David Liu

Principles of Programming Languages

Lecture Notes for CSC324 (Version 1.1)

Department of Computer Science
University of Toronto

Many thanks to Peter Chen, Rohan Das, Ozan Erdem, Itai David Hass, Hengwei Guo, and many anonymous students for pointing out errors in earlier versions of these notes.

Contents

Prelude: The Lambda Calculus

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.

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.

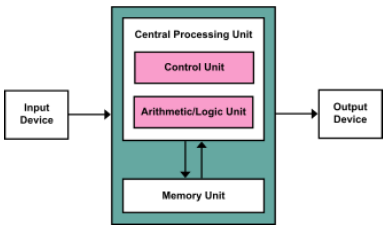


Figure 1: wikipedia.org/wiki/Von_Neumann_architecture.

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.

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

Alonzo Church, 1903-1995

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.

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

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

The Lambda Calculus

Alonzo Church modelled computation with the **lambda calculus**, which was heavily influenced by the mathematical notion of a function. This model has just three ingredients:

1. **Names**: a, x, $yolo$, etc. These are just symbols, and have no intrinsic meaning.
2. **Function Creation**: $\lambda x \mapsto x$. This expression creates a function that simply returns its argument – the identity function.
3. **Function Application**: $f\ x$. This expression applies the function f to the value x.

The exact syntax of the lambda calculus is not too important here. The main message is this: the lambda calculus views *functions* as the primary ingredient of computation; functions can be created and applied to values, which are themselves functions, and so through combining functions we may create complex computations.

Though if this were a fourth-year class, this would certainly be a topic of study in its own right!

Quite importantly, these functions are *mathematically pure* – they take some number of inputs, and produce an output. They 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.

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

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 of 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-eval-print loop (REPL)** where you can type in expressions for immediate evaluation. Make use of this when learning Racket!

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

There are three basic literal types we'll use in Racket. These are illustrated below (you should follow along in your Interactions window by typing in each line and examining the result). Racket uses the semicolon to delimit one-line comments.

```
; numbers
3
3.1415

; booleans. Can also use true and false instead of #t and #f.
#t
#f

; strings
"Hello, World!"
"Notice the DOUBLE-quotes. They're important."
```

With this in mind, let us see how the lambda calculus is realised in Racket. We're going to talk about the three elements in a different order than the Prologue, starting from the most familiar to the least familiar.

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 standard operators.

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

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

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

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 a boolean function, a common naming 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
> (* 2 -9)
-18
> (equal? 3 4)
#f
> (equal? 2 2.0)
#t
> (<= 10 3)
#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)
```

The Nature of Functions

We are used to thinking about functions from imperative languages as specifying a sequence of operations to perform. Sometimes a value is returned; other times, a value is read from or written to an external source, like the console, your monitor, or the Internet. In stark contrast, functions in functional programming are (generally) *pure mathematical functions*, analogous to $f(x) = x^2$.

Pure functions are completely defined by the values they output, and in particular, they have **no side-effects** like printing to the standard output or changing values in memory; intuitively, the only thing they "do" is output a value. Moreover, the values pure functions return depend **only on their arguments**; put another way, if in Racket we call some pure function `(f 1)` and the output value is "`One`", then **every single time `(f 1)` is called, the output value is "`One`"**.

This property of a function value depending *only* on its input might seem strange in a programming context, but it is extremely natural in math. If I tell you that $f(x) = x^2$, then it makes perfect sense to say that $f(5)$ *always* has value 25.

Working with pure functions might feel strange at first, but they offer a great opportunity to simplify your reasoning about your code. More on this later, for pretty much the rest of the course.

Function Creation

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

```
> (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
```

Once again, a function is a value, and so can be entered into the interpreter.

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.*

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 run the risk of ending up with 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)
```

Not even DrRacket's indentation will 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 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.

Unlike the imperative programming languages you've used so far, names in (pure) functional programming represent **immutable** values – once you've bound them to a particular value (or function, because remember functions are values) you can't change them.

This is the first major difference between functional and imperative programming: the former's general avoidance of mutation when possible. This will come up again and again throughout the course, because working without mutable state seems so foreign to those of us coming from imperative languages. Why would we want this? For one, it helps make programs easier to reason about (i.e., prove correct), as we don't have to worry about tracking how variables change! In particular, the whole issue of whether an identifier represents a *value* or a *reference* is rendered completely moot here, and will be for the rest of the term!

In particular, we won't talk much about the difference between call-by-value and call-by-reference parameter passing in this course.

This leads us to an extremely powerful concept know as **referential transparency**. We say that a name is referentially transparent if it can be replaced with its value without changing the meaning of the pro-

referential transparency

gram. Referential transparency is extremely desirable because it means we can use names and values interchangeably when we reason about our code, and in pure functional programming, referential transparency is enforced *as part of the programming language*!

Now, Racket is actually an *impure* functional programming language, meaning (among other things) that it does support mutation; however, in this course mutation is explicitly disallowed. Remember that the point is to get you thinking about programming in a different way, and believe it or not, this ban is supposed to *simplify* your thinking!

