A Quarter of Haskell

Stuart Kurtz

uchicago

A Quarter of Haskell

Stuart Kurtz

v1.1
Compiled by Ravi Chugh
January 2023

(v1.0: December 2022)

Released under a
CC BY-NC-SA 4.0 License

uchicago

Artwork by UChicago Creative

Foreword

Stuart Kurtz wrote this book for *Honors Introduction to Computer Science I (CMSC 16100)*, taught at the University of Chicago from 2009 to 2021. It was my privilege to teach this course alongside Stu for seven of these years, and my pleasure to compile his work into this informal volume. The subject at hand is introductory Haskell programming (a quarter of the vast topic, let's say), mostly covered in a 10-week term (a "quarter" at UChicago).

This book is dense, demanding, and—ultimately—rewarding. For motivated readers without programming experience, it may serve as an inspirational foundation for prolonged study. For those with prior experience, it offers a mind-bending perspective on program design—more likely than not, radically different than familiar notions of coding. In either case, I urge you to curl up with a pen, some paper, and perhaps a computer, dedicate ample time to read and think, and enjoy!

Ravi Chugh
August 2022

Contents

I Haskell Basics **1**

1 Introduction **2**

2 Lists **13**

3 Algebraic Data Types **23**

4 Case Study: Peano Arithmetic **33**

5 Functions **43**

6 Type Classes **55**

7 A Brief Introduction to Haskell I/O **64**

8 Case Study: Rot-13 **72**

9 A Few More Things to Comprehend **80**
9.1 Records 80
9.2 `newtype` 83
9.3 List Comprehensions 85

II Core Type Classes and Instances **87**

10 Maybe Monad is Not So Scary **88**

11 Functors **89**

12 Applicative **96**

13 Monads **105**

14 The IO Monad **116**

15 Case Study: The Animal Game **127**

16 Monoids **134**

17 Foldable **143**

18 Traversable **146**

19 Writer, Reader **153**

20 State **163**
20.1 State, I 163
20.2 State, II 173

21 Case Study: Functional Parsing **182**
21.1 Introduction to Functional Parsing 182
21.2 Practical Functional Parsing 191

III Towards the Real World **198**

22 Monad Transformers **199**
22.1 Maybe Monad Transformer is Not So Scary 199
22.2 StateT Example: Sudoku 200
22.3 MTL: Implementation 201

23 Scalability **211**

24 Seq and All That **215**

25 Case Study: Propositional Logic **227**
25.1 Mathematical Preliminaries 227
25.2 Parsing 233
25.3 A Tautology Checker 240
25.4 A Quine's Method Theorem Prover 245

Part I

Haskell Basics

Chapter 1

Introduction

Getting Started[1]

You should set up the Glasgow Haskell system. The easy way is to follow the platform specific instructions on the Haskell Platform page. Note that you many need to add certain directories to your `PATH` environment variable.

Haskell

Haskell is a strongly-typed, lazy, pure, functional language. We'll get around to defining these terms in due course, but suffice it to say that this is not your grandfather's FORTRAN, nor your father's C, nor even the Java or C++ you might have learned in High School. In traditional procedural programming language, programs are compiled (i.e., translated) into a sequence of instructions for manipulating memory; whereas in functional languages, computation can be understood as the mathematical process of disciplined substitution. Programming in Haskell often feels like "programming with mathematics."

Our decision to teach Haskell may seem peculiar, idiosyncratic, and perhaps even iconoclastic. Students who favor languages that are more familiar to them, and more important in the here-and-now commercially, often feel that way. We don't want to get into a debate over our choices today, but instead will lay out our priorities, and ask you to suspend disbelief. A useful metaphor, drawn from hockey (this is a hockey town), is "skating to where the puck is going to be." Time and again, we have seen features, concepts, and even idioms of Haskell taken up by more traditional languages. Our goal is to prepare you for the future, and Haskell's the best vehicle we know for doing so.

[1] RC: Unless otherwise (foot)noted, the material in this book was up-to-date as of 2021. Some things—such as these installation instructions—may have since changed. (See `https://www.haskell.org/` for the latest installation instructions.)

The Past

The past is never dead. It's not even past.
William Faulkner

Early attempts to formalize mathematics ran into a variety of paradoxes, i.e., self-contradictory statements. One such was Russell's Paradox, in which Bertrand Russell defined the set

$$R = \{x \mid x \notin x\},$$

using Frege's axiom of comprehension, which allows one to create a set out of the sets that satisfy some predicate. The paradox comes from asking whether or not $R \in R$. This reduces, by the definition of R, to

$$R \in R \iff R \notin R,$$

a self-contradictory statement, a paradox.

There were several distinct approaches developed used to avoid Russell's paradox.

The most influential approach was Ernst Zermelo's, in which comprehensions defined *classes* rather than sets, and where the intersection of a class with a set is also a set. As only sets can be elements of classes or sets, and one can conclude that R is a proper class, i.e., a class which is not a set, and thereby resolve the contradiction.

Another approach was Russell-Whitehead's theory of ramified types, which required assigning an integer to each variable used in forming a statement, e.g., we'll use the notation $x :: i$ to mean that the variable x is given the type (integer) i. These types had to be used consistently, i.e., if we used both $x :: i$ and $x :: j$ in an expression, then we must have $i = j$. They then restricted the use of $\in$ to be strictly monotonic in type, i.e., $(x :: i) \in (y :: j)$ could be formed only if $i < j$. This resolves the paradox by making the comprehension $(x :: i) \in (x :: i)$ at the core of Russell's paradox syntactically invalid, as no integer i can be chosen to meet the constraint $i < i$.

