newF` to obtain the completely equivalent expression `(&&) True True`.

It turns out that Haskell treats *all* functions as unary functions, and that function application is indicated by simply separating two expressions with a space. An expression like `(&&) False True` might look like a single function application on two arguments, but the true syntactic form parsed by Haskell is `((&&) False) True`,[8] where `((&&) False)` is evaluated to create a new function, which is then applied to `True`. In other words, the following definitions of `(&&)` are equivalent:

[8] function application is *left*-associative

```
-- Usual definition
(&&) x y = if x then y else False
-- As a higher-order function
(&&) x = \y -> if x then y else False
```

These two definitions are *completely equivalent* in Haskell, even though they are not in most other programming languages! Going a step further, the following ternary function can be defined in Haskell in four completely equivalent ways:

```
f x y z = x + y * z
f x y = \z -> x + y * z
f x = \y -> (\z -> x + y * z) -- parentheses shown for clarity
f = \x ->
  (\y -> (\z -> x + y * z))
```

Because binary operators are primarily meant to use infix, Haskell provides a special syntax for currying infix operators called **sectioning**. Let's return to `(&&)` as an example of this syntax.

```
f = (True &&)  -- equivalent to f x = True && x
g = (&& True)  -- equivalent to g x = x && True
```

Note that unlike standard currying, in which partial application must be done with a left-to-right argument order, sectioning allows us to partially apply an operator by fixing either the first or the second argument.

In practice, partial application is a powerful tool for enabling great flexibility in combining and creating functions. Consider the function `addToAll`, which adds a number to every item in a list.

```
addToAll n lst = map (\x -> x + n) lst
```

This function is already quite concise because of our use of `map`, but using what we've learned in this section, we can given an even more elegant definition.

First, rather than creating a new anonymous function `(\x -> x + n)`, we can use sectioning to express an "add n" function directly:

```
addToAll2 n lst = map (+ n) lst
```

Taking advantage of automatic currying, we can remove `lst` from both sides, leaving just

```
addToAll3 n = map (+ n)
```

It might seem almost magical to define this function in such a short fashion. As you get more comfortable with functional programming and thinking in terms

of combining and creating functions, you can interpret the final definition quite elegantly as "`addToAll3` maps the 'add `n`' function."

Defining types in Haskell

Now that we have been introduced to Haskell's type system, we'll study how to define our own types in this language.

The running example we'll use in this section is a set of types representing simple geometric objects. There's quite a bit of new terminology in this section, so make sure you read carefully! First, a point `(x,y)` in the Cartesian plane can be represented by the following `Point` type.

```
data Point = Point Float Float
```

You can probably guess what this does: on the left side, we are defining a new type `Point` using the `data` keyword. On the right, we define how to create instances of this type; the `Point` is a **value constructor**, which takes two `Float` arguments and returns a value of type `Point`. Let's play around with this new type.

```
> p = Point 3 4
> :t p
p :: Point
> :t Point
Point :: Float -> Float -> Point
```

There is a common source of confusion when it comes to Haskell's treatment of types that we are going to confront head-on. In Haskell, there is a strict separation between the *expression language* used to define names and evaluate expressions, and the *type language* used to represent the types of these expressions. We have seen some artifacts of this separation already: for example, the `->` operator is used in expressions as part of the syntax for a lambda expression, and in types to represent a function type. We have also seen the mingling of these in type annotations, which come in the form `<expression> :: <type>`.

A type declaration is a different instance of this mingling. In our above example, the `Point` on the left is the name of a new *type*, while the `Point` on the right is the name of a *function* that returns values of that type. In typical object-oriented languages, the *constructor* of a class is a special method, and its significance is enforced semantically (in part) by having its name be the same as the class name. This is actually not true in Haskell, as we illustrate below:

```
-- Point is the type and MyPoint is its constructor.
data Point = MyPoint Float Float

```

```
> p = MyPoint 3 4
> :t p
p :: Point
> :t MyPoint
MyPoint :: Float -> Float -> Point
```

Operating on points

This type definition is much more in the spirit of C structs than classes in the familiar object-oriented setting, in that this allows us to define members of the type (in the case of points, two floats), but not to bundle operations on these types. In Haskell, we provide such operations by defining top-level functions, but naively doing so almost immediately poses a problem:

```
-- Compute the distance between two points
distance :: Point -> Point -> Float
distance p1 p2 = ???
```

How can we access the `Float` attributes of the point values? Just as we define functions by pattern matching on primitive values and lists, we can also pattern match on *any value constructor*.[9]

[9] In fact, the cons operator `(:)` is actually a value constructor for the built-in list type, so this is a feature you've been using all along!

```
distance :: Point -> Point -> Float
distance (Point x1 y1) (Point x2 y2) =
    let dx = abs (x1 - x2)
        dy = abs (y1 - y2)
    in
        sqrt (dx*dx + dy*dy)
```

In another testament to the beauty of Haskell's syntax, the left side of this function definition perfectly matches how we would call the function:

```
> distance (Point 3 4) (Point 1 2)
```

Here's another example where we create new `Point` values.

```
-- Take a list of x values, y values, and return a list of Points.
makePoints :: [Float] -> [Float] -> [Point]
makePoints xs ys = zipWith (\x y -> Point x y) xs ys

-- Or better:
makePoints2 = zipWith (\x y -> Point x y)
```

Even though `makePoints2` looks quite simple already, it turns out there's an even simpler representation using the fact that *the `Point` value constructor is just another function*! So in fact, the most elegant way of writing this function is simply:

```
makePoints3 = zipWith Point
```

Unions

The previous section introduced struct-based types in Haskell, in which a constructor like `Point` is used essentially as a mechanism to group pieces of data into one value. In this section, we'll look at how Haskell supports union types, starting with the most basic form that you may be familiar with already: *enumerations*.

Consider a type used to represent a day of the week: Monday, Tuesday, etc. While we could use either a string or integer to represent such a type, both of these primitive types have the drawback that they are *more* expressive than required, and so checks for "invalid" days could not be caught by the type system. Instead, since there are a small and finite number of possible days, we can create an *enumeration type* by explicitly giving a list of possible constructors:

```
data Day = Monday      --
         | Tuesday     --
         | Wednesday   --
         | Thursday    --
         | Friday      --
         | Saturday    --
         | Sunday      --
```

Here, the vertical bar `|` should be read as an "or", separating the different possible value constructors for `Day`. Note that `Monday`, `Tuesday`, etc. are indeed value constructors, same as `Point` in the previous section, except that they do not take in any arguments:[10]

```
> :t Monday
Monday :: Day
```

[10] Due to Haskell's laziness, we can view these constructors either as nullary functions, or simply as singleton values.

This use of alternative constructors in Haskell is pretty standard in other languages as well. What is more unusual is Haskell's ability to combine *constructors that take arguments* with *alternatives* to declare types that have radically different forms:

```haskell
data Shape
    = Circle Point Float      -- ^ centre and radius
    | Rectangle Point Point   -- ^ top-left and bottom-right corners
```

Here it is *really* important to get the terminology correct. This code creates a new type `Shape`, which has two value constructors: `Circle` and `Rectangle`. Here's how we might use this new type.[11]

[11] Remember: `Circle` and `Rectangle` are *functions*, while `Shape` is a *type*.

```haskell
> shape = Circle (Point 3 4) 1
> :t shape
shape :: Shape
> :t [Circle (Point 3 4) 5, Rectangle (Point 0 0) (Point 1 1)]
[Circle (Point 3 4) 5, Rectangle (Point 0 0) (Point 1 1)] :: [Shape]
```

Here's another example of using pattern-matching to write a function operating on `Shapes`. Note that each constructor gets its own pattern-match rule, and we can even nest patterns.

```haskell
area :: Shape -> Float
area (Circle _ r) = pi * r * r
area (Rectangle (Point x1 y1) (Point x2 y2)) =
    abs ((x2-x1) * (y2-y1))

> area (Circle (Point 3 4) 1)
3.141592653589793
> area (Rectangle (Point 4 5) (Point 6 2))
6.0
```

