David Liu

Principles of Programming Languages

Lecture Notes for CSC324 (Version 1.2)

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, Ian Stewart-Binks, Anthony Vandikas, and many anonymous students for their helpful comments and error-spotting in earlier versions of these notes.

Contents

Prelude: The Lambda Calculus

> 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

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! 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 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.

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

But this is not their story.

Alan Turing, 1912-1954

The *Entscheidungsproblem* ("decision problem") asked whether an algorithm could decide if a logical statement is provable from a given set of axioms. Turing showed no such algorithm exists.
Finite state controls are analogous to the *deterministic finite automata* that you learned about in CSC236.

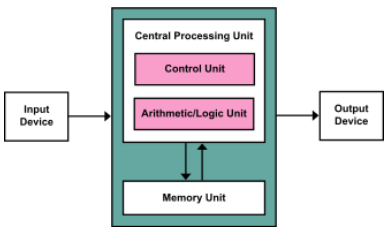


Figure 1: wikipedia.org/wiki/Von_Neumann_architecture.

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.

Alonzo Church, 1903-1995

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?

To make this amazing idea a little more concrete: no existing programming language and accompanying hardware can solve the Halting Problem. None.

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.*For non-grammar buffs, the* imperative *verb tense is the tense we use when issuing orders: "**Give** me that" or "**Stop** talking, David!"* Though such statements are composed of smaller expressions, these expressions typically do not appear on their own; consider the following odd-looking, but valid, Python program:

```
1 def f(a):
2   12 * a - 1
3   a
4   "hello" + "goodbye"
```

Even though each of the 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

We will return to the idea of "sequencing expressions" later on.

bodies of those functions contain statements for the computer.

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 the operations corresponding to those statements, but rather that it *evaluates* the single expression.

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

1. A **variable/name**: a, x, $yolo$, etc. These are just symbols, and have no intrinsic meaning.
2. A **function**: $\lambda x \mapsto x$. 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**: $f\ expr$. This expression applies the function f to the expression $expr$.

We use non-standard notation here. This function would normally be expressed as $\lambda x.x$.

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, variables and functions 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`:

$$(\lambda x \mapsto x)hi$$

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

Pretty simple, eh? But as straight-forward as this sounds, this 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 evaluation (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 the lambda calculus only allows us to define *pure functions*, because the only thing we can do 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 *steps*, nor is there any external or global *state* which 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 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.

If you're curious, ask me about this! It's probably one of my favourite things to talk about.

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.

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 more functional programming styles are being adopted in conventional 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; in particular, the latter's object system is heavily-influenced by its function-centric design.

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. 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 its own right.

Those of you who are particularly interested in compilers should really take CSC488.

Course Overview

We will begin our study of functional programming with Racket, a dialect of Lisp commonly used for both teaching and language research. Here, we will explore language design features like scope, function call strategies, and tail recursion, comparing Racket 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: recursion; the list functions map, filter, and fold; and higher-order functions and closures. We'll conclude with a particularly powerful feature of Racket: *macros*, which give the programmer to add new syntax and semantics to the language in a very straight-forward manner.

We will then turn our attention to Haskell, another language founded on the philosophy of functional programming, but with many stark contrasts to Racket. We will focus on two novel features: a powerful static type system and lazy evaluation, with a much greater emphasis on the former. We will also see how to use one framework – an advanced form of function composition – to capture programming constructs such as failing computations, mutation, and even I/O.

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

Finally, we will turn our attention to Prolog, a programming language based on an entirely new paradigm known as **logic programming**, which models computation as queries to a set of facts and rules. This might sound like database querying in the style of SQL, and the basic philosophy of Prolog is quite similar; how the data is stored and processed, however, is rather different from likely anything you've seen before.

Racket: Functional Programming

> 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 invented **Lisp**, a bare bones programming language based on Church's lambda calculus. Since then, Lisp has spawned many dialects (languages based on Lisp with some deviations from its original specifications), among which are **Scheme**, which is widely-used as an introductory teaching language, and **Clojure**, which compiles to the Java virtual machine. To start our education in functional programming, we'll use **Racket**, an offshoot of Scheme.

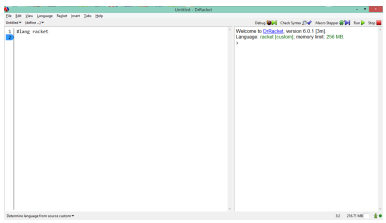
Lisp itself is still used to this day; in fact, it has the honour of being the second-oldest programming language still in use. The oldest? Fortran.

The old name of Racket is PLT Scheme.

Quick Introduction

Open DrRacket, the integrated development environment (IDE) we'll be using in this chapter. You should see two panes: an editor, and an interactions windows. Like Python, Racket features an interactive **read-evaluate-print loop (REPL)** where you can type in expressions for immediate evaluation. Make use of this for quick and easy experimentation when learning Racket!

You can install Racket here: `http://racket-lang.org/download/`.



Though Racket is about as close as you'll get to a bare-bones lambda calculus implementation, one of its usability advantages over the theoretical model is that it provides a standard set of *literals* which are considered fully simplified expressions, and hence values. There are three basic literal types we'll use in Racket. These are illustrated below (you should follow along in your Interactions window by entering each line and examining the result).

```
; numbers
3, 3.1415
; booleans. Can also use 'true' and 'false' instead of #t and #f.
#t, #f
; strings
"DOUBLE quotes are required for strings."
```

Racket uses the semi-colon to delimit one-line comments.

With this in mind, let us see how the three main expression types of the lambda calculus are realised in Racket.

Function Application

Unlike most programming languages, Racket uses Polish prefix notation for *all* of its function applications. Its function application syntax is:

The surprising part, as we'll see below, is that this applies to operators which are infix in most languages.

```
(<function> <arg-1> <arg-2> ... <arg-n>)
```

Every function is applied in this way, which means **parentheses are extremely important in Racket!** Almost every time you see parentheses, the first expression is a function being applied.

Every time except for syntactic forms and macros, which we'll get to later.

Here are some examples of function application. We'll leave it as an exercise for you to determine what these functions do.

As part of your learning for this course, you should become comfortable reading the documentation for a new language and finding common functions to use.

```
> (max 3 2)
3
> (even? 4)
#t
> (sqrt 22)
4.69041575982343
> (string-length "Hello, world!")
13
```

Note the use of the ? for naming boolean function, a common convention in Racket.

Something that looks a little strange to newcomers is that the common *operators* like + and <= are *also* functions, and use the same syntax.

```
> (+ 3 2)
5
> (+ 1 2 3 4 5 6)
21
> (<= 10 3)
#f
> (equal? 3 4)
#f
```

The + function, like many other Racket functions, can take an arbitrary number of arguments.

Oh, and by the way: try putting each of the following two lines into the interpreter. What happens?

```
> +
> (2)
```

Function Values

Recall that the lambda calculus used the Greek letter *lambda*, λ, to denote the creation of a function value. Racket follows this tradition for its function creation syntax:

```
(lambda (<param-1> ... <param-n>) <body>)

> (lambda (x) (+ x 3))
#<procedure>
> ((lambda (x) (+ x 3)) 10)
13
> (lambda (x y) (+ x y))
#<procedure>
> ((lambda (x y) (+ x y)) 4 10)
14
```

By the way, the shortcut `Ctrl+\` produces the Unicode symbol λ, which you can use in Racket in place of `lambda`.

Let's take this opportunity use the editor pane; we can write more complex functions split across multiple lines.

```
#lang racket

((lambda (x y z)
   (+ (- x y)
      (* x z)))
 4
 (string-length "Hello")
 10)
```

The first line tells the Racket interpreter which flavour of Racket to use. For this entire course, we're going to stick to vanilla `racket`.

If we run this code (`Racket` → `Run`), the value `39` will be output in the interactions pane.

By the way, I deliberately chose a more complex function application to illustrate **correct indentation**, which DrRacket does for you. **Don't fight it!** The creators of DrRacket spent a great deal of energy making sure the indentation worked nicely. If you don't appreciate it, you *will* get lost in the dreaded Parenthetical Soup. At any point, you can select a block (or all) of your code and use `Ctrl + I` to reindent it.

Sometimes late at night as I'm drifting off to sleep, I still hear the cries: "I should have respected DrRacket's indentation..."

This is all well and good, but we're starting to see a huge readability problem. If we combine creating our own functions with the ability to nest function calls, we risk writing monstrosities like this:

```
((lambda (x y) (+ x
                  ((lambda (z) (* z z)) 15)
                  (/ (+ 3
                        (- 15 ((lambda (w) (+ x w)) 13)))
                     ((lambda (v u)
                        (- (+ v x) (* u y)))
                      15
                      6))))
 16
 14)
```

No indentation convention in existence can save us here! Of course, this kind of thing happens when we try to combine every computation at once, and is not at all a problem specific to Racket. This leads us to our next ingredient: saving intermediate calculations as names.

You may recall the common rule of thumb that a function not span more than around fifteen lines of code.

Names

You have seen one kind of name already: the formal parameters used in `lambda` expressions. These are a special type of name, and are bound to values when the function is called. In this subsection we'll look at using names more generally. In the lambda calculus, non-parameter variables were considered values, and could not be simplified further. This is emphatically *not* the case in Racket (nor in any other programming language): all non-parameter names must be explicitly bound to a value, and the *evaluation* of a name replaces that name with its bound value.

In other words, real programming languages introduce a second simplification rule: name-value lookup.

However, it is important to note that we will be using names as a convenience, to make our programs easier to understand, but not to truly extend the power of the lambda calculus. We'll be using names for two purposes:

1. "Save" the value of subexpressions so that we can refer to them later.
2. Refer to a function name within the definition of the function, allowing for 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, even without the use of named functions.

This is very cool. Look up the "Y combinator" for details.

Unlike the imperative programming languages you've used so far, names in (pure) functional programming represent **immutable** values – once bound to a particular value, that name cannot be changed, and so is simply an alias for a value. This leads us to an extremely powerful concept known as **referential transparency**. We say that a name is referentially transparent if it can be replaced with its value in the source code without changing the meaning of the program.

referential transparency

Another parallel from math: when we say "x equals 5" in a calculation or proof, we don't expect the value of x to change.

This approach to names in functional programming is hugely different than what we are used to in imperative programming, in which changing the values bound to names is not just allowed but required for some common constructs. Given that mutable state 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?

e.g., loops

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 variable 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.

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

Now, Racket is actually an *impure* functional programming language, meaning (among other things) that it does support mutation. However, for about 90% 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 thinking!

Global definitions

The syntax for a global definition uses the keyword `define`:

```
(define <name> <expr>)
```

This definition first evaluates `<expr>`, then binds the resulting value to `<name>`. Here are some examples using `define`, including a few that bind a name to a function, because of course functions are proper values in Racket.

```
(define a 3)
(define b (+ 5 a))
(define add-three
  (lambda (x) (+ x 3)))
(define almost-equal
  (lambda (x y) (<= (abs (- x y)) 0.001)))
```

Because function definitions are so common, there is a more concise way of binding function names. We're going to use this more convenient notation for the rest of the course, but keep in mind that this is merely "syntactic sugar" for the lambda expressions!

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

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

One final note: Racket, like many other programming languages, enforces the rule that name bindings must be placed before they are used:

```
; a name and then a usage -> name in scope
(define a 3)
(+ a 7)

; a usage and then name -> error will be raised
(+ a b)
(define b 10)
```

We see here a tradeoff between purity and practicality. In a logical sense, we *have* specified the value of `b` in the program. Time has no place in the pure lambda calculus. However, we cannot escape the fact that the Racket interpreter reads and evaluates the program sequentially, from top to bottom.

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.

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

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

We can also explicitly create a local scope using the keyword `let`:

```
(let ([<name> <expr>] ...)
  <body>)
```

A `let` expression takes pairs `[<name> <expr>]`, evaluates each `<expr>`, then binds the resulting value to the corresponding `<name>`. Finally, it evaluates `<body>` (which might contain references to the `<name>`s), and returns that value as the value of the entire `let` expression.

In many other programming languages, control flow blocks like `if` and loops also possess local scope; however, we express such constructs using functions and expressions in pure functional programming, so there is no distinction here.

```
(define x 3)

(let ([y (+ x 2)]
      [z 10])
  (* x y z))          ; 150

x  ; 3
y  ; error!
```

However, suppose in the previous example we wanted to define `z` in terms of `y`; `let` doesn't allow us to do this because it doesn't recognize the `y` binding until all of the local bindings have been created. We use `let*` instead, which behaves similarly to `let`, except that it allows a local name binding to be used in bindings below it.

```
(define x 3)

(let* ([y (+ x 2)]
       [z (+ y 1)])
  (* x y z)))           ; 3 * 5 * 6 = 90
```

Finally, we can use `letrec` to make recursive bindings:

```
(define (f x)
  (letrec ([fact (lambda (n) (if (equal? n 0)
                                 1
                                 (* n (fact (- n 1))))))])
    (fact x)))
```

The fact that recursive functions require a new keyword is a hint that something unusual is going on.

Variable shadowing

You might think that name bindings in Racket play the same role as assignment statements ("`x = 3`") in statement-based languages. However, there is one crucial difference we alluded to earlier: you cannot use `define` on the same name twice to change its value.

```
(define x 3)
(define x 4) ; error "duplicate definition for identifier"
```

However, Racket supports a permissive language feature called variable (or name) shadowing. Consider the following Racket code, which contains two name bindings for the same name, except that one is global and one is local:

```
(define x 10)
(let ([x 3])
  (+ x 1))    ; What is output here?

x             ; What is output here?
```

The first thing to note is that this is valid Racket code, and does not raise an error. The two `x` names defined here are treated as completely separate entities, even though they happen to share the same spelling in the source code.

Variable shadowing is when a local binding (including function parameters) shares the same name as an outer binding. To prevent any ambiguity, Racket and other programming languages use a very simple rule to resolve these conflicting bindings. Every time an identifier is used, its value is obtained from the *innermost* binding for that name; in other words, each level of nested local bindings takes precedence over outer bindings. We say that the inner binding *hides* the outer one, and the outer one is *shadowed* by the inner one.

variable shadowing

If you haven't done so, run the above code, and make sure you understand the two outputs! Variable shadowing is a simple rule, but if you aren't conscious of it, you are bound to make mistakes in your code.

Special Forms: `and`, `or`, `if`, `cond`

In addition to the primitive data types and the three elements of the lambda calculus, Racket features a few different special syntactic forms. First, here they are in their usage:

```
; logical AND, which short-circuits
(and #t #t)        ; #t
(and #f (/ 1 0))   ; #f, even though (/ 1 0) is an error!
(and #t (/ 1 0))   ; error

; logical OR, also short-circuits
(or #t #t)         ; #t
(or #t (/ 1 0))    ; #t
(or #f (/ 1 0))    ; error

; (if <condition> <consequent> <alternative>)
; Evaluates <condition>, then evaluates EITHER
; <consequent> or <alternative>, but not both!
(if #t 3 4)        ; 3
(if #f 3 4)        ; 4
(if #t 3 (/ 1 0))  ; 3
(if #f 3 (/ 1 0))  ; error

; (cond [(cond-1 expr-1)] ... [(cond-n expr-n)] [(else expr)])
; Continuously evaluates the cond-1, cond-2, ..., until one evaluates
; to true, and then evaluates and returns the corresponding expr-i.
; (else expr) may be put at end to always evaluate something.
; Note: at most ONE expr is ever evaluated;
; all of the cond's might be evaluated (or might not).
(cond [(> 0 1) (+ 3 5)]
      [(equal? "hi" "bye") -3]
      [#t 100]
      [else (/ 1 0)])              ; 100
```

Technically, `if` and `cond` interpret any non-`#f` value as true. For example, the expression `(if 0 1 2)` evaluates to `1`.

First note that even though `if` and `cond` seem to possess the familiar control flow behaviour from imperative languages, there is an important distinction: they are *expressions* in Racket, meaning that they always have a value, and can be used anywhere an expression is allowed. For example, inside a function call expression:

This is analogous to the "ternary operator" of other languages.

```
> (max (if (< 5 10) 10 20)
       16)
16
```

However, even though all of `and`, `or`, `if`, and `cond` look like plain old functions, they aren't! What makes them different?

Eager evaluation

Recall from our earlier discussion of the lambda calculus that we view computation as the evaluation of functions, and that functions are evaluated with the simple rule of *substitution*.

We'll use the $\rightarrow$ to signify an evaluation step for a Racket expression.

```
((lambda (x) (+ x 6)) 10)
; → (substitute 10 for x)
(+ 10 6)
; →
16
```

However, function evaluation as substitution is only part of the story. Any real interpreter does not just need to know that must substitute arguments into function bodies to evaluate them: it needs to know the *order* in which to perform such substitutions. Consider this more complex example:

```
((lambda (x) (+ x 6)) (+ 4 4))
```

We now have a *choice* about how to evaluate this expression: either evaluate the `(+ 4 4)`, or substitute that expression into the body of the outer lambda. Now, it turns out that in this case the two are equivalent, and this is true in most typical cases.

More formally, the **Church-Rosser Theorem** says that for any expression in the lambda calculus, if you take different evaluation steps, there exist further evaluation steps for either choice that lead to the same expression. Intuitively, all roads lead to the same place.

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

Or,

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

However, for impure functions – or pure functions which may be non-terminating or generate errors – the order of evaluation matters greatly! So it's important for us to know that Racket does indeed have a fixed evaluation order: arguments are always evaluated in *left-to-right* order, *before being passed* to the function. So in the above example, Racket would perform the *first* substitution, not the second.

This is a very common evaluation strategy known as **left to right strict/eager evaluation**. When we get to Haskell, we will study an alternate evaluation strategy known as lazy evaluation.

eager evaluation
Python, Java, and C are all strict.

What makes the special syntactic forms special?

Now that we know that Racket uses strict evaluation, we can see what makes `and`, `or`, `if`, and `cond` special: none of these are guaranteed to evaluate all of their arguments! To illustrate the point, suppose we tried to write our own "and" function, which is simply a wrapper for the built-in `and`:

```
(define (my-and x y) (and x y))
```

Even though it looks basically identical to the built-in `and`, it's not, simply because of evaluation order.

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

This point is actually rather subtle, because it has nothing to do with Racket at all! In *any* programming language that uses eager evaluation, it is impossible to write a short-circuiting "and" function.

Exercise Break!

1. 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)))`.

2. Draw the *expression tree* associated with the previous expression, where each internal node represents a function call, whose children are the arguments to that function.
3. We have two special keywords `lambda` and `define`. Neither of these are functions; how do you know?

Lists

The list is one of the most fundamental data structures in computer science; in fact, the name "Lisp" comes from "**LIS**t **P**rocessing". Racket views lists as a *recursive* data structure:

- The empty list is a list, represented in Racket by `'()` or `empty`.
- If `lst` is a list, and `item` is a value, then we can create a new list whose first element is `item` and whose other items are the ones from `lst`. This is done in Racket with the `cons` function:

  `(cons item my-list)`.

The `cons` function, standing for "construct," is used more generally to create a pair of values in Racket, though we will use it primarily for lists in this course.

Try playing around with lists in DrRacket, following along with the commands below.

```
; the empty list is denoted by empty or '()
(define empty-1 empty)
(define empty-2 '())
(equal? empty-1 empty-2)          ; #t

; list with one element - an item cons'd with an empty list
(cons 3 empty)                    ; equivalently, '(3 . ())

; list with two elements
(cons 3 (cons (+ 10 2) '()))

; this will get tedious very quickly - so let's switch notation!
(list 1 2 3 4 5 6 7)
(list (+ 3 2) (equal? "hi" "hello") 14)
(list 1 2 (list 3 4) (cons 2 '()))

; Not a list!
(define a (cons 1 2))
(list? a)                         ; #f
```

Note that `equal?` checks for value equality, not reference equality.

Remember: a list is created with a `cons` where the second element must be a list. In the last example, `2` is not a list.

Aside: quote

You probably noticed that Racket represents lists using the concise notation `'(1 2 3)`, analogous to the common list representation `[1, 2, 3]`. We call the symbol `'` at the beginning of this expression the *quote*. Pretty amazingly, you can also use the quote *in* your code to create lists. This offers a shorter alternative to either `cons` or `list`, and even works on nested lists:

```
'(1 2 3 4 5)
'(1 2 (3 4 5) (6 7))
```

However, the quote only works for constructing lists out of *literals*, and doesn't allow you to evaluate expressions when creating lists:

```
> (define x '(1 2 (+ 3 4)))
> (third x)
'(+ 3 4)
> (first (third x))
'+
```

The `'+` is a result of the quote interpreting the `+` as a **symbol**, rather than a function. We will not talk much about symbols in this course.

Instead, you can use the Racket `quasiquote` and `unquote` syntactic forms, although this is a just a little beyond the scope of the course. Here's a taste:

```
> `(1 2 ,(+ 3 4))
'(1 2 7)
```

We use the backtick `` ` `` and comma `,` in this expression.

We generally recommend using the `list` function to be explicit in your intention and avoid the accidental conversions to symbols.

List functions

Since lists occupying such a central place in Racket's data types, the language offers a number large number of functions to perform computations on lists. Here are a few basic ones; for a complete set of functions, check the Racket documentation.

`http://docs.racket-lang.org/reference/pairs.html`

```
> (first '(1 2 3))
1
> (rest '(1 2 3))
'(2 3)
> (length '(1 2 "Hello"))
3
> (append '(1 3 5) '(2 4 6))
'(1 3 5 2 4 6)
```

Recursion on lists

Imperative languages naturally process lists using loops. This approach uses time and state to keep track of the "current" list element, and so requires mutation to work. In pure functional programming, where mutation is not allowed, lists are processed by using their recursive structure to write recursive algorithms.

Generally speaking, the *base case* is when the list is empty, and the recursive step involves breaking down the list into its first element and all of the other elements (which is also a list), applying recursion to the rest of the list, and then combining it somehow with the first element. You should be familiar with the pattern already, but here's a simple example anyway:

```
(define (sum lst)
  (if (empty? lst)
      0
      (+ (first lst) (sum (rest 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 come back to in much detail later.