A first impression is that the theory of ramified types is *ad hoc*; it gets around Russell's paradox by a seeming "magic bullet" that allows us to ignore it. But at a deeper and more sympathetic level, the ramified theory of types prevents circularities in the use of $\in$, by insisting on definitions in which the use of $\in$ is "well-founded in type."

Computers and Programming

Computers are complex devices. We use them to write papers, run spreadsheets, store and manipulate photos and videos, to play games, etc. And their design and construction is likewise complex. Yet at their mathematical core, computers rest on very simple ideas, e.g., a bit is logical element that can be in one of two distinct states (on/off, 0/1, true/false). The processor, in a step-wise fashion, changes the value contained in some bits in a lawful way based on the values contained in other bits. Bits are to computers as sets are to mathematics—the foundation upon which all other representations are built.

Types enter programming as means of interpretations of various collections of bits, e.g., these bits represent an ac-

count balance, those bits represent a photograph, those other bits represent a telephone number. Programming often involves building representations of new types out of the representations of old types, and the defining manipulations of values of those new types in terms of manipulations of the values of old types.

Traditional Programming Languages

In traditional compiled languages like `C`, `C++`, and `Java`, we declare types for every name used in a program. The compiler infers the type of every expression, in a strict bottom-up way (the type of an expression is defined by the types of its constituent parts). In doing so, the compiler enforces the consistent use of these names, e.g., typically you can't add an integer to a floating-point number, but must first convert one to the type of another.

A problem, especially with older languages, is that their type systems were inflexible, which often meant that code either had to be duplicated to work at different types (and duplicated code is a bad thing), or that various nefarious tricks had to be used to get the type system to accept code that couldn't be typed, mooting the point of having a type system at all.

Figure 1.1: Reproduced from https://xkcd.com/1537/ without change. (CC BY-NC 2.5)

Scripting Languages

> *There are three great virtues of a programmer: laziness, impatience, and hubris.*
> Larry Wall

For doing quick-and-dirty programming, traditional programming languages often offend programmers who exhibit Wall's three great virtues. The discipline of declaring types for all names offends their laziness. The time required for compilation offends their sense of impatience. And the idea that their use of the language is constrained by the compiler's need to ensure consistent usage offends their hubris. And so, scripting languages like `Perl` and `Python` eschew a static type-discipline, in favor of dynamic runtime type-checking, and interpretation (which typically has shorter latencies, and facilitates dynamic loading) over compilation.

To make this work, the values stored in memory have to encode their type, and these types have to be checked at run time to select appropriate implementations of various operations, or to produce an error if no such implementation exists. To facilitate rapid development, these languages will often convert types "on the fly" as needed to make sense of a programming construct, e.g., if you try to add an integer to a floating point number in `Python`, the integer will first be converted into a floating-point number, and then the two floating point numbers will be added. A problem is that the specific coercions may not be expected or desired, cf., the adjacent image courtesy of `xkcd.com`.

But this happens automatically, and so panders to the laziness and impatience of the programmer, and sometimes spares them the consequences of their hubris, when the latter is ill-founded. The problem with this is the weasel word "sometimes."

Scripting programmers are confident in their own ability to reason about the actual types that will be encountered in these at runtime, and so are confident that operations will only be called with appropriate and compatible arguments. The right word for this is *hubris*. In practice, many bugs in scripting language programs are revealed through runtime type incompatibilities, which exposes that their programmer's confidence has been misplaced. Moreover, code often requires debugging, and is subsequently maintained, often not by the original programmer. Code is *fragile*, in the sense that small changes can invalidate these chains of reasoning about type usage, and those chains of reasoning are often not available when code is being reworked.

Sooner or later, an end user encounters a runtime error when the code tries to multiply `'dirt'` times `'curtains'`, and no one can figure out how something quite so absurd came to be.

Haskell's Approach to Types

> *Computers should think, and people should play.*
> Carl Smith

Haskell uses a strong, flexible type system. Moreover, it makes widespread use of powerful type inference algorithms that moot the need for most type declarations. This isn't to say that Haskell programmers don't include type declarations, but rather that the declarations tend to serve a different function: they're typically a compiler verified "comment," comprising a contract about name use.

Let's consider some examples, from within the `ghci` Haskell interpreter:

```
$ ghci
GHCi, version 8.10.5: https://www.haskell.org/ghc/  :? for help
Prelude> 1 :: Integer
1
```

This is pretty easy stuff. We're asking the interpreter to evaluate the expression `1`, and telling it that the result should be of type `Integer`. [Note that the `Integer` type is infinite-precision.] Haskell does not actually need the type annotation, cf.,

```
Prelude> 1
1
```

This is all very simple. We can see a subtle distinction if we ask Haskell to evaluate `1`, but telling it that the result should be a floating-point number:

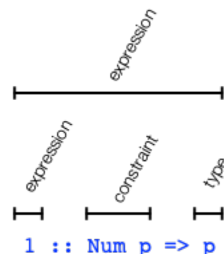
```
Prelude> 1 :: Double
1.0
```

The `1` is a numeric literal, i.e., a bit of syntax that can be interpreted as a number, and that `Double` is the typical type used to represent floating-point values. Moreover, when Haskell prints a `Double`, it always includes a decimal point with at least one fractional digit.