So we use names only as a way of storing intermediate values to make our computation more readable:

```
(define a 3)
(define b (+ 5 a))
(equal? (< 0 (* a b))
        true)
```

We use the exact same syntax for binding names to functions. Remember: functions are *exactly the same* as values like numbers and strings!

```
(define add-three (lambda (x) (+ x 3)))
(define almost-equal (lambda (x y) (<= (abs (- x y)) 0.001)))
(define return-0 (lambda () 0))

> (add-three 4)
7
> (return-0)
0
```

Because function definitions are so common, there is a special, 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))
(define (return-0) 0)
```

STOP. What is the difference between `(define return-0 0)` and `(define (return-0) 0)`?!

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 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-`false` 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 imparative 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

To answer this question, we want to understand how function calls work in Racket, or more generally, in functional programming. First, consider mathematical functions: if I give you a function $f(x) = x^2 + 5$ and ask you to *evaluate* $f(5)$, what would you do? Simple: *substitute* 5 for x in every value of x on the right side. Thus $f(5) = 5^2 + 5 = 30$.

Now, here is the important point: for pure function calls, we can model their evaluation as substitution in the *exact* same way.

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

We'll use the → to signify an evaluation step for a Racket expression.

But things get complicated with nested function applications:

```
((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

((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 **strict** or **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 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.

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". Functional programming views lists as a *recursive* data structure, which can be defined as follows:

- The empty list is a list, represented in Racket by `empty` or `'()`.
- If `my-list` is a list, and `item` is any value, then we can create a new list whose first element is `item` and whose other items are stored in `my-list`. 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
```

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 Python list representation `[1, 2, 3]`. 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`, 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.

List functions

Racket offers a number of basic list functions. Here's a few of the most basic ones; we'll talk about a few more soon.

```
> (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)
```

For a complete set of functions, check the Racket documentation.

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

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!

1. Write a function to determine the length of a list.
2. Write a function to determine if a given item appears in a list.
3. Write a function to determine the number of duplicates in a list.
4. Write a function to remove all duplicates from a list.
5. Given two lists, output the items that appear in both lists (intersection). Then, output the items that appear in at least one of the two lists (union).
6. 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)`).
7. Write a function which takes an item and a list of lists, and inserts the item at the front of every list.
8. 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.
9. Write a function taking a list with no duplicates, and a number k, and returns all subsets of size k of that list.
10. Modify your function to the previous question so that the parameter k is optional, and if not specified, the function returns all subsets.
11. 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)`).
12. A **sublist** of a list is a series of *consecutive* items of the list. Given a list of numbers, find the maximum sum of any sublist of that list. (Note: there is a $O(n)$ algorithm 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.

13. We have two special keywords `lambda` and `define`. Neither of these are functions; how do you know?
14. 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)))`.
15. 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.

Aside: 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. A quick Python example, in which the number of function calls is $\mathcal{O}(n)$:

```python
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 be returned to the original caller.

tail call elimination

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!

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.

```racket
(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!

1. 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 – 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.

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

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

; Apply the same function to an argument twice in a row
(define (apply-twice f x)
  (f (f x)))
(apply-twice sqr 2)                  ; 16
```

In case you didn't guess, `(sqr x)` squares `x`.

Delaying evaluation

Using higher-order functions, we can find a way to delay evaluation, sort of. 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 0-arity function :

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

; (filter function lst)
; Creates a new list whose elements are those in lst
; that make function output true
> (filter (lambda (x) (> x 1))
          '(4 -1 0 15))
'(4 15)
```

Both `map` and `filter` 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))))
```

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

Though all three of map, filter, and fold are extremely useful in performing most computations on lists, both map and filter are constrained in having to return lists, while fold can return any data type. This makes fold 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))))
```

Exercise Break!

1. 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.

2. Reimplement all of the previous exercises using map, filter, and/or fold, and *without* using explicit recursion.
3. 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`.
4. Is `foldl` tail-recursive? If so, explain why. If not, rewrite it to be tail-recursive.
5. Implement `map` and `filter` using `foldl`. (Cool!)
6. 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 `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))))
```

Scope and 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*! To really understand this language feature, we need to get a basic understanding of how scope works in Racket.

When we bind names, the code "knows" that these were bound, so that when we refer to them later, the computer "knows" what their values are. The **scope** of a name is the places in the source code where that name is defined. Every position in a program's source code has a **(lexical) context**, which is the total collection of all names and their bindings at that point in the code. On top of this, when the program is run, the **(execution) context** is all of the bindings from the corresponding lexical context, together with all other names defined on the call stack.

Note that this is true both with and without mutation.

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 the program sequentially, from top to bottom.

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 `let`:

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)])
  (* x y))          ; 15

x ; 3
y ; error!
```

Almost all languages, including Racket, support *variable shadowing*: the ability to define bind a name in a local scope with the same name as a previously defined name. At any point in the code, any use of the name refers to the innermost definition of that name.