```
(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))]))
```

Exercise Break!

4. Write a function to determine the length of a list.
5. Write a function to determine if a given item appears in a list.
6. Write a function to determine the number of duplicates in a list.
7. Write a function to remove all duplicates from a list.
8. 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).
9. Write a function which takes a list of lists, and returns the list which contains the largest item (e.g., given `'((1 2 3) (45 10) () (15))`, return `'(45 10)`).

10. Write a function which takes an item and a list of lists, and inserts the item at the front of every list.
11. Write a function which takes a list with no duplicates representing a set (order doesn't matter). Returns a list of lists containing all of the subsets of that list.
12. Write a function taking a list with no duplicates, and a number k, and returns all subsets of size k of that list.
13. Modify your function to the previous question so that the parameter k is optional, and if not specified, the function returns all subsets.
14. Write a function that takes a list, and returns all *permutations* of that list (recall that in a permutation, order matters, so `'(1 2 3)` is distinct from `'(3 2 1)`).
15. 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 which does this, although you should try to get an algorithm that is correct first, as the $O(n)$ algorithm is a little more complex.)

It involves a helper function.

Tail call elimination

Roughly speaking, function calls are stored on the *call stack*, a part of memory which stores information about the currently active functions at any point during runtime. 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 $\mathcal{O}(n)$:

```
def f(n):
  if n == 0:
    return 0
  else:
    return f(n-1)
# Produces a RuntimeError because the call stack fills up
f(10000)
```

More precisely, the CPython implementation guards against stack overflow errors by setting a maximum limit on recursion depth.

In fact, the same issue of recursion taking up a large amount of memory occurs in Racket as well. However, Racket and many other languages 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 the value will

tail call elimination

be returned to the original caller. 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 form the call stack, leading to a very small ($\mathcal{O}(1)$) space usage for the equivalent Racket function:

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))))
```

Our `sum` is not tail-recursive, because the result of the recursive call must first be added to `(first lst)` before being returned. However, we can use a tail-recursive helper function instead.

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

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

Because recursion is extremely common in function programming, converting recursive functions to tail-recursive functions is an important technique that you want to practice. The key strategy is to take your function and add an extra parameter to *accumulate* results from previous recursive calls, eliminating the need to do extra computation with the result of the recursive call.

In the above example, the parameter `acc` plays this role, accumulating the sum of the items "processed so far" in the list. We use the same idea to accumulate the "multiples of three seen so far" for our `multiples-of-3` function:

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

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

Exercise: why didn't we use `cons` here?

Exercise Break!

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

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, we said at the beginning that in the lambda calculus, functions are values, so it is natural to ask: can functions operate on other functions?

Indeed, in the pure lambda calculus all *values* are actually *functions*

The answer 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 output new ones. Let's see some simple examples.

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 mathematicians don't use this terminology. By the way, so is the indefinite integral.

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

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

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

Delaying evaluation

Using higher-order functions, we can find a way to simulate delaying the evaluation of arguments to functions. Recall that our problem with an `my-and` function is that every time an expression is passed to a function call, it is evaluated. However, if a *function* is passed as an argument, Racket does not "evaluate the function" (i.e., call the function), and in particular, does not touch the body of the function.

Remember, there is a large difference betweeen a function being passed as a value, and a function call expression being passed as a value!

This means that we can take an expression and delays its evaluation by using it as the body of a o-arity function :

o-arity: takes zero arguments

```
; short-circuiting "and", sort of
(define (my-and x y) (and (x) (y)))

> (my-and (lambda () #f) (lambda () (/ 1 0)))
#f
```

Higher-Order List Functions

Let's return to the simplest recursive object: the list. With loops, time, and state out of the picture, 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, functional programmers often use three critical higher-order functions to compute with lists. The first two are extremely straightforward:

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

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

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

`map` can be called on multiple lists; check out the documentation for details!

To illustrate the third core function, let's return to the previous (tail-recursive) example for calculating the sum of a list. It turns out that the pattern used is an extremely common one:

```
(define (function lst)
  (helper init lst))

(define (helper acc lst)
  (if (empty? lst)
      acc
      (helper (combine acc (first lst)) (rest lst))))
```

A good exercise is to rewrite `multiples-of-3` in this form.

This is interesting, but also frustrating. As computer scientists, this kind of repetition begs for some abstraction. Note that since the recur-

sion is fixed, the only items that determine the behaviour of `function` are `init` and the `combine` function; by varying these, we can radically alter the behaviour of the function. This is precisely what the `foldl` function does.

You may have encountered this operation previously as "reduce."

```
; sum...
(define (sum lst)
  (sum-helper 0 lst))

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

; is equivalent to
(define (sum2 lst) (foldl + 0 lst))
```

Notice, by the way, that this abstraction is only possible because we can pass a *function* `combine` to `foldl`. In languages where you can't pass functions as arguments, we'd be out of luck.

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. This makes `foldl` both the most powerful and most complex of the three. Study its Racket implementation below carefully, and try using it in a few exercises to get the hang of it!

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

`foldl` intuition tip

Here is a neat way of gaining some intuition about what `foldl` actually does. Consider a call `(foldl f 0 lst)`. Let's step through the recursive evaluation as the size of `lst` grows:

```
(foldl f 0 '())
; →
0
```

```
(foldl f 0 '(1))
; →
(foldl f (f 1 0) '())
; →
(f 1 0)

(foldl f 0 '(1 2))
; →
(foldl f (f 1 0) '(2))
; →
(foldl f (f 2 (f 1 0)) '())
; →
(f 2 (f 1 0))
```

Of course, this generalizes to a list of an arbitrary size:

```
(foldl f 0 '(1 2 3 ... n))
; →
(f n (f (- n 1) ... (f 2 (f 1 0))...))
```

Exercise Break!

17. 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.
18. Reimplement all of the previous exercises using `map`, `filter`, and/or `foldl`, and *without* using explicit recursion.
19. 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`.
20. Is `foldl` tail-recursive? If so, explain why. If not, rewrite it to be tail-recursive.
21. Implement `map` and `filter` using `foldl`. (Cool!)
22. The "l" in `foldl` stands for "left", because items in the list are combined with the accumulator in order from left to right.

```
(foldl f 0 '(1 2 3 4))
; →
(f 4 (f 3 (f 2 (f 1 0))))
```

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

```
(foldr f 0 '(1 2 3 4))
; →
(f 1 (f 2 (f 3 (f 4 0))))
```

apply

To round off this section, we will look at one more fundamental Racket function. As a warm-up, consider the following (mysteriously-named) function:

The name will seem less mysterious later in the course, we promise.

```
(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 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, and one that takes ten?

Try it!

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` which does almost this:

```
; (apply f lst)
; Call f with arguments being the value in lst
(apply + '(1 2 3 4))
; → (equivalent to)
(+ 1 2 3 4)

; More generally,
(apply f (list x1 x2 x3 ... xn))
; →
(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 *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.

Exercise Break!

23. 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)`. You can (and should) use `apply` in your solution.

Lexical Closures

We have now seen functions that take primitive values and other functions, but so far they have all output primitive values. Now, we'll turn our attention to another type of higher-order function: a function that *returns* a function. Note that this is extremely powerful: it allows us to create new functions *at runtime*! Here is a simple example of this:

```
(define (make-adder x)
  ; Notice that the *body* of this function (which is what's returned)
  ; is a function value expression
  (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 `10` 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))
```

If you understand evaluation as substitution, then there really isn't much new here, and you can reason about functions which return functions just as well as any other function. However, the actual implementation of this behaviour in Racket is quite a bit more subtle, and more interesting. But before you move on...

Exercise Break!

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

25. 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.
26. 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`.
27. 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.
28. Write the following 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
|#
```

To accomplish this, look up **rest arguments** in Racket.

29. 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
|#
```

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

Closures

Suppose we call `make-adder` multiple times: `(make-adder 10)`, `(make-adder -1)`, `(make-adder 9000)`, etc. 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, and differ only in their value of `x`.

And in fact, Racket does not create a new function every time `make-adder` is called. When Racket evaluates `(make-adder 10)`, it returns (a pointer to) the function `(lambda (y) (+ x y))` *with* the name binding `{x:10}`. The next call, `(make-adder -1)`, returns a pointer to the *same* function body, but a different binding `{x:-1}`.

The function body together with this name binding is called a **closure**. When `(add-10 3)` is called, Racket *looks up* the value of `x` in the closure, and gets the value 10, which it adds to the argument `y` to obtain the return value 13.

closure

Remember that when the function is called, the argument `3` gets bound to `y`.

We have lied to you: all along, our use of `lambda` has been creating *closures*, not *functions*! Why has this never come up before now? Closures are necessary only when the function body has a **free identifier**, which is an identifier that is not local to the function. Intuitively, this makes sense: suppose every identifier in a function body is either a parameter, or bound in a `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.

free identifier

However, in the definition of `make-adder`, the body of the `lambda` expression has a free identifier `x`, and this is the name whose value will need to be looked up in the closure.

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

In summary, a closure is both a pointer to a function body, as well as a collection of name-value bindings *for all free identifiers in that function body*. If the body doesn't have any free names (as has been the case up until now), then the closure can be identified with the function body itself. Note that in order for the closure to be created, all of the identifiers inside the function body must still be in scope, just not necessarily local to the function. This is true for the previous example: `x` is a free identifier for the inner `lambda`, but is still in scope because it is a parameter

of the enclosing outer function `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)))
```

Okay, so that's what a closure is. To ensure that different functions can truly be created, closures are made when the `lambda` expressions are evaluated, and each such expression gets its own closure.

Though remember, multiple closures can point to the same function body!

```
(define (make-adder x)
  (lambda (y) (+ x y)))
(define add-10 (make-adder 10))
(define add-20 (make-adder 20))

> (add-10 3)
13
> (add-20 3)
23
```

Lexical vs. Dynamic scope

Knowing that closures are created when `lambda` expressions are evaluated does not tell the whole story. Consider the following example:

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

> (add-z 5)
15
```

The body of `make-z-adder` is a function with a free name `z`, which is bound to the global `z`. So far, so good. But happens when we shadow this `z`, and then call `make-z-adder`?

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

Now the `lambda` expression is evaluated after we call the function in the local scope, but what value of `z` gets saved in the closure? Put more concretely, does this expression output `15` or `105`? Or, does the closure created bind `z` to the global one or the local one?

The question of how to resolve free identifier bindings 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, even if closures are created at runtime, *how* the closures are created (i.e., which values are used) is based only where they are created in the source code.

Note that this can be fully determined before running any code.

This means that in the previous example, when `make-z-adder` is called, the `z` in the closure is still bound to value of the global `z`, because that is the one which is in scope where the `lambda` expression is written. If you try to evaluate that expression, you get 15, not 105.

More generally, you can take any identifier and determine at compile time which variable it refers to, simply by taking its location in the source code, and proceeding outwards until you find where this identifier is declared. This is called **lexical scope**, and is used by almost every modern programming language.

lexical scope

In contrast to this is **dynamic scope**, in which names are resolved based on the closest occurrence of that name on the call stack; that is, where the name is used, rather than where it was defined. 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`. In other words, lexical scoping resolves names based on context from the source code, which dynamic scoping resolves names based on context from the program state at runtime.

dynamic scope

Initial programming languages used dynamic scope because it was easier to implement; however, dynamic scope makes programs very difficult to reason about, as the values of non-parameter names of a function now depend on how the function is used, rather than simply where it is defined. Lexical scope was a *revolution* in how it simplified programming tasks, and is ubiquitous today. And it is ideas like that which motivate research in programming languages!

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

By the way, here is an example of how dynamic scoping works in the (ancient) bash shell scripting language:

```
X=hey
function printX {
  echo $X
}
function localX {
  local X=bye
  printX
}

localX    # will print bye
echo $X   # will print hey
```

A Python puzzle

Closures are often used in the web programming language Javascript to dynamically create and bind callbacks, functions meant to respond to events like a user clicking a button or entering some text. A common beginner mistake when writing 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 *refer to the same* `i` variable (in the enclosing scope). That is, the closures here store a *reference*, and not a *value*. 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!

Remember: since we have been avoiding mutation up to this point, there has been no 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.

```
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()
```

Here, each `print_i` function has a closure looking up `x`, which is bound to different values and not changed as the loop iterates.

Secret sharing

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

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

  def alice(s):
    nonlocal secret
    secret = s

  def bob():
    print(secret)

  return alice, bob

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

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

Exercise Break!

31. In the following `lambda` expressions, what are the free variables (if any)? (Remember that this is important to understand 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))
```

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

Summary

In this chapter, we looked at the basics of functional programming in Racket. Discarding mutation and the notion of a "sequence of steps", 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 control flow entirely; in our study of evaluation order, we learned precisely how Racket evaluates functions, and how special syntactic forms distinguish themselves from functions precisely because of how their arguments are evaluated.

Our discussion of higher-order functions culminated in our discussion of closures, allowing us to even 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.

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*, will likely be very familiar to you, while the second, *logic programming*, will not. However, because of our limited time in this course, we will not treat either topic with as much detail as they deserve. In particular, we will stay in Racket, rather than switching programming languages. Instead, we will build support for these paradigms into the language of itself, and so kill two birds with one stone: you will learn how to use and advanced macro system to fundamentally extend a language, and gain a deeper understanding of these paradigms by actually implementing simple versions of them.

Other offerings of this and similar courses often use a pure object-oriented language like Ruby or Eiffel and a logic programming language like Prolog.

Basic Objects

> 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 simple object-oriented system.

Recall the definition of a closure: a function together with name bindings that are saved when the function is created. It is this latter part – "stored values" – which is reminiscent of objects. In traditional object-oriented languages, an **object** is a value which stores data in **attributes** with associated functions called **methods** which can operate on this data.

object

This paradigm was developed as a way to organize and encapsulate data, while exposing a public interface to define how others may operate on this data. Unlike the pure functions we have studied so far, a method always takes special argument, an associated object which we often 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 the object calls.

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 to model computation. We have already seen how closures provide encapsulation when they are returned from function calls. If we put the notion of an object as something that receives and responds to a message into the context of functional programming, well, an object is just a particular type of function. Here is a simple example of that idea, in Racket:

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

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

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

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, preventing reusability.

One solution to the latter problem is to create a point **class**, a template which 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, getting their attributes and methods from the class definition. 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.

class

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.

To translate this into our language of functions, a point constructor 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.

```
(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
```

We follow the convention of giving the constructor the same name as the class itself.

Of course, this is where we require the use of closures: in the returned function, `x` and `y` are free, and so must have values bound in a closure when the `Point` constructor returns. And of course the `x` and `y` identifiers are local to the `Point` function, so even though their values are stored in the closure, they can't be accessed without passing a message to the object.

And *lexical* scope 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.

One might quibble that this isn't true encapsulation, because nothing prevents the user from passing the right messages to the object. We'll let you think about how to define private attributes in this model.

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

Basic methods

Next, let's add two simple methods to our Point class. Because Racket gives its functions first-class status, we can treat attributes and methods in the same way: a method is just an attribute that happens to be a function. But remember from our earlier discussion of OOP that inside the body of a method, we expect to be able to access all attributes of the class. Turns out this isn't an issue, since we define methods within the body of the enclosing constructor.

Of course, this isn't a trivial difference. In the same way that functions enable complex computation over primitive values, methods enable computation with internal state rather than just reporting an attribute value.

```
(define (Point x y)
  (lambda (msg)
    (cond [(equal? msg "x") x]
          [(equal? msg "y") y]
          [(equal? msg "to-string")
           (lambda ()
             (string-append "("
                            (number->string x)
                            ", "
                            (number->string y)
                            ")"))]
          [else "Unrecognized message"])))

> (define p (Point 10 2))
> (p "to-string")
#<procedure>
> ((p "to-string"))
"(10, 2)"
```

Finally, let's define a method that takes a parameter that is another point instance, and calculates the distance between the two points.

```
(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)))))]
          [else "Unrecognized message"])))

> (define p (Point 3 4))
> ((p "distance") (Point 0 0))
5
```

Note that `(p "distance")` is a function, so this expression is just a nested function call.

More on reusability

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 you would have to write each time. What we will 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:

message handling with `cond` and `equal?`, "Unrecognized message"

```
(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!

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 output 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. Take a look at the previous `Person` example. Even though it is currently invalid Racket code, the intent should be quite clear. Write a

`Person` class in the same style as the `Point` class given in the notes. This will ensure that you understand our approach for creating classes, so that you are well prepared for the next section.

Macros

Though we have not mentioned it explicitly, Racket's extremely simple syntax – every expression is a nested list – not only makes it easy to parse, but also to *manipulate*. One neat consequence of this is that Racket has a very powerful macro system, with which developers can quite easily extend the language by adding new keywords, and even entire domain-specific languages.

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

Simply put, a **macro** is a function that transforms a piece of Racket syntax into another. Unlike simple function calls (which "transform" a function call expression into the body of that function), macros are not part of the evaluation of Racket expressions at runtime. After the source code is parsed, but *before* anything is evaluated, the interpreter performs a step called **macro expansion**, in which any macros appearing in the code are transformed to produce new code, and it is this resulting code that gets evaluated.

macro

Why might we want to do this? The main use of macros we'll see is to *introduce new syntax* into the programming language. In this section, we'll build up a nifty syntactic construct: the **list comprehension**.

First, a simple reminder of what the simplest list comprehensions look like in Python.

We take our immediate inspiration from Python, but many other languages support similar constructs.

```
>>> [x + 2 for x in [0, 10, -2]]
[2, 12, 0]
```

The list comprehension consists of three important parts: an output expression, a variable name, and an initial list. The other characters in the expression are just syntax necessary to indicate that this is indeed a list comprehension expression instead of, say, a list.

What we would like to do is mimic this concise syntax in Racket:

By the way, even though this looks like function application, that's not what we want. After all, `for` and `in` should be part of the syntax, and not arguments to the function!

```
> (list-comp (+ x 2) for x in '(0 10 -2))
'(2 12 0)
```

Let's first talk about how we might implement the high-level *functionality* in Racket, ignoring the syntactic requirements. If you're comfortable with the higher-order list functions, you might notice that a list comprehension is essentially a `map`:

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

Now, we do some pattern-matching to generalize to arbitrary list comprehensions:

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

; leads to the following generalization.
(list-comp <expr> for <var> in <list>)
(map (lambda (<var>) <expr>) <list>)
```

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 an equivalent map. What remains is actually telling Racket what to do, and that's a macro.

```
(define-syntax list-comp
  (syntax-rules (for in)
    [(list-comp <expr> for <var> in <list>)
     (map (lambda (<var>) <expr>) <list>)]))
```

Let's break that down. The top-level `define-syntax` takes two arguments: a name for the syntax, and then a `syntax-rules` expression. The first argument of `syntax-rules` is a list of all of the *literal keywords* that are part of the syntax: in this case, the keywords `for` and `in`.

There are other, more complex types of macros that we won't cover in this course.

This is followed by the main part of the macro: one or more syntax pattern rules, which are pairs specifying the old syntax pattern to match, and the new one to transform it into. Evaluating this code yields the desired result:

There's only one syntax rule here, but that will change shortly.

```
> (list-comp (+ x 2) for x in '(0 10 -2))
'(2 12 0)
```

However, it is important to keep in mind that there are two phases of execution here, unlike normal function calls: first, the `list-comp` expression is transformed into a `map` expression, and then that `map` expression is evaluated. We can see the difference in the steps by trying 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) for x in 10)
map: contract violation
  expected: list?
  given: 10
  argument position: 2nd
  other arguments...:
  #<procedure>
```

Note that 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) for x in 10)` might be *syntactically* valid, but it expands into `(map (lambda (x) (+ x 2)) 10)`, which raises a runtime error.

The purpose of literal keywords

Our syntax rule makes use of both pattern variables and literal keywords. A *pattern variable* is an identifier which can be bound to an arbitrary expression in the old pattern; during macro expansion, this expression is then substituted for the variable in the new pattern.

We can actually view macro expansion as a generalization of the function call: both operate on the basis of substitution, but the latter has one particular syntax it must follow.

One built-in example you've already seen is the `else` keyword that can appear inside a `cond`.

On the other hand, *literal keywords* are parts of the syntax that must appear literally in the expression, and cannot be bound to any other expression. If we try to use `list-comp` without the two keywords, we get a *syntax* error – the Racket interpreter does not recognize the expression:

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

To avoid confusion, we'll generally name pattern variables using angle brackets, but be warned that this isn't required by Racket, so always make sure to double-check what your keyword literals are in `syntax-rules`.

Extending our basic macro

Python list comprehensions also support filtering:

```
>>> [x + 2 for x in [0, 10, -2] if x >= 0]
[2, 12]
```

To achieve this form of list comprehension in Racket, we simply add an extra syntax rule to our macro definition:

```
(define-syntax list-comp
  (syntax-rules (for in if)
    ; This is the old pattern.
    [(list-comp <expr> for <var> in <list>)
     (map (lambda (<var>) <expr>) <list>)]

    ; This is the new pattern.
    [(list-comp <expr> for <var> in <list> if <cond>)
     (map (lambda (<var>) <expr>)
          (filter (lambda (<var>) <cond>)
                  <list>))]))

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

Ignore the syntax highlighting for the `if`; here, it's just a literal!

In this case, the two old patterns in the rules are mutually exclusive. However, it is possible to define two syntax rules that express overlapping patterns; in this case, the *first* rule which has a pattern that matches an expression is the one that is used to perform the macro expansion.

Hygienic Macros

If you've heard of macros before learning a Lisp-family language, it was probably from C or C++. The C macro system operates on the source text itself, in large part because C's syntax is a fair bit more complex that Racket's. Even though this sounds similar to operating on entire expressions like Racket macros, there is one significant drawback.

The typesetting language LaTeX also uses macros extensively.

Identifiers in C macros are treated just as strings, and in particular their scope is not checked when the substitution happens. This means that inside a macro, it is possible to refer to variables defined outside of it, a phenomenon known as *variable/name capture*. This is illustrated in the following example.

Note that this is *not* the same as name shadowing, for which there are two different values that share the same name.