If we look more deeply still, we can ask Haskell for the type of `1`, and we get a surprisingly complicated answer:

```
Prelude> :t 1
1 :: Num p => p
```

What this tells us is that `1` is an expression which has some type `p` which is an instance of the `Num` type class.

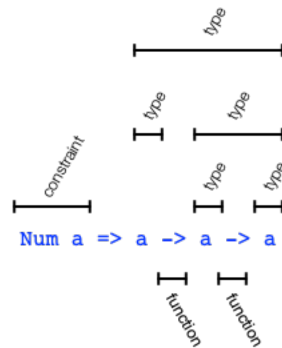


There's a lot going on here! But the point is that when asked Haskell to evaluate `1 :: Integer`, Haskell first had to verify that `Integer` is an instance of `Num`, and then use capabilities of the `Integer` type that all instances of type

class `Num` are required to provide. Likewise, when we asked Haskell to evaluate `1 :: Double`, Haskell first had to verify that `Double` is an instance of `Num`, and then use capabilities of the `Double` class that all instances of type class `Num` are required to provide. In this case, the chain of reasoning went top-down, although since there's only one level, it's a bit tricky to see. Let's consider a more complex example.

```
Prelude> 2 + 3 :: Integer
5
```

This is deceivingly simple. We're asking Haskell to evaluate the expression `2 + 3`, producing an `Integer`. A first bit of complexity here is the use of infix addition. Haskell permits us to use various infix operators, and this is a great notational convenience, but the type system is actually assumes *prefix* operators, and so the expression `2 + 3` is translated to `(+) 1 2`, where `(+)` is the syntax for binary addition written in prefix form. We want this whole expression to have the type `Integer`, and so we have to consider the type of `(+)` which is `Num a => a -> a -> a`.



At this point, it should be clear that the language of types is *also* an expression language, albeit with different operators. What this means is that for any type `a` which is an instance of the `Num` type class, `(+)` is a function that takes two arguments of type `a`, and produces a result of type `a`. [Note: we're telling a bit of a white lie here, cf. currying, which we'll deal with in an upcoming lecture.] Since Haskell knows that the result is supposed to be of type `Integer`, it must first check that `Integer` is an instance of `Num` (which it is), and then it can infer that the arguments are of type `Integer` also. This enables it to interpret `1` and `2` appropriately as numeric literals that represent `Integer` values.

Haskell uses a similar line of reasoning to handle

```
Prelude> 2 + 3 :: Double
5.0
```

Albeit, with a visible difference due to the way Haskell prints floating point numbers.

Still, the Haskell type system has done such a good job of working invisibly behind the scenes that we might wonder why it is there at all. So let's break things. Consider

```
Prelude> (2 :: Integer) + 3 :: Double
```

What could possibly go wrong?

We're asking Haskell to add 2, considered as a value of type `Integer`, to some value represented by the numeric literal 3, obtaining a result that's a `Double`. This is a conundrum for `(+) :: Num a => a -> a -> a`. The type constraint on its first argument requires that `a` be interpreted as `Integer`, whereas the overall type constraint on the expression requires that `a` be interpreted as `Double`, and these are different types! Indeed, that's what the error message we get back from `ghci` says:

```
<interactive>:1:1: error:
    • Couldn't match expected type ‘Double’ with actual type ‘Integer’
    • In the expression: (2 :: Integer) + 3 :: Double
      In an equation for ‘it’: it = (2 :: Integer) + 3 :: Double
```

This brings up an important point. *Compiler error messages can be long and scary*. They're long because the compiler writer doesn't know exactly what information you need to figure out what's wrong, and so they tend to err on the side of giving you too much information, and they're scary because you won't necessarily know how to interpret all of that information, and there's a tendency to assume that because the compiler thought it mattered, you should too.

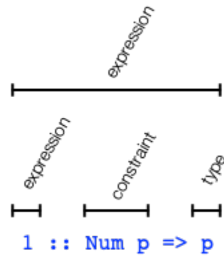
The first line tells us where the error occurred, in this case, in an interactive session, on the first line, at the nineteenth character position. This is TMI for now, but localization is crucial when our programs are spread out over multiple, long files. The second line tells us about the error: at some point, our code seems to require that `Double` and `Integer` are the same type, an inconsistency. The third and fourth lines consists of the expressions whose type analysis failed, and the fifth hints at a capability of the interpreter we haven't yet learned.

Exercise 1.1 *Consider the expression `(1 :: Integer) + 2 * (3 :: Double)`. This will cause a type error, because the two different type constraints are inconsistent. One bit of trickiness is that the inconsistency could be revealed either in the analysis of `(+)` or `(*)`. Provide both analyses.*

A Taste of Programming

The process for programming in Haskell typically involves writing code in a *Haskell Source file*, which will typically have the extension `.hs`. Note that these code files must be plain text files (which is to say, you can't edit them using MS Word or similar word-processing programs, but can using Visual Studio Code or similar text editors). More complex builds will require the use of the `cabal` build system. We'll learn about that later, and for now can rely on a simpler process.