Unions in C

You might be familiar with `struct` and `union` types in C, which together approximate algebraic data types from Haskell. That said, the C syntax is quite a bit more verbose. And for unions, there is additional legwork required of programmers to keep track of what part of the union is active. Here is how our `Shape` example might look in C. Note that the `circle` and `rectangle` members will overlap in memory: only one of them is available at a time.

```c
#include <stdio.h>

struct Point {
    float x, y;
};

union Shape {
    struct Circle {
        struct Point centre;
        float radius;
    } circle;
    struct Rectangle {
        struct Point corner1, corner2;
    } rectangle;
};

int main(void) {
    union Shape s;
    s.circle = (struct Circle){.radius=1};
    printf("Circle has radius %f\n", s.circle.radius);
    s.rectangle = (struct Rectangle) {{4, 5}, {6, 2}};
    printf(
        "Rectangle with corner (%f, %f)\n",
        s.rectangle.corner1.x,
        s.rectangle.corner1.y
    );

    return 0;
}
```

Now, what happens if we want to pass a `union Shape` to a function?

```c
void area(union Shape s) {
    if (... s is a circle) {
        ... operate on the circle
    } else {
        ... operate on the rectangle
    }
}
```

How do we know whether `s` is a circle or a rectangle? We don't! And, unlike in Haskell, we certainly can't pattern-match to find out. The C solution is for the programmer to explicitly maintain a *tag field* that indicates what the union is currently storing.[12] Unfortunately, this requires that we embed the union in a struct that houses both the tag field and the union—compare to Haskell, which

[12] This is reminiscent of what happens when we pass an array to a function; we have to manually keep track of the array's length.

keeps track of all of this for us!

```c
#include <stdio.h>
#include <math.h>
#define pi 3.141592653589793

struct Point {
    float x, y;
};

struct Shape {
    enum {CIRCLE, RECTANGLE} shape_tag;
    union {
        struct Circle {
            struct Point centre;
            float radius;
        } circle;
        struct Rectangle {
            struct Point corner1, corner2;
        } rectangle;
    } shape;
};

float area(struct Shape s) {
    if (s.shape_tag == CIRCLE) {
        float r = s.shape.circle.radius;
        return pi * r * r;
    } else { // rectangle
        float x1 = s.shape.rectangle.corner1.x;
        float y1 = s.shape.rectangle.corner1.y;
        float x2 = s.shape.rectangle.corner2.x;
        float y2 = s.shape.rectangle.corner2.y;
        return abs ((x2-x1) * (y2-y1));
    }
}

int main(void) {
    struct Shape s;
    s.shape_tag = CIRCLE;
    s.shape.circle = (struct Circle){.radius=1};
    printf("%f\n", area(s));
    s.shape_tag = RECTANGLE;
    s.shape.rectangle = (struct Rectangle) {{4, 5}, {6, 2}};
    printf("%f\n", area(s));

    return 0;
}
```

Algebraic data types

You now have the two main building blocks you need to create and understand user-defined types in Haskell. We use value constructors (like `Circle`) to group individual values together into a compound type with multiple fields, and use unions (indicated by the vertical bar `|`) to specify the different ways that a value of a particular type can be constructed. We call types that are created using combinations of constructors and unions **algebraic data types**.[13]

[13] The term "algebraic" here is suggestive of the fact that such types are built from a simple set of basic types by applying operations on them.

Polymorphism I: type variables and generic polymorphism

Let's now turn our attention to a more advanced feature of Haskell's type system: its support of polymorphism. The term **polymorphism** comes from the Greek words *poly*, meaning "many", and *morphe*, meaning "form" or "shape." In programming language theory, polymorphism refers to the ability of an entity (e.g., function or class) to have valid behaviour in contexts of different types. That's a pretty abstract definition—let's look at what this means in Haskell starting from a familiar concept: the list.

The list type, for real

As we have previously discussed, Haskell's type system imposes a restriction on programmers, that a list value must contain elements of the same type:

```
> :t [True, False, True]
[True, False, True] :: [Bool]
> :t [(&&) True, not]
[(&&) True, not] :: [Bool -> Bool]
```

At the same time, lists are quite generic: as long as we adhere to this restriction, we Haskell programmers can create lists of *any* type. But this raises the question: what is the type of functions that operate on lists, like `head`? We know that `head` takes as input a list and returns its first element, but the type of this element depends on the type of the list, and we just said this could be anything!

```
> :t (head [True, False, True])
(head [True, False, True]) :: Bool
> :t (head "abc")
(head "abc") :: Char
> :t head
head :: [a] -> a
```

The type of `head` is `[a] -> a`. This differs from all other type signatures we have seen so far because it contains a **type variable** `a`.[14] A type variable is an identifier in a type expression that can be instantiated to any type, such as `Bool` or `Char`. This type signature tells us that `head` works on any type of the form `[a]`, and returns a value of type `a`. Each time we call `head`, we match the parameter type `[a]` against the type of the argument, and use this to instantiate the type variable `a` to a concrete type to do the type-checking. For example, here is what happens when we type-check the expression `head [True, False, True]`:

[14] In type expressions, type variables must start with a lowercase letter, distinguishing them from types themselves.

1. Haskell determines that the type of the argument list is `[Bool]`.
2. Haskell matches the argument type `[Bool]` against the parameter type `[a]`, and instantiates `a = Bool` in the function type.
3. Haskell takes the return type `a`, with `a` instantiated to `Bool`, to recover the final concrete type `Bool` for the return type, and therefore the overall type of the function call.

Let's look at a more complicated example: recall the `filter` function:[15]

[15] Another instance of sectioning: `(>= 1)` is equivalent to `\x -> x >= 1`.

```
> filter (>= 1) [10, 0, -5, 3]
[10,3]
```

What is the type of `filter`? We can think of `filter` in the usual sense of taking two arguments (putting currying aside). The second argument could be a list of any type: `[a]`. The first argument must be a function mapping each `a` to a `Bool`, i.e., `a -> Bool`.[16] `filter` returns a list of elements that have the same type as the original list (since these elements were elements of the original list). Putting this together we get that the type of `filter` is

[16] Haskell generally avoids the boolean coercions of other languages; it doesn't have a notion of "truthy" or "falsey" values.

```
filter :: (a -> Bool) -> [a] -> [a]
```

These are some examples of built-in list functions that have type variables—but what about the list type itself? How does Haskell know to allow the programmer to create lists of different types? This is in the very definition of the list type itself. Before we get to the "real" definition, let's build some intuition by trying to define a list type ourselves. Here's a type that represents a list of integers. Such a list is either empty, or an integer "cons'd" with another list of integers.

```
data IntList = Empty | Cons Int IntList
```

What if we wanted a list of strings instead?

```haskell
data StrList = Empty | Cons String StrList
```

The only thing that differs in this definition is the type of the first argument to `Cons`, as this represents the type of the items actually being stored in the list. To generalize this definition, we do exactly the same thing as with functions: introduce a parameter to represent the varying part.

```haskell
data List a = Empty | Cons a (List a)
```

Here, `a` is a *type variable* (same terminology as above), and can be instantiated with different types (like `Int` or `String`) to create different types. Note how this exactly mimics the syntax for function definitions! We call the `List` itself a **type constructor**, as it is not exactly a type, but rather a function that takes a type `a` and creates a new type `List a`.[17]

It should come as no surprise that this is precisely how Haskell defines its built-in list type, albeit with terser names:[18]

```haskell
data [] a = [] | (:) a ([] a)
```

Now remember, these value constructors are just functions! So the type of the second constructor `(:)` shouldn't be a surprise, but the constructor for the empty list might be:

```
> :t (:)
(:) :: a -> [a] -> [a]
> :t []
[] :: [a]
```

The type of `[]` is `[a]`, which is what allows the empty list value `[]` to be mixed with lists of *any* type, without any additional overhead on the programmer's part.