```
#define INCI(i) {a = 0; ++i;}
int main(void) {
    int a = 0, b = 0;
    INCI(a);
    INCI(b);
    printf("a is now %d, b is now %d\n", a, b);
    return 0;
}
```

The top line is a macro: it searches the source for text of the form `INCI(_)`, and when it does, it replaces it with the corresponding text in the body of the macro.

```c
int main(void) {
    int a = 0, b = 0;
    {a = 0; ++a;};
    {a = 0; ++b;};
    printf("a is now %d, b is now %d\n", a, b);
    return 0;
}
```

But in line 4 the statement `a = 0;` from the macro body resets the value of `a`, and so this program prints `a is now 0, b is now 1`. The local use of `a` in the macro *captures* the variable defined in `main`. In essence, what `a` in the macro refers to depends on which `a` is in scope *where the macro is used* – in other words, C macros are dynamically scoped.

In contrast, Racket's macro system obey lexical scope, a quality known as **hygienic macros**, and so doesn't have this problem. Here's a simple example:

```racket
(define-syntax-rule (make-adder x)
  (lambda (y) (+ x y)))

(define y 10)
(define add-10 (make-adder y))

> (add-10 100)
```

We're using `define-syntax-rule`, a slight shorthand for macro definitions when there is just a single syntax rule and no literal keywords.

The final line does indeed evaluate to `110`. However, with a straight textual substitution, we would instead get the following result:

```racket
(define y 10)
; substitute "y" for "x" in (make-adder y)
(define add-10 (lambda (y) (+ y y)))

> (add-10 100)
200
```

Macros with ellipses

It is often the case that we want a macro can be applied to an arbitrary number of expressions. Unfortunately, we cannot explicitly write one pattern variable for each expression, since we don't know how many there will be when defining the macro. Instead, we can use the ellipsis '...' token to bind to an arbitrary number of repetitions of a pattern.

Think `and`, `or`, or `cond`.

Here is one example of using the ellipsis in a recursive macro that implements `cond` in terms of `if`. To trigger your memory, 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)))
```

Let us write a macro which does the initial expansion from the first expression (just a `cond`) to the second (a `cond` inside an `if`).

```
(define-syntax my-cond
  (syntax-rules (else)
    [(my-cond [else <val>]) <val>]
    [(my-cond [<test> <val>] <next-pair> ...)
     (if <test> <val> (my-cond <next-pair> ...))]))
```

Note that `else` is a literal keyword here.

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. For example, the first syntax rule will match the expression `(my-cond [else 5])`, but *not* `(my-cond else 5)`.

This rule will also match `(my-cond (else 5))` – the difference between () and [] is only for human eyes, and Racket does not distinguish between them.

The second is the `<next-pair> ...` part of the second pattern, which matches "1 or more expressions." 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.

```
(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>`. For example, you cannot use the ellipsis in the new syntax pattern without `<next-pair>`.

```
(define-syntax my-cond-bad
  (syntax-rules (else)
    [(my-cond-bad [else <val>]) <val>]
    [(my-cond-bad [<test> <val>] <next-pair> ...)
     ; This is actually a *syntax* error.
     (if <test> <val> (my-cond-bad ...))]))
```

But this goes both ways – `<next-pair>` can't appear without the ellipsis, either. In other words, 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.

Try it!

Exercise Break!

3. Explain how a macro is different from a function. Explain how it is *similar* to a function.
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 is to make sure you understand the difference!

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

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

5. Write macros to implement `and` and `or` in terms of `if`. Note that both syntactic forms take an arbitrary number of arguments.
6. Why could we not accomplish the previous question with functions?
7. Consult the official Python documentation on list comprehensions. One additional feature we did not cover is list comprehensions on multiple variables:

```
>>> [(x,y) for x in [1,2,3] for y in [x, 2*x]]
[(1, 1), (1, 2), (2, 2)]
```

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

Objects revisited

At the beginning of this chapter, we developed a basic technique for implementing classes and objects using function closures. Here is an abbreviated Point class with only the two attributes.

```
(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. Instead, we turn to macros to introduce a new syntactic form into the language:

Why can functions not help us declare new identifiers?

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

Performing the same pattern-matching procedure as before, we can see how we might write a skeleton of the macro.

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

This is the first macro we've seen which expands into a `define` expression. This works just as you'd expect; remember, nothing is evaluated when macros are expanded.

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

```
(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 strings 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 string. We can do this using two built-in names:

`quote` turns an identifier into a symbol, and then `symbol->string` turns a symbol into the corresponding string.

```
[(equal? msg (symbol->string (quote x)) x)]
```

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

```
[(equal? msg (symbol->string (quote x)) x)]
[(equal? msg (symbol->string (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:

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

I hope you're suitably impressed. I know I am.

Adding methods

Now, let us augment our macro to allow method definitions. We will use the following syntax, mimicking Racket's own macro pattern-matching syntax:

```
(class <class-name> (<attr> ...)
  [(<method-name> <arg> ...)
   <body>]
  ...)
```

Once again, the easiest way to write the 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 running point example:

```
; What we want to write:
(class Point (x y)
  [(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:

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

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 parenthesization really help distinguish the ellipsis pairings.

One warning about macro ellipses: because they must always be paired with a pattern variable, you cannot use them to express *zero* occurrences of a pattern. In the above macro, the single pattern will only match classes which define at least one method, but not on classes that have no methods (these will result in a syntax error). In order to fix this problem, you need to explicitly define another syntax pattern rule for the macro.

There is another limitation in this macro along the same lines – do you see it?

Late binding

Our implementation of objects-as-functions has one consequence we haven't yet discussed, but with huge implications for the behaviour of our class system. We implemented attribute access and method calls in terms of message passing, but these messages are resolved *at runtime*. This is a property of object-oriented systems called **late (or dynamic) binding**, and contrasts with early (or static) binding, in which method calls are checked at compile time.

Think about when a user actually would see the "Unrecognized message" string.

One might wonder what purpose late binding serves, given that we lose the ability to have the compiler check for the existence of methods before running any code. But by deferring object name resolution until runtime, we gain the ability to add attributes and methods to objects during the execution of a program.

Moreover, our mechanism can easily be extended during runtime to respond to more messages by simply "adding" extra cases to the `cond`.

Of course, we still aren't using mutation.

Here is a simple example of this in action:

```
(define p (Point 3 2))
(define p2
  (lambda (msg)
    (cond [(equal? msg "sum-coords")
           (+ (p2 "x") (p2 "y"))]
          [else (p msg)])))

> (p2 "x")
3
> (p2 "y")
2
> (p2 "sum-coords")
5
> (p "sum-coords")
"Unrecognized message!"
```

By the way, note that `p2` is a recursive function. We lack a `this` or `self` keyword to explicitly refer to the calling object.

While this might seem like rather surprising behaviour, it is in fact permitted in Python, Ruby, and JavaScript (among others) to add methods to objects and classes at runtime. On the other hand, let's see one example where late binding of an method allows us to do something silly.

Suppose we have a `classes-self` macro, which behaves identically to our class macro, except is allows us to use `self` inside class methods. We use this to define a class method which refers to another method not defined in that class.

```
(require "classes-self.rkt")

(class-self A (x)
  [(f y)
   ; Call method "g" of the calling object
   (+ y ((self "g")))])

(define a (A 1))

(define a1
  (lambda (msg)
    (cond [(equal? msg "g") (a1 "x")]
          [else (a1 msg)])))
```

You can download this source file on the course website to try out the code for yourself. You aren't responsible for the implementation of the macro, though.

We now have a situation in which `a` and `a1` will behave differently when given the message `"f"`. Because of the possibility of late binding, we cannot look at the class definition for `A` and predict what its method `f` does before runtime.

Exercise: determine the outputs of calling `"f"` on each of these three values.

By the way, you've probably learned about something similar previously in your study of object-oriented programming under the name of *dynamic dispatch*. This is a phenomenon where a particular implementation of a method is chosen based on the runtime type of a value. In Java, for example, the compiler can check whether a method is defined for a particular object or not (methods cannot be added at runtime), but in the case of inheritance, the implementation of the method used is determined by the particular subclass to which the object belongs, and this is only possible to determine at runtime.

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.

9. Modify the `class` macro to add support for **private attributes**.
10. Modify the `class` macro to add support for **inheritance** (e.g., by using a `super` keyword).
11. Add support for a `self` keyword, or at least understand the implementation you can download in `classes-self.rkt` well enough to be able to recreate it with just the help of the Internet.

Non-deterministic choice

In the first half of this chapter, we used macros to develop a embed the very familiar concept of class-based objects in Racket. Hopefully this gave you a taste at 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.

For the remainder of this chapter, we will repeat this process to integrate a more unfamiliar concept: non-deterministic choice powered by backtracking search. We will show how to use choice expressions to write purely declarative code that expresses *what* we want to compute, rather than *how* to compute it. Our implementation of such expressions will introduce one final technical aspect of Racket: the continuation, a data structure storing with the control flow at a given point in the execution of a program.

Defining `-<` and `next`

Because you will likely be less familiar with this language feature, we will describe the functionality of the main two expressions before discussing their implementation. The main one is a macro called `-<` (pronounced "amb", short for "ambiguous"), and it behaves as follows.

coined by John McCarthy, inventor of Lisp

- `-<` takes an arbitrary, but non-zero, number of arguments
- It returns the value of first argument.
- If it is passed more than argument, it creates a "choice point," storing the remaining arguments so that they can be accessed, one at a time, by calling a separate function `next` (which we'll also define).
- Once all of the arguments have been evaluated, every time `next` is called again it returns the string `"false."`

Note that we describe its effects but not the order in which these effects take place - this is relevant only to the implementation. Here is an example of how we would like to use `-<` and `next`:

```
> (-< 1 2 (+ 3 4))
1
> (next)
2
> (next)
7
> (next)
"false."
```

However, we are not content to just use `-<` as a top-level expression. We will also be able to use this expression to create choice points embedded within larger expressions. For example:

```
> (+ 2 (-< 10 20 30))
12
> (next)
22
> (next)
32
> (next)
"false."
```

We will even be able to define multiple choice points within the same expression:

```
> (+ (-< 10 20) (-< 2 3))
12
> (next)
13
> (next)
22
> (next)
23
> (next)
"false."
```

The order of the returned values shown here is certainly not the only plausible one. What other orders might you expect?

An initial try at implementation

To get started, let's write a skeleton definition for both `-<` and `next`. Note that we will define `-<` as a macro, but `next` as a function.

As you read through this section, think about why we didn't make `-<` a function.

```
(define next (lambda () "false."))

(define-syntax -<
  (syntax-rules ()
    [(-< <expr1>) <expr1>]
    [(-< <expr1> <expr2> ...) <expr1>]))
```

Note that in our macro, we have separate rules for whether `-<` is given one or more than one argument, in anticipation of the different behaviours. We have also separated `<expr1>` from `<expr2> ...` because we expect to treat `<expr1>` differently from the rest of the arguments.

Why couldn't we just use `[(-< <expr1> ...) <expr1>]`?

You should verify that our current implementation always correctly evaluates and returns the value of the first argument, but it is pretty clear that it fails to "store" the other arguments, leaving `next` to always return `"false."`

In fact, the current macro is an example of an extremely short-circuiting macro; it does not evaluate any of its arguments after the first one!

The fact that this macro can take an arbitrary number of arguments suggests a recursive approach; however, as is often the case when writing recursive programs, we will try to start slow and write code which works for a small, fixed number of arguments. So first suppose we give `-<` two arguments. Then we would like to mutate `next` to make it return the second argument, in addition to returning the first. Here we introduce two new keywords: `begin`, which allows us to evaluate a sequence of expressions rather than just a single expression, and `set!`, which is a mutating primitive in Racket.

`begin` evaluates each of its expressions, but returns only the value of the last one. The prior expressions are only useful for their *side-effects* (like mutation), which explains why we haven't seen `begin` before this.

```
(define-syntax -<
  (syntax-rules ()
    [(-< <expr1>) <expr1>]
    [(-< <expr1> <expr2>)
     (begin
       (set! next (lambda () <expr2>))
       <expr1>)]))
```

This approach has a few problems: it isn't truly recursive (making it hard to generalize), and there's nothing that changes `next` back to returning `"false."` after it has been called.

```
> (-< 1 2)
1
> (next)
2
> (next)
2
```

We can fix both of these problems by making small alterations the two syntax rules:

```
(define-syntax -<
  (syntax-rules ()
    [(-< <expr1>)
     (begin
       ; Update 'next' to return "false."
       (set! next (lambda () "false."))
       <expr1>)]
    [(-< <expr1> <expr2>)
     (begin
       ; Replace <expr2> with (-< <expr2>)
       (set! next (lambda () (-< <expr2>)))
       <expr1>)]))
```

Now `next` is set by the second pattern to call `(-< <expr2>)`, which will not only evaluate and return `<expr2>`, but also set `next` back to `"false."` This generalizes directly to handling an arbitrary number of arguments:

```
(define-syntax -<
  (syntax-rules ()
    [(-< <expr1>)
     (begin
       (set! next (lambda () "false."))
       <expr1>)]
    [(-< <expr1> <expr2> ...)
     (begin
       (set! next (lambda () (-< <expr2> ...)))
       <expr1>)]))

> (-< 1 2 (+ 3 4))
1
> (next)
2
> (next)
7
> (next)
"false."
```

Of course, we aren't done yet. While this implementation is able to "store" the other choices, that's *all* it saves.

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

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

In other words, even though the remaining choices in an `-<` expression are saved in `next`, the computational context in which the expression occurs, i.e., what further steps in the computation must be performed using the result of that expression, is completely lost. To fix this problem, we'll need some way of saving this computational context; luckily, this is precisely what continuations give us.

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

Continuations

In our functional programming to date, analysis of control flow in Racket has been limited to the *eager evaluation order* of nested function call expressions. We accepted as fact that Racket uses eager evaluation, and thus were able to understand, but not influence, the program execution. 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

Contrast this with imperative programming languages, in which not just function calls but also keywords like 'if', 'while', and 'return' directly impact control flow.

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:

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

We know from our understanding of eager evaluation that the program execution proceeds in three function calls:

1. `(* 3 4)` is evaluated to produce `12`
2. `(first '(1 2 3))` is evaluated to produce `1`
3. `(+ 12 1)` is evaluated to produce `13`

For each subexpression – `(* 3 4)`, `(first '(1 2 3))`, and even the entire `(+ (* 3 4) (first '(1 2 3)))` – there is a well-defined notion of "what will happen after that subexpression is evaluated," based solely on Racket's evaluation strategy. More formally, the **continuation of an expression** is a representation of the control flow from immediately after the expression has been evaluated until the end of the evaluation of the enclosing top-level expression.

continuation

Warning: students often incorrectly believe that the continuation of an expression includes the evaluation of that expression. An expression's continuation represents only the control flow *after* its evaluation.

Even though this notation makes the continuation look like a function, it's not a function! A continuation is a direct representation of control flow, but a function is not. More on this later.

For example, the continuation of the subexpression `(* 3 4)` is, in English, "evaluate `(first (1 2 3))` and then add it to the result." Or, as a function: `(lambda (x) (+ x (first '(1 2 3))))`. Similarly, the continuation of the subexpression `(first '(1 2 3))` is `(lambda (x) (+ 12 x))`; note that the subexpression `(* 3 4)` would already be evaluated when we reach this point in the program execution! Finally, the continuation of the whole expression `(+ (* 3 4) (first '(1 2 3)))` is simply "return the value," i.e., the identity "function" `(lambda (x) x)`.

Moreover, since values and names are themselves expressions which must be evaluated by the interpreter, they also have continuations. For example, the continuation of the `4` is `(lambda (x) (+ (* 3 x) (first '(1 2 3))))`, and the continuation of `first` is `(lambda (x) (+ 7 (x '(1 2 3))))`.

Continuations in Racket

So far, continuations sound like a rather abstract representation of control flow. What's quite spectacular about Racket is that it has continuations as a first-class data type, meaning that Racket programs can access and manipulate continuations just like any other value.

We say that Racket *reifies* continuations.

In order to do this, we need to be able to access the current continuation at a point during execution. 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 using some sophisticated macros, this is beyond the scope of the course. Instead, we will use Racket's syntactic form `let/cc` to capture continuations. This has the following usage:

Those interested should look up "continuation-passing style."

```
(let/cc id body)
```

When a `let/cc` expression is evaluated, Racket determines the current continuation of this expression, then does two things:

1. Bind the continuation to `id` (which must be an identifier).
2. Evaluate `body`, which may contain one or more expressions, and return the value of the last expression

For example, here's a slight modification to the previous example:

```
> (+ (* 3 (let/cc cont 4))
     (first '(1 2 3)))
13
```

As before, the continuation of the expression `(let/cc cont 4)` is `(lambda (x) (+ (* 3 x) (first '(1 2 3))))`, and this continuation is bound to the name `cont`. Next, the body of the `let/cc` expression, `4`, gets evaluated and returned, and so the entire expression still evaluates to 13. Before moving on, make sure you understand the following output:

```
> (+ (* 3 (let/cc cont (* 5 20)))
     (first '(1 2 3)))
301
```

Of course, binding the continuation to the name but not referring to it is not very useful. Moreover, `cont` is local to the `let/cc` expression, so we cannot refer to it after the expression has been evaluated. We'll use mutation to save the stored continuation.

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

```
(define global-cont (void))

(+ (* 3
      (let/cc cont
        (set! global-cont cont)
        4))
   (first '(1 2 3)))
```

Just like `begin`, there are two expressions in the body. The first is evaluated just for its side-effect, while the second - `4` - is actually returned.

```
> global-cont
#<continuation>
```

Now that we have saved the continuation, we can *call* it, using the syntax `(<cont> <arg>)`, where `<arg>` is the value to pass to the continuation.

Remember, continuations always take just one argument.

```
> ; global-cont is (lambda (x) (+ (* 3 x) (first '(1 2 3))))
> (global-cont 4)
13
> (global-cont 100)
300
```

Warning: continuations ≠ functions

Even though we've been representing continuations using `lambda` notation, and even though the syntax for calling continuations is identical to function call syntax, continuations are emphatically *not* functions. Function calls operate within Racket's standard control flow mechanism of eager evaluation, while continuation calls interrupt the existing control flow, replacing it with what is stored in the continuation. Here is a simple example to distinguish the difference:

```
(define global-cont (void))

(+ (* 3
      (let/cc cont
        (set! global-cont cont)
        4))
   (first '(1 2 3)))

(define f
  (lambda (x) (+ (* 3 x) (first '(1 2 3)))))
```

After the second expression (with the `let/cc`) is evaluated, it might seem like `global-cont` and `f` do the same thing:

```
> (f 4)
13
> (global-cont 4)
13
> (f 100)
301
> (global-cont 100)
301
```

But they don't!

```
> (+ 1 (f 4))
14
> (+ 1 (global-cont 4))
13
```

The function `f` exists within the larger control flow of evaluating the expression `(+ 1 (f 4))`: the continuation of `(f 4)` is "add `1` to the result". The continuation of `(global-cont 4)` is exactly the same, but calling a continuation discards the current continuation of the call expression, and replaces it the continuation being called. To drive the point home, we can see that continuations even interrupt Racket's normal eager evaluation control flow:

This is similar to, but more powerful than, jump mechanisms like `goto` in C.

```
> (+ (global-cont 4) (/ 1 0))
13
```

Back to `-<`

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! In other words, we need to use a `let/cc`:

```
(define-syntax -<
  (syntax-rules ()
    [(-< <expr1>)
     (begin
       (set! next (lambda () "false."))
       <expr1>)]
    [(-< <expr1> <expr2> ...)
     ; cont is bound to the continuation of the (-< ...) expression
     (let/cc cont
       (set! next (lambda () (-< <expr2> ...)))
       <expr1>)]))
```

Note that we can remove the `begin` because `let/cc` allows an multiple subexpressions in its body.

Inside the second pattern, we bind `cont` to the current continuation, which is precisely the continuation of the `-<` expression. This can difficult to see, so let's actually do a bit of manual macro expansion to see this in action:

```
; The continuation of (-< 1 2) is (lambda (x) (* 4 (+ 1 x)))
(* 4 (+ 1 (-< 1 2)))

; => (Macro expansion)
; cont is bound to that continuation
(* 4 (+ 1
        (let/cc cont
          (set! next (lambda () (-< 2)))
          1)))
```

Of course, our macro is incomplete, as it doesn't "store" `cont` anywhere. Doing this is straight-forward: we need to change the behaviour of `next` so that, when called, it doesn't simply return the value of `(-< expr2 ...)`, but *calls the continuation on this value*:

```
(define-syntax -<
  (syntax-rules ()
    [(-< <expr1>)
     (begin
       (set! next (lambda () "false."))
       <expr1>)]
    [(-< <expr1> <expr2> ...)
     (let/cc cont
       (set! next (lambda () (cont (-< <expr2> ...))))
       <expr1>)]))
```

Since `cont` is free in the function being bound to `next`, we are using a closure to "store" it. Hey, remember those?

And now we get our desired behaviour:

```
> (+ 1 (-< 10 20)) ; cont is (lambda (x) (+ 1 x))
11
> (next)
21
> (next)
"false."
```

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 in Racket.

Multiple choices

Even with this breakthrough, we aren't quite done yet. Our overwriting of `next` means that at any point in time, our program is only aware of only one choice point. In the example below, the second choice clobbers the first:

```
> (+ (-< 10 20) (-< 2 3))
12
> (next)
13
> (next)
"false."
```

How to fix this? Obviously, just adding more global variables (`next1`, `next2`, ...) 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.

Before we modify our macro, we introduce a new global variable `choices` to represent the stack of choices, as well as two helper functions to emulate stack behaviour.

```
(define choices '())  ; initialized to be empty

; Push a new choice onto the stack
(define (push-choice choice)
  (set! choices (cons choice choices)))

; Pop the top choice from the stack
(define (pop-choice)
  (let ([choice (first choices)])
    (set! choices (rest choices))
    choice))
```

Remember, the *front* of the list represents the top of the stack.

With this in mind, we'll modify our `-<` macro to update the `choices` variable rather than `next`.