Let's suppose we want to be able to compute the length of the hypotenuse of a right triangle, given the lengths of the sides. We know the pythagorean theorem from our high school math classes. Given a right triangle



the sides a, b, and c are related by the formula $a^2 + b^2 = c^2$. A quick reorganization of this formula gives us

$$c = \sqrt{a^2 + b^2},$$

which determines c as a function of a and b, and which can be easily expressed as code.

```
-- Hypotenuse.hs

hypotenuse a b = sqrt (a^2 + b^2)
```

This is a simple Haskell source file. The first line is a *comment*, identifying this as the file `Hypotenuse.hs`. Comments are annotations we make to the code that the compiler doesn't interpret, but may help future readers (most likely, us!) understand something about the code. The third line defines `hypotenuse` to be a function that takes two arguments, and computes a value based on them. Assuming we've done this correctly, we can load this file into `ghci`:

```
Prelude> :l Hypotenuse
[1 of 1] Compiling Main             ( Hypotenuse.hs, interpreted )
Ok, 1 module loaded.
*Main>
```

The change in prompt isn't important, but we've now added a new function to our system, and that is:

```
*Main> hypotenuse 3 4
5.0
```

At this point, we might consider ourselves done. But... real Haskell programmers wouldn't be happy leaving things here, and your instructors and graders won't be happy if you do. Haskell programmers understand laziness much more deeply than Perl programmers, both as a language evaluation strategy, and as attribute of good programming. In particular, avoiding a little bit of disciplined work now, at the risk of having to do much more in the future isn't lazy, it's a self-defeating strategy that costs us unnecessary work over time. So we'll start by adding a `module` declaration to our program, and some specially formatted comments that will save us time later in using this code.

```
-- | A module for working with triangles.

module Hypotenuse where

-- | Compute the length of the hypotenuse of a triangle from the lengths
--   of its sides.

hypotenuse a b = sqrt (a^2 + b^2)
```

Note that as University of Chicago students, we expect that your comments will be clear and eloquent in their own right, with correct spelling, grammar, and punctuation. It is often said that, "Documentation is the armpit of the industry." We can do better. You *will* do better.

These are `Haddock` comments, and are used to create Haskell Documentation. Note an ordinary comment begins with a double-hyphen `--`, whereas a Haddock comment adds a space and a vertical bar `-- |`. The first comment describes the purpose of the `module`, which typically consists of a number of related definitions, and the comment before the definition of `hypotenuse` describes its intended meaning/use. If we load this, we'll see a minor difference in `ghci`'s prompts:

```
$ ghci Hypotenuse
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Hypotenuse       ( Hypotenuse.hs, interpreted )
Ok, 1 module loaded.
*Hypotenuse>
```

Note how the module name is now incorporated in the prompt.

It is conventional to declare the types of all names defined at the top-level. This requires that we figure out what the type of `hypotenuse` actually is. Fortunately, the compiler has already done this for us, and we can ask:

```
*Hypotenuse> :t hypotenuse
hypotenuse :: Floating a => a -> a -> a
```

At this point, we have a choice. We can simply copy the type that Haskell has computed for us back to our source file, or we can chose a more restrictive type. Let's say, for the sake of argument, that we're confident that we'll only use this with `Double` values. We add the appropriate type declaration our source file as follows:

```
-- | A module for working with triangles.

module Hypotenuse where

-- | Compute the length of the hypotenuse of a triangle from the lengths
```

```
--   of its sides.

hypotenuse :: Double -> Double -> Double
hypotenuse a b = sqrt (a^2 + b^2)
```

And then reload our source

```
*Hypotenuse> :r
[1 of 1] Compiling Hypotenuse       ( Hypotenuse.hs, interpreted )
Ok, 1 module loaded.
*Hypotenuse>
```

Our code works as we'd expect, with the change that `hypotenuse` now has the more restrictive type we declared.

```
*Hypotenuse> hypotenuse 3 4
5.0
*Hypotenuse> :t hypotenuse
hypotenuse :: Double -> Double -> Double
```

One final change, albeit an extremely pedantic one, is to note the duplication of code, the `^2` part. Duplicated code is a bad thing, because it is often very difficult to keep it all in sync. Codebases evolve over time, and divergent evolution of duplicated functions leads to unnecessary complication and size. But we'll illustrate how to deal with this by adding a `square` function:

```
-- | A module for working with triangles.

module Hypotenuse where

-- | Compute the length of the hypotenuse of a triangle from the lengths
--   of its sides.

hypotenuse :: Double -> Double -> Double
hypotenuse a b = sqrt (square a + square b)

-- | Square a number.

square :: Num n => n -> n
square x = x ^ 2
```

Note that in this case, we decided to go with a more general type for `square`. Note as well that our definition of `square` follows its use. This is not a problem (it would be in some languages), as Haskell considers all the declarations in a module when it does its type analysis.

One of the nice things about abstracting out duplicate code is that it often makes it worthwhile for us to think about alternative implementations, e.g.,

```
square x = x * x
```

which won't make a perceptible difference here, but may in more complicated contexts.

The use of Haddock comments enables us to produce nicely formatted documentation. We'll cover how to do this later.

A final remark for today: Haskell programmers care a lot about code quality. The language itself facilitates surprisingly rapid development, but this doesn't mean that real-world Haskell programmers knock off early to hit the bars. They are far more likely continue working on their code *after it meets its specifications*, looking for ways to make it more robust, more general, and more concise. This "tending of the garden" pays huge dividends over time.