Generic polymorphism (in Haskell and beyond)

In Haskell, lists are an example of **generic polymorphism**, a form of polymorphism in which an entity (e.g., function or class) behaves in the *same way* regardless of the type context. For example, Haskell lists are generically polymorphic (or just "generic" for short): they can be used to store elements of *any type*, and how these lists are constructed (`[]` and `(:)`) is the same regardless of what type of element is being stored. Similarly, almost every built-in list function is generic,

[17] So in this context, a type variable is a parameter of a type constructor.

[18] The trickiest part of this definition is that the symbol `[]` plays two different roles: in the type language, `[]` is a *type constructor* that takes a type `a` and returns a new type representing a "list of as"; in the expression language, `[]` represents an empty list, which is one of two list constructors. Note that `[] a` is the expanded form of the more familiar `[a]`.

meaning they operate on their input list regardless of what this input list contains. Put more poetically, generic functions operate on the "outer shape" of their input data (e.g., the fact that the data is a linear ordered sequence) without care to the "inner shape" of the data's contents.

In terms of types, there is for now a good approximation to tell whether a given function is generic: if its type signature contains a type variable, then it's generic, and if it contains no type variables, it is not generic—in the latter case, we say that the type is *concrete*. Please note that this heuristic is not quite complete, and we'll formalize this properly a bit later in this chapter.

As an aside, the concept of generic polymorphism is extremely useful, as it allows for great reductions in code repetition by writing one data structure or function once and having it work on all sorts of different types. If you've programmed in Java or C++, generics and type variables shouldn't be novel. For example, Java's standard `ArrayList` follows basically the same principle:[19]

[19] as does Java's entire Collections framework

```
class ArrayList<T> {
  ...
}

public static void main(String[] args) {
  ArrayList<Integer> ints = new ArrayList<Integer>();
}
```

In the class definition of `ArrayList`, `T` is a type variable, and is explicitly instantiated as `Integer` when `ints` is created. Of note is the creation of a new empty list; prior to Java 7, the `Integer` type annotation was required to make this code type-check, making it more verbose than the simple `[]` in Haskell, which is itself a generic value. Java 7 updated the language so that programmers would be able to omit the `Integer` from the right-hand side, naming the empty `<>` the *diamond operator*.

Templates in C++ play the same role, allowing type variables in both functions and classes. Here's an example of a function template, which also illustrates a more limited form of type inference found in C++.[20]

[20] This example is taken from `http://en.cppreference.com`.

```
#include <iostream>
template<typename Type>
void f(Type s) {
  std::cout << s << '\n';
}

int main() {
  f<double>(1);       // instantiates and calls f<double>(double)
  f<>('a');           // instantiates and calls f<char>(char)
  f(7);               // instantiates and calls f<int>(int)
  void (*ptr)(std::string) = f; // instantiates f<string>(string)
}
```

Polymorphism II: Type classes and ad hoc polymorphism

In the previous section, we looked at *generic polymorphism*, a useful tool in promoting abstraction by defining types and functions that would operate on not one but many data types. However, this form of polymorphism isn't the only useful one, and indeed isn't appropriate in a lot of circumstances. In this section, we'll look at another form of polymorphism supported by Haskell, starting with a simple example: how can we inspect `Shapes` in the Haskell interpreter?

Showing shapes

Recall our `Point` and `Shape` types from earlier in this chapter:

```
data Point = Point Float Float

data Shape
    = Circle Point Float     -- ^ centre and radius
    | Rectangle Point Point  -- ^ top-left and bottom-right corners
```

Unfortunately, right now we can't view any instances of our custom types directly in the interpreter, because GHCi doesn't know how to display them as strings. This is where type classes come in.

A **type class** is a set of types plus a set of functions that must be able to operate on these types, similar to the notion of an abstract interface in object-oriented programming. We say that a type is a **member** of a type class if the type class' functions have been implemented for that type. Haskell has the built-in type class `Show`, containing the function `show`, which is roughly defined as follows:[21]

[21] This is a simplification of the full type class definition, but it suffices for our purposes.

```
class Show a where
    show :: a -> String
```

This type class definition contains a type variable `a`, with a slightly different meaning than before. Here, the `a` should be interpreted as "a type `a` belongs to `Show` when there is an implementation of a function `show` that takes a value of type `a` and returns a string."

Most built-in Haskell types like `Int` and `Bool` are members of the `Show` type class. When GHCi is asked to evaluate an expression, it tries to call `show` on the resulting value to display the result back to the user, and this only works if the expression's type is a member of the `Show` type class.

To make our class `Point` a member of `Show`, we need to implement the `show` function for our type, wrapped in a special syntactic form using the `instance` keyword:

```
instance Show Point where
  show (Point x y) = "(" ++ (show x) ++ ", " ++ (show y) ++ ")"

> show (Point 1.0 3.0)
"(1.0, 3.0)"
```

Finally, a question: what exactly is `show`? At first this seems silly, as we've both referred to it and used it as a function. Yet this is a function whose initial declaration and type signature is decoupled from its implementation, and in fact has multiple implementations scattered throughout the Haskell standard library and in our own `Point` code, which is unlike any function we've seen before.

Well, the answer isn't too complicated, but is worth digging into just a little bit. `show` is a single function with multiple implementations, one for each type that is a member of `Show`. Its type signature is a little different:

```
> :t show
show :: Show a => a -> String
```

This type signature also has a type variable, but this type variable is modified by a new piece of type expression syntax, the `Show a =>`. We call the `Show a` before the `=>` a **type class constraint**: its meaning is that the type variable `a` can only be instantiated to a member of the `Show` type class.[22] We say that the type variable `a` is a *constrained* type variable, because of the presence of this constraint.

[22] A function type can have multiple type constraints, e.g. `Eq a, Ord b => ...`.

So now our polymorphism detection heuristic is a bit more complicated. If a function's type signature is concrete (no type variables), it is not polymorphic. If a function's type signature contains an unconstrained type variable, it is generically-polymorphic (in that variable): the function *must* behave the same way for any type instantiation of that variable. But what does it mean when we see a constrained type variable in a function type signature?

Ad hoc polymorphism

In contrast to generic polymorphism, **ad hoc polymorphism** is the ability of an entity to have different behaviours depending on the type context. In Haskell, this is precisely what type classes give us. By making types an instance of a type class, we give different implementations of the same function for each type separately.[23] Then, the presence of type class constraints in a function definition signals the ad hoc polymorphic nature of that function: when we see `Show a =>`, we expect different behaviours for the function for different instantiations of `a`.

[23] The term "ad hoc" in the name reflects this: each time we write an implementation for one type, we are free to make the implementation as tailored as possible to that type, without worrying about making it generic.

A different approach to ad hoc polymorphism is implemented in Java through *method overloading*, which allows the programmer to implement the same method in different ways, giving different parameter signatures for each one.[24]

[24] Note: this type of overloading is *not* permitted in Haskell, as it limits the kinds of type-checking that can be performed.

```java
public int f(int n) {
    return n + 1;
}

public void f(double n) {
    System.out.println(n);
}

public String f(int n, int m) {
    return "Yup, two parameters.";
}
```

One particular limitation of that technique is that the method's type does not contain a type class constraint; the only way to determine if a Java method is ad hoc polymorphic is by scanning all of the function signatures in the source code (or relying on an IDE that does so). This plays back to this chapter's recurring theme of using types to encode information about a value as being a fundamental purpose of a type system.

Some built-in type classes

In this part, we'll briefly describe some type classes you'll likely encounter in your study of Haskell. First, here are two type classes for comparing values:

- `Eq`: members of this type class support the `(==)` and `(/=)` functions to test for equality. Only `(==)` needs to be implemented; `(/=)` is given a *default implementation* in terms of `(==)`.
- `Ord`: members can be ordered using `(<)`, `(<=)`, `(>)`, and `(>=)`. Members must also be members of the `Eq` type class; we say that `Ord` is a **subclass** of `Eq`. Only `(<)` (and `(==)` from `Eq`) need to be implemented.