```
; Now this is a stack of functions
(define choices '())

(define-syntax -<
  (syntax-rules ()
    [(-< <expr1>)
     <expr1>]
    [(-< <expr1> <expr2> ...)
     (let/cc cont
       (push-choice (lambda () (cont (-< <expr2> ...))))
       <expr1>)]))

> (+ (-< 1 2) (-< 3 10))
4
> choices
'(#<procedure> #<procedure>)
```

Note that after evaluating the expression, which involves evaluating *two* choice expressions, we do get two different choices on the stack. Since we know arguments are evaluated left-to-right, we expect the top of the stack to contain the choice for `(-< 10)`, and we can confirm this by calling this function directly:

```
> ((first choices))
11
```

Of course, we don't want to manually call the function by using list methods; we need to re-implement `next` so that it evaluates the top choice on the stack:

```
(define (next)
  (if (empty? choices)
      "false."
      ((first choices))))
```

Unfortunately, this does not achieve quite the behaviour we want:

```
> (next)
"false."
> (+ (-< 1 2) (-< 3 10))
4
> (next)
11
> (next)
11
> (next)
11
> choices
'(#<procedure> #<procedure>)
```

Of course, the problem is that when we use `next` to get the next choice, we do not "consume" that choice, instead leaving it on the stack to be called again. We can easily fix that by popping from the stack, rather than simply accessing the first element. And we're done!

```
(define (next)
  (if (empty? choices)
      "false."
      ((pop-choice))))
```

Remember that `pop-choice` will return a procedure, which must then be called to actually recover the stored choices.

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 continuation of the top choice expression is** `(lambda (x) (+ 1 x))`, because the left choice expression has already been 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)`: `(lambda (x) (+ x (-< 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 function with a new continuation onto the stack.

The `3` is of course added to the `2`, resulting in `5`.

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)
"false."
```

Predicates and Backtracking

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. In this section, we will learn about a new programming paradigm called **logic programming**. Unlike imperative programming, which views computation as a sequence of steps to perform, or functional programming, which uses functions as a atomic unit of computation, logic programming asks a simple question: "is this true?" In logic programming, we will express computations as logical statements, and the *execution* of these computations as searching for values which make these statements true.

logic programming

To start, we will define a **query operator `?-`**, which takes in a unary predicate (i.e., function that takes one argument and returns a boolean) and an expression, and returns the value of the expression if it satisfies the predicate, and `"false."` if it doesn't.

```
(define (?- pred expr)
  (if (pred expr)
      expr
      "false."))

> (?- even? 10)
10
> (?- even? -3)
"false."
```

This function does not look quite impressive, but all of the work we did in the previous sections now pays off: we can call this function with a choice expression!

Indeed, implementing this function should be quite straight-forward for you at this point; if it isn't, go back to previous chapter and do some review!

```
> (?- even? (-< 1 2 3 4))
"false."
> (next)
2
> (next)
"false."
> (next)
"false."
```

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 `(lambda (x) even? x)`, and so each time `next` is called, the next choice gets passed to the query. In the language 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.

Pretty cool, if you ask me.

backtracking

When making queries with choices, it is often the case that we'll only care about the choices which succeed (i.e., satisfy the predicate), and ignore the others. Therefore it would be nice not to see the intermediate `"false."` outputs; we can make one simple change to our predicate to have this available:

```
(define (?- pred expr)
  (if (pred expr)
      expr
      ; Try the next choice
      (next)))

> (?- even? (-< 1 2 3 4))
2
> (next)
4
> (next)
"false."
```

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.

Automated search

Probably the easiest way to appreciate the power that this combination of `?-` and `-<` 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 which satisfy a predicate one at a time, rather than all at once.

We can extend this fairly easily to passing in multiple choice point to search through "all possible combinations" of a pair of expressions.

Though we can achieve analogous behaviour with plain higher-order list functions, the simplicity of this approach is quite beautiful. The one downside is that because our `?-` function only takes unary predicates, we'll need to pass all of the choices into a list of arguments.

```
(define (f lst)
  (equal? 12
          (* (first lst) (second lst))))

> (?- f (list (-< 1 2 3 4 5 6)
              (-< 1 2 3 4 5 6)))
'(2 6)
> (next)
'(3 4)
> (next)
'(4 3)
> (next)
'(6 2)
> (next)
"false."
```

See that? In just five lines of code, we expressed a computation for finding factorizations `12` in a purely declarative way: specifying *what* we wanted to find (in `f`), rather than *how* to compute them, as we would in a procedural or even functional style.

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 let's put 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 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:

`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)
"false."
```

Recursive choices

So far, there has been a clear separation between our predicates and the choice expressions defining the search space of the inputs to those predicates. Sometimes, however, it is possible to define a predicate itself using choices, and so express more complex relationships.

Consider the following problem: you are given a list and a value, and want to insert the value into the list at some position, but don't care which one, and in fact would like to see all possibilities. For example,

given the list `'(1 2 3)` and the value `10`, we would want to see `'(10 1 2 3)`, `'(1 10 2 3)`,`'(1 2 10 3)`, and `'(1 2 3 10)`.

order doesn't matter... yet

How could we implement this in Racket? Essentially, we need to break down the choices that we could make in where to insert the `10`. One possibility is to insert the item at the front of the list, so let's write that down:

```
(define (insert lst val)
  (cons val lst))
```

What are the other possibilities for the new value's position? One might think: "after the first element, after the second, after the third..." but of course this would depend on the length of the list. So instead, we turn to our old friend for structurally handling lists: recursion. When we view this problem in terms of "first" and "rest," there are really only two possibilities: either the new value is inserted at the first position, or it's inserted somewhere in the rest of the list. And this is precisely our implementation (with an extra check if `lst` is empty, because we don't make any choices in that case).

```
(define (insert lst val)
  (if (empty? lst)
      (list val)
      (-< (cons val lst)
          (cons (first lst)
                (insert (rest lst) val)))))
```

We strongly encourage you to run this code for yourself!

And if this doesn't do it for you, we can use `insert` as a helper function to compute the permutations of a list. Unlike the other code examples in this section, we won't spend any time explaining this; instead, we'll let the code speak for itself. It's a worthy exercise for you to play around with the code in DrRacket to understand what's going on.

Hint: think as above about the *logical* definition of a permutation, using the definition of `insert` as a "helper."

```
(define (permutation lst)
  (if (empty? lst)
      '()
      (insert (permutation (rest lst))
              (first lst))))
```

Exercise Break!

12. In our `insert` code, reverse the order of the two choices. What happens, and why?

Haskell and Types

> 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

In 1987, it was decided at the conference *Functional Programming Languages and Computer Architecture* to form a committee to consolidate and standardize existing non-strict functional languages, and so Haskell was born. Though mainly still used in the academic community for research, Haskell has become more widespread as functional programming has become, well, more mainstream. Like Racket and 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. Having learned the basics of functional programming in Racket, we'll use our time with Haskell to explore some of these differences: a completely different evaluation strategy known as *lazy evaluation*, and (in much greater detail) a strong static type system.

The conference is now part of this one: `http://www.icfpconference.org/`.

Submit your best hipster joke here.

Quick Introduction

As usual, the CDF labs have Haskell installed, but if you want to get up and running at home, you'll need to download the Haskell Platform. Like Python and Racket, Haskell comes with its own REPL, called GHCi, which you can run directly from the terminal.

This can be downloaded at `http://www.haskell.org/platform/`.

The command is simply `ghci`.

Haskell is a bit more annoying to experiment with than Racket for a few reasons: the syntax in the REPL for name bindings is slightly different than the syntax you normally use in source files, and you aren't

allowed to have "top-level" expressions in files, so you can't just list a series of expressions and run the file to evaluate of all them. This is a consequence of Haskell source files being *compiled* when loaded into GHCi, rather than simply interpreted, as in Racket.

However, you can make the best of both worlds by defining *names* for functions and primitive values in files, and then loading them into the interpreter to play around with. For example, if you define a function `myFunction` in file `functions.hs`, then after starting GHCi, you can type `:load functions` and then call `myFunction` as much as you want.

Warning: you must start GHCi in the same directory as the source file that you want to load!

alias `:l functions`

Of course, Haskell has some Integrated Development Environments, but I haven't yet found one that I love; the CDF labs don't have any installed, as far as I know.

Basic Syntax

```
1  -- This is a comment.

2  -- primitive data types
3  1, -10.555
4  True, False
5  "Hello, World"             -- String, double quotes
6  'a'                        -- Char, single quotes
7  ()                         -- "null" value, called "unit"

8  -- built-in binary operators, written infix
9  1 + 3
10 2 == 15
11 2 /= 15                    -- "not equal" operator
12 True && (3 >= -0.112)

13 -- lists
14 []
15 [1,2,3,4]
16 1:[2,3,4]                  -- cons!

17 -- function application does not use parentheses
18 max 3 4                    -- 4
19 head [1,2,3,4]             -- 1
20 tail [1,2,3,4]             -- [2,3,4]
21 length [1,2,3,4]           -- 4
22 take 3 [10,20,30,40,50]    -- [10,20,30]
23 [1,2,3] ++ [4,5,6]         -- [1,2,3,4,5,6]
```

```haskell
-- Names are immutable; definition happens with =.
-- Haskell is pure - no mutation allowed!
x = 4
y = x * 3
x = 5                       -- ERROR: reassignment!

-- Like Racket, 'if' is an expression (has a value).
if x > 0 then 23 else 200
```

Note that we use the familiar = for assignment. While in other programming languages this causes confusion to beginners because of the difference in meaning from mathematics, there is no such tension in Haskell. When you see "equals," this is equality in the mathematical sense!

Function Creation

As in Racket, functions are just another type of value in Haskell.

```haskell
double x = x + x
sumOfSquares x y = x * x + y * y
absVal x =
    if x >= 0 then x else -x

-- Because functions are values, you can also bind them directly.
foo = double
foo 12
```

It shouldn't come as a surprise that lambda syntax exists in Haskell too, and can be used both in binding function names and for anonymous functions. However, as with Racket, avoid using lambda notation to define named functions; the other syntax is nicer, and has a very useful syntax extension we'll see shortly.

```haskell
> (\x -> x + x) 4
8
> let double2 = \x -> x + x
> double2 4
8
> let sumOfSquares2 = \x y -> x*x + y*y
```

This illustrates the use of `let` in GHCi to bind names.

And here's an example of a familiar higher-order function.

```haskell
> let makeAdder x = \y -> x + y
> (makeAdder 10) 5
15
```

Pattern Matching

In most languages, we use conditional branching expressions to implement functions which have different behaviours depending on their inputs. While Haskell can certainly do this too, it also uses *pattern match-*

ing as an elegant tool for defining different function behaviours. For example:

```
comment 7 = "Lucky number 7!"
comment 13 = "Unlucky!!!"
comment _ = "Meh"
```

What's going on, have we defined the function `comment` three times? No. All three lines define the same function `comment` through a series of pattern rules. When `comment` is called on an argument `x`, `x` is matched against the three rules, and stops at the *first* matching rule, and evaluates the right side of that. An example:

Notice here that this is strictly not mathematically pure, as the order of pattern matching goes top-down, just like Racket macros.

```
> comment 7
"Lucky number 7!"
> comment 5
"Meh"
```

Haskell provides *guard* syntax for more complex branching patterns that we don't have time to get into here.

The underscore matches any expression. You can think of this as an "else" clause that always gets executed if it is reached. If you omit the underscore and the function is called on an argument with no matching pattern, a runtime error is raised.

If Haskell only allowed you to pattern match on value equality, this would not be a very interesting language feature. However, Haskell also provides the ability to pattern match on the emphstructure of values. Let's see an example of this on lists; we define a function `second` that returns the second item in a list.

```
second [] = undefined
second [x] = undefined
second (x:y:rest) = y
```

`undefined` is a built-in Haskell function that raises an error. We won't go much into error handling in this course, though.

In this example, the first line defines the behaviour of `second` on an empty list, and the second defines the behaviour of `second` on a list of length `1` (single element `x`). The third line is the interesting one: the pattern `(x:y:rest)` uses the cons operator twice; it is equivalent to `(x:(y:rest))`. Then the three names `x`, `y`, and `rest` are bound to the first, second, and remaining items of the input list, respectively. This allows us to use `y` in the body of the function to return the second element. Note that this pattern will only match on lists of length two or more.

In other words, the cons operator in Haskell is *right*-associative.

A note about style: it is considered bad form to use a name to bind an argument or parts of an argument if that name is never used in the body of the function. In this case, it is better to use an underscore to match the subexpression that is unused. Thus we should rewrite `second` as follows:

```
second [] = undefined
second [_] = undefined
second (_:y:_) = y
```

The ability to pattern match on the structures of values is one of the most practical features of Haskell. Not only does with work with built-in types like lists, but also with user-defined types, as we'll see later in this chapter.

Let

Like Racket, Haskell allows binding of local names in expressions everywhere expressions are allowed. Unlike Racket, Haskell doesn't have a `let*`, and so you can freely use bound names in later bindings in the same `let` expression.

In fact, Haskell doesn't have a `letrec` either, and so local names can be bound to recursive and even mutually recursive definitions!

```
f x =
  let y = x * x
      z = y + 3
      a = "Hi"
  in x + y + z + length a
```

Infix operators

Even though Haskell uses infix notation for the standard operators like +, it turns out this is nothing more than special syntax for function application. Just like Racket, Haskell makes no *semantic* difference between the standard operators and regular functions, but unlike Racket, it gives them a special syntax. However, because operators are simply functions, they can be used in prefix syntax, with just a minor modification.

```
-- Standard infix operator.
-- The real name of the + operator is (+).
-- The "(" and ")" are used to signal that
-- the function is meant to be used as infix.
> 3 + 4
7
> (+) 4 5
9
```

```
-- Use backticks `...` to use binary functions as infix.
> let add x y = x + y
> add 3 4
7
> 5 `add` 10
15
```

We can also define our own infix binary operators:

Unlike regular functions, operators must consist of special punctuation marks. See the Haskell documentation for details.

```
> let (+-+) x y = 3 * x + 4 * y
> (+-+) 1 2
11
> 1 +-+ 2
11
```

Two higher-order binary operators

Two very useful operators in Haskell that you'll often encounter are `(.)` and `($)`. The first is simply function composition:

```
-- f . g is "f composed with g": function that applies g, then f.
f . g = \x -> f (g x)

> let func = (\x -> x + 1) . (\x -> x * 3)
> func 10
31
```

The second operator takes two arguments, a unary function and a value, and applies the function to the value.

You might recall `apply-unary` from the first chapter.

```
f $ x = f x
```

Given that Haskell's function application syntax is already so lightweight, you might wonder what purpose the `($)` function could possibly serve. Because function application has the highest precedence in Haskell, we often need to use parentheses to separate a function and its argument(s) – we saw this in `f (g x)` above – which can lead to many parentheses. Because `($)` has extremely low precedence and is right-associative, it provides an alternate way to correctly group a function and its arguments.

Remember Racket?

```
> f (g x)
-- equivalent to
> f $ g x
```

Our recommendation for beginners is not to use this operator, since it is an extra thing to remember and is unfamiliar. On the other hand, you might see this "in the wild," so we wanted to show it to you as a neat built-in binary operator.

Folding and Laziness

Let's greet our old friends, the higher-order list functions. Of course, these are just as important and ubiquitous in Haskell as Racket.

```
map (\x -> x + 1) [1,2,3,4]              -- [2,3,4,5]
filter (\x -> x `mod` 2 == 0) [1,2,3,4]  -- [2,4]
foldl max 0 [4,3,3,10]                   -- 10
```

We haven't talked about efficiency too much in this course, but there is one time/space issue that Haskell is so notorious for that it would be a travesty for me not to talk about it here. Let's start with the standard `foldr`, which folds a value across a list, starting at the end.

This is in contrast with the `foldl` we saw earlier with Racket, which folds a value across a list starting at the front.

```
foldr _ init [] = init
foldr f init (x:xs) = f x (foldr f init xs)
```

Note the neat use of list pattern matching `(x:xs)`, and the underscore.

You can think about this as taking `init`, and combining it with the last element in the list, then taking the result and combining it with the second last element, etc.:

```
-- foldr (+) 0 [1,2,3,4]
-- (1 + (2 + 3 + (4 + 0)))
```

Unfortunately, this is not at all tail-recursive, and so suffers from the dreaded stack overflow if we evaluate the following expression:

```
foldr max 0 [1..1000000000]
```

This problem is a result of requiring the recursive call to be evaluated before `f` can be applied. Thus when the list is very long, one recursive call is pushed onto the stack for each element. Recall that the left fold from Racket *was* tail-recursive, as it computed an intermediate value and passed that directly to the recursive call. In fact, Haskell does the exact same thing:

Though we'll soon see that the recursive call need not always happen!

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

However, when we try running `foldl max 0 [1..100000000]`, we *still* get a memory error! To understand the problem, we need to look at how Haskell's unusual evaluation order.

Evaluation order revisited: lazy evaluation

Unlike most programming languages you've encountered so far, Haskell uses **lazy evaluation** rather than eager evaluation. This means that when expressions are passed as arguments to functions, they are *not* evaluated right away, before the body of the function executes. Instead, they are only evaluated if and when they are used, and so if an argument is never used, it is never evaluated. Consider the following example:

lazy evaluation
Technically, Haskell's evaluation order is *non-strict*, not necessarily *lazy*, but this is a technical point that we'll ignore here.

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

The function `onlyFirst` only cares about its first argument, because its second argument is never used to evaluate the function call. So in the first call, the argument subexpression `(head [])` is never evaluated.

The big deal we made about short-circuiting vs. non-short-circuiting in Racket is rendered moot here. All functions in Haskell are short-circuiting, because of lazy evaluation.

Indeed, the desire to extend short-circuiting to new functions, or more generally control evaluation behaviour, was another main motivation of macros.

```
> let myAnd x y = if x then y else False
> myAnd False (head [])                    -- this didn't work in Racket
False
```

Back to folds

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 `foldl`, laziness means that the `let` bindings don't actually get evaluated until absolutely necessary – when the end of the list is reached – and this is what causes the memory error. Because laziness presents its own unique problems, Haskell provides ways of explicitly *forcing* strict evaluation. One of these is the function `seq x y`, which forces `x` to be evaluated, and then returns `y` (with or without evaluating it). We can use this in our definition of `foldl` to force the let binding to be evaluated before the recursive call:

```
foldl' f init [] = init
foldl' f init (x:xs) = let init' = f init x
                       in seq init' (foldl f init' xs)
```

And finally, we get the desired space-efficient behaviour: `foldl' max 0 [1..100000000]` doesn't cause a stack overflow.

Though it will take some time to evaluate!

Oh, and why did we say Haskell is notorious for this issue? The more naturally-named function, `foldl`, behaves this way and so is generally discouraged, with the more awkwardly-named `foldl'` recommended in its place. Go figure.

`foldl'` must also be imported from the `Data.List` module.

Lazy to Infinity and Beyond!

One of the flashiest consequences of lazy evaluation is the ability to construct and manipulate infinite lists (and other data structures) using finite time and space. As long as it has recursive structure (as lists do), we'll always be able to compute any finite part of it we want. Here are some simple examples.

```
myRepeat x = x : myRepeat x
> take 3 (myRepeat 6)
[6,6,6]

plus1 x = x + 1
nats = 0 : map plus1 nats
> take nats 5
[0,1,2,3,4]
```

Later, we'll see a much more concise way of representing `plus1`.

What's going on with that `nats` definition? Consider a simpler case: `take nats 1`. This is supposed to return a list containing the first element of `nats`. That's easy: right from the definition, the first element is 0. We didn't need to look at the recursive part at all!

But now what about `take nats 2`? Here, we need both the first and second element of nats. We know the first element is 0; what's the second? Due to referential transparency, it's actually quite easy to see what the second element is:

```
nats = 0 : map plus1 nats
-- →
nats = 0 : map plus1 (0 : map plus1 nats)
-- → (using the definition of map)
nats = 0 : (plus1 0) : map plus1 (map plus1 nats)
-- →
nats = 0 : 1 : map plus1 (map plus1 nats)
```

So the first two elements are 0 and 1, and once again we do not need to worry about anything past the first two elements. Similarly, the third element of `nats` is `plus1 (plus1 0)`, which is simply 2, and so on.

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.

Exercise: how does this work?

```
-- an example of zipWith, which combines two lists element-wise
zipWith (*) [1,2,3] [4,5,6] -- 4 10 18

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

Exercise Break!

1. Reimplement all of the list functions from the exercises in the Racket chapter, this time in Haskell, with or without using explicit recursion. For extra practice, use both!
2. Make sure you understand the meaning of `seq`, and how to use it correctly.
3. 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.
4. Define an infinite list containing all rational numbers.
5. (Joke) Define an infinite list containing all real numbers. Bonus: why is this funny?
6. Look up **Pascal's Triangle**. Represent this structure in Haskell.

Types in programming

Here's a problematic Racket function.

```
(define (f x)
  (+ x (first x)))
```

Trying to run this function on *any* input will fail! This is a *type error*: the parameter `x` cannot simultaneously be a number and a list.

Even though it has been hidden from you so far, Haskell treats *types* much more like Java than like either Python or Racket. That is, Haskell

and Java are **statically-typed**, but Python and Racket are **dynamically-typed**. Let's explore what that means.