Exercise 1.2 *The law of cosines is a generalization of the Pythagorean Theorem, which allows us to compute the length c of the third side of a triangle, given the lengths of the two other sides a and b, and the included angle γ.*

Expand the Haskell script file `Hypotenuse.hs` to include a function `law_of_cosines` which takes three arguments: `a`, `b`, and `gamma`, and returns the length of `c`.

Some notes: your function should take the angle `gamma` in degrees, *but you need to be aware that the Haskell's built in `cos` function expects its argument to be in* radians, *i.e.,*

```
> cos pi
-1
```

Note that `pi` is a predefined constant in Haskell for the mathematical constant π.

Floating point arithmetic in Haskell (like almost all programming languages) is finite precision, and only approximately corresponds to the real numbers of mathematics. My implementation returned the following results, which you might want to use as test data:

```
> law_of_cosines 1 1 60
0.9999999999999999
> law_of_cosines 1 1 120
1.7320508075688772
> law_of_cosines 3  4 90
5.0
> law_of_cosines 3 4 0
1.0
> law_of_cosines 3 4 180
7.0
```

Chapter 2

Lists

Lists are an important and useful data structure in functional programming. Indeed, the name of the first widely used functional programming language, Lisp, is a portmanteau of "List Processing."

In Haskell, a list is a sequence of objects of the same type. Often, we'll describe a list by an explicit enumeration of its elements, e.g.,

```
[1,2,3,4,5]
```

This is a nice notation, but it is important to understand that it is syntactic sugar, i.e., a clear and concise notation that reflects a commonly used pattern. For all it's considerable merits, this notation obscures the essential fact that there are only two kinds of lists:

- empty lists (`[]`), and
- lists that contain at least one element, and so are constructed using `(:)`, which is pronounced "cons."

More pedantically, the list above could be written as

```
1 : (2 : (3 : (4 : (5 : []))))
```

or more tersely (using the useful fact that `(:)` associates to the right) as

```
1:2:3:4:5:[]
```

Now, if you have any syntactic taste at all, you'll prefer the first form, `[1,2,3,4,5]` to the second and third forms. But this misses an important point—it is one thing to have a concise notation for lists, but if you want to write code that manipulates list structure, you have to understand how they're actually constructed.

Defining List Functions By Recursion

Let's start by implementing the standard `length` function:

```
-- | Functions for manipulating lists.

module ListFunction where

import Prelude hiding (length)

-- | Count the number of elements in a list.

length :: [a] -> Int
length [] = 0
length (c:cs) = 1 + length cs
```

There's a fair bit going on here! A first thing to note is that `length` is a predefined function, and Haskell's not happy if the same name has two definitions. We avoid this problem by an `import` of the `Prelude` which hides the definition of `length`. This isn't all that common in practice, but it crops up a lot with list-based functions and introductions to Haskell. Note that the `length` function in the `Prelude` has a different (and more general) type.

Note the type declaration. The `length` function that takes as arguments a list (over an arbitrary base type `a`), and returns the number of elements it contains. This is a *polymorphic* definition, and the resulting function can be applied to lists over any type. We've not seen the `Int` type before, but it is just a finite precision (usually 64-bit, these days) integer. In this case, we don't need a constraint on `a`, because we're not going to do anything that depends on the elements of the function.

The definition above is based on *pattern matching*. Instead of naming the arguments to a function via a variable, as we've done before, the argument positions are inhabited by *patterns*, which either match or fail to match. Each of the clauses of a pattern matching definition are considered in order, and the equation corresponding to the first matching pattern is used.

This definition is also a *natural recursion* on list structure, i.e.,

- There's a natural correspondence between the equations we use to define `length` and the list constructors (`[]` and `(:)`).
- In the case where one of the constituent values of an argument (e.g., the `cs` in the second line above) has the same type as whole argument, our definition applies the function we're defining to that argument.

Let's take the definition of `length` apart, piece by piece

```
length [] = ...
length (c:cs) = ...
```

The parentheses around the cons `(:)` in the second line are not optional. Function application binds more tightly than infix operations in Haskell, and so, without the parentheses, it would interpret `length c:cs` as `(length c):cs`, and interpret your equation as an attempt to define `(:)`!

In the first case, our pattern matches only the empty list `[]`, which contains no elements, so we can define the result directly:

```
length [] = 0
```

In the second case, the pattern matches a `(:)`, and so we're dealing with a list that adds an element onto the front of another list. We can use a recursive call to account for the length of that sublist:

```
length (c:cs) = ... length cs
```

and we can use this result to compute the length of the original list, which has precisely one element more:

```
length (c:cs) = 1 + length cs
```

Finally, an experienced Haskell programmer would make one further change. Portions of a pattern that we don't need can be matched using just an underscore _, thus,

```
length [] = 0
length (_:cs) = 1 + length cs
```

This idiom allows us to focus on the parts of the pattern that are important in reducing an expression.

We can use the same approach in defining the `sum` function, which adds up the elements of a list.

```
-- | Compute the sum of the numbers in a list.

sum :: Num n => [n] -> n
sum [] = 0
sum (x:xs) = x + sum xs
```

In this case, the `Num` constraint is necessary because of our use of ()+.