And now that we know about type classes, we can *finally* understand Haskell's numeric types. Let's start with addition.

```
> :t (+)
(+) :: Num a => a -> a -> a
```

You might have come across the `Num a` class constraint before and had this explained to you as saying that `a` is a numeric type, and this is basically correct. More precisely, `Num` is a type class that supports numeric behaviours, and which

concrete types like `Integer` and `Float` belong to. There are three main numeric type classes in Haskell:

- `Num` provides basic arithmetic operations such as `(+)`, `(-)`, and `(*)`. Notably, it doesn't provide a division function, as this is handled differently by its subclasses. It also provides the function `fromInteger`, which we'll discuss a bit further below.
- `Integral` represents integers, and provides the `div` (integer division) and `mod` functions.

```
> div 15 2
7
> :t div
div :: Integral a => a -> a -> a
```

- `Fractional` represents non-integral numbers, and provides the standard division operator `(/)`, among others.

So that's it for the numeric functions. It turns out that numeric literals are also impacted by type classes, which is a pretty idiosyncratic characteristic of Haskell. Let's inspect the type of the literal `1`:

```
> :t 1
1 :: Num a => a
```

Huh, it's not `Integer`? That's right, `1` (and all other integer literals) is a *polymorphic* value, like `[]`: `1` can take on any numeric type, depending on its context. This is the purpose of the `fromInteger :: Num a => Integer -> a` function in the `Num` type class, which is called implicitly to convert integer literals into whatever numeric type is necessary for the surrounding context.[25]

[25] The `Fractional` type class has an analogous function, `fromRational`, that handles the conversion from fractional literals to fractional types.

By the way, this should explain why the expression `1 + 2.3` correctly returns `3.3` in Haskell, even though we previously said Haskell does not have any type coercion. When Haskell attempts to infer the type of this expression, it doesn't automatically assume `1` is an `Integer` nor `2.3` a `Float`. Instead, it sees that the type of `1` must be a member of the `Num` type class, and the type of `2.3` must be a member of the `Fractional` type class, and that these two types must be equal because `(+)` takes two arguments of the same type. The Haskell compiler can correctly instantiate the types to satisfy these constraints, and so the type check passes.

Higher-order type classes

Let's now look at one impressive feature of Haskell's type class implementation, which goes beyond what you see in most other languages. Consider the following problem: in addition to lists, Haskell has many other "container" types,

including sets and vectors. One of the particularly useful higher-order transformations we have seen before is `map`, which is used to apply some function to each element in a list, producing a new list as a result:

```
map :: (a -> b) -> [a] -> [b]
```

While this works for lists, we want such a function for sets and vectors as well:

```
setMap :: (a -> b) -> Set a -> Set b
vectorMap :: (a -> b) -> Vector a -> Vector b
```

In Racket, this is achieved by simply implementing different "mapping" functions for each data type, avoiding name collisions by prefixing by the data type name, just as we illustrated above. But this approach has the limitation that we cannot write elegant code that is polymorphic over a container type. For example, consider defining a function that "maps" an "add 1" function over a collection of elements, that should work regardless of the container type. In Racket, we would do something like the following:

```
(define (add1 x) (+ x 1))

(define (add-1-all items)
  (cond
    [(list? items) (map add1 items)]
    [(set? items) (set-map add1 items)]
    [(vector? items) (vector-map add1 items)]))
```

As you are probably thinking, the solution in Haskell seems pretty straightforward: why don't we make `map` part of a type class, so that we can implement it separately for each container type? And this is indeed what Haskell does, but there is one subtlety: since this type class would represent "container" types, its members are *type constructors*, not types themselves. Let's take a look at the definition of the built-in "mappable" type class, which Haskell calls `Functor`:[26]

```
class Functor f where
    -- the "f" in fmap stands for "Functor"
    fmap :: (a -> b) -> f a -> f b
```

[26] Many of the higher-order type classes have names derived from an abstract branch of mathematics called *category theory*. This isn't relevant for this course, but you might be interested in exploring the deep relationships between type classes and category theory on your own time.

The type variable `f` represents the member of `Functor`, but we can tell from the type signature of `fmap` that `f` is *not* a primitive type like `Int` or `Bool`, but instead a type constructor like the list (`[]`), `Set`, or `Vector`. And because Haskell makes these different data types instances of `Functor`, we can implement a "map plus 1" function to work on arbitrary containers:

```haskell
add1All :: Functor f => f Int -> f Int
add1All items = fmap (+1) items

-- Or, using currying:
add1All = fmap (+1)
```

Representing failing computations

Here's an interesting type that really illustrates the purpose of having multiple data constructors. Suppose you have an initial value, and want to perform a series of computations on this value, but each computation has a chance of failing. We would probably handle this in other programming languages by using exceptions or by checking each step to see if the returned value indicates an error. Both of these strategies often introduce extra *runtime* risk into our program, because the compiler cannot always determine whether (or what kind of) an exception will be raised,[27] or distinguish between valid and erroneous return values, leaving it to the programmer to remember to make the check.

[27] Though this is mitigated in Java with checked exceptions.

Lest we get too high on our Haskell horse, be reminded that we have seen unsafe functions in Haskell as well: `head`, `tail`, and any incomplete pattern matches, for example. In this section, we will see how to incorporate this possibility of error in a statically-checkable way, using Haskell's type system. That is, we can indicate in our programs where potentially failing computations might occur, and make this known to the compiler by defining a data type representing the result of a computation that may or may not have been successful.

We start by introducing the fundamental type constructor, `Maybe`, used to represent such values.[28]

[28] Java 8 introduced the analogous `Optional` type.

```haskell
data Maybe a = Nothing | Just a
```

Intuitively, the type `Maybe a` represents values that could have two possible states: either they're a `Nothing`, because they were produced by some computation that failed, or they're an actual `a` value, because the computation succeeded. Many programming languages encode this idea implicitly through a special value like `None` or `null` or `nil`; Haskell goes a step further, using not just an analogous value `Nothing`, but also encoding this in the very type of the expression: `Maybe Int` is *not* the same as `Int`, and the compiler can tell the difference!

Here are some simple functions that illustrate this data type; note that the "wrapped" type `a` can be anything, as `Maybe` is a generic type.

```haskell
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (div x y)

safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:_) = Just x

safeTail :: [a] -> Maybe [a]
safeTail [] = Nothing
safeTail (_:xs) = Just xs

assertNonNegative :: Int -> Maybe Int
assertNonNegative n =
    if n >= 0
    then
        Just n
    else
        Nothing
```

The last example is a bit odd, since we aren't "doing" anything to the input, but instead checking some property. The generalized version is perhaps more suggestive:

```haskell
assert :: (a -> Bool) -> a -> Maybe a
assert pred x =
    if pred x
    then
      Just x
    else
      Nothing
```

However, in order to appreciate how to use this `assert` function in an interesting way, we need to expand our scope beyond just individual functions that return `Maybe`s, to thinking about how to use such functions in a larger expression.

Lifting pure functions

Recall that we already have an operator, `(.)`, which composes regular functions:

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Let's start with a simple example: defining a function that takes the first element of a list of numbers, and adds `1` to that number. If we weren't worried about runtime errors, we could define this function as follows:[29]

```
add1ToHead :: [Int] -> Int
add1ToHead = (+1) . head
```

[29] We deliberately write this in a terser style to emphasize the *composition* of the main functions involved. Make sure you understand this definition before moving on!