You are familiar with the intuitive notion of the type of some data, but we'll give a definition that takes a more general approach. A **type** is simply an annotation on a value or identifier that carries some semantic meaning, often in the form of constraints on that value/identifier. We typically see types as constraining two things: (1) the possible values of the data, and (2) the possible *functions* that can operate on the data. For example, if we know that `x` is an integer, then we would be shocked if `x` took on the value `"Hello, world!"`, and equally surprised if we tried to write `x[0]` or `(first x)`, as these behaviours are not defined on integers.

type

Python uses **duck typing**, a form of dynamic typing that uses types *only* as restrictions on what functions are defined for the value. "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck." –James Whitcomb Riley

Strong typing vs. weak typing

One of the more confused notions is the difference between static/-dynamic typing and strong/weak typing. We haven't yet discussed static/dynamic typing yet, but we'll take a bit of time to explain the 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 at any point during the execution of a program. Keep in mind that this isn't strictly necessary: data of every type is stored in memory as 0's and 1's, and it is possible that the types are not stored at all!

strong typing

Note that this does *not* imply the type of a value never changes during the run of a program. 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 perfectly valid, raising no runtime errors. Many languages interpret 6 as the string `"6"`, and concatenate the strings to obtain the string `"56"`. A Java feature called *autoboxing* allows primitive values to be coerced into their corresponding wrapper classes (e.g., `int` to `Integer`) during runtime.

weak typing

Be aware of your biases! 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:

```c
int main(void) {
  printf("56" + 1);
}
```

We don't want to ruin the fun – trying running this program yourself!

Depending on who you ask, different levels of type coercion might warrant the title of strong or weak typing. For example, basically every programming language successfully adds `3.0 + 4` without producing a type error. So the takeaway message is that strong/weak typing is not a binary property, but set of nuanced rules about what types can be coerced and when. It goes without saying that mastering these rules is important for any programming language.

Static typing vs. 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*?" It is the answer to this question that differentiates static from dynamic typing. In **statically-typed** languages, the type of every expression and name is determined directly from the source code at compile time, *before any code is actually executed*. Variables typically have a fixed type; even in languages that permit mutation, variables cannot be reassigned to values of a different type than the one they started with. For example, in Java types must be specified explicitly in variable declarations, and it is a compile error if a variable is ever assigned a value of the wrong type.

static typing

```
int x = 5; Point p = Point(1,2);
```

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 for the during of the program's run, we can then drop the runtime type checks when functions are called.

In contrast, **dynamically-typed** languages do not analyse types during compilation, but instead the types of values are checked during runtime (e.g., when a function is called). Any type errors are raised only when the program is run, as in the initial Racket example:

dynamic typing

```
1 (define (f x)
2   (+ x (first x)))
```

As we'll discuss in the next section, statically-typed languages perform type checking as a compile time operation, 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 which 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 write code without any type constraints.

We'll see a few examples of this when we study Haskell's type system in the next section.

Types in Haskell

In Haskell, types are denoted by capitalized names like `Bool`, `Char`, and `Integer`. In GHCi, you can see the type of every expression and value with the `:type` command. Some examples follow; since Haskell's numeric types are a bit sophisticated, we're deferring them until later.

This is why identifiers must begin with a lowercase letter.

alias `:t`

```
> :t True
True :: Bool
> :t "Hello"
"Hello" :: [Char]
> :t ()
() :: ()
```

Yes, the type of `()` is `()`, also called "unit." While this may seem confusing at first, it should be clear from the context context when you read `()` whether it is being used as a value or as a type.

Haskell also provides a type synonym for `[Char]` called `String` - more on type synonyms later.

Note the square brackets surrounding the `Char` indicates a *list*: like C, Haskell interprets strings simply as a list of characters. 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. This restriction also applies to `if` expressions: both the `then` and `else` subexpressions must have the same type!

```
> :t (if 3 > 1 then "hi" else "bye")
(if 3 > 1 then "hi" else "bye") :: [Char]
```

Note that the other type restriction on subexpressions is an obvious one: the `if` condition must have type `Bool`. There is no "truthy/falsey" type coercion.

Function types

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 outputs a `Bool` value. Note that the very existence of a type signature for 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 returned either a list or `#f`.

Type annotations

When we define our own functions (or other values), we can specify our own types using a **type annotation**.

type annotation

```
1 isEven :: Integer -> Bool
2 isEven x = (x `mod` 2) == 0

3 > :t isEven
4 isEven :: Integer -> Bool
```

Wait. The line `isEven :: Integer -> Bool` does not look a like a comment; if we change the type annotation to `isEven :: Char -> Bool`, the program will fail to compile. On the other hand, we've defined other names so far without type annotations.

One of the common naïve complaints about static typing is the amount of boilerplate code it forces you to write. Think of Java, your favourite statically-typed language. Types are everywhere! What gives? Are types actually optional in Haskell?

Type Inference

The missing ingredient is actually quite obvious (and something you've probably realised yourself in your career so far): you can usually determine the types of values (including functions) *by inspection, based on how they are used in the code*. And so by carefully parsing the source code at compile time, Haskell does indeed assign fixed types to all expressions, variables, and functions – and it doesn't need your help to do it!

Well, at least most of the time.

This ability to determine types at compile time without explicit type annotations or executing code is called **type inference**, and is one of Haskell's most powerful features. With it, Haskell provides all of the safety associated with static typing *without* the boilerplate. However, keep in mind that every language that has type inference *also* supports type annotations, allowing programmers to explicitly state a type if desired.

type inference

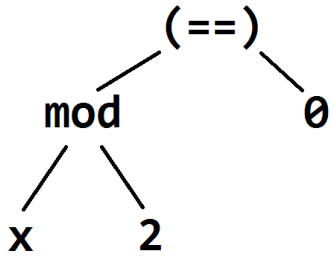
To infer types, Haskell uses an algorithm based on the Hindley-Milner type system, a truly beautiful invention in it's own right, dating back to the 1970s. We won't go into the algorithmic details in this course, but to get a taste of the kind of reasoning involved, let's take a

The type inference algorithm from this system, **Algorithm W**, runs almost in *linear* time!

second look at `isEven`.

```
isEven x = (x `mod` 2) == 0
```

When doing type inference, it is helpful to break apart compound expressions into subexpressions in an *abstract syntax tree*. The abstract syntax tree is one of the fundamental steps in modern compilers, regardless of type inference. So you really can view type inference as simply an additional step to the compilation process.



Remember that the "function name" of an operator like == is obtained by enclosing it in parentheses: `(==)`.

First we consider `mod`; since we know (by consulting the Haskell documentation) that `mod` has two `Integer` parameters, we can *infer* that the type of `x` is `Integer`. Next we consider `(==)`, which must take two arguments *of the same type*, and then returns a `Bool`. So the type inference algorithm checks that the return value of `mod` has the same type as `0` (which it does), and then infers that the whole expression has a `Bool` result.

Let's return to the original "bad type" Racket example, transliterated into Haskell.

```
badType x = if x then x + 1 else head x
```

In Haskell, this definition produces an error at *compile time*, because the type of `x` cannot be inferred: from the `if x`, `x` must have type `Bool`; but later, its use in other operations means it must have a numeric or list type! Each time the variable `x` is used, some constraint is put on its type; the type inference algorithm must *unify* all of these constraints by finding a type for `x` that satisfies them. If this unification succeeds, the program is successfully type-checked; if it fails, Haskell emits an error.

```
> True && "Hello"
...
  Couldn't match expected type 'Bool' with actual type '[Char]'
```

Finally, an interesting connection to our earlier discussion of lazy evaluation. Type inference occurs at compile time, before any evaluation occurs. One can say that type inference is lazy in the strongest possible way! Recall an earlier example:

```
> :t (if 3 > 1 then "hi" else "bye")
(if 3 > 1 then "hi" else "bye") :: [Char]
```

You might have wondered why the output was not `"hi" :: [Char]`; the reason is indeed because the expression is not evaluated to determine its type. This may seem a little strange for this example because

not just the type but also the value of the expression can be determined at compile time. But consider a function `g :: Integer -> Bool` which is exponential time; evaluating the following expression is much more expensive than determining its type!

```
> :t (if g 5000 then "hi" else "bye")
(if g 5000 then "hi" else "bye") :: [Char]
```

Multi-Parameter Functions and Currying

So far, we've only considered functions which take in a single parameter. 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 is `Bool -> (Bool -> Bool)`. This is quite suggestive: somehow 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 *composition of single-valued functions*.

currying

The *second* great notion named after the logician Haskell Curry (1900-1982).

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`.

Interesting side note: Haskell is able to determine even the *number of parameters* that `newF` should take, without you explicitly writing them in the definition!

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 `(&&) True True` might look like a single function application on two arguments, but its true syntactic form parsed by Haskell is `((&&) True) True`, where `(&&) True` is evaluated to create a new function, which is then applied to `False`. In other words, we can define the `and` function in two ways:

function application is *left*-associative

```haskell
-- 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! Similarly, we could define a function that takes three arguments in four equivalent ways:

```haskell
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))
```

Sectioning

All multi-parameter Haskell functions can be curried, including binary operators. We just saw an example of partial application of the `(&&)` function, but Haskell also provides a special syntax for currying operators called **sectioning**.

sectioning

```haskell
f = (True &&)
-- equivalent to f x = True && x
g = (&& True)
-- equivalent to g x = x && True
```

Note that unlike regular 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.

Of course, this only matters for non-symmetric operators.

One last example

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.

```haskell
addToAll n lst = map (\x -> x + n) lst

-- Rather than creating a new anonymous function using a lambda
-- expression, we can just use partial application of (+):
addToAll2 n lst = map (+ n) lst

-- And rather than writing lst twice, we can simply curry map:
addToAll3 n = map (+ n)
```

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."

Type Variables and Polymorphism

As we mentioned earlier, Haskell lists must contain elements of the same type:

```haskell
> :t [True, False, True]
[True, False, True] :: [Bool]
> :t [(&&) True, not, (|| False)]
[(&&) True, not, (||) False] :: [Bool -> Bool]
```

However, lists are also quite generic: as long as all of the elements have the same type, that single type could be *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 the first element in the list, but the type of the output depends of the type of the elements in the list, and we just said this could be anything!

```haskell
> :t (head [True, False, True])
(head [True, False, True]) :: Bool
> :t (head "abc")
(head "abc") :: Char

> :t head
head :: [a] -> a
```

type variable

In the type expression `[a] -> a`, `a` is a **type variable**, which is an identifier in a type expression that can be instantiated to any type. This type signature tells us that `head` works on any type of the form `[a]`, i.e., any list type. Similarly, the type of tail is `[a] -> [a]`.

Let's look at a more complicated example. As we briefly mentioned earlier, Haskell has a list `filter` function, which can be used as follows:

```
> filter (>= 1) [10,0,-5,3]
[10,3]
```

Another instance of sectioning: `(>= 1)` is equivalent to `\x -> x >= 1`.

We can think of filter in the "usual" sense of taking two arguments (putting currying aside). The second argument is a list of `a`'s, or `[a]`. The first argument must be a function mapping each `a` to a `Bool`, i.e., `a -> Bool`. `filter` outputs a list of elements that have the same type as the original list (since these were elements in the original list). Putting this together with currying, we get that the type of `filter` is

```
filter :: (a -> Bool) -> [a] -> [a]
```

Generics, templates, and polymorphism

If you've programmed in Java or C++, type variables shouldn't be novel. Indeed, Java's standard `ArrayList` follows basically the same principle:

in fact, Java's entire Collections framework

```
class ArrayList<T> {
  ...
}

public static void main(String[] args) {
  ArrayList<Integer> ints = new ArrayList<Integer>();
}
```

Interestingly, Java 7 introduced a limited form of type inference with the *diamond operator*, which allows the "`Integer`" in the constructor to be omitted.

In the class definition of `ArrayList`, `T` is a type variable, and is explicitly instantiated as `Integer` when `ints` is created. Note that this would result in a compile-time error if not explicitly instantiated, something that Haskell's powerful type inference lets us avoid.

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++.

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)
}
```

All three of Haskell lists, Java `ArrayLists`, and the above C++ `f` function are **polymorphic**, meaning they can act of different types of arguments. All three of these examples use a particular type of polymorphism known as **parametric** or **generic polymorphism**, in which the *types* of the entities (functions or classes) vary depending on one or more type variables, but whose behaviour remains the same. Note the term "generic": such functions ignore the actual values of the type variables. Consider common list functions like `head`, `tail`, and `filter`: they all do the same thing, independent on whether the input is a list of numbers, of strings, or even of functions.

The term "polymorphism" comes from the greek words *polys*, meaning "many", and *morphe*, meaning "form, shape."

parametric polymorphism

Thought experiment: suppose you are given a value of type `a`, and you know nothing else about it. What could you possibly do with that value?

Ad hoc and subtype polymorphism

There are two other important types of polymorphism in programming languages, both of which you have seen before in your programming experiences. The first is **ad hoc polymorphism**, in which the same function name can be explicitly given different behaviours depending on the types and/or number of the arguments. This is commonly introduced as *function overloading* in Java. In contrast to parametric polymorphism, in which the behaviour of the function on different types of arguments is identical, ad hoc polymorphism enables radically different behaviour on a per-type basis.

ad hoc polymorphism

Or at least, that's what it was called when I learned it.

```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."; }
```

By the way, this is explicitly disallowed in Haskell. Though we've seen we can use different pattern matching rules to define a single function `f`, each rule must operate on the *same number of parameters*, and the parameters of each rule must also have the *same types*.

Even though functions cannot be overloaded, we'll later see a different form of ad hoc polymorphism in Haskell.

The other main kind of polymorphism is called **subtype polymorphism**. In languages that support some kind of subtyping (for instance, object-oriented programming languages with inheritance), it is possible to declare subtype relationships between types. If, for example, we say that `B` is a subtype of `A`, then we expect that functions which operate on a values of type `A` work equally well on values of type `B`. Here's a simple example using Python's class system.

This is essentially the *Liskov substitution principle*.

```python
class A:
    def f(self):
        return 'Class A'

    def g(self):
        return "Hey there, I'm from "

# B inherits from A, i.e., B is a subtype of A
class B(A):
    # Overloaded method
    def f(self):
        return 'Class B'

# By calling g, this method really expects class A objects.
# However, due to Python inheritance, subtype polymorphism
# ensures that this also works on class B objects
def greet(obj):
    return obj.g() + obj.f()

if __name__ == '__main__':
    a = A()
    b = B()
    print(greet(a))
    print(greet(b))
```

Note that though the function `greet` is subtype polymorphic, it isn't parametric polymorphic, as it certainly fails on objects of other types. Nor is it an example of ad hoc polymorphism, as it isn't defined to have any other behaviour on other types/numbers of parameters.

Exercise Break!

7. Infer the types of each of the following Haskell values (note: functions are also values). For simplicity, you may assume that the only numeric type is `Integer`.

```haskell
x = if 3 > 5 then "Hi" else "Bye"
f = \a -> a + 3
y = [\a -> a + 3, \b -> 5, \c -> c * c * c]
z a = if (head a) > 5 then tail a else []
w lst = head (tail lst)
w1 lst = head (head lst)
w2 lst = (head (head (lst))) * 4
```

```haskell
g a b = a + 3 * b
h a b c = if a then b + 3 else c + 3
app g h = g h
app2 g h x = g h x  -- Remember associativity!
p = app not         -- same app as above
q = app not True
r = (+3)
```

8. Make sure you can infer the types of `map`, `filter`, `foldl`, and `foldr`.
9. Go back over previous list exercises, and infer the types of each one! (Note that you should be able to do this without actually implementing any functions.)
10. Rewrite the definition of `nats`, using sectioning.
11. Use currying to write a function that squares every integer in a list.
12. Use currying to write a function that takes a list of lists of numbers, and appends 1 to the front of every list. (This should sound familiar from `subsets-k`).
13. Explain the difference between the functions `(/2)` and `(2/)`.

User-Defined Types

Now that we have a solid understanding of Haskell's type system, let's see how to define and use our own types. The running example we'll use in this section is a set of data 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.

```haskell
data Point = Point Float Float
```

You can probably guess what this does: on the left side, we are defining a new **data 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.

data type

value constructor

```haskell
> let 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*, in which we define names and evaluate expressions, and the *type language* used to represent the types of these expressions. We have seen some artefacts 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 data type declaration is a different instance of this mingling. The left side is a type expression, `data Point`, which defines a new type called `Point`. The right side, `Point Float Float`, defines the function used to create the type and specifies the type of its argument(s). The type and its value constructor can have different names, as shown in the example below.

```
data Point = MyPoint Float Float
> let p = MyPoint 3 4
> :t p
p :: Point
> :t MyPoint
MyPoint :: Float -> Float -> Point
```

Operating on points

Of course, we would like to define functions acting on this new type. However, approaching this naïvely leads to a pickle:

```
-- Compute the distance between two points
distance :: Point -> Point -> Float
distance p1 p2 = ???
```

The problem is clear: how can we access the `x` and `y` 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*.

In fact, the cons operator `(:)` is a value constructor for the 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)
```

Note that this function definition perfectly matches how we would call the function:

```
> distance (Point 3 4) (Point 1 2)
```

Here's another example where we use the value constructor to 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 best way of writing this functions is simply:

```
makePoints3 = zipWith Point
```

Multiple Value Constructors

In typical object-oriented languages, the *constructor* of a class is a special method, and its significance is enforced syntactically (in part) by having its name be the same as the class name. Therefore it might not have been surprising that the value constructor of `Point` was also named `Point`, although we've already pointed out that this is not necessarily the case. In this section, however, we'll explore a novel aspect of Haskell types: creating instances of types with different value constructors.

```
data Shape = Circle Point Float |
             Rectangle Point Point
```

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.

```
> let 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]
```

Note that the types are based on `Shape`, and *not* `Circle` or `Rectangle`!

Here's another example of using pattern matching to write functions operating on `Shapes`.

```
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
```

Type classes

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** in Haskell defines a set of functions or operations that can operate on a type, similar to a Java interface. We say that a type is a **member** of a type class if the programmer has implemented the type class' functions for that type, akin to implementing an interface. Most built-in types in Haskell are **members** of the `Show` type class, which the interpreter requires to display instances of those types.

type class

member

One notable except is function types; try entering just `(+)` into ghci

To make our class `Point` a member of `Show`, we need to know the functions `Point` should implement! Luckily, there's only one: `show`, which takes an instance of our class and returns a string. Here's the syntax for making `Point` a member of `Show`.

```
instance Show Point where
  show (Point x y) = "(" ++ (show x) ++ ", " ++ (show y) ++ ")"

> let p = Point 1 3
> p
(1.0, 3.0)
```

Haskell also provides a "default" implementation for `show` (and other type classes), which we can use to quickly create type class instances of the basic type classes using the `deriving` keyword: `data Point = Point Float Float deriving Show`.

Here are two built-in type classes for comparing values:

- `Eq`: members of this type class support the `(==)` and `(/=)` functions to test for equality. Note that only `(==)` should be implemented; `(/=)` is implemented by default in terms of it.
- `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.

Class constraints and polymorphism

Let's take a step back from our 2D geometric types. Suppose we want to use type classes more generally – for example, to check if an argument is between two other arguments, using whatever comparison functions their type implements. The implementation is quite straightforward:

```
between x low high = low <= x && x <= high
```

The interesting question is, what is the type of `between`? Your first instinct may to say some sort of numeric type, but in Haskell many other types – like lists and even booleans – can also be compared. In fact, any type that is a member of `Ord` type class should be supported by `between`. In Haskell, `(<=)` must take two arguments of the same type, and so `x`, `low`, and `high` must have the same type. A good guess for the type of `between` would be `between :: a -> a -> a -> Bool`. However, this incorrectly allows types that aren't members of `Ord` – such as function types – to be used. We must specify that `a` is a member of `Ord`, which we can do using the following syntax. The `Ord a =>` is called a **type class constraint**, and indicates that the type variable `a` can only be instantiated to a member of the `Ord` type class.

```
False <= True
```

type class constraint

By the way, a type can have multiple type constraints, on both the same or different type parameters. The syntax for this is `Eq a, Ord b => ...`.

```
between :: Ord a => a -> a -> a -> Bool
```

While type classes are useful in abstracting common behaviour across different types, every member of a type class carries its own implementation of each function. Thus type classes give Haskell a way of supporting ad hoc polymorphism, allowing different behaviours for each type depending on the these underlying implementations.

Moreover, this kind of ad hoc polymorphism is signaled by a class constraint in the function's type. This is a significant difference from Java, in which the only way to determine if a function is ad hoc polymorphic is by scanning all of the function signatures in the source code. This recurring theme of types encoding information about a value is one of the most fundamental ideas of this chapter.

or using an IDE

Numeric types

Now that we know what a class constraint is, we can finally, *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 which supports numeric beharious, and which concrete types like `Integer` and `Float` belong to. There are three main numeric type classes in Haskell:

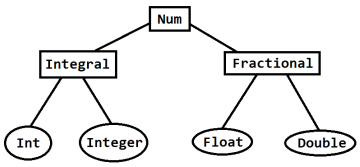
1. `Num` provides basic arithmetic operations such as `(+)`, `(-)`, and `(*)`. Notably, it doesn't provide a division function, as this is handled differently by its subclasses.
2. `Integral` represents integers, and provides the `div` (integer division) and `mod` functions.

```
> :t div
div :: Integral a => a -> a -> a
```

3. `Fractional` represents non-integral numbers, and provides the standard `(/)` operator, among others.

```
> :t (/)
(/) :: Fractional a => a -> a -> a
```

A diagram of Haskell's numeric types is shown. Note that the concrete types are in ovals, distinguishing them from the various type classes in rectangles.



So that's it for the numeric functions. Let's briefly talk about literals, because they're a little subtle. Let's inspect the type of `1` in Haskell:

```
> :t 1
1 :: Num a => a
```

Huh, it's not `Integer`? That's right, `1` is a *polymorphic* value: it can take on any numeric type, depending on its context! Here are two more examples of polymorphic literals:

Under the hood, each time an integer literal appears, it it implicitly passed to `fromInteger :: Num a => Integer -> a`, which converts the literal's type.

```
> :t 2.3
2.3 :: Fractional a => a
> :t []
[] :: [a]
```

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 in the `Num` type

class, and the type of `2.3` in the `Fractional` type class, and that these two types must be equal. It can correctly instantiate the types so that both `1` and `2.3` are treated as `Doubles`, and hence the type check passes.

Error handling with `Maybe`

Here's an interesting type that really illustrates the purpose of having multiple value 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 with exceptions or checks 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, or distinguish between valid and erroneous return values, leaving it to the programmer.

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.

Java 8 adopted this approach with its **Optional** types.