Let's consider a slightly different problem—summing the squares of the elements of a list. We're going to consider this simple problem from several angles.

A first approach would be a direct implementation, like this:

```
-- | Compute the sum of the squares of the numbers in a list.

sumSquares :: Num n => [n] -> n
sumSquares [] = 0
sumSquares (x:xs) = x^2 + sumSquares xs
```

You will soon be able to write definitions like this pretty quickly, e.g., a sum of cubes function might be written like this:

```
 -- | Compute the sum of the cubes of the numbers in a list.

sumCubes :: Num n => [n] -> n
sumCubes [] = 0
sumCubes (x:xs) = x^3 + sumCubes xs
```

Higher Order List Functions

As natural as this is, and as comfortable as it becomes, experienced programmers want to avoid writing the same code over and over again—so this will inspire them to find appropriate *abstractions* that capture the relevant commonalities, and then to express the particular versions as special cases.

For example, we might abstract away that we're summing functions applied to elements of a list. This gives rise to a definitions like this:

```
-- | Given a function f, compute the sum of the images under f of the elements of a list

sumf :: Num n => (a -> n) -> [a] -> n
sumf f [] = 0
sumf f (c:cs) = f c + sumf f cs

-- | Square a number

square :: Num n => n -> n
square x = x^2

-- | Compute the third power of a number

cube :: Num n => n -> n
cube   x = x^3

-- | Compute the sum of the squares of the numbers in a list.
```

```
sumSquares :: Num n => [n] -> n
sumSquares xs = sumf square xs

-- | Compute the sum of the cubes of the numbers in a list.

sumCubes :: Num n => [n] -> n
sumCubes xs = sumf cube xs
```

Although the second implementation of sumSquares is a bit longer (four lines vs. two), this second version is to be preferred because it achieves a clean factoring of the problem into a recursive summing part, and a function computing part, which makes it easier to build functions that sum other things, whereas in the first version, these aspects are intertwined. Moreover, we've only started with the second version, and there is room for improvement.

One objection to the code above is that we've had to add top-level definitions of the square and cube functions, even though they're not something that we're interested in directly. Of course, at this point, we only know how to do top level definitions! We can simplify this conceptually by adding using local definitions of square and cube where needed:

```
-- | Compute the sum of the squares of the numbers in a list.

sumSquares :: Num n => [n] -> n
sumSquares xs = sumf square xs where
    square x = x^2

-- | Compute the sum of the cubes of the numbers in a list.

sumCubes :: Num n => [n] -> n
sumCubes xs = sumf cubes xs where
    cube x = x^3
```

We can include many definitions within a single where clause, but they all have to indented (and by the same amount) relative to the higher-level clause in which they occur. An alternative to where is let:

```
-- | Compute the sum of the squares of the numbers in a list.

sumSquares :: Num n => [n] -> n
sumSquares xs =
    let square x = x^2
    in sumf square xs

-- | Compute the sum of the cubes of the numbers in a list.

sumCubes :: Num n => [n] -> n
sumCubes xs =
    let cube x = x^3
```

```
    in sumf cubes xs
```

The difference between `let` and `where` is more subtle than whether the definitions come first or last. The `let` construct is part of the *expression* syntax of Haskell, whereas the `where` construct is part of the *definition* syntax.

Exercise 2.1 *Consider the expression* `sumf (sumf square) [[1,2],[3,4]]`. *Do a step-by-step substitution-based evaluation of this expression (you may omit trivial steps, e.g.,* `square 4 => 16` *is permitted).*

But, as they say on late-night commercials, we're not done yet!

Let's factor the problem somewhat differently. In the current implementation, the process of building the sum and evaluating the function remain intertwined, even as we've abstracted out the particular function being evaluated. They can be separated. To that end, let's consider the `map` function, which might be implemented as follows:

```
-- | Map a function across a list.

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

This is another natural recursion, which builds a new list, gathering into a list the image under the given function of each of the elements of the original list. For example

```
> map square [1..4]
[1,4,9,16]
```

Note also another Haskellism for constructing a list. Certainly, writing `[1..1000]` is a lot easier than writing out the list long hand, but it's also, and more importantly, clearer and less error prone.

With `map` in hand, we can write

```
sumSquares xs = sum (map square xs)
```

This is literally a one-liner (assuming we've defined `square`), because `sum` and `map` are predefined in `Prelude.hs`, and it's superfluous to code them ourselves. It may not be clear that we've gained anything, but we're not done yet. Haskell programmers like to manipulate their code, applying meaning-preserving transformations that result in code that more concise and more flexible.

One such tranformation is η-reduction. (The glyph 'η' is the Greek letter "eta.") The way this works is that if we have a definition of the form

```
f x = N x
```

where `N` is an expression not containing `x`, we can cancel the `x` from both sides, and write

```
f = N
```

The mathematical idea underlying η-reduction is the *principle of extensionality*, the idea that two functions are equal if they have the same domain, and have the same value at every point of that domain,

$$(\forall x.\ f(x) = g(x)) \rightarrow f = g.$$

Returning now to our earlier definition, we have

```
sumSquares xs = sum (map square xs)
```

This doesn't take the form we need for η-reduction, but it's close: there's only one occurrence of `xs` on the right-hand side of the definition, and it comes at the end (albeit embedded within essential parentheses).

Haskell is actually a *curried* language, in which all functions are unary. Thus, a function like `map`, formally takes a single argument (e.g., `square` in the example above), which returns a unary function. In Haskell, application associates to the left, so the right hand side of this is actually