```
> (define x 3)
> (define (my-f x) x)  ; x is a parameter, not 3
> (my-f 10)
10
> (let ([x 15]) x)     ; x is 15, not 3
15

> x                    ; global x is unchanged
3
```

Remember, no mutation! A name binding will never change values; however, it can be *shadowed* with another binding to the same name.

Aside: let, let, letrec*

Using `let` to bind local names is essential to writing clean code in Racket. There are three flavours that we'll mention here; it is important that you understand the difference between all three.

```
(define (f x)
  (let ([y 3]
        [z (+ x 1)])
    (+ x y z)))
(f 10) ; 24
```

The standard `let` takes two arguments: a list of local name bindings (one per `[...]`), and then an expression where you use those bindings. However, suppose in the previous example we want 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 allows a local name to be used in bindings *beneath* its definition.

```
(define (f x)
  (let* ([y 3]
         [z (+ y 1)])
    (+ x y z)))
(f 10) ; 17
```

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)))
```

Back to scopes

Now consider the following function, which *returns* a new function.

```
(define (make-adder x)
  (lambda (y) (+ x y)))
(make-adder 10) ; #<procedure>

(define add-10 (make-adder 10))
(add-10 3) ; 13
```

The body of `add-10` seems to be `(+ 10 y)`; using the substitution model of evaluation for `(make-adder 10)`, we see how the `(+ x y)` becomes `(+ x y)`:

```
(make-adder 10)
; → (substitute 10 for x)
(lambda (y) (+ 10 y))
```

However, the actual implementation of how Racket achieves this is more complex, and more interesting. But before you move on...

Exercise Break!

1. 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.
2. 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.
3. 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`.
4. 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.
5. 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.

6. 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
|#
```

7. 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 brand-new functions each time, when really all of the created functions have the same body, and differ only in their value of `x`.

This insight allows Racket to greatly simplify its handling of such functions. 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`.

I 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**, i.e., one 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 "lookup" necessary.

free identifier

However, in the definition below, 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.

Note: there is a difference between a name being in scope and being free; in this example, `x` is a free identifier, but is still in scope because of the enclosing outer function.

```
(define (make-adder x)
  (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.

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 full picture. 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? The value used is the one bound to the name that is in scope **where the `lambda` expression is 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 the global z, and gets a value of 10. 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.

dynamic scope

Initial programming languages used dynamic scope because it was the natural behaviour 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 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 callbacks, which are 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.

Basic Objects

You may have noticed that the act of creating functions with closures – variable values that are local to that function – may seem reminiscent of object-oriented programming, or at least the simpler structs of C. For in its purest form, an object is nothing but a value that stores certain data, and responds to *messages* requesting attributes or methods.

Using closures, we can model (immutable) objects in exactly this way:

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

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

Note that `(p "distance")` is just a function here.

Cool! The main downside of this approach is that there seems to be a lot of extra syntax required, and in particular a lot of repetition. However, we'll next see a way to *manipulate the syntax of Racket* to achieve the exact same behaviour in a much more concise, natural way.

Exercise Break!

1. 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))
```

2. 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.

Macros

Though we have not mentioned it explicitly, Racket's extremely simple syntax – essentially a nested list structure – not only makes it extremely 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 create domain-specific languages.

Previous offerings of this course (and similar courses at other institutions) take an in-depth look at how to write a Racket interpreter in Racket.

macro

Simply put, a **macro** is a program that transforms the source code of one Racket program into another. Unlike simple function calls (which "transform" a function call expression into the body of that function call), macros are processed by the Racket interpreter *before the program is run*. 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 Python syntactic construct: the **list comprehension**.

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

```
>>> [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.

So what we would like to do is mimic this concise syntax in Racket by doing the following:

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 actually *implement* this functionality in Racket. If you're comfortable with the higher-order list functions, you might notice that this is basically what `map` does:

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

; 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>)
```

We can now write a macro to transform the former into the latter.

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

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

Note: the < and > are optional here, but are helpful to distinguish the pattern variables from the keywords.

Let's break that down. The `define-syntax` takes two arguments: a name for the syntax, and then a set of one or more syntax rules. 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`. This is followed by one or more syntax rules, which are pairs specifying the old syntax pattern, and the new one.

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

To understand what is meant by "literal keywords", consider the following erroneous example:

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

> (list-comp (+ x 2) 3 x "hi" '(0 10 -2))
'(2 12 0)
```

The macro now treats `for` and `in` as patter variables, which match any expression, including `3` and `"hi"`.

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 both forms 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)
    [(list-comp <expr> for <var> in <list>)
     (map (lambda (<var>) <expr>) <list>)]
    [(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)
```

Note that each syntax rule is enclosed by [], just like `cond` cases or `let` bindings.

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

Hygienic Macros

If you've heard of macros before learning a Lisp-family language, it was probably from C. The C macro system operates on the source text rather than the parse tree, in large part because C's syntax is a fair bit more complex that Racket's. One of the most significant drawbacks of operating directly on the source code is that the macros are not **hygienic**, meaning they might inadvertently capture some existing variables. This is illustrated in the following example.

The typesetting language LaTeXalso uses macros extensively.

```
#define INCI(i) {int 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.

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

But now a local variable `a` is created, and the global variable is not incremented; this program prints `a is now 0, b is now 1`. The local declaration in the macro *captures* the name `a`.