We start with a simple illustration of this idea with `MaybeInt`, which represents the result of some computation that might return an `Integer`, or might fail:

```haskell
data MaybeInt = Success Integer |
                Failure deriving Show

-- Divide x by y, but fail if y == 0
safeDiv :: Integer -> Integer -> MaybeInt
safeDiv _ 0 = Failure
safeDiv x y = Success (div x y)

-- Succeed except when given 10
blowUpTen :: Integer -> MaybeInt
blowUpTen 10 = Failure
blowUpTen x = Success x

-- Succeed when greater than 3
biggerThanThree :: Integer -> MaybeInt
biggerThanThree x = if x > 3 then Success x else Failure
```

Note that `Failure :: MaybeInt` is a value constructor that takes no arguments. You can think of it as a singleton value.

Also note the use of `deriving Show` to automatically make `MaybeInt` an instance of `Show`. This is just for our convenience in exploring this topic in the REPL.

Composing erroneous functions

Now let us return to original goal: composing, or chaining together, computations that might fail. Recall that we already have on operator, `(.)`, which composes regular functions:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

You might wonder why we can't just use this to chain together failing computations as well. Of course, the answer lies in the types: the expression `blowUpTen . safeDiv` fails to type check because `safeDiv` returns a `MaybeInt`, but `blowUpTen` expects just a regular `Integer`. Instead, let's see how to use pattern matching to "unwrap" the return value of `safeDiv` to pass it to `blowUpTen`:

```
computeInt :: Integer -> MaybeInt
computeInt n =
    let x = safeDiv 100 n
    in
        case x of
        Failure -> Failure
        Success y -> blowUpTen y

> computeInt 4
Success 25
> computeInt 0
Failure
> computeInt 10
Failure
```

Here we introduce **case expressions**, which allow pattern matching in arbitrary expressions.

One limitation of this approach is that chained computations don't preserve information about *where* the computation failed. You can fix this by attaching an error message to the value, which you can look up with the `Either` Haskell types.

What happens if we want to add 3 to the final result?

```
computeInt2 :: Integer -> MaybeInt
computeInt2 n =
    let x = safeDiv 100 n
    in
        case x of
        Failure -> Failure
        Success y ->
            let z = blowUpTen y
            in
                case z of
                Failure -> Failure
                Success w -> Success (w + 3)

> computeInt 4
Success 28
```

Even though this code make sense and is quite concise, we can see this pattern of "deconstruct a `MaybeInt` and only do something when the value is a `Success`" will be repeated quite a lot in our code. As programmers, any sort of repetition should make us uneasy, and so it should not come as a surprise that we will resolve this by *abstracting away this composition of failing computations*.

You may have noticed in `computeInt2` that we actually had two different compositions going on: a composition of two failing computations `safeDiv` and `blowUpTen` to get a new failing computation, and a composition of this new failing computation with `(+3)`, which is a regular computation with no possibility of failure. Since the *types* of these two compositions are different, you can expect to handle them differently.

Note the use of sectioning in `(+3)`

Lift: using pure computations with `MaybeInt`

Suppose we have a `MaybeInt` value, the result of some failing computation, and we wish to call plain `Integer -> Integer` function like `(+3)` on the result if it was a `Success`. We can do this manually using a `case` expression as above, but the problem is we'll have to do this every time we want to apply `(+3)` to a `MaybeInt` value. Instead, we could create the following function:

```
add3Maybe :: MaybeInt -> MaybeInt
add3Maybe Failure = Failure
add3Maybe (Success x) = Success (x + 3)
```

Of course, we still need to worry about the `(+4)`, `(/5)`, and `(*9000)` functions. Since higher-order functions don't scare us, we'll abstract again and define a function that takes an `Integer -> Integer` function, and returns an equivalent `MaybeInt -> MaybeInt` function!

```
lift :: (Integer -> Integer) -> MaybeInt -> MaybeInt
lift f Failure = Failure
lift f (Success x) = Success (f x)

> let add3Maybe' = lift (+3)
> add3Maybe' (Success 10)
Success 13
> add3Maybe' Failure
Failure
```

Note the use of function pattern matching taking the place of a `case` expression.

Now let us see how to compose these computations. Since each call to `lift` returns a `MaybeInt`, it is rather simple to chain together `lifts`:

```
> lift (+3) (lift (*2) (lift (+1) (Success 8)))
Success 21
-- or equivalently,
> lift (+3) $
  lift (*2) $
  lift (+1) $
  Success 8
```

However, you might note that all three arithmetic functions can be composed directly using `(.)`, because they all take and return `Integers`, and so we can rewrite this expression to the very elegant

```
> lift ((+3) . (*2) . (+1)) (Success 8)
Success 21
```

Pretty cool! But there is another way to express this sequence of function applications that is truly beautiful. For the nicest syntax, we'll define the following `(~>)` operator:

```
(~>) :: MaybeInt -> (Integer -> Integer) -> MaybeInt
Failure ~> _ = Failure
(Success x) ~> f = Success (f x)

> (Success 8) ~> (+1) ~> (*2) ~> (+3)
Success 21
> Failure ~> (+1) ~> (*2) ~> (+3)
Failure
```

Don't let the puncutation marks fool you – we are just rewriting `lift` as an infix operator here, with the arguments flipped.

Finally, this operator gives us an extremely clean way of handling the second composition from `computeInt2`:

```
computeInt3 :: Integer -> MaybeInt
computeInt3 n =
    let x = safeDiv 100 n
    in
        case x of
        Failure -> Failure
        Success y ->
            blowUpTen y ~> (+3)
```

Recall that function application has higher precedence than any infix operator, so the correct grouping of the last line is `(blowUpTen y) ~> (+3)`.

Bind: composing failing computations

The concept of lifting is an extremely powerful one, but only works with functions of type `Integer -> Integer`, which never produce errors themselves. However, we want to compose not just regular functions, but also `Integer -> MaybeInt` functions too. To do this, we define a new operator `(>~>)` (pronounced "bind") in an analogous way to lift:

Compare with the type of `(~>)` above!

```
(>~>) :: MaybeInt -> (Integer -> MaybeInt) -> MaybeInt
Failure >~> _ = Failure
-- No need to wrap result of (f x) in a Success
(Success x) >~> f = f x

> (Success 3) >~> (safeDiv 15) >~> blowUpTen ~> (+3)
Success 8
> (Success 3) >~> (safeDiv 30) >~> blowUpTen ~> (+3)
Failure
```

And using this operator, we now have an extremely concise way of expressing `computeInt3`:

For clarity, we have separated each function onto a new line.

```
computeIntFinal :: Integer -> MaybeInt
computeIntFinal n =
  safeDiv 100 n >~>
  blowUpTen ~>
  (+3)
```

Generalizing to `Maybe`

Of course, in general we want to capture erroneous computations that return values of type other than `Integer`, and this is where `Maybe` comes in. This is a built-in Haskell construct, which is defined in the following way:

```
data Maybe a = Just a | Nothing
```

Notice this data type declaration has a type variable `a`. This might seem strange, but remember that our familiar list type `[a]` is defined in the exact same way. Technically, we call `Maybe` a **type constructor**: you can think of it as a function that takes a type `a`, and outputs a new type call `Maybe a`. By using a type variable in the data definition of `Maybe`, we can create `Maybe` values for *any* type! For example:

type constructor

```
> :t (Just True)
(Just True) :: Maybe Bool
> :t (Just ["Hi", "Bye", "Cowabunga"])
(Just ["Hi", "Bye", "Cowabunga"]) :: Maybe [[Char]]
> :t Nothing
Nothing :: Maybe a
```

You guessed it – `Nothing` is another polymorphic value!

Pretty sweet: we can now chain together any number of computations that may or may not fail, with arbitrary intermediate types computed along the way! We end this section by defining our operators `(~>)` and `(>~>)` in an analogous fashion; **pay special attention to their types**. (In fact, you should be able to *infer* their types based on the implementations!!)

```
(~>) :: Maybe a -> (a -> b) -> Maybe b
Nothing ~> _ = Nothing
(Just x) ~> f = Just (f x)

(>~>) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >~> _ = Nothing
(Just x) >~> f = f x
```

Note, in fact, that the implementations haven't changed one bit. Instead, we've just generalized them!

Exercise Break!

14. Define a function `maybeHead :: [a] -> Maybe a`, which is the same as `head` except it encodes a failure if the list is empty.
15. Do the same for `maybeTail :: [a] -> Maybe [a]`.

State in a Pure World

One of the central tenets of functional programming we've had in the course so far was avoiding the concepts of *state* and *time*; that is, writing 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 modeling the real world must take into account constantly evolving state! 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 reassignment of variables?"

A mutable stack

Perhaps unsurprisingly, the answer is simply to interpret functions as transformations in time, whose output correspond to a "later state" of their inputs. As a simple example, consider a stack with two operations,

`push` and `pop`, which respectively add an item to the top of the stack, and remove an item from the top of the stack (and returns it). Though we cannot create a variable to represent a stack and mutate it directly, we can simulate this behaviour by creating functions that take in a stack and output a new stack.

```haskell
-- Here a stack is simply a list of integers,
-- where the top of the stack is at the *front* of the list.
type Stack = [Integer]

pop :: Stack -> (Integer, Stack)
pop (top:rest) = (top, rest)

push :: Integer -> Stack -> Stack
push x stack = x:stack
```

This is the same representation as our "choices" stack from the previous chapter.

By the way, notice that the pattern match for `pop` is incomplete. This means an error will be raised if we attempt to `pop` from an empty stack.

Unfortunately, this can lead to cumbersome code when chaining multiple operations on stacks, as the naïve way to do this is to store the intermediate stacks as local variables:

```haskell
-- Switch the top two elements on a stack.
switchTopTwo :: Stack -> Stack
switchTopTwo s =
    let (x, s1) = pop s      -- (1, [2,3,4,10])
        (y, s2) = pop s1     -- (2, [3,4,10])
        s3      = push x s2  -- [1,3,4,10]
        s4      = push y s3  -- [2,1,3,4,10]
    in  s4
```

Notice the use of pattern matching on the *left side of a let binding* – pretty useful!

This code is certainly easy to understand, and its size may not even suffer in comparison to other languages. But hopefully it seems at least somewhat inelegant to have to manually keep track of this state. This is precisely analogous to the clumsy use of `case` expressions to manually process `Failure` and `Success x` values from the previous section. So, inspired by the abstraction from the previous section, we will define higher-order functions that will enable us to elegantly compose functions that operate on stacks, keeping track of the underlying state automatically.

We're essentially "capturing time" in intermediate variables.

The first step is abstract away the common behaviour of `push` and `pop`: both are functions that take a `Stack`, and return another `Stack`. However, the function `pop` has the additional effect that it returns a value (the top of the original stack). Therefore, we will define a function type which captures both effects:

```haskell
type StackOp a = Stack -> (a, Stack)
```

Note the use of type parameter `a` to differentiate different kinds of stack operations.

You should interpret the return type of a `StackOp`, `(a, Stack)`, as explicitly encoding both the standard "return value" of the function in the familiar imperative sense, as well as the mutation that has occurred. For example, the traditional pop stack method has a non-mutating effect – return the top item of the stack – as well as a mutating one – remove the top item of the stack. Note how this is captured by the two components of the returned tuple in our new `pop` implementation:

```
pop :: StackOp Integer
pop (top:rest) = (top, rest)
```

Our `push` method is a little different, in that it takes an additional `Integer` argument, and it really doesn't "return" anything, instead only mutating the stack. We capture these details using the Haskell unit value `()`:

```
push :: Integer -> StackOp ()
push x = \s -> ((), x:s)
-- equivalently, push x s = ((), x:s)
```

Remember that `()` can represent either a value or a type, depending on the context.

Composing stack operations

Now that we have the necessary types in place, let us return to the main task at hand: elegantly composing `StackOps`. Consider a simplification of `switchTopTwo` in which we only pop off the top two items, returning the second (and losing the first).

```
popTwo :: StackOp Integer
popTwo s = let (_, s1) = pop s
           in  pop s1
```

Note the use of the underscore to indicate a part of a pattern that will not be used.

Our new operator `(>>>)`, pronounced "then", is a simple generalization of this:

```
-- Combine two StackOps, performing the first, then the second.
-- Any returned value of the first operation is lost.
(>>>) :: StackOp a -> StackOp b -> StackOp b
op1 >>> op2 = \s ->
    let (_, s1) = op1 s
    in  op2 s1

> let popTwo = pop >>> pop
> popTwo [10,2,5,1,4]
(2, [5,1,4])
```

Hey, alright! With this, we can simplify one of the three function compositions from `switchTopTwo`:

```
switchTopTwo2 s =
    let (x, s1) = pop s         -- (1, [2,3,4,10])
        (y, s2) = pop s1        -- (2, [3,4,10])
    in  (push x >>> push y) s2
```

Of course, this operator has the significant disadvantage that it throws away the result of the first operation. This prevents us from doing something like `pop >>> pop >>> push x >>> push y`; the `x` and `y` must come from the result of the `pop` operations. What we need to do is define an operator that will perform two `StackOps` in sequence, but have the second `StackOp` 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 creating new functions dynamically with behaviours depending on some other values.

So rather than take two fixed `StackOps` like `(>>>)`, our new operator will take a first `StackOp`, and then **a function that takes the result of that `StackOp`, and returns a new `StackOp`**. This operator will produce a new `StackOp` that does the obvious thing:

1. Apply the first `StackOp`.
2. Use the result to create a second `StackOp`.
3. Apply the second `StackOp` to the new stack, and return the result.

We'll use the same name `(>~>)` ("bind") for this operator as from our discussion of `Maybe`; by looking at the type signatures, you might see why...

```
(>~>) :: StackOp a -> (a -> StackOp b) -> StackOp b
(f >~> g) s =
    let (x, s1) = f s
        newStackOp = g x
    in  newStackOp s1

> (pop >~> push) [1,2,3]
((), [1,2,3])
> let addTenToTop = pop >~> \x -> push (x + 10)
> addTenToTop [5,3,7,1]
((), [15,3,7,1])
```

And now with this operator in hand, we can complete the transformation of `switchTopTwo`:

```
switchTopTwo3 s =
    (pop >~> \x ->
        (pop >~> \y ->
            (push x >>> push y)
        )
    ) s
```

Well, almost. We have used indentation and parentheses to delimit the bodies of each lambda expression, which causes our code to look a little ugly. However, because the `->` operator has very low precedence, the parentheses are unnecessary. Moreover, we can remove the redundant `s` from the function definition:

```
switchTopTwo4 =
    pop >~> \x ->
        pop >~> \y ->
            push x >>> push y
```

And finally, we note that indentation has no special significance in either applying the operators or the lambda expressions, so the way we have broken the lines allows a very nice vertical alignment:

```
switchTopTwoFinal =
    pop >~> \x ->
    pop >~> \y ->
    push x >>>
    push y
```

One recursive example

As a final problem, we're going to use these operators to calculate the sum of the numbers in a stack.

```
sumOfStack :: StackOp Integer
```

This can be done recursively in three steps:

1. Pop off the first item.
2. Recursively compute the sum of the rest of the stack.
3. Add the first item to the sum.

Here is an initial implementation; make sure you understand how this code corresponds to the algorithm outlined above.

```haskell
sumOfStack :: StackOp Integer
sumOfStack [] = 0
sumOfStack s = (
    pop >~> \x ->
    sumOfStack >~> \y ->
    \stack -> (x + y, stack)
    ) s
```

Note that `\stack -> (x+y, stack)` is a `StackOp Integer`.

This looks pretty nifty already, but that line 6 is a little inelegant, as it is a manifestation of the problem we've been trying to avoid in the first place: manually keeping track of state. Note that the third line, in which we add `x` and `y`, is a particularly special type of `StackOp` in that *it has no mutating effect*. We will define a third and final operator which composes a `StackOp` with a plain function that operates on the result (but not the stack). We'll call this operator "lift" `(~>)`, and again this is no coincidence: as before with `Maybe`, we are taking a normal function with no connection to stacks, and making it work in this stack context.

You can think of this as being analogous to pure functional programming.

```haskell
-- "Lift": Do f, and then apply g to the result,
-- without mutating the stack.
(~>) :: StackOp a -> (a -> b) -> StackOp b
f ~> g = \s ->
    let (x, s1) = f s
    in  (g x, s1)

sumOfStack [] = 0
sumOfStack s = (
    pop >~> \x ->
    sumOfStack ~>
    (+x)
    ) s
```

Why are we not able to eliminate the `s` here? Try it!

Exercise Break!

16. Implement the functions `removeSecond :: StackOp ()` and `removeThird :: StackOp ()`, which remove the second-highest and third-highest item from the stack, respectively. (Here "highest" refers to an item's position on the stack, and not the item's value.)

17. Implement the function `removeNth :: Integer -> StackOp ()`, which takes a positive integer `n` and returns a `StackOp` that removes the `n`th-highest from the stack.

18. One of the drawbacks of `pop` is that it raises a runtime error on an empty stack. Implement `safePop :: StackOp (Maybe Integer)`, which behaves similarly to `pop`, except it does not have this problem.

So in case you've been lulled into a false sense of security by static typing, runtime errors occur in Haskell too!

19. Implement `returnVal :: a -> StackOp a`, which takes a value and returns a new `StackOp` with no mutating effect, and which simply returns the value as the "result." Sample usage:

```
> returnVal 5 [4,2,6,10]
(5, [4,2,6,10])
> returnVal [True,False] [4,2,6,10]
([True,False], [4,2,6,10])
```

20. Using `returnVal`, `sumOfStack` so that it does not mutate the stack.
21. Using `returnVal`, re-implement `removeSecond`, `removeThird`, and `removeNth` so that they also return the removed item.
22. Implement `len :: StackOp Integer`, which computes the number of elements on the stack. Do not mutate the stack.
23. Implement `stackMap :: (Integer -> Integer) -> StackOp ()`, which takes a function `f` and mutates the stack by applying `f` to every item.
24. Implement `stackFilter :: (Integer -> Bool) -> StackOp ()`, which creates a `StackOp` that removes all elements in the stack that don't satisfy the input predicate.
25. Implement `stackFold :: (a -> Integer -> a) -> a -> StackOp a`, which folds a function and initial value across a stack. Leave the stack unchanged.
26. Implement `(>>>)` and `(~>)` in terms of `(>~>)`.

Haskell Input/Output

In addition to mutating state, another common type of *impure* function is one that interacts with some entity external to the program: standard in/out, a graphical display, a database or system files, 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 to actually display the output to the user as well. Put another way, if Haskell had no I/O capabilities, we would never know if a program actually worked or not, since we would have no way to observe its output!

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*.

We will now briefly study how Haskell programs can interact directly with standard in and standard out; while the primitives we'll use in this section might look underwhelming, all of the other I/O operations we listed 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 `StackOp a` functions captured two behaviours: a pure part, the computation performed to return 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 are those that do.

There was nothing special about having a stack as the context; you probably can guess that a natural generalization to our work in the previous section is using a more general data structure like a list or tree as the underlying *state*. But there is no need to limit our function context to the program's heap-allocated memory. By widening the scope of this context to user interactions, files, and faraway servers, we can capture any type of I/O we want!

Think about the possibilities of key-value pairs as the underlying state...

Standard input/output

Recall that we built complex `StackOp`s using two primitives, `pop` and `push`. With standard I/O there are two built-in functions that we're going to cover: `putStrLn`, which prints a string, and `getLine`, which reads a string. A natural question to ask when in an unfamiliar environment is about the types of the values. Let's try it:

Look up more functions in the `System.IO` module.

```
> :t getLine
getLine :: IO String
> :t putStrLn
putStrLn :: String -> IO ()
```

Here, `IO` plays an analogous role to `StackOp`: rather than indicating that the function changes the underlying stack context, it indicates that the function has some sort of interaction with an external source. `IO` has a type parameter `a`; the type `IO a` represents an I/O action that produces a value of type `a`. 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.

We can now define a function which prints something to the screen:

Finally. A Hello World program. Geez, David.

```
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 the print function twice, in sequence; but remember that we work with *expressions* and not *statements*, and we can only have one expression in the definition of any type of value, including functions.

After all, we wouldn't write `length [1,2,3] length [2]`.

What we want is to chain together two I/O actions, a way to say "do this operation, **then** do that one." This hopefully rings a bell from the previous section, when we chained together two `StackOps` with two operators "then" (`>>>`) and "bind" (`>~>`). And this is where things get interesting: we have the exact same operators for I/O, albeit with slightly different name spellings:

```
1  -- "then"
2  (>>) :: IO a -> IO b -> IO b
3  -- "bind"
4  (>>=) :: 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:

```
1  greet2 :: IO ()
2  greet2 =
3    putStrLn "Hello, world!" >>
4    putStrLn "Second line time!"

5  prompt :: IO ()
6  prompt =
7    putStrLn "What's your name?" >>
8    getLine >>
9    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 we saw this with our "bind" operator:

```
1  prompt2 :: IO ()
2  prompt2 =
3    putStrLn "What's your name?" >>
4    getLine >>= \name ->
5    putStrLn "Nice to meet you, " ++ name ++ "!"