```
sum ((map square) xs)
```

The pattern `f (g x)` appears a lot in functional code, so naturally enough there's an operator `(.)`, called *composition* such that `f (g x) == (f . g) x`. We can use this to re-write the definition above as

```
sumSquares xs = (sum . map square) xs
```

and η-reduce to

```
sumSquares = sum . map square
```

which is pretty tight. But was this all worth the effort? For a programmer, this is going to boil down to clarity, efficiency, and maintainability. This may not seem too clear to you just yet, but it will grow on you. You can think about a succession of functions that get applied to a list, read right-to-left, possibly including a summarization function (like `sum`) at the end. And it's very easy to think about changing the parts or order, e.g., altering the summarization function so that a `sum` is replaced by a `product`.

This style of programming is sometimes called "point-free," we don't name the "point" in the domain to define the function. Instead, we use function *combinators*.

Exercise 2.2 *Implement the* `product` *function. This should take a list of numbers, and return their product. Unsur-*

prisingly, `product` *is defined in the* `Prelude`*, which creates a conflict. You can avoid this by using a* `hiding` *clause as above.*

Use your implementation of `product` *to determine the product of the squares of the first numbers 1 through 10.*

Let's suppose now that we wanted to sum the squares of the odd numbers from one to one-hundred. This involves a new programming construct, *guarded equations*:

```
-- | Compute the sum of squares of the odd integers in a list

sumSquaresOfOdds :: Integral n => [n] -> n
sumSquaresOfOdds [] = 0
sumSquaresOfOdds (x:xs)
    | odd x     = x^2 + sumSquaresOfOdds xs
    | otherwise = sumSquaresOfOdds xs
```

This captures a different sort of definition by cases: patterns consider the *structure* of the arguments to a function, guards consider the *values* of those constituents. Note that patterns can introduce new bindings, whereas guards do not. The evaluation of a block of guarded equations works much like the evaluation of a block of pattern-based equations: each guard is considered in turn, and the equation associated with the first true guard is used.

After our discussion of `map`, perhaps you can anticipate the next step. Here we're actually mixing together three distinct things: filtering a list for elements that meet a particular test, squaring each resulting element, and combining the results via `sum`. In this case, the filtering is the new part:

```
-- | Return a sublist comprised of the elements of a list that satisfies a predicate.

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

Note that the `Bool` type consists of values that can be `True` or `False`. Again, `filter` is a built-in function in the `Prelude`, so we don't actually need to implement it. But after our experience from simplifying `sumSquares`, the final form of our solution practically writes itself:

```
sumSquaresOfOdds = sum . map square . filter odd
```

Exercise 2.3 *Let's consider the following problem: compute the sum of the first 100 natural numbers which are divisible by 2 and 3, but not 4 or 9. We'd like to do this in a way that makes it easy to perform similar computations in the future.*

It's not hard to see that we're going to need to use `sum` *and* `filter`*. There's a very nice function in the* `Prelude` *named*

`take`, *which will return the first* `n` *elements of a list. With this, the problem boils down to*

```
result = sum . take 100 . ?? $ [0..]
```

There's some new syntax here:

- *The* `($)` *operator is simply function application, but it differs in a couple of important ways from the usual use of juxtaposition as application:*
  - *juxtaposition has highest precedence (effectively precedence level 10), whereas* `($)` *has the lowest (precedence level 0),*
  - *juxtaposition is left-associative, whereas* `($)` *is right-associative*
- *It isn't necessary to provide an upper-bound on a range expression, thus* `[0..]` *represents the infinite list of natural numbers. Haskell's lazy evaluation strategy makes it possible to use such values, as only as much of the list that is actually needed for the calculation will be built.*

How can we fill in the `???` *First off, it would be nice to have a predicate* `divisibleBy` *such that* `divisibleBy d n` *evaluates to* `True` *if and only if* `d` *evenly divides* `n`*. With such a predicate, we could solve the problem this way:*

```
result = sum
       . take 100
       . filter (divisibleBy 2)
       . filter (divisibleBy 3)
       . filter (not . divisibleBy 4)
       . filter (not . divisibleBy 9)
       $ [0..]
```

This isn't terrible, but it feels just a bit cumbersome. It would be nice to have a function `allp` *which takes two arguments, a list of predicates* `ps`*, and a value* `x`*, and which returns* `True` *if and only if* `p x` *evaluates to* `True` *for every* `p` *in* `ps`*. With this, we could write:*

```
result2 = sum
        . take 100
        . filter (allp [ divisibleBy 2
                       , divisibleBy 3
                       , not . divisibleBy 4
                       , not . divisibleBy 9
                       ])
        $ [0..]
```

This feels a lot better, because it is fairly easy for us to insert or delete tests. But we can due just a bit better still, writing another function `filterAll` *that combines* `filter` *with* `allp`*, so that*

```
result3 = sum
        . take 100
        . filterAll [ divisibleBy 2
                    , divisibleBy 3
                    , not . divisibleBy 4
                    , not . divisibleBy 9
                    ]
        $ [0..]
```

For this exercise, you should define

- `divisibleBy`
- `allp`
- `filterAll`

And verify that all three results are the same. Strive for simplicity and clarity in your code.