It is tempting to make this function "safe" by swapping in our `safeHead` from above, and returning a `Maybe`:

```
safeAdd1ToHead :: [Int] -> Maybe Int
safeAdd1ToHead = (+1) . safeHead
```

Unfortunately, this implementation doesn't type-check. The problem is that `(+1)` expects a "pure" numeric type (like `Int`), but we're giving it a `Maybe Int` instead. Now, we could define a new function to handle this explicitly using pattern-matching:

```
add1Maybe :: Maybe Int -> Maybe Int
add1Maybe Nothing = Nothing
add1Maybe (Just n) = Just (n + 1)
```

But of course this approach doesn't scale; we don't want to have to define "`Maybe` versions" of every function we want to use. Instead we define a *higher-order* function that will take any pure `a -> b` function and turn it into one that works on `Maybe a` instead. We call this operation `lift`, taking the term from mathematics, usually denoting the transformation of some object to move it into a more abstract or general context.

```
lift :: (a -> b) -> (Maybe a -> Maybe b)
lift _ Nothing = Nothing
lift f (Just x) = Just (f x)
```

With this function, we can complete a correct implementation of `safeAdd1ToHead`:

```
safeAdd1ToHead :: [Int] -> Maybe Int
safeAdd1ToHead = (lift (+1)) . safeHead
```

But hang on: doesn't the type signature for `lift` look a little familiar? Keep in mind that `->` is right-associative in type expressions, and so that second pair aren't necessary:

```haskell
lift :: (a -> b) -> Maybe a -> Maybe b
```

You guessed it! This expression exactly matches the one for `fmap`, and this isn't a coincidence: `Maybe` is indeed a member of the `Functor` type class.[30]

[30] Indeed, we can think about `Maybe` as a "container" that holds exactly zero or one value of its contained type.

```haskell
safeAdd1ToHead :: [Int] -> Maybe Int
safeAdd1ToHead = (fmap (+1)) . safeHead
```

Composing failing computations

Now let's consider defining a function that returns the second element of a list:

```haskell
second :: [a] -> a
second = head . tail
```

Our first attempt at making this function safe is to replace the list functions with the safe variants we defined above:

```haskell
safeSecond :: [a] -> Maybe a
safeSecond = safeHead . safeTail
```

It looks like we have a similar problem as before: `safeTail` returns a `Maybe [a]`, while `safeHead` expects just a `[a]`. What happens if we try using `fmap` (or `lift`)?

```haskell
safeSecond :: [a] -> Maybe a
safeSecond = (fmap safeHead) . safeTail
```

Unfortunately, this doesn't quite work: the return type is a `Maybe (Maybe a)`, rather than just a `Maybe a`! This isn't what we want, so we're going to take a different (presumably also higher-order) approach. But what? Let's start by writing a correct, if cumbersome, implementation of `safeSecond`.[31]

[31] This also introduces `case` expressions, which enable pattern-matching in arbitrary expression contexts.

```haskell
safeSecond :: [a] -> Maybe a
safeSecond xs =
    let xs' = safeTail xs
    in
        case xs' of
        Nothing -> Nothing
        Just xs'' -> safeHead xs''
```

To see what's going on, let's inspect the types:

- `xs` has type `[a]`, which is the original input to the function
- `xs'` has type `Maybe [a]`, which is what's returned by `safeTail`
- `xs''` has type `[a]`: it is an "unwrapped" version of `xs'`, which is safe to do inside the `case` expression because we provide pattern-match rules for both possibilities for `xs'`.

Unfortunately, using the `case` expression does not generalize very elegantly, as we compose more and more computations. Suppose we wanted to access the fourth element of a list:

```
safeFourth :: [a] -> Maybe a
safeFourth xs =
    let xs' = safeTail xs
    in
        case xs' of
        Nothing -> Nothing
        Just xs1 ->
            let xs1' = safeTail xs1
            in
                case xs1' of
                Nothing -> Nothing
                Just xs2 ->
                    let xs2' = safeTail xs2
                    case xs2' of
                    Nothing -> Nothing
                    Just xs3 -> safeHead xs3
```

To remove the code duplication, we move the pattern-matching to a helper:[32]

[32] The name `andThen` should suggest the idea of sequencing multiple computations.

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen Nothing _ = Nothing
andThen (Just x) f = f x
```

Like `lift`, the implementation of `andThen` is very simple. The only difference is that we do *not* wrap the second line's body in a `Just` (logically, we shouldn't assume that calling `f` will succeed!). Let's use this helper to see how to simplify `safeSecond`:

```
safeSecond :: [a] -> Maybe a
safeSecond xs =
    let xs' = safeTail xs
    in
        andThen xs' safeHead
```

However, this isn't that readable. The reason we chose the name `andThen` only becomes apparent when we use backticks to call `andThen` from an infix position:

```
safeSecond :: [a] -> Maybe a
safeSecond xs =
    let xs' = safeTail xs
    in
        xs' `andThen` safeHead

-- Or simply,
safeSecond xs = safeTail xs `andThen` safeHead
```

With this use of `andThen`, we can entirely remove all of the boilerplate from `safeFourth` as well, resulting in a truly beautiful implementation.

```
safeFourth :: [a] -> Maybe a
safeFourth xs =
    safeTail xs `andThen`
    safeTail `andThen`
    safeTail `andThen`
    safeHead
```

Error reporting

One limitation of using `Maybe` is that chained computations don't preserve information about where or why the computation failed. A `Nothing` conveys a failure, but nothing else. In practice, the `Either` type constructor is often used instead:

```
data Either a b = Left a | Right b
```

`Either` is a versatile type constructor that can be used to represent an alternative between two different types (e.g., "this function returns either an `Int` or a `String`"), but one of its most common uses is to use the `Left` constructor to represent a failure with some error information, and a `Right` to represent a success, analogous to `Just`. For example, a bare-bones approach might store an error message as a string:

```
safeDiv2 :: Int -> Int -> Either String Int
safeDiv2 _ 0 = Left "Division by 0 error"
safeDiv2 x y = Right (x `div` y)
```

More sophisticated systems might use a custom data type to store more elaborate error information as the "left" value.

Modeling mutation in pure functional programming

One of the central tenets of functional programming we've had in the course so far was avoiding the concept of *state*; that is, writing code without mutation. Though we hope you have seen the simplicity gained by not having to keep track of changing variables, it is often the case that mutable state is the most natural way to model problem domains. So the question we'll try to answer in this section is, "How can we simulate changing state in Haskell, which does not even allow re-binding of identifiers?"

Postorder tree labeling

Consider the following problem: given a binary tree, label each node with its position in a postorder traversal of that tree. Here is some code in Python to solve this problem, using a *global mutable variable* to keep track of the "current" position in the traversal.

```
i = 0

def post_order_label(tree):
    global i

    if tree.is_empty():
        return
    else:
        post_order_label(tree.left)
        post_order_label(tree.right)
        tree.root.label = i
        i += 1
```

This code is very elegant, but the reason for its elegance is the use of this global variable `i` that is mutated through the recursive calls to `post_order_label`. Our goal for this section is to implement this same algorithm in Haskell.

Stateful computations

Believe it or not, we have already seen the basic "mutating state" technique in this course. Recall the basic loop iteration pattern:

```
acc = init
for x in lst:
    acc = update(x, acc)
```

We achieved the same affect using `foldl`, and the key idea was that the update function passed to `foldl` was interpreted to "mutate" the accumulator. Concretely, the function took as input the *old state* of the accumulator, and returned its *new state*. By chaining together such functions—each taking the result of the previous—we can simulate a mutating "accumulated state" over many different computations.

For any kind of mutable state, the two primitive operations are *read* and *write*, which here we'll label `get` and `put`. We could implement each one as follows:

```
get :: s -> s
get state = state

put :: s -> s -> s
put item state = item
```

This looks a bit strange, so let's unpack our intentions. - `get` takes in the current state and simply returns it; this simulates "reading" from the state. - `put` takes an item and the current state, and returns the item, representing the *new state* (note that the old state is completely discarded). Right now, it is impossible to tell from the return types of these two functions that one of them returns a lookup value (but leaves the state unchanged), and the other actually modifies the stored state. To resolve this issue, we unify their return types by using a tuple, where the first element of the tuple represents what value (if any) is "produced" by the computation, and the second element represents the new state.