In contrast, Racket's macro system doesn't have this problem. Here's a simple example:

```
(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.

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

```
(define y 10)
(define add-10 (lambda (y) (+ y y)))

> (add-10 100)
200
```

Macros with ellipses

It is often the case that we will want to write a macro that is applied to an arbitrary number of expressions; for this task, we cannot explicitly write one pattern variable for each expression. Instead, we'll use the ellipsis `...` to match such expressions.

Here is one example of using the ellipsis in a recursive macro. To implement `cond` in terms of `if`. To trigger your memory, recall that any branching of "else if" expressions can be rewritten in terms of nested if expressions:

```
(cond [c1 x]
      [c2 y]
      [c3 z]
      [else w])

; as one cond inside an if...
(if c1
    x
    (cond [c2 y]
          [c3 z]
          [else w]))

; eventually expanding to...
(if c1
    x
    (if c2
        y
        (if c3 z w)))
```

Let us write a macros 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]) (if test val (void))]
    [(my-cond [test val] next-pair ...)
     (if test val (my-cond next-pair ...))]))
```

Note that `else` is a literal here.

This example actually illustrates two important concepts with Racket's pattern-based macros. The first is the use of square brackets inside the pattern. For example, the first syntax rule will match the pattern `(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 ellipsis matches "0 or more expressions", and can be used in the resulting expression **as long as it appears after** `next-pair`. So rather than thinking of the ellipsis as a separate entity, you can think of the ellipsis as being paired with the `next-pair` identifier; every time `next-pair` appears in the body of the macro, the ellipsis must appear as well, and vice versa.

Classes revisited

Earlier, we developed a basic class using function closures:

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

However, this is rather verbose; there is a lot of "boilerplate code." We can use macros to define a simpler syntax like `(class Point x y)` A skeleton of our macro:

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

Notice the use of `attr ...` to match an arbitrary number of attributes. Unfortunately, this macro is incomplete, and would generate the following code:

```
(class Person x y)
; =>
(define (Person x y)
  (lambda (msg)
    (cond [else "Unrecognized message!"])))
```

What's missing, of course, is the other expressions in the `cond` that match strings corresponding to the attribute names. For simplicity, 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 a trick:

This turns an identifier into a symbol, and then a symbol into the corresponding string.

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

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

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

So the question is how to generate these expressions for *each* of the attributes in `attr ...` – and it turns out that Racket's use of ellipses is precisely what we want:

```racket
(define-syntax class
  (syntax-rules ()
    [(class class-name attr ...)
     (define (class-name attr ...)
       (lambda (msg)
         (cond [(equal? msg (symbol->string (quote x)) x)]
               ...
               [else "Unrecognized message!"])))]))
```

This use of the ellipsis causes the *entire previous expression* to repeat, replacing only the occurrences of `attr` with the other arguments to the macro.

Exercise Break!

1. Explain how a macro is different from a function. Explain how it is *similar* to a function.
2. Below, we define a macro, and then use it in a few expressions. Write the resulting expressions after the macros are expanded.

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

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

3. Write macros to implement `and` and `or` in terms of `if`.
4. Why could we not accomplish the previous question with functions?

5. Consult the official Python documentation on list comprehensions. One additional feature we did not cover is nested comprehensions, e.g.

```
>>> [(x,y) for x in [1,2,3] for y in [x, 2*x] if x + y < 5]
[(1, 1), (1, 2), (2, 2)]
```

   Modify our existing `list-comp` macro to handle this case. Hint: first convert the above expression into a *nested* list comprehension.
6. Modify the `class` macro to add support for class **methods**.
7. Modify the `class` macro to add support for **private attributes**.
8. Modify the `class` macro to add support for **inheritance** (e.g., by using a `super` keyword).

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. And finally, we learned about macros, enabling us to directly manipulate the syntax of Racket programs.

Haskell: Types

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

Software and other things

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 here: `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 *functions* in files, and then loading them into the interpreter and playing around with them there. For example, if you define a function `myFunction` in a file `functions.hs`, then after starting GHCi, you could type `:load`

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

`functions` (or `:l functions`) and then call `myFunction` as much as you want.

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

```haskell
-- This is a comment.

-- primitive data types
1, -10.555
True, False
"Hello, World"              -- String, double quotes
'a'                         -- Char, single quotes
()                          -- "null" value, called *unit*

-- operators, written infix
1 + 3
2 == 15
2 /= 15                     -- "not equal" operator
True && (3 >= -0.112)

-- lists
[]
[1,2,3,4]
1:[2,3,4]                   -- cons!

-- function application does not use parentheses
max 3 4                     -- 4
head [1,2,3,4]              -- 1
tail [1,2,3,4]              -- [2,3,4]
length [1,2,3,4]            -- 4
take 3 [10,20,30,40,50]     -- [10,20,30]
[1,2,3] ++ [4,5,6]          -- [1,2,3,4,5,6]

-- 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
```

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

It shouldn't come as a surprise that the 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 some more aesthetic bonuses we'll discuss shortly.