6  > prompt2
7  What's your name?
8  David
9  Nice to meet you, David!
```

The "David" on line 8 is user input.

Aside: standalone Haskell programs

Though we've mainly used GHCi, Haskell programs can also be compiled and *run*. Such programs require a `main` function with type `IO ()`, which is executed when the program is run.

Run `ghc -make myfile.hs` on the command line to compile a Haskell file, and `./myfile` to run it. Alternatively, use `runhaskell myfile.hs`.

The type of `main` is very suggestive. Don't let our exploration of pure functional programming fool you. True *software* written in Haskell will inevitably need to interact with the "outside world," and this behaviour is encoded in the type of `main`.

Purity through types

You have now seen two examples of impure functions in Haskell: the state-changing `StackOp`, and the console-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. However, even though this is technically correct, it misses a very important point: unlike other languages, Haskell 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 object-oriented programming in the context of class design. 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 the very beginning. 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 which combine values of `IO` types, or with a function that *returned* a value of these types. But once the `IO` was introduced, there was no way to get rid of it. In other words, **any Haskell function that uses console I/O has a `IO` in its type.** Similarly, if any function wants to make use of our stack context for mutation purposes, it must take the stack and output a value and new stack. That is, our notion of mutation *requires* a certain function type.

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 which doesn't have `IO` in its type is guaranteed not to interact with the console, and this guarantee is provided *at compile time*, just through the use of types! In other words, a type signature in Haskell can not tell us not just about the expected arguments and return value, but also what can and cannot happen when the function is called.

One more abstraction

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. 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.

We already did that in Racket.

But defining type constructors like `Maybe`, `StackOp`, and `IO` to represent elements of these computations was only a small part of what we did. After defining `MaybeInt` and `StackOp`, 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 combine primitive functions to create 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 – "lift", "then", and "bind" – to work for each one. To make this a bit more concrete, let's look at the type signatures for the one function common to all three:

```
1 (>~>) :: Maybe a -> (a -> Maybe b) -> Maybe b
2 (>~>) :: StackOp a -> (a -> StackOp b) -> StackOp b
3 (>>=) :: IO a -> (a -> IO b) -> IO b
```

And though the implementations differed for each context, 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 constant core, which presents an opportunity for abstraction.

In fact, this is exactly what happens in Haskell: the similarity between `Maybe`, `StackOp`, and `IO` here is abstracted into a single type class, called `Monad`. In fact, we misled you in our discussion of I/O: if you query the type of either `(>>)` or `(>>=)`, you'll see a type class constraint that generalizes `IO`:

```
1 > :t (>>)
2 (>>) :: Monad m => m a -> m b -> m b
3 > :t (>>=)
4 (>>=) :: Monad m => m a -> (a -> m b) -> m b
```

It is no coincidence that these functions were the ones we have explored in great context; they are two of the three functions which are part of the Monad definition:

```haskell
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'll find that the other function is also very familiar, especially if you've been keeping up with the exercises.

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 contents of the first one.
- `(>>=)`: chain together two monadic values; the result *can* depend on the contents 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 document these functions in the `IO` context as follows:

- `return`: create an I/O action which does nothing, except produces the value.
- `(>>)`: combine two I/O actions into a new I/O action which performs the two in sequence, producing the result of the second.
- `(>>=)`: create a new I/O action which performs the first one, takes the result and uses it to create a second action, and then performs that second action.

This is mainly used to lift a value into the IO context, so that it be chained with other I/O actions.

Perhaps unsurprisingly, `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" 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 `StackOp` an instance of `Monad`, and to describe precisely what the three operations would mean in that context.

Why Monad?

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.
- Take a list of stack operations, perform them all on the same stack, 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`, `StackOp`, 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.

```
1 seqM :: Monad m => [m a] -> m [a]
2 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]`).

```
1 seqM (m1:ms) =
2     let rest = seqM ms
3     in
4         ...
```

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:

```haskell
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 stack?" 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!

```haskell
> seqM [Just 10, Just 300, Just 15]
Just [10, 300, 15]
> (seqM [pop, pop, pop]) [5,2,17,3,23,10]
([5,2,17], [3,23,10])
> (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!

27. Implement `seqM` using `foldl`.

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 idiosyncracies and minutiae of our programming languages, focusing instead on universal theoretical and practical concepts like 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 to choose from.

Large, monolithic frameworks are essentially specialized languages themselves.

It is easy to get lost in the myriad technical and non-technical details which 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. Newer, modern programming languages extend

the functional programming and static analysis ideas in interesting ways. Two languages of note which 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.

www.elm-lang.org
www.rust-lang.org

Be the person your dog thinks you are.

Appendix A: Prolog and Logic Programming

Author's note: This chapter used to be a fully-fledged part of these course notes. Now they are not, although the central ideas presented here are explored in a different context in an earlier chapter. It seemed to me a shame to throw out this chapter entirely, so here it is, more or less in the same condition as last year (so the flow may be off). Enjoy!

The two programming language paradigms you have seen so far have taken rather different approaches to computation. The central metaphor of imperative programming, inspired by the Turing machine, was interpreting computation as a *series of instructions* given to a machine to execute, often involving the manipulation of mutable state. In contrast, functional programming viewed computation as the *evaluation of (function call) expressions*, in which simple functions are combined to accomplish complex tasks.

In this chapter, we'll study **logic programming**, a new paradigm that views computation as *determining the truth or falsehood of statements*, according to some specified knowledge base. Our study of logic programming will be done using the programming language `Prolog`, which is a portmanteau of "**pro**grammation en **log**ique." A Prolog "program" will consist two parts: facts and rules, which encoding the things Prolog knows about the world; and queries to ask whether a given statement can be *proven* using these rules. At first, we will treat Prolog as a *declarative* language, in which we specify *what* to compute rather than *how* to compute it. However, we'll soon see that Prolog is not purely declarative; its implementation is easy to state, yet contains many subtleties that lead to pitfalls for newcomers. Understanding both how Prolog works and how to influence its execution will see a marked departure from the purity of functional programming from the first two chapters.

named by Philippe Roussel

SQL is similar, except its knowledge base is always a relational database, and its queries are not framed as true/false questions.

Getting Started

In this course, we'll use SWI-Prolog, a free Prolog implementation that can be found online. As with Prolog, you'll divide your time between Prolog source files (`*.pl`), where you'll store facts and rules, and the

`http://www.swi-prolog.org`

interpreter, where you'll load these files and make queries. The syntax for loading a file `myfile.pl` is very straightforward: enter `[myfile].` into the SWI-Prolog interpreter.

You can start the interpreter by running `swipl` in the terminal.

Facts and Simple Queries

A **fact** is a statement that is unconditionally true. These are expressed using *predicates*, which can take one or more *terms*.

fact (Prolog)

```
1  student(bob).
2  student(lily).
3  teacher(david).
4  teaching(david, csc324).
5  taking(lily, csc324).
```

Though we chose meaningful words for both predicates and terms, these of course have no intrinsic meaning to Prolog. Any strings starting with a lowercase letter would do; we could replace every instance of `student` with `twiddletwaddle` and achieve the exact same behaviour.
Students **often** miss the terminating period!

Facts always take the form `predicate(name1, name2, ...).`, where all predicates and terms must begin with a lowercase letter (for now). For names that contain multiple words, `camelCase` is generally used. After loading these facts into the interpreter, we can ask **queries**. Again, don't forget to put in the period!

```
1  ?- student(bob).
2  true.
3  ?- teacher(bob).
4  false.
5  ?- person(david).
6  false.
7  ?- teacher(david).
8  true.
9  ?- teaching(david, csc324).
10 true.
11 ?- student(lily), student(bob), taking(lily, csc324).
12 true.
13 ?- student(lily), student(david).
14 false.
```

Note the use of the comma to separate predicates in a query; you can think of the comma as the logical **and**.

If you're most familiar with C-based languages, it is easy to fall into the trap of treating predicates as functions, because of their syntax similarity. However, this could not be further from the truth; a predicate is a special structural construct that has no semantic meaning in Prolog. Consider:

- Unlike functions, predicates are never "called" (a redirection of control flow in a computation)
- Unlike functions, predicates cannot be "evaluated" to be replaced by a simpler result

- Unlike functions, predicates can be compared for equality at a purely structural level

more details on this later

So the Prolog interactive console is *not* a standard REPL, in which you enter expressions and get back results which are mathematically equivalent to your input. Instead, it's an oracle: you ask it questions in the form of predicates, and it answers yes or no. To underline the difference, we will use some specific terminology for Prolog queries that have nothing to do with function evaluation. A predicate in a query will be called a **goal**, which can either **succeed** of **fail**, which are signified by `true` and `false`. You can interpret query interactions as a dialogue:

This makes Prolog sound vaguely magical; by the end of this chapter, you'll have a firm understanding of how this "oracle" actually works!

> Is `bob` a student?
> Yes.
> Is `bob` a teacher?
> No.
> Is `lily` a student, and `bob` a student, and `lily` taking `csc324`?
> Yes.
> Is `lily` a student and `david` a student?
> No.

Variables in Queries

If all we could do is query literal facts, Prolog wouldn't be very interesting! However, Prolog is also able to handle *variables* in its queries, enabling the user to ask more general questions: "Who is a student?" rather than "Is `david` a student?" Variables can also have arbitrary names, but are distinguished from literal predicates and names by always being Capitalized.

What's the difference between a name and a variable? In the simplest case, when you use a variable in a goal, Prolog tries to *instantiate* that variable's value to make the goal succeed:

```
?- student(X).
X = bob
```

But that's not all! Notice how the cursor stays on that line, rather than prompting for a new query. Pressing `Enter` or `.` will halt the computation, but `;` will result in the following:

```
?- student(X).
X = bob ;
X = lily.
```

The semi-colon prompts Prolog to search for a *new* value with which to instantiate `X` to make the goal succeed. However, notice that after `lily` comes a period; Prolog has finished searching, and knows there are

no more queries. You can think about this interaction as the following dialogue:

> Is X a student?
Yes, if `X = bob`.
> Any other possibilities? (the semicolon)
Yes, if `X = lily`.

Sometimes *no* values can make a query succeed:

"Is X a student and is X a teacher?"

```
?- student(X), teacher(X).
false.
```

Notice that this reveals something meaningful about the role of variables in a query. Even though each goal `student(X)` and `teacher(X)` can succeed on their own, there is no instantiation of `X` to make both of these goals succeed *simultaneously*. On the other hand, the following query does succeed:

```
?- student(X), taking(X, csc324).
X = lily.
```

We've seen that the same variable name is linked across multiple goals separated by commas. However, it doesn't come as a surprise that we can instantiate different variables separately:

```
?- student(X), teacher(Y).
X = bob,
Y = david ;
X = lily,
Y = david.
```

Variables in facts

Interestingly, we can also use variables in facts. You can think about the variable `X` in `same(X, X)` below as being *universally quantified*: "for all `X`, `same(X, X)` holds." Using variables this way in facts lets us make global statements of truth, which can be quite useful.

```
sameName(X, X).

?- sameName(bob, bob).
true.
?- sameName(bob, lily).
false.
```

That said, unconditionally true statements are kind of boring. We would really like to build *definitions*: a predicate is true *if* some related

predicate holds. To use the terminology of first-order logic, we've now seen predicates, AND, and universal quantification. Now we want implication.

Rules

Consider adding the following **rules** to our program:

rule (Prolog)

```
happy(lily) :- taking(lily, csc324).
happy(bob) :- taking(bob, csc324).
```

Unlike facts, rules represent *conditional* truths. We interpret these rules by replacing the `:-` with "if": "lily is happy *if* lily is taking csc324." The following queries should be pretty straight-forward.

Just remember that in our running example `taking(lily, csc324)` is a fact, and `taking(bob, csc324)` is not.

```
?- happy(lily).
true.
?- happy(bob).
false.
```

Imagine we had more people in our universe; it would be rather tedious to write a separate "happy if taking `csc324`" rule for each person! Luckily, we can use variables in rules, too.

```
happy(X) :- taking(X, csc324).

?- happy(lily).
true.
?- happy(bob).
false.
```

Here's a predicate `taughtBy`, which represents the idea that "`X` is taught by `Y`." Intuitively, this predicate succeeds if there is a course that `X` is taking and `Y` is teaching, and this is precisely what the following rule expresses. We use the comma to mean "and" on the right side a rule, just as we did for our queries. When we have multiple (comma-separated) goals on the right side, they must *all* succeed in order for the left side goal to succeed.

```
taughtBy(X, Y) :- taking(X, C), teaching(Y, C).

?- taughtBy(lily, david).
true.
?- taughtBy(bob, david).
false.
```

Note that you can think of the `C` as being *existentially* quantified.

We have now seen facts, queries, and rules, which are the fundamental building blocks of Prolog programs. There are a few more special forms we'll see later, but with just this core alone, you can already begin writing some sophisticated Prolog programs!

By the way, mentioning both facts and rules is a little redundant; you can view a fact as a rule with no goals on the right side .

Exercise Break!

Translate each of the following English statements into Prolog. You should use the predicates from the previous section, but you will probably have to define some of your own, too. Note that some statements may require may require multiple lines of Prolog code to fully express.

1. There is a student named `julianne` who is taking `csc209`.
2. `Jen` is teaching `csc207` and `csc108`.
3. All students are taking `csc108`.
4. All students who are taking `csc236` have previously taken `csc165`.
5. Only students are taking courses. (Hint: write this as an implication: "if ... then ...")
6. Two students are classmates if they are taking the same course.

Recursion

It shouldn't come as a surprise that we can define *recursive* rules in Prolog. For example, suppose we want to determine if someone is a descendent of someone else. Here's a simple way to do this in Prolog.

```
1 child(alice, bob).
2 child(bob, charlie).
3 child(charlie, daenerys).

4 descendant(X, Y) :- child(X, Y).
5 descendant(X, Y) :- child(X, Z), descendant(Z, Y).
```

As you've probably guessed, `child(X,Y)` denotes the fact that `X` is a child of `Y`.

Note that we've defined two separate rules for descendant: a base case and a recursive rule. These match our intuitive recursive understanding of descendants: if `X` is a descendant of `Y`, then either `X` is a child of `Y`, or `X` is a child of some descendant `Z` of `Y`.

Using just our logical reasoning, all of the following queries make perfect sense.

```
?- descendant(alice, daenerys).
true.
?- descendant(bob, alice).
false.
?- descendant(X, daenerys).
X = charlie ;
X = alice ;
X = bob ;
false.
```

You can ignore the final `false` for now.

However, something that we've swept under the rug so far is precisely *how* Prolog computes the answers to its queries, and in particular, why it returned the values `charlie`, `alice`, and `bob`, in that order. Of course, from the point of view of logic the order of the results doesn't matter at all. But it is still interesting to think about how Prolog actually works behind the scenes, and this will be our next major topic of study. But first, one common application of recursion that we've seen twice already: list processing.

Lists

As in Python and Haskell, Prolog represents list values using square brackets, e.g. `[a, b, c]`. As you might expect, variables can be instantiated to list values, but they can also be pattern matched to literals inside lists, as demonstrated in the following example.

```
special([a, b, c]).
?- special([a, b, c]).
true.
?- special(X).
X = [a, b, c].
?- special([a, X, c]).
X = b.
?- special([X, Y, Z]).
X = a,
Y = b,
Z = c.
?- special([X, X, X]).
false.
```

We can match the *head and tail* of a list using the syntax `[a|[b, c]]`:

```
?- special([a|[b, c]]).
true.
?- special([X|Y]).
X = a
Y = [b, c].
```

Let's see how to use recursion on lists to write a simple "membership" predicate.

```
member(X, [X|_]).
member(X, [_|Tail]) :- member(X, Tail).
```

Note the use of the underscore to match any term that we do not need to reuse.

Note that this is essentially a recursive approach, except that there's no base case! This is a consequence of an even larger contrast between logic programming and other paradigms: we are defining a function that returns a boolean value; instead, we are defining *the conditions that make a predicate true*:

> "X is a member of a list when X is the head of the list."
> "X is a member of a list when X is a member of the tail of the list."

In case you haven't noticed by now, Prolog does not treat truth and falsehood symmetrically. In our source files, we only explicitly define in our knowledge base what is *true*; Prolog determines which statements are false if and only if it is unable to determine that the statement is true from its knowledge base. And since `member(X, [])` is false for any `X`, we do not explicitly need to pattern match this case in our definition.

This reasoning is perfectly acceptable for modelling boolean functions; however, what if we wanted to capture more complex functions, like list concatenation (which returns a new list)? It might be tempting to try to define something like `append(X, Y)`, but this falls into the trap of thinking of predicates as functions. Remember that Prolog predicates are statements of truth, and cannot "return" list values!

So instead, we define a predicate `myAppend(X, Y, Z)`, which encodes the statement "`Z` is the concatenation of `X` and `Y`." Note that `Z` here plays the role of the "output" of the corresponding append function from other programming languages. This is a common idea that you'll see used again and again in Prolog to capture traditional computations.

There is already a built-in `append` predicate

```
myAppend([], Y, Y).
myAppend([X|Xs], Y, [X|Zs]) :- myAppend(Xs, Y, Zs).

?- myAppend([a], [b,c], [a,b,c]).
true.
?- myAppend([a], [b,c], [a,b]).
false.
```

By leaving the third argument uninstantiated, we get to simulate the behaviour of a real function call. However, don't be fooled; remember that there are no functions being called, but instead a variable being instantiated to make a goal succeed.

```
?- myAppend([a], [b,c], Z).
Z = [a, b, c].
```

We finish off this section by using `myAppend` to illustrate the power of logic programming. Even though we began by trying to simulate a list concatenation *function*, we have actually created something more. A traditional function in both imperative and functional programming has a very clear distinction between its inputs and its output; but by including an output variable as an argument to the `myAppend` predicate, we have completely erased this distinction. And so not only can we use Prolog to instantiate an output variable; we can use it to instantiate input variables as well:

```
?- myAppend(X, Y, [a,b,c]).
X = [],
Y = [a,b,c] ;
X = [a],
Y = [b,c] ;
X = [a,b],
Y = [c] ;
X = [a,b,c],
Y = [].
```

Exercise Break!

7. Implement the predicate `takingAll(Student, Courses)`, which succeeds if and only if `Student` is taking all of the courses in the list `Courses`.

8. Implement the list predicate `reverse(L, R)`, which succeeds if and only if `R` is the reversal of the list `L`.

9. Implement `sorted(List)`, which succeeds if and only if `List` is sorted in non-decreasing order.

10. Implement `removeAll(List, Item, NewList)`, which succeeds if and only if `NewList` is the same as `List`, except with all occurrences of `Item` removed.

Prolog Implementation

We have seen a few times in our study of functional programming the inherent tension in reconciling the abstraction of the lambda calculus with the deterministic implementations of both Racket and Haskell that make notions of time and memory inescapable. The same is true of Prolog: while logic programming derives its meaning solely from first-order logic, writing equivalent code in Prolog often requires more thought than just direct transliteration, because care must be taken to incorporate the subtleties of the Prolog implementation.

Luckily, the basic algorithm Prolog uses is quite simple, and so with some effort we can learn how this works and how to ensure our Prolog code is compatible with it. The algorithm consists of two main ingredients, one big and one small. We'll start with the small one.

Previous offerings of this course have actually implemented most of Prolog in Racket!

Unification

Remember type inference? This is something the Haskell compiler did to determine the types of names and expressions without needing explicit type annotations. We mentioned that you could think about this as the compiler taking the operations used on a variable and generating *constraints* on it, and then determining the most general type that satisfies these constraints.

When you enter a query into Prolog, Prolog attempts to **unify** the query with facts or rules, which is analogous to type inference, except acting on literal values and predicate structure rather than types. For terms with no variables, unification is just the usual notion of (literal) equality:

```
?- bob = bob.
true.
?- bob = lily.
false.
?- student(bob) = student(bob).
true.
?- student(bob) = student(lily).
false.
?- taking(lily, csc324) = taking(csc324, lily).
false.
?- [1, 2, 3] = [3, 2, 1].
false.
?- f(a, b(c)) = f(a, b(c)).
true.
```

The `=(X,Y)` predicate succeeds if and only if `X` and `Y` can be unified. It is used here with *infix* syntax, but keep in mind `bob = bob` can be rewritten as `=(bob, bob)`.

Order in predicates and lists matters.

Nested predicates are supported.

Unification with variables is a little complex, but you have probably already developed a mental model of how this works from your own

experimenting with queries. Formally, terms with variables are said to unify (match) if it is possible to instantiate all their variables so that the resulting terms, containing only literals, match. As we've noted earlier, variables don't need to be instantiated to literal names like `bob` or `csc324`; they can also take on predicate and list values like `taking(lily, csc324)` and `[a, b, 3]`.

```
?- X = bob.
X = bob.
?- student(X) = student(bob).
X = bob.
?- X = student(bob).
X = student(bob).
?- taking(X, csc324) = taking(lily, csc324).
X = lily.
?- taking(X, Y) = taking(lily, csc324).
X = lily,
Y = csc324.
?- taking(X, X) = taking(lily, csc324).
false.
?- taking(X, csc324) = taking(lily, Y).
X = lily,
Y = csc324.
```

One last point that we've also seen before is that when two terms are being unified, they share the same variable namespace. That is, if both terms have a variable `X`, and the first term has `X` instantiated to the value `cool`, then the second term must also instantiate `X` to `cool`. We illustrate this in a few examples below.

```
?- X = X.
true.
?- [X, 2] = [2, X].
X = 2.
?- taking(X, csc324) = taking(lily, X).
false.
```

Now that we rigorously understand how Prolog unifies terms, let's see how Prolog makes use of this in its main algorithm. The key question we'll answer is "How does Prolog decide which terms to match, and when to match them?"

Search I: Facts, Conjunctions, and Rules

Despite the build-up that's happened so far, the Prolog implementation of query search is actually quite simple, and can be summarized as follows:

> Each goal attempts unification with the facts and rules in the source code, one at a time, *top-to-bottom*. The goal succeeds if it is successfully unified with a fact, or with the left side of a rule, and all goals on the right side of that rule also succeed. Otherwise, the goal fails. A query succeeds if all of its goals succeed.

For example, consider the following simple knowledge base and queries.

```
1 student(bob).
2 student(lily).
3 teacher(david).

4 ?- student(lily).
5 true.
6 ?- student(david).
7 false.
8 ?- student(bob), teacher(bob).
9 false.
```

When the query is entered, Prolog attempts to unify the goal `student(lily)` with facts one at a time, starting from the top. The first unification fails, as `student(lily) != student(bob)`, but the second unification is successful, and so the query succeeds, and Prolog outputs `true`. On the other hand, the query goal `student(david)` fails to unify with *all three* of the facts, and so this query fails.

What about conjunctions like `student(bob), teacher(bob)`? Simple: Prolog runs the above unification-checking algorithm on each goal separately, left-to-right. Though the goal `student(bob)` succeeds, `teacher(bob)` fails, leading to the whole query to fail. This is also a simple example of Prolog implementation details contrasting with pure logic. In first-order logic, the statements "bob is a student and bob is a teacher" and "bob is a teacher and bob is a student" are completely equivalent. However, Prolog short-circuits its goal-checking, meaning that Prolog will halt and output `false` the *first* time a goal fails. Had we queried `teacher(bob), student(bob)` instead, Prolog would not have checked if `student(bob)` succeeds at all. This might seem obvious coming from other languages which short-circuit their and operator, but understanding this aspect of Prolog control flow is crucial to understanding more sophisticated implementation details later on.

Finally, let's introduce a simple rule and see how Prolog handles a query which involves that rule.