```
get :: s -> (s, s)
get state = (state, state)

put :: s -> s -> ((), s)
put item state = ((), item)
```

Now let's interpret this new code: - `get` takes the old state and returns its value as both elements of the tuple. The *first* element represents what's being produced (the value is looked up), while the second represents the new state, which is actually exactly the same as the old state. - `put` updates the state, which is

why the second element in the returned tuple is `item`—the old state has been replaced. The `()` in the first position is called **unit** in Haskell,[33] and is used to denote the fact that the `put` doesn't produce a value. This is analogous to a `void` return type in Java or C.

[33] Somewhat confusingly, `()` is both a value and a type, both called "unit".

That's better, but we'll go one step further and use Haskell's built-in `State` data type, which essentially wraps around the "updating state" function type:

```
data State s a = State (s -> (a, s))

get :: State s s
get = State (\state -> (state, state))

put :: s -> State s ()
put item = State (\state -> ((), item))
```

Warning: `State` is actually a pretty confusing name. In Haskell, this `State` type constructor doesn't refer to the state itself, but rather to a *stateful computation* that takes in and returns state, as we described above. This is a pretty unfortunate historical name, so please pay attention to the name when we use it through this section!

So using this language, we say that `get` is a stateful computation that produces what was stored in the state and leaves the state unchanged, while `put` is a stateful computation that takes an item, and replaces the existing state with that item, but doesn't produce a value.

Running and combining stateful computations

One of the side effects of using the `State` type to represent stateful computations is that the underlying function is now wrapped, and can't be called directly. For example, calling `get 5` produces an error rather than the expected `(5, 5)`. To resolve this, Haskell has a built-in function `runState` whose only purpose is to return the wrapped function:

```
runState :: State s a -> (s -> (a, s))
runState (State op) = op
```

But in the above code, `op` is a function; in order to actually "execute" the stateful computation, we need to pass in a state value, which we think of as representing the initial state.

```
> runState get 5
(5,5)
```

With this in mind, we can chain together stateful computations as well. As an example, let's translate the following Java code:

```
public String f() {
    int x;
    x = 10;
    x = x * 2;
    return str(x);
}
```

The idea is pretty simple once we get the hang of it: except for the variable declaration and initialization, every other line consists of a `get`, `put` or pure computation (like `(*2)`). Our approach is to perform each of these computations in turn, using `let` to bind the intermediate states so that each `State` computation gets as input the state from the previous one.[34]

[34] To make the code a little nicer, we use pattern-matching on the left-hand side of the `let` bindings.

```
f :: State Int String
f = State (\state0 ->
    let (_, state1) = runState (put 10) state0
        (x, state2) = runState get state1
        (_, state3) = runState (put (x * 2)) state2
        (x', state4) = runState get state3
    in
        (show x', state4)
    )
```

This code is certainly easy to understand, and its size may not even suffer (much) in comparison to other languages. But hopefully it seems at least somewhat inelegant to have to manually keep track of this state. This is analogous to the clumsy use of pattern-matching to distinguish between `Nothing` and `Just` values in the previous section. So, as before we'll define a higher-order function to abstract away the composition of two `State` operations. For reasons that will become clear shortly, the main function we'll implement is `andThen`. Here's a preliminary (*not final*) version:

```
-- Perform two state computations and return the second one's value.
andThen :: State s a -> State s b -> State s b
andThen op1 op2 = State (\state0 ->
    let (x1, state1) = runState op1 state0
        (x2, state2) = runState op2 state1
    in
        (x2, state2)
    )
```

This is a good start, but it is not quite general enough for our purposes. The problem is that the "produced" value `x1` of the first computation is unused by the second computation. But in general, when we sequence two stateful computations, we want the second to be able to use the result of the first. So what we need to do is define a function that will perform two `State` operations in sequence, but have the second depend on the result of the first one. On the face of it, this sounds quite strange—but remember that closures give languages a way of dynamically creating new functions with behaviours depending on some other values.

So rather than take two fixed `State` operations, `andThen` will take a first `State` operation, and then *a function that takes the result of the first `State` operation and returns a new `State` operation.* Think about this second parameter as an "incomplete" `State` function, which gets completed by taking the value produced by the first function. This might seem pretty abstract, but should become clearer when we look at the new type signature and implementation:

```
andThen :: State s a -> (a -> State s b) -> State s b
andThen op1 opMaker = State (\state0 ->
    let (x1, state1) = runState op1 state0
        op2 = opMaker x1
        (x2, state2) = runState op2 state1
    in
        (x2, state2)
    )
```

That is, `andThen` does the following:

1. Run its first `State` computation (`op1`).
2. Use the produced result (`x1`) and its second argument (`opMaker`) to create a second `State` computation.
3. Run the newly-created `State` computation, and return the result (and updated state).

With this function in hand, we can now greatly simplify our example function:

```
f :: State Int String
f = put 10 `andThen` (\_ ->
        get `andThen` (\x ->
            put (x + 2) `andThen` (\_ ->
                get `andThen` (\x' ->
                    State (\state -> (show x', state))))))
```

Well, almost. We have used indentation and parentheses to delimit the bodies of each lambda expression, neither of which are technically necessary. Removing them and veritcally-aligning the operations cleans up the code dramatically:

```haskell
f :: State Int String
f = put 10 `andThen` \_ ->
    get `andThen` \x ->
    put (x + 2) `andThen` \_ ->
    get `andThen` \x' ->
    State (\state -> (show x', state))
```

Finally, we note that the final `State` value is special, since it simply "produces" a value (`show x'`) without *any* interaction with the underlying state at all. We'll call this type of computation a "pure" one, and use a helper function to simplify the code further:

```haskell
pureValue :: a -> State s a
pureValue item = State (\state -> (item, state))

f :: State Int String
f = put 10 `andThen` \_ ->
    get `andThen` \x ->
    put (x + 2) `andThen` \_ ->
    get `andThen` \x' ->
    pureValue (show x')
```

Back to post-order labeling

It turns out that we have everything we need to implement a post-order binary tree labeling in Haskell. First, recall the mutating Python version:

```python
i = 0

def post_order_label(tree):
    global i

    if tree.is_empty():
        return
    else:
        post_order_label(tree.left)
        post_order_label(tree.right)
        tree.root.label = i
        i += 1
```

Here is the corresponding data type we'll use to represent binary trees in Haskell.[35]

[35] Note the similarity to our recursive list definition!

```haskell
data BTree a = Empty
             | Node a (BTree a) (BTree a)
```

To keep things immutable, we won't mutate a `BTree` directly, but rather return a new `BTree` with all nodes labelled properly. To accomplish this using the same algorithm as shown in the Python code, we'll need one `Int` of mutable state, and so our type signature will be:[36]

[36] In the returned tree, each node stores its original value paired with its post-order label.

```haskell
postOrderLabel :: BTree a -> State Int (BTree (a, Int))
```

The base case is pretty easy to implement, and can be done without any statefulness at all:

```haskell
postOrderLabel Empty = pureValue Empty
```

As a first pass for the recursive step, let's just try translating the two recursive calls in Python into the equivalent chained Haskell code, using our `andThen`.[37]

[37] Note that we ignore the original node label, since the goal here is to set the label ourselves.