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

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

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

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

Note that The last line show that function application is *left-associative*, meaning the correct bracketing of `makeAdder 10 5` is `(makeAdder 10) 5`. (Remember that `makeAdder 10` is a function!)

Pattern Matching

Haskell uses *pattern matching* as its main tool for defining different function behaviour on different inputs. For example:

```haskell
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 many more branching patterns that we don't have time to get into here – guards, case statements, and more!

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

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

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

By the way, `undefined` is just 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 expression, but signify that it will not be used. 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 defining 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 +, 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, allows for special syntax for them. However, because operators are simply functions, they can be used in prefix syntax, with just a minor modification.

```
-- standard infix operators
-- 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

-- standard binary operator
-- use backticks ‘...‘ to do 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
```

Folding and Laziness

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

Which way to fold?

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.

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

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:

```
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 first set of exercises in the Racket chapter, this time in Haskell, with or without using explicit recursion.
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.
6. Look up **Pascal's Triangle**. Represent this structure in Haskell.

For extra practice, use both!

Bonus: why is this funny?

Static and Dynamic Typing

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 let's state it anyway. A **type** specifies: (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 `Int`, then we would be shocked if `x` took on the value `"Hello, world!"`, and equally surprised if we tried to call `len(x)` or `(first x)`, as these functions are not defined on integers.

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

Aside: Strong typing vs. weak typing

One of the most 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() {
  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 truly a binary property, but set of nuanced rules about what types can

Although they might differ in precisely what they output.

be coerced and when. It goes without saying that mastering these rules is important for any programming language.

Back to static/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 "*how are types determined*?" In fact, 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. For example, in Java we declare a new variable with a type, and throughout the code the variable can only take on values of that type.

static typing

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

Static typing is really a form of program verification (like program correctness from CSC236), and by proving 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 takes 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 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, meaning any type errors are raised only when the program is run, as in the initial Racket example:

dynamic typing

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

The main advantage of static typing is the extra guarantees of program correctness *before runtime*. On the other hand, the main advantage of dynamic typing is less rigid and often verbose code, and more flexibility in how you use variables and functions in your code.

A TA pointed out that one can view variables in dynamically-typed languages as variables of a huge union type in a statically-typed language (and thus simulate dynamic typing in statically-typed languages). Thus dynamic typing doesn't necessarily increase the expressive power of a language, but for practical purposes it sometimes makes code easier to write.

Types in Haskell

In Haskell, types are denoted by *capitalized* names, such as `Bool`, `Char`, and `Integer`. In GHCi, you can see the type of every expression and value with the `:t` 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.

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

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

Note that 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 `[Bool, 'a']` is rejected by Haskell. This restriction also applies to `if` expressions: both the `then` and `else` subexpressions must have the same type!

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

```
> :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`.

Function types

Not only do our basic values have types: since functions are just values, they also have types!

```
> :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.

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

type annotation

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

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

Wait. The line `isEven :: Integer -> Bool` does not look a like a comment; so it's read by the Haskell compiler and used to specify the type of `isEven`. But we've defined other functions so far without type annotations, and (in the interpreter) we've defined variables without explicitly naming their types as well.

For fun, try changing the type annotation to `isEven :: Char -> Bool`. Does the program compile?

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, reading 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 of Haskell to determine types at compile time without explicit type annotations is called **type inference**, and is one of Haskell's most powerful features. With it, Haskell provides all of the type safety associated with static typing *without* all of the normal boilerplate code. However, keep in mind that every language that has type inference *also* supports explicit 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 second look at `isEven`.

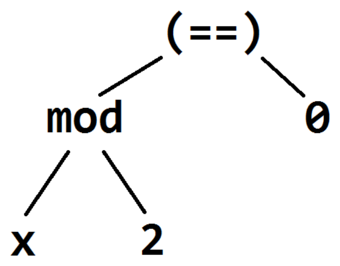
The type inference algorithm from this system, Algorithm W, runs almost in *linear* time!

```haskell
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.

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.



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

Let's return to the original "bad type" Racket example, in Haskell form.

```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.

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 also 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!

Recall that type inference takes nearly linear time.

```
> :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 `(&&)`?

```
> :t (&&)
(&&) :: Bool -> Bool -> Bool
```

Huh, that's weird. I probably would have expected something more like `(Bool, Bool) -> Bool` to denote that it 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 `(&&)` is 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).

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, we have created a new function**. That's not so bad; the *real* insight is that because functions are values, the `y` is completely unnecessary to defining `newF`. That is, the following definition is a completely equivalent definition of `newF`.

```haskell
newF = (&&) True
```

This makes sense when we think in terms of *referential transparency*: using this definition of `newF`, every time we see an expression like `newF True`, we can *substitute* this definition 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!

We saw earlier that function application is *left-associative*, meaning the expression `(&&) True True` might look like a single function application on two arguments, but its true syntactic form 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:

```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!

Aside: currying in the lambda calculus

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. In the example below, we'll use the more familiar Haskell syntax with lambda expressions, though you can translate this almost directly to the lambda calculus.

```haskell
-- Standard multi-variable function...
\x y -> x + y
-- is really syntactic sugar for the following:
\x -> (\y -> x + y)
```

If you understand how the above two definitions are equivalent, you understand currying.