```
1 student(bob).
2 student(lily).
3 teacher(david).
4 peers(lily, bob) :- student(lily), student(bob).

5 ?- peers(lily, bob).
6 true.
```

As before, Prolog attempts to match `peers(lily, bob)` with each of the three facts, but of course these all fail. But Prolog also attempts to use rules by matching the goal against the **left side term** of each rule it encounters. In our example, the goal matches the left side of the rule at line 4.

Then when this unification is successful, Prolog **checks the goals on the right side of the rule** to make sure they all succeed. Intuitively, it says: "I want to left side to succeed: the rule tells me it does when the right side succeeds, so now I have to check if the right side succeeds." In our case, both of the goals `student(lily)` and `student(bob)` succeed, and so the original goal also succeeds. Note that this means Prolog's query answering is a *recursive* procedure: every time a rule is used to resolve a query, Prolog recursively makes new subqueries, one for each goal on the right side.

To make sure you understand this, try to predict what happens with the following query:

```
1 f(a) :- g(a).
2 g(a) :- f(a).

3 ?- f(a).
```

Search II: Backtracking

Consider the following addition to our example:

```
1 student(bob).
2 student(lily).
3 teacher(david).

4 peers(lily, bob) :- teacher(lily), teacher(bob).
5 peers(lily, bob) :- student(lily), student(bob).

6 ?- peers(lily, bob).
7 true.
```

Here we've added an extra rule whose right side is certainly not true

according to our facts. We know that the query `peers(lily, bob)` actually succeeds, because of the second rule. The original query goal now can unify with two rules, one of which will cause the query to fail, and the other to succeed. So the question is how does Prolog *know* which rule to use?

The answer: **it doesn't**. The algorithm we described in the previous section still holds true, with the query goal being unified with the facts and rules one at a time, top-to-bottom. And because the first rule does match, Prolog selects that rule and begins a subquery for the first goal, `teacher(lily)`, which fails.

Now here is the important part. Prolog should not have chosen the first rule, but instead the second rule. So Prolog does not simply give up and output `false`, which it would do if we asked it the query `teacher(lily), teacher(bob)` directly. Instead, Prolog now **backtracks** to the point where it chose to use the first rule, and rather than choosing this rule, it skips it and uses the second rule. Of course, the right-side goals for the second rule succeed, making the original goal a success.

So when Prolog reaches a fact or rule that can unify with the current goal, it saves a **choice point**, which you can think of as a point in the computation where Prolog must make a choice to actually use the fact/rule, or skip it. Of course, this is just metaphor; Prolog doesn't actually "make choices," it always uses a matching fact/rule when it encounters it for the first time. However, if Prolog backtracks to this choice point, it *makes a different choice*, skipping the fact/rule instead.

We have seen that Prolog backtracks *automatically* when it encounters a failing subquery as part of a rule. However, when a query succeeds, but there is still a choice point that could lead to alternatives, Prolog allows the user to *prompt backtracking* – by pressing semicolon.

```
1  student(bob).
2  student(lily).
3  teacher(david).

4  f(a) :- student(bob).
5  f(a) :- student(lily).
6  f(a) :- teacher(david).

7  ?- f(a).
8  true ;
9  true.
```

The first `true` occurs because of the first rule (`student(bob)` succeeds), and the second occurs because of the third rule (`teacher(david)` succeeds). Note that no intermediate `false` is printed.

Search III: Variable Instantiation

The previous example may have surprised you; you are normally used to pressing the semicolon in the context of asking for additional variable instantiations to make a query succeed. Let us now carefully look at how Prolog handles queries containing variables. By the end of this section, you should be able to predict not only *the values* of variable instantiations from a query (this can be done solely through understanding the logic), but also the *order* in which the instantiations occur.

```
1 student(bob).
2 student(lily).
3 teacher(david).

4 ?- student(X).
5 X = bob ;
6 X = lily.
```

So let's see what happens when Prolog checks the goal `student(X)`. As before, Prolog will attempt to unify this goal with the facts and rules one at a time, top-to-bottom. The first such check is with `student(bob)`. We know from our earlier discussion of unification that `student(X) = student(bob)` when `X = bob`, so Prolog does indeed create a choice point for this rule, and then chooses to use it.

If Prolog were a much simpler language, it could simply note that `student(X)` and `student(bob)` can be unified and then use the fact to make the query succeed, and output `true`. The beauty of Prolog is that it also *saves* the variable instantiation `X = bob` to make the unification succeed, and this instantiation is precisely what is output instead of merely `true`.

These variable instantiations are tied directly to a choice of fact/rule for unification. Therefore after the user presses `;`, the first fact is skipped, and the instantiation of `X` is undone. Of course, the second fact is chosen, with the instantiation `X = lily` saved and reported instead.

Search IV: Subquery backtracking

You may have gotten the impression so far that queries can be modelled as a stack, just like function calls.

```
student(bob).
student(lily).
teacher(david).

peers(lily, bob) :- student(lily), student(bob).
happy(lily) :- peers(lily, bob).

?- happy(lily, bob).
true.
```

That is, the original query `happy(lily)` spawns the subquery `peers(lily, bob)`, which then first queries `student(lily)`, and then queries `student(bob)`. One of the fundamental things about the stack-based function call model is that program execution is sequential: at any time only one function is executing (other calling functions may be paused), and after that function is complete, there's no going back.

No parallelism here

In Prolog, it is certainly the case that even with subqueries, there is only ever one currently executing query. And the backtracking we have seen so far could be considered consistent with the idea that one a query is completely finished, one cannot return to it – we have only seen backtracking for the current query. So it may surprise you that this is not actually how backtracking works: it is quite possible (and *extremely* common) to backtrack into subqueries that have already completed. Prolog saves all choice points for the entire duration of a query, *including choice points of completely subqueries*. For example:

```
student(bob).
student(lily).

happy(david) :- student(X).

?- happy(david).
true ;
true.
```

Note that no variable instantiations are printed because only variables in the *original* query have their instantiations output.

Here, the subquery `student(X)` is unified with the first fact, and so succeeds, causing the original goal `happy(david)` to also succeed. But after this, even though the subquery `student(X)` exited, Prolog is able to backtrack within this subquery (unifying with the second fact), causing another `true` to be output.

As we've seen earlier, the currently executing query has a single

namespace, meaning all variables with the same name must have the same value. The ability to backtrack into exited subqueries is how Prolog supports rules involving multiple uses of the same variable.

```
1 student(bob).
2 student(lily).
3 taking(lily, csc324).

4 happy(X) :- student(X), taking(X, csc324).

5 ?- happy(X).
6 X = lily.
```

The first subquery `student(X)` first unifies with the first fact, setting `X = bob`. The next subquery made is *not* `taking(X, csc324)`, but instead `taking(bob, csc324)` (!!), which fails! If Prolog were a simpler language, this would be the end of it: no backtracking can occur for `happy(X)`, since it only unifies with the one rule. However, because Prolog saves the choice point of the subquery `student(X)`, backtracking occurs to this choice point, setting `X = lily`!

By the way, keep in mind that the goals in the rule are queried left-to-right; had we switched the order to `happy(X) :- taking(X, csc324), student(X)`, the `X` would have been *first* instantiated to `lily`, and *never* instantiated to `bob`.

Tracing Recursion

You haven't really understood backtracking until you've had to trace a bit of recursion, so let's do that now! Recall our example of children and descendants from earlier:

```
1 child(alice, bob).
2 child(bob, charlie).
3 child(charlie, daenerys).

4 descendant(X, Y) :- child(X, Y).
5 descendant(X, Y) :- child(X, Z), descendant(Z, Y).

6 ?- descendant(charlie, daenerys).
7 true ;
8 false.
```

You may have already seen the choice point-`false` combination already; let's understand why. The goal `descendant(charlie, daenerys)` goes through the following steps:

1. This is unified with the rule on line 5, with `X = charlie` and `Y=`

`daenerys`.

2. The subquery `child(charlie, daenerys)` succeeds, and hence so does the original query (`true`) is output.
3. Backtracking! Line 6 is now chosen, and the subquery `child(charlie, Z)` is made. Note the free variable `Z`, which is then instantiated to `daenerys`.
4. The next subquery `descendant(daenerys, daenerys)` **fails**. (Why? Exercise!)
5. Since no other choice points remain, a `fail` is output.

Remember that the variables have been instantiated at this point.

Note that even though we could tell by inspection that the second rule would not be helpful in making `descendant(charlie, daenerys)` succeed, Prolog is unable to determine this without actually trying it out!

Now let's look at a query with a somewhat surprising result:

```
1 ?- descendant(X, daenerys).
2 X = charlie ;
3 X = alice ;
4 X = bob ;
5 false.
```

Though the values are obviously correct, why are they produced in this order?

1. Remember that facts and rules are checked from top to bottom. So `descendant(X, daenerys)` unifies with the first rule, producing the new query `child(X, daenerys)`. Note that at this point, `X` hasn't been instantiated with a literal yet.
2. The goal `child(X, daenerys)` unifies with `child(charlie, daenerys)`, and `X` is instantiated to `charlie`. There are no goals left, so the original query succeeds, producing the output `X = charlie`.
3. Then backtracking occurs to the goal `child(X, daenerys)`, which fails because it cannot match any other rule or fact.
4. Then backtracking occurs *again*, and the original goal `descendant(X, daenerys)` skips over the first rule, and instead matches the second, leaving the new goals `child(X, Z), descendant(Z, daenerys)`.
5. The first goal `child(X, Z)` matches the very first rule `child(alice, bob)`, with instantiations `X = alice, Z = bob`. We'll call this choice point (CP1).
6. The remaining goal `descendant(bob, daenerys)` matches the first rule, but this fails (since `child(bob, daenerys)` doesn't match any fact/rule).
7. Backtrack: `descendant(bob, daenerys)` matches the second rule, and the goals `child(bob, Z), descendant(Z, daenerys)` are added.

Technically speaking, `X` is instantiated with a fresh variable to be unified with the rule left side `descendant(X, Y)`.

We've deliberately separated steps 3 and 4 to indicate that these are really *two separate instances* of backtracking.

8. `child(bob, Z)` matches the fact `child(bob, charlie)`, and the remaining goal `descendant(charlie, daenerys)` succeeds, as in step 2. No goals are left, and the output `X = alice` is printed (remember that we're still under the "scope" of step 5).
9. More backtracking: to (CP1), in which `X` was instantiated to `alice`.
10. The goal `child(X, Z)` then matches the second rule `child(bob, charlie)`, and the remaining goal `descendant(charlie, daenerys)` succeeds, leading to the output `X = bob`.
11. Backtracking occurs: `child(X, Z)` matches with `child(charlie, daenerys)`, but the remaining goal `descendant(daenerys, daenerys)` fails.
12. Backtracking occurs again, except now `child(X, Z)` has no more facts or rules to match against, so it fails. Since no more choice points are available, a final `false` is output.

Note that instantiations of intermediate variables aren't printed, just the values of the variables appearing in the original query.

Again, exercise!

Left Recursion Hairiness

The recursive form of the `descendant` recursive rule might seem pretty normal, but in this subsection we'll investigate how another plausible rule leads to infinite recursion, not because of *logical* errors, but because of Prolog's implementation.

Consider the following modification of `descendant`:

```
1 descendant(X, Y) :- descendant(Z, Y), child(X, Z).
2 descendant(X, Y) :- child(X, Y).
```

This is certainly logically equivalent to the previous implementation: reordering goals left-to-right doesn't matter, nor does reordering the rules top-to-bottom. But from a procedural perspective, this is very different. For example, the query `descendant(alice, bob)` causes Prolog to now go into an infinite loop! Why? It is *impossible to reach the base case.*

1. The goal `descendant(alice, bob)` matches the first rule, which spawns the subqueries `descendant(Z, bob), child(alice, Z)`.
2. The goal `descendant(Z, bob)` matches the first rule, which adds the goals `descendant(Z', bob), child(Z, Z')`. Notice that the two `Z`'s do *not* refer to the same variable at runtime.
3. The goal `descendant(Z', bob)` matches the first rule, which adds the goals `descendant(Z", bob), child(Z', Z")`. Uh oh.

The problem actually isn't the order of the rules, but the fact that the recursive rule is now **left-recursive**, meaning that the first goal on the right side is recursive. When we first learn recursion as a programming technique, we take special care that the arguments passed to a recursive call must be closer to a base case than the original arguments; the

consequence of ignoring this is stack overflow. In Prolog, we typically ensure progress towards a base case by initializing variables with extra predicates *before* the recursive predicate.

A notable exception to this is using pattern-matching on lists to ensure that the arguments to the recursive predicate are shorter than the original arguments.

Exercise Break!

11. Implement the predicate `atLeastTwo(Student)`, which succeeds if and only if `Student` is taking at least two different courses. (Hint: how can you assert that two variables are different?)
12. Consider the following variations on `descendant`. Give an example of a query that fails to work (e.g., goes into infinite recursion). Make sure you can **explain why the query fails**!

```
descendant(X, Y) :- child(X, Y).
descendant(X, Y) :- descendant(X, Z), child(Z, Y).
```

13. Suppose we introduced the new fact `child(daenerys, alice)`. Using the *original* definition of `descendant`, explain what can go wrong.
14. Give an example of a Prolog source file and query that produces the following output, and explain why.

```
true ;
false.
```

15. Give an example of a Prolog source file and query that produces the following output, and explain why.

```
true ;
true ;
true ;
false.
```

Cuts

The final Prolog topic we'll explore in this course is one more implementation detail: **cuts**. When we first started our study of Prolog, we were concerned purely with the *logical meaning* of Prolog facts, rules, and queries. Then we turned our attention to understanding how Prolog actually processes our queries.

In this final section, we'll discuss one very useful tool for *explicitly controlling* Prolog backtracking: the **cut**. Using cuts will enable us to

make more *efficient* Prolog programs, though you should keep in mind throughout that many of the programs we'll write have logically equivalent ones that do not contain cuts.

Consider the following rules.

```
student(willow).
taking(willow, csc324).
teacher(david).

happy(X) :- student(X), taking(X, csc324).
happy(X) :- teacher(X).
```

The two rules assert that every student taking `csc324` is happy, and every teacher is happy. Consider the query:

In particular, `david` is happy. :)

```
?- happy(willow).
true ;
false.
```

Note that backtracking occurs, because there are two rules that can unify with the goal `happy(willow)`. However, if we assume that no student is also a teacher, then this is a little inefficient: because we know that the goals of the first rule `student(willow), taking(willow, csc324)` succeeded, there is no reason to try the second rule: `teacher(willow)` is doomed to fail! But so far we lack the ability to stop Prolog from backtracking, and this is where cuts come in.

A **cut**, denoted `!`, is a special goal in Prolog that *always succeeds* – but with a side-effect that after the goal has succeeded, Prolog fixes two choices for the current query:

cut

- The choice of rule.
- The choices made in all subqueries of the rule prior to the cut.

Intuitively, the cut goal prevents Prolog from backtracking to *before* the cut took place for the current goal.

A student described it as a "wall" that prevents backtracking past it.

```
student(willow).
taking(willow, csc324).
teacher(david).

happy(X) :- student(X), !, taking(X, csc324).
happy(X) :- teacher(X).

?- happy(willow).
true.
```

Let's trace through this query. The goal `happy(willow)` is unified with the first rule, and then the new goal `student(willow)` succeeds. The next goal `!` is the cut, which we know always succeeds. The final goal, `taking(willow, csc324)` also succeeds, leading to an overall success: `true` is printed. However, unlike before, the query ends. This is because the cut *prevents backtracking on the choice of the current rule*, meaning the initial goal `happy(willow)` is never unified with the second rule.

To make the cut action more explicit, suppose there was a person who was both a student and a teacher.

We know we just said this isn't possible, but this is illustrative.

```
student(willow).
taking(willow, csc324).
teacher(david).

student(dan).
teacher(dan).

happy(X) :- student(X), !, taking(X, csc324).
happy(X) :- teacher(X).

?- happy(dan).
false.
```

Logically, `happy(dan)` should succeed, because of the second rule. However, the goal is unified with first rule, the subquery `student(dan)` succeeds, and so the cut is reached, locking in the rule. The final subquery `taking(dan, csc324)` fails, and so the whole conjunction fails. At this point *without* the cut, Prolog would backtrack and use the second rule to obtain a success, but now the cut prevents precisely this backtracking, and so the query fails.

Because the cut also prevents backtracking in subqueries made before the cut, it can also fix variable isntantiations, leading to surprising consequences when the original query contains an uninstantiated variable.

```
student(xander).
student(willow).
taking(willow, csc324).
teacher(david).

happy(X) :- student(X), !, taking(X, csc324).
happy(X) :- teacher(X).

?- happy(willow).
true.
?- happy(david).
true.
?- happy(X).
false.
```

Woah. Clearly there are *two* happy people, `willow` and `david`. Let's trace this carefully to see how the cut changed the meaning of the program. The goal `happy(X)` is unified with the rule, spawning the subqueries `student(X), !, taking(X, csc324)`, *in that order*.

But we know that `student(X)` is unified with `student(xander)`, with `X` being instantiated to `xander`. **This choice is locked in by the cut**, and the final goal `taking(xander, csc324)` fails. The cut prevents backtracking on `student(X)` (which would have led to `X = willow`), as well as on the choice of rule (which would have led to `X = david`). As a result, the entire query fails.

Backtracking with cuts

A common mistake that students make when learning about cuts is that they think it prevents *all* backtracking, which is much too powerful! A cut prevents backtracking just for the **current goal**. That is, if the current goal finishes, backtracking can indeed occur:

```
student(xander).
student(willow).
taking(willow, csc324).
cool(xander).
cool(willow).

happy(X) :- student(X), !, taking(X, csc324).
veryHappy(X) :- cool(X), happy(X).

?- veryHappy(X).
X = willow.
```

What happens with this query? It shouldn't come as a surprise that the first subquery `cool(X)` succeeds with the instantiation `X = xander`,

but then the next subquery `happy(xander)` fails. But because the cut goal is reached, you might think that the cut prevents all backtracking, just like in the previous example. But this is not the case; after `happy(xander)` fails, backtracking *does* occur for `cool(X)`, which succeeds with the instantiation `X = willow`.

That is, the cut prevents all backtracking within the subquery `happy(willow)`, but if that subquery fails, the all other backtracking in the query `veryHappy(X)` proceeds as normal. On the other hand:

```
1 student(xander).
2 student(willow).
3 taking(willow, csc324).
4 cool(xander).
5 cool(willow).

6 happy(X) :- student(X), !, taking(X, csc324).
7 veryHappy(X) :- happy(X), cool(X).

8 ?- veryHappy(X).
9 false.
```

Here, the first subquery made is `happy(X)` (note the uninstantiated variable!), which fails, as the cut prevents backtracking after `X` is instantiated to `xander`.

Failures as Negation

Even though we've called Prolog a logic programming language, there's one fundamental ingredient from logic that we've been missing: negation. That is, if we've defined a predicate `p(X)`, how can we write a goal representing "not `p(X)`". It turns out that representing negation is hard to do in Prolog, but here is one common heuristic known as the **cut-fail combination**.

```
1 not-p(X) :- p(X), !, fail.
2 not-p(X).
```

How does this work? Let's start with `fail`. Like cut, `fail` is a special built-in goal, but unlike cut, it always *fails*. How could this possibly be useful? When combined with cut, `fail` can cause a whole conjunction and rule to fail.

In our example, if `p(X)` succeeds, then the goal `!` fixes the choice of this rule, and then `fail` causes the query to fail. On the other hand, if `p(X)` fails, then backtracking *can* occur (the cut is never reached), and so `not-p(X)` is matched with the fact on the second line, and succeeds.

In slightly more detail: the goal `fail` fails, and since the cut restricts backtracking, Prolog cannot choose a different fact/rule to use, and so it fails.

Even though this cut-fail combination seems to behave like logical

negation, it's not! Here's one example illustrating the difference.

```
1 student(bob).
2 teacher(david).
3 not-student(X) :- student(X), !, fail.
4 not-student(X).

5 happy(X) :- not-student(X), teacher(X).
```

The following queries works just fine (we'll leaving tracing them as an exercise):

```
?- happy(bob).
false.
?- happy(david).
true.
```

But suppose we want to ask the query "is anyone happy?" Clearly, the answer is yes (`david`). However, Prolog answers differently.

```
?- happy(X).
false.
```

What went wrong? Let's trace through the steps:

1. First, the goal `happy(X)` is unified with the rule on the last line, and the new goals `not-student(X), teacher(X)` are added.
2. Then the goal `not-student(X)` is matched with the rule on line 3, which adds the goals `student(X), !, fail`.
3. The `student(X)` goal succeeds (`X = bob`), and then the cut-fail combination causes `not-student(X)` to fail. Note that no backtracking can occur.
4. But since `not-student(X)` fails, the whole conjunction `not-student(X), teacher(X)` from the `happy` rule also fails.
5. This causes Prolog to backtrack to before the choice of using the rule; but since no other facts or rules are left to match `happy(X)`, the original goal fails.

Exercise Break!

16. Express the following English statement in Prolog: "every student except bob is taking csc324."
17. Consider the following variation of `descendant`:

```
descendant(X, Y) :- child(X, Y), !.
descendant(X, Y) :- child(X, Z), descendant(Z, Y).
```

It may seem like putting a cut at the end of a rule is harmless, but this is not exactly true.

(a) Give an example of a query where this cut makes the program more efficient, i.e., prevents unnecessary backtracking.

(b) Given an example of a query where this cut causes the program to output something different than what we would see without the cut. Explain what happens.

(c) Can you generalize your answers from parts (a) and (b) to characterize which queries will be okay, and which won't?