```haskell
postOrderLabel (Node item left right) =
    postOrderLabel left `andThen` \newLeft ->
    postOrderLabel right
```

Not too bad, although these two recursive calls aren't really doing anything (other than spawning other useless calls). Let's first increment our "`i`":

```haskell
postOrderLabel (Node item left right) =
    postOrderLabel left `andThen` \newLeft ->
    postOrderLabel right `andThen` \newRight ->
    get `andThen` \i ->
    put (i + 1)
```

That version almost compiles, but unfortunately ends with a `State` value of the wrong type: `State Int ()` instead of `State Int (BTree (a, Int))`. To fix this, let's actually consider what we want to return: a new `BTree` with all the nodes labeled correctly. Through the power of recursion, we can assume that both `newLeft` and `newRight` (produced by the recursive calls) are indeed correctly labeled. So then to put everything together, we need to create and return a new root node, containing the original item, an updated label, and new subtrees:

```haskell
postOrderLabel (Node item left right) =
    postOrderLabel left `andThen` \newLeft ->
    postOrderLabel right `andThen` \newRight ->
    get `andThen` \i ->
    put (i + 1) `andThen` \_ ->
    pureValue (Node (item, i) newLeft newRight)
```

And that's it! Not too shabby indeed—and quite close to what we'd write in Python if we had to return a new tree rather than mutate the original.

Impure I/O in a pure functional world

In addition to mutating state, another common type of impure function is one that interacts with some entity external to the program:[38] standard input or output, a graphical display, the filesystem, or a server thousands of kilometres away. Even though we've been programming without explicit mention of I/O since the beginning of the term, it's always been present: any REPL requires functions that take in user input, and that actually display the output to the user as well. If Haskell had no I/O capabilities whatsoever, we would never know if a program actually worked or not, since we would have no way to observe its output!

[38] Functions receiving external input are impure because their behaviour changes each time they are run; functions writing to some output are impure because they have a *side-effect*.

In this section, we'll briefly study how Haskell programs can interact directly with the standard input and standard output provided by the operating system. While the primitives we'll use in this section might look underwhelming, all of the other I/O operations we mentioned above behave exactly the same. In fact, even more is true: it turns out that all I/O follows our approach to mutable state from the previous section. Even though both I/O and mutation are side-effectful behaviours, they seem quite different at first glance, and so this might come as a surprise. Let's take a closer look at what's going on.

Our `State s a` values represented computations with two behaviours: a "pure" part, the computation performed to produce a value, and an impure part, a "mutation" operation, representing an interaction with the *underlying context of the computation*. We can think of pure functions as those that have no context (e.g., external mutable state) they need to reference, and impure functions as those that do.

Even though our post-order labeling example used a `State Int` representing just one integer's worth of mutable state, the beauty of the `State` type constructor is that it can be parameterized on the type of state `s` being used, and so in more complex programs we might have `State String` or `State [Int]` or even `State (Map String Expr)`.[39] But even this view is too narrow: there is no need to limit our computational context to the program's heap-allocated memory. By widening the scope of this context to user interactions, files, and faraway servers, we can model any type of I/O computation at all.

[39] Think back to our previous discussions of the *environment*...

Standard I/O primitives

Recall that we built complex `State` computations using two primitives, `get` and `put`. With standard I/O there are two built-in values that we're going to cover: `putStrLn`, which prints a newline-terminated string to standard output, and `getLine`, which reads a string from standard input.[40] A natural question to ask when in an unfamiliar environment is about the types of the values. Let's try it:

[40] Look up more I/O functions in the `System.IO` module.

```
> :t getLine
getLine :: IO String
> :t putStrLn
putStrLn :: String -> IO ()
```

Here, `IO` plays an analogous role to `State s`, except rather than indicating that the function changes the underlying state context, it indicates that the function has some sort of interaction with an external entity. `IO` is a unary type constructor, where the type `IO a` represents an I/O action that produces a value of type `a`.[41] So `getLine` has type `IO String` because it is an I/O action that produces a string, and `putStrLn` has type `String -> IO ()` because it takes a string, prints it out, but produces nothing.

[41] We use "produce" in the same sense as the previous section: it's the value that is obtained from performing the computation.

We can now define a function which prints something to the screen:[42]

[42] A Hello World program. Finally.

```
greet :: IO ()
greet = putStrLn "Hello, world!"

> greet
Hello, world!
```

Now suppose we want to print out two strings. In almost every other programming language, we would simply call a printing function twice, in sequence. But in our pure functional programming setting, we work with expressions and not statements, and so can't simply write one expression after the other.[43]

[43] This would be interpreted as a *function call* in Haskell!

What we want is to chain together two I/O actions, a way to say "do this computation, *and then* do that one." This hopefully rings a bell from the previous section, when we chained together two `State` computations using two different versions of `andThen`. And this is where things get interesting: we actually have both versions as infix operators for I/O:

```
-- andThen, but discard the result produced by the first
-- By convention, called "then"
(>>) :: IO a -> IO b -> IO b
-- andThen, using the first result to create the second
-- By convention, called "bind"
(>>=) :: IO a -> (a -> IO b) -> IO b
```

It shouldn't be a big surprise that we can use the former to sequence two or more I/O actions:

```
greet2 :: IO ()
greet2 =
    putStrLn "Hello, world!" >>
    putStrLn "Second line time!"

prompt :: IO ()
prompt =
  putStrLn "What's your name?" >>
  getLine >>
  putStrLn "Nice to meet you, ____!"
```

How do we fill in the blank? We need to take the string produced by the `getLine` and use it to define the next I/O action. Again, this should be ringing lots of bells, because this is exactly what we did with our final version of `andThen`, corresponding to the `I/O` operator `(>>=)`:

```
prompt2 :: IO ()
prompt2 =
  putStrLn "What's your name?" >>
  getLine >>= \name ->
  putStrLn "Nice to meet you, " ++ name ++ "!"

> prompt2
What's your name?
David   -- This line is user input
Nice to meet you, David!
```

Standalone Haskell programs

Though we've mainly used GHCi, Haskell programs can also be compiled and *run*.[44] Such programs require a `main` function with type `IO ()`, which is executed when the program is run. The type of `main` is very suggestive: the program as a whole is an `IO` action that can interact with the outside world, using `main` as its entry point. In this course we've been mainly using the REPL as our entry point, but in practice any kind of I/O behaviour would start under `main`.

[44] Say, by using the command `runhaskell myfile.hs`.

Types as constraints

You have now seen two examples of impure values in Haskell: the state-"mutating" `State` computations, and the external-world-interacting `IO`. Despite our prior

insistence that functional programming is truly different, it seems possible to simulate imperative programs using these functions and others like them. This is technically correct, but it misses a very important point: unlike other languages, Haskell's type system enforces a strict separation between pure and impure code.

The *separation of concerns* principle is one of the most well-known in computer science, and you've probably studied this extensively in the context of designing classes, libraries, and larger application architectures like MVC. However, in most languages, is easy for impure and pure code to be tightly wound together, because side-effecting behaviour like mutation and I/O is taught to us from our very beginnings as programmers. But as we've discussed earlier, this coupling makes programs difficult to reason about, and often in our refactoring we expend great effort understanding and simplifying such side-effects.

In stark contrast, such mixing is explicitly forbidden by Haskell's type system. Think back to `IO`. We only used functions that combine values of `IO` types (like `(>>=)`), or functions that *returned* a value of these types. But once the `IO` was introduced, there was no way to get rid of it: there is no function `IO a -> a`.[45] This means that *any Haskell function that uses standard I/O must have a `IO` in its type.*

[45] This isn't strictly true, but the functions that do exist are used rarely and with great care in Haskell production.

It might feel restrictive to say that every time you want to print something in a function, you must change the fundamental type (and hence structure) of that function. However, as a corollary of this, any function that doesn't have `IO` in its type is guaranteed not to interact with the terminal, and this guarantee is provided *at compile time*, just through the use of types! This is perhaps the best illustration of this chapter's central principle: a type signature in Haskell tells us not just how to use a function and what it returns, but what it *cannot do* as well.

One last abstraction: monads

When we first started our study of types and Haskell, you might have thought we would focus on the *syntax* of creating new types in Haskell. But even though we did introduce quite a few new keywords for Haskell's type system, our ambitions were far grander than recreating a classical object-oriented system in Haskell.[46] Rather than focusing on types representing concrete models or actors in our programs, we have focused on using types to encode generic modes of computation: failing computations, stateful computations, and I/O computations.

[46] We (basically) already did this in Racket.

But defining type constructors like `Maybe`, `State`, and `IO` to represent elements of these computations was only a small part of what we did. After defining `Maybe` and `State`, we could have performed any computation we wanted to with these types simply by using `let` bindings, and called it a day. The novel part was really our use of higher-order functions to abstract away even how we computed on these types, allowing us to elegantly compose primitive values of these types to express complex computations. And this, of course, is the true spirit of

functional programming.

But there is one additional layer of abstraction that we've only hinted at by our choices of names for our operators. Though the contexts we studied were quite different, we built the same operators to work for each one. To make this explicit, let's look at the type signatures for the one function common to all three:

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen :: State s a -> (a -> State s b) -> State s b
(>>=) :: IO a -> (a -> IO b) -> IO b
```

And though the implementations differed for each context,[47] their spirit of composing context-specific computations did not. As always when we see similarities in definitions and behaviour, we look to precisely identify the commonalities, which presents an opportunity for abstraction.