Sectioning

All multi-parameter Haskell functions can be curried, including 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

```
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 can 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.

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.

```
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 contains elements of the same type:

```
> :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!

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

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

Tired of hearing it yet? Strings are lists of `Char`!

In the type expression `[a] -> a`, `a` is a *type variable*. This type signature tells us that `head` works on any type of the form `[a]`. 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:

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

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

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`. It 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 `(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:

```
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.

Generics in C++ play the same role, allowing type variables in both functions and classes.

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 change their behaviour depending on their context. 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. 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

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 basically identical, ad hoc polymorphism enables radically different behaviour depending on the type.

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!

1. 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
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)
```

2. Make sure you can infer the types of `map`, `filter`, `foldl`, and `foldr`.
3. 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.)
4. Rewrite the definition of `nats`, using sectioning.
5. Use currying to write a function that squares every integer in a list.
6. 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`).
7. 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

```
> 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 expression of this mingling. The left side is a type expression: **data Point** defines a new type called `Point`. The right side `Point Float Float` defines the function used to create the type, specifying the type of its argument(s). There is no reason for the type and its value constructor to have the same name:

```
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? Indeed, with the way we've defined the `Point` type, we can't! However, 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.

```haskell
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)
```

Here's another example where we use the value constructor to create new `Point` values.

```haskell
-- 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:

```haskell
makePoints3 = zipWith Point
```

Multiple Value Constructors

In typical object-oriented languages, the *constructor* of a class is special, 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.

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

Here it is *really* important to get the terminology correct. This line creates a new type `Shape`, which has two value constructors: `Circle` and `Rectangle`. Here's how we might use this new type.

Note that the types are based on `Shape`,

```
> let shape1 = Circle (Point 3 4) 1
> let shape2 = Rectangle (Point 0 0) (Point 3 3)
> :t shape1
shape1 :: Shape
> :t shape2
shape2 :: Shape
> let shapes = [Circle (Point 3 4) 5, Circle (Point 0 0) 2]
> :t shapes
shapes :: [Shape]
```

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
```

Typeclasses

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 typeclasses come in.

A **typeclass** in Haskell defines a set of functions or operations that can operate on a type, similar to a Java interface. Most built-in types in Haskell is a **member** of the Show typeclass, which the interpreter uses to display instances of those types. You can think of membership in a typeclass as implementing an interface.

typeclass

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 typeclasses), which we can use to quickly create typeclass instances of the basic typeclasses using the deriving keyword: data Point = Point Float Float deriving Show.

Here are two other built-in typeclasses that you should know:

- `Eq`: members of this typeclass 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` typeclass. Only `(<)` (and `(==)`) need to be implemented.

Class constraints and polymorphism

Let's take a step back from our 2D models. Suppose we want to use our new-found typeclasses 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 of course many other types – like lists and even booleans – can also be compared. In fact, any type that is a member of `Ord` typeclass should be supported by `between`. Of course, for `x`, `low`, and `high` must have the same type, so a good guess would be `between :: a -> a -> a -> Bool`. But this allows types that aren't members of `Ord` – such as function types – to be used, which is incorrect. We must specify that `a` is a member of `Ord`, which we can do using the following syntax:

`False <= True`

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

The `Ord a =>` is called a **class constraint**, and indicates that the type variable `a` must be a member of the `Ord` typeclass.

class constraint

Typeclasses give Haskell a way of supporting ad hoc polymorphism. By creating different type which implement typeclass functions in different ways, we can make functions with typeclass constraints behave in different ways, depending on the underlying implementations of these functions. Note that this sort of ad hoc polymorphism is signalled by a typeclass 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 important ideas of this chapter.

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 => ....`

Aside: 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 – and now we know precisely what that means. There are three main numeric typeclasses 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
   ```

As we've hinted at previously with `Ord` and `Eq`, typeclasses can have subclasses.

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

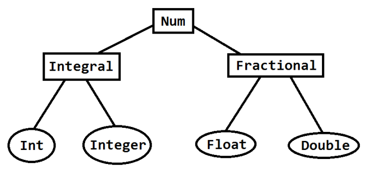


Figure 2: Some numeric typeclasses.

One last thing: what is the type of 1?

```
> :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]
```

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.

Using types, we can incorporate this form of error handling in a completely type-safe way. That is, we can indicate in our programs where potentially failing computations might occur, and make this known to the compiler by defining a data type representing the result of a computation that may or may not have been successful.

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

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

```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 x = if x > 3 then Success x else Failure
```

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 with ghci.

Composing erroneous functions

Now let us return to original goal: composing, or chaining together, computations that might fail. As a warm-up, here's a simple example which "composes" two failing computations, `safeDiv` and `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.