[47] We don't even know how `(>>=)` is implemented for `IO`!

In fact, this is exactly what happens in Haskell: the similarity between `Maybe`, `State`, and `IO` here is abstracted into a single type class, called `Monad`. We misled you earlier in our discussion of I/O: if you query the type of either `(>>)` or `(>>=)`, you'll see a type class constraint that generalizes `IO`:

```
> :t (>>)
(>>) :: Monad m => m a -> m b -> m b
> :t (>>=)
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

It is no coincidence that these functions were the ones we have explored in great detail; they are two of the three functions which are part of the `Monad` definition:[48]

[48] You'll find that the other function is also very familiar, especially if you've been keeping up with the exercises.

```
class Monad m where
  return :: a -> m a
  (>>) :: Monad m => m a -> m b -> m b
  (>>=) :: Monad m => m a -> (a -> m b) -> m b
```

You should have some intuition for these functions already based on how we've used them in the various contexts, but here is a brief description of each one:

- `return`: take some pure value, and "wrap" it in the monadic context.
- `(>>)`: chain together two monadic values; the result *cannot* depend on the result of the first one.
- `(>>=)`: chain together two monadic values; the result *can* depend on the result of the first one.

Of course, generic terms like "chain" and "contents" change meaning depending

on the context, and this is reflected when we actually make instances of `Monad`. For example, we can describe these functions in the `IO` context as follows:

- `return`: create an I/O action that does nothing except produces the given value.
- `(>>)`: combine two I/O actions into a new I/O action that performs the two in sequence, producing the result of the second.
- `(>>=)`: create a new I/O action that performs the first one, takes the result and uses it to create a second action, and then performs that second action.

`Maybe` is also an instance of `Monad`, with the following definition:

```haskell
instance Monad Maybe where
  return = Just

  Nothing >> _ = Nothing
  (Just _) >> x = x

  Nothing >>= _ = Nothing
  (Just x) >>= f = f x
```

While the "bind" and "return" should look familiar, the "then" operator is new, although it makes sense: it returns the second `Maybe` only when the first one is a success.

We'll leave it as an exercise to make `State s` an instance of `Monad`, and to describe precisely what the three operations would mean in that context.

Why monads?

Because these contexts are so different, you might wonder what purpose it serves to unite them under a single type class at all. It turns out that the common set of operations—`return`, `(>>)`, and `(>>=)`—are powerful enough to serve as building blocks for much more complex, generic operations. Consider, for example, the following situations:

- Take a list of results of possibly-failing computations, and return a list of the success values if they were all successes, otherwise fail.
- Take a list of I/O actions, perform them all, and return the list of the produced results.
- Take a list of stateful operations, perform them all in sequence on the same initial state, and return a list of the results.

All of these actions have essentially the same type: `[m a] -> m [a]`, where `m` is either `Maybe`, `State s`, or `IO`. Moreover, all of them involve the type of chaining that is captured in the monadic operations, so this should be a hint that we

can define a single function to accomplish all three of these. Because operating with the monadic functions is relatively new to us, let us first try to implement this—you guessed it—recursively.

```
seqM :: Monad m => [m a] -> m [a]
seqM [] = return []
```

The first thing to note is that we use `return` here in a very natural way, to put a base value—the empty list—into a monadic context. For the recursive part, we consider two parts: the first monadic value in the list (with type `m a`), and the result of applying `seqM` to the rest of the list (with type `m [a]`).

```
seqM (m1:ms) =
    let rest = seqM ms
    in
        ...
```

What do we want to do with these two values? Simple: we want to "cons" the contained `a` value from the first one with the contained `[a]` from the second, and then return the result:

```
seqM :: Monad m => [m a] -> m [a]
seqM [] = return []
seqM (m1:ms) =
    let rest = seqM ms
    in
        -- extract the first value
        m1 >>= \x ->
        -- extract the other values
        rest >>= \xs ->
        return (x:xs)
```

"But wait," you say. "What happened to all of the context-specific stuff, like handling `Nothing` or mutating the state?" The brilliance of this is that *all* of that is handled entirely by the implementation of `(>>=)` and `return` for the relevant types, and `seqM` itself doesn't need to know any of it!

```
> seqM [Just 10, Just 300, Just 15]
Just [10, 300, 15]
> (seqM [putStrLn "Hello", putStrLn "Goodbye", putStrLn "CSC324!!!"])
Hello
Goodbye
CSC324!!!
[(),(),()]
```

Now, this one example doesn't mean that there aren't times when you want, say, `Maybe`-specific code. Abstraction is a powerful concept, but there is a limit to how far you can abstract while still accomplishing what it is you want to do. The `Monad` type class is simply a tool that you can use to design far more generic code than you probably have in other languages. But what a tool it is.

Exercise Break!

3.1 Implement `seqM` using `foldl`.

---

4 *In Which We Say Goodbye*

> If all you have is a hammer,
> everything looks like a nail.
>
> Law of the instrument

When teaching introductory computer science, we often deemphasize the idiosyncrasies and minutiae of the programming language(s) we use, focusing instead on universal theoretical and practical concepts like control flow, data structures, modularity, and testing. Programming languages are to computer science what screwdrivers are to carpentry: useful tools, but a means to an end. If you have a cool idea for the *Next Big Thing*, there are dozens, if not hundreds, of programming languages and application frameworks[1] to choose from.

[1] Large, monolithic frameworks are essentially specialized languages themselves.

It is easy to get lost in the myriad technical and non-technical details that we use to compare languages. Though these are very important—and don't get us wrong, we covered several in this course—we must not lose sight of the forest for the trees. At its core, the study of programming languages is the study of *how we communicate to computers*: how language constructs and features inform the properties of the code we can express.

Our first big idea was exploring how first-class functions allow us to write programs in a pure functional style, achieving greater abstraction through higher-order functions and writing elegant code without side-effects. We then turned to using macros to add two new constructs to Racket: class-based objects through closures, and backtracking through declarative choice expressions using continuations. Finally, we saw how to use a strong static type system in Haskell to encode common language features like errors, mutation, and I/O in an explicit and compile-time checkable way.

We hope you got the sense from this course that programming language research is a truly exciting and fascinating field. If you are interested in learning more, **CSC465** studies a rigorous, mathematical approach for understanding the semantics of (classical imperative) programming languages. **CSC410** studies how to specify and ensure the correctness of programs, using a combination of static and dynamic analysis techniques. And **CSC488**, our compilers course, is fundamentally the study of *translation*: taking a human-readable, high-level language and turning it into the sequence of primitive operations directly understandable by a computer.

Newer, modern programming languages extend the functional programming and static analysis ideas in interesting ways. Two languages of note that draw inspiration from other domains are *Elm*, which uses the declarative *functional reactive programming* paradigm to build web applications, and *Rust*, which uses strong static analysis to add powerful compile-time safety guarantees at the low level of systems programming. And of course, if you would like to talk about any of the material in this course or beyond, don't be afraid to ask! That's what we're here for.