But what happens if we try 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
> computeInt 0
Failure
```

Even though this code make sense and is quite concise compared to other programming languages, 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.

Lift: using `Integer` computations with `MaybeInt`

Suppose we have a `MaybeInt` value, the result of some failing computation, and we wish to perform normal `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:

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

```haskell
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!

```haskell
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 to avoid a `case` expression.

Now let us see how to compose these computations. First, a quick look at how regular function composition works using the `(.)` function, which takes two functions `f` and `g`, and returns a new function that applies `g` and then `f`:

```haskell
> ((+3) . (*2)) 8
19
> ((+3) . (*2) . (+1)) 8
21
```

Suppose we want to replace `8` with `Success 8`. Using `lift`, we can still get this composing behaviour:

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

That's a bit ugly. Keep in mind that we can lift arbitrary `Integer -> Integer` computations, so we can "factor out" the `lift`:

```
> 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.

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`, and so 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:

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

```haskell
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 the `Maybe` type comes in. This is a built-in Haskell type, which is defined in the following way:

```haskell
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. By using a type variable in the data definition of `Maybe`, we can create `Maybe` values for *any* type! For example:

```haskell
> :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!!)

```haskell
(~>) :: 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
```

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. However, this isn't always desired for our programs – for creating complex models of the real world, which certainly has 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 transforming their inputs into outputs. 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
```

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

We're essentially "capturing time" in intermediate variables.

precisely analogous to the clumsy use of `case` expressions to manually process `Nothing` and `Success x` values from the previous section. So, inspired by the combinators 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.

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:

```haskell
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 `()`:

```haskell
push :: Integer -> StackOp ()
push x = \s -> ((), x:s)
-- equivalently, push x s = ((), x:s)
```

Don't get hung up on the overloaded notation. Depending on the context, it's quite easy to tell if `()` represents a value or a type!

Composing stack operations

Now that we have the necessary types in place, let us return to the main task at hand: elegantly composing `StackOp`s. Consider a simplification of `switchTopTwo` in which we only pop off the top two items, returning the second (and losing the first).

Note the use of the underscore to indicate a part of a pattern that will not

```
popTwo :: StackOp Integer
popTwo s =
    let (_, s1) = pop s
    in  pop s1
```

Our new operator >>> is a simple generalization:

pronounced "then"

```
-- (>>>) combines two StackOps into one,
-- performing the first, then the second
-- Note that 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
```

But 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 `StackOp`s 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 `StackOp`s 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 start to see why...

```haskell
(>~>) :: 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`:

```haskell
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:

```haskell
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:

```haskell
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.

```haskell
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 the final line is a `StackOp Integer`.

This looks pretty nifty already, but that last line 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 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 "lift" g up to apply it to the result
(~>) :: 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?

Exercise Break!

1. 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.)
2. 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.
3. 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!

4. 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])
```

5. Using `returnVal`, `sumOfStack` so that it does not "mutate" the stack.
6. Using `returnVal`, re-implement `removeSecond`, `removeThird`, and `removeNth` so that they also `return` the removed item.
7. Implement `len :: StackOp Integer`, which computes the number of elements on the stack. Do not "mutate" the stack.
8. Implement `stackMap :: (Integer -> Integer) -> StackOp ()`, which takes a function `f` and mutates the stack by applying `f` to every item.
9. Implement `stackFilter :: (Integer -> Bool) -> StackOp ()`, which creates a `StackOp` that removes all elements in the stack that don't satisfy the input predicate.
10. Implement `stackFold :: (a -> Integer -> a) -> a -> StackOp a`, which folds a function and initial value across a stack. Leave the stack unchanged.
11. Implement `(>>>)` and `(~>)` in terms of `(>~>)`.

Haskell Input/Output

In addition to mutating state, another common type of *impure* function is one that interacts with a user or some external data storage; in other words, input and output. Note that even though we've been programming without explicit mention of I/O since the beginning of the term, it's always been present: the very idea of a REPL requires functions that take in user-generated input, and to actually display the output to the user as well.

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*.

So these actions are certainly supported in Haskell, even though we haven't used them so far. It turns out that I/O is handled in Haskell basically the same way as our approach to mutable state from the previous section. Even though both I/O and mutation are impure, they seem quite different at first glance, so this might come as a surprise. Let's take a closer look to understand what's going on.

When we were mutating stacks, our `StackOp a` functions captured two behaviours: returning values, and mutating the underlying stack. We can think about these functions as have a pure part, the computation performed to return a value, and an impure part, a mutation of the stack, which represents the underlying context of the computation. So in functional programming, *pure* functions are simply those that have no context, and impure functions are those that do.

But 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 to both files and user interaction (keyboard, mouse, monitor), we now can translate every we've learned from stack manipulation to I/O!

Think about the possibilities of key-value pairs as the underlying state...

Standard I/O

When we were dealing with stacks, there were only two base mutating functions, `pop` and `push`. Of course, the relevant functions depending on the input sources and output targets, and Haskell provides different libraries for these different types of interaction. With basic console I/O there are only two functions that we're going to cover: `putStrLn`, which prints a string to the console, and `getLine`, which reads a string from the console.

As you are probably getting the hang of Haskell's type system, a natural question to ask when in an unfamiliar environment is to ask about the types of things. Let's try it:

```
> :t getLine
getLine :: IO String
> :t putStrLn
putStrLn :: IO ()
```

`String` is a type synonym of `[Char]`; these types are completely interchangeable in Haskell.

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 the console. The `IO` data type has a type parameter `a`; the type `IO a` represents an I/O action that returns a value of type `a`. So `getLine` has type `IO String` because it is an I/O action that returns a string, and `putStrLn` has type `IO ()` because it returns nothing.

Suppose we want to chain multiple I/O actions together. We did this with `StackOp` by defining two operators `(>>>)` and `(>~>)`. Completely equivalent operators are defined in Haskell with the names `(>>)` and `(>>=)` for this data type:

```
printStuff :: IO ()
printStuff = putStrLn "Hey there" >>
putStrLn "Bye now"

repeatLine :: IO ()
repeatLine = getLine >>= \x ->
putStrLn x

> printStuff
Hey there
Bye now
> repeatLine
David
David
```

Note that the first `David` is user input

Aside: compiled Haskell

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 as second nature 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! As a corollary of this, any function which (say) 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! Awesome.

Types as abstractions

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 surrounding Haskell's types, our ambitions were far grander than recreating a classical object-oriented system in Haskell. Rather than focusing on types representing concrete structures 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.

But defining types like `Maybe`, `StackOp`, and `IO` to represent elements of these computations was only a small part of what we did. After all, 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 even to abstract away how we computed on these types, allowing us to elegantly combine basic functions to create more 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 function for each one. And though their literal definitions differed for each context, their spirit did not. And as always, when we see similarities in definitions and behaviour, we look for an opportunity for abstraction, to precisely identify the constant core. These ideas are expressed in Haskell's type system via a collection of more abstract typeclasses, of which `Monad` is the most famous. If you're interested, I strongly encourage you to visit `https://www.haskell.org/haskellwiki/Typeclassopedia` to start learning more about this new layer of abstraction.

Prolog: Logic Programming

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 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.

`http://www.swi-prolog.org`

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)

```
student(bob).
student(lily).
teacher(david).
teaching(david, csc324).
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!

```
?- student(bob).
true.
?- teacher(bob).
false.
?- person(david).
false.
?- teacher(david).
true.
?- teaching(david, csc324).
true.
?- student(lily), student(bob), taking(lily, csc324).
true.
?- student(lily), student(david).
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.

```
child(alice, bob).
child(bob, charlie).
child(charlie, daenerys).

descendant(X, Y) :- child(X, Y).
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!

1. Implement the predicate `takingAll(Student, Courses)`, which succeeds if and only if `Student` is taking all of the courses in the list `Courses`.

2. Implement the list predicate `reverse(L, R)`, which succeeds if and only if `R` is the reversal of the list `L`.

3. Implement `sorted(List)`, which succeeds if and only if `List` is sorted in non-decreasing order.

4. 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.

```
student(bob).
student(lily).
teacher(david).

?- student(lily).
true.
?- student(david).
false.
?- student(bob), teacher(bob).
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:

```
f(a) :- g(a).
g(a) :- f(a).

?- f(a).
```

Search II: Backtracking

Consider the following addition to our example:

```
student(bob).
student(lily).
teacher(david).

peers(lily, bob) :- teacher(lily), teacher(bob).
peers(lily, bob) :- student(lily), student(bob).

?- peers(lily, bob).
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.

```
student(bob).
student(lily).
teacher(david).

f(a) :- student(bob).
f(a) :- student(lily).
f(a) :- teacher(david).

?- f(a).
true ;
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.

```
student(bob).
student(lily).
teacher(david).

?- student(X).
X = bob ;
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.

```
student(bob).
student(lily).
taking(lily, csc324).

happy(X) :- student(X), taking(X, csc324).

?- happy(X).
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:

```
child(alice, bob).
child(bob, charlie).
child(charlie, daenerys).

descendant(X, Y) :- child(X, Y).
descendant(X, Y) :- child(X, Z), descendant(Z, Y).

?- descendant(charlie, daenerys).
true ;
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.

Remember that the variables have been instantiated at this point.

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.

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:

```
?- descendant(X, daenerys).
X = charlie ;
X = alice ;
X = bob ;
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`.

Technically speaking, `X` is instantiated with a fresh variable to be unified with the rule left side `descendant(X, Y)`.

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)`.

We've deliberately separated steps 3 and 4 to indicate that these are really *two separate instances* of backtracking.

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.

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`:

```
descendant(X, Y) :- descendant(Z, Y), child(X, Z).
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!

1. 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?)
2. 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).
```

3. Suppose we introduced the new fact `child(daenerys, alice)`. Using the *original* definition of `descendant`, explain what can go wrong.
4. Give an example of a Prolog source file and query that produces the following output, and explain why.

```
true ;
false.
```

5. 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.

```prolog
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:

```prolog
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:

```
student(xander).
student(willow).
taking(willow, csc324).
cool(xander).
cool(willow).

happy(X) :- student(X), !, taking(X, csc324).
veryHappy(X) :- happy(X), cool(X).

?- veryHappy(X).
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**.

```
not-p(X) :- p(X), !, fail.
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!

1. Express the following English statement in Prolog: "every student except bob is taking csc324."
2. 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?