There's something quite deep happening with Exercise 2.3, in that this code produces a result in finitely much time, even though some of the subexpressions (consider `[0..]`) describe *infinite* lists. The key feature of Haskell that makes this possible is *laziness*: we don't need to build all of these infinite lists, just enough so that the first 100 elements of the top-level list are defined. So that's all the evaluation that Haskell does!

Exercise 2.4 *The definition of* `allp` *you gave for the previous exercise was probably a recursive definition in the style of the definition of* `map` *or* `filter`*. If you think about the problem a bit, you'll see that you the definition can be reduced to mapping application of a list of functions to a given point with a function that takes a list of booleans, and returns* `True` *if and only if all of the elements of that list are* `True`*. The later function already exists in the* `Prelude`*, as* `and`*. This means that you can define* `allp` *without an explicitly recursive definition, all you need to do is come up with a function that evaluates another function at a given point.*

Give such a definition of `allp`.

Chapter 3

Algebraic Data Types

Haskell provides a rich collection of atomic types to the programmer. There are floating-point types `Double` and `Float`, integer types `Integer` (infinite precision) and `Int` (machine precision, these days, 64-bit), a character type `Char` (unicode) and `Char8` (ASCII), etc.

Naturally, the type system is extensible—we can and often do introduce new types within our programs. One of the principal mechanisms for doing this is algebraic data types (ADTs).

Today, we'll take a look at some (simplified versions) of predefined ADTs. You shouldn't try to redefine these. Bad things will happen. Very bad things.

Simple Algebraic Data Types

The simplest Algebraic Data Type (ADT) is

```
data ()
    = ()
    deriving (Eq,Ord,Show)
```

This is the oddly opaque "unit" type. It is a type that has a single value, which happens to have the same name as it has (and an unusual name it is!). Despite seeming to offer nothing, `()` plays an important role in Haskell, and you'll see it leak out as an argument to a polymorphic type from time to time, e.g., `main :: IO ()`. It may be useful to think of this as a `0`-tuple. The `deriving` clause gives us default implementations of the `Eq`, `Ord`, and `Show` type classes, which work as expected.

```
> :t ()
() :: ()
```

Next up, we have the `Bool` type, for boolean values:

```
data Bool
    = False
    | True
    deriving (Eq,Ord,Show)
```

This defines a data type that has two distinct values, `True` and `False`. It's just a bit! The `Bool` data type gets used a lot, as predicates (condition tests) are naturally boolean valued, and so there are special operators and syntax devoted to `Bool`. The three standards boolean operators are defined via pattern matching definitions, much as we saw last lecture with lists.

```
-- | Boolean negation.

not :: Bool -> Bool
not True  = False
not False = True

-- | Boolean conjunction, a.k.a., "and."

(&&) :: Bool -> Bool -> Bool
False && _ = False
True  && y = y

-- | Boolean disjunction, a.k.a., "or."

(||) :: Bool -> Bool -> Bool
True  || _ = True
False || y = y
```

This brings us to an important rule of Haskell syntax. Each non-operator constructor (and type) begins with a capital letter, whereas variables (i.e., non-operator function names) start with a lower-case letter.

Boolean conjunction `a && b` will be `True` only if `a` and `b` are both `True`, and likewise boolean disjunction `a || b` will be `False` only if `a` and `b` are both `False`.

We've already seen the special syntax (guards) associated with `Bool`. Consider the following predicate definitions:

```
-- | A predicate for even Integers.

even :: Integer -> Bool
even n = mod n 2 == 0

-- | A predicate for odd Integers.

odd :: Integer -> Bool
```

```haskell
odd n = not (even n)
```

The `even` function will return `True` for even `n`, and `False` for odd `n`. Note that `==` is the *equality predicate* and is a function in the `Eq` type class, as distinct from `=` which is used in definitions. (Ordinary mathematical notation often conflates these two distinct meanings.) Note also that there are predefined `even` and `odd` functions of slightly more general type `Integral a => a -> Bool` defined in the `Prelude`.

Guards allow us to use predicates to give different defining equations for based on different *values* of its arguments, in addition to pattern matching, which enables us to make distinctions based on the *structure* of the arguments. Consider

$$|x| = \begin{cases} x & \text{if } x \geq 0, \\ -x & \text{otherwise} \end{cases}$$

the familiar absolute value function for real numbers. We could define

```haskell
-- | The absolute value function

abs x
    | x >= 0 = x
    | otherwise = -x
```

were it not for the fact that `abs` is defined in `Prelude` as a member of the `Num` type class. Let's ignore that nuance for now, and consider the definition via *guarded equations*. Here, rather than having a sequence of equations that involve patterns, we have a sequence of alternative definitions associated with *guards*, i.e., boolean predicates of the arguments. Evaluation occurs by considering each guarded equation in the order they appear in the definition, and using the first equation whose guard evaluates to `True`. Haskell also supports an `if ... then ... else ...` construct, so we could have defined

```haskell
abs x = if x >= 0
        then x
        else -x
```

Haskell syntax doesn't get along with unary operators, and so it's often necessary to write `(-x)`, but we're ok with the bare unary negation here (because it is preceded by `else`, which is a keyword and not an ordinary variable). You might wonder why there are different syntaxes for what are basically the same thing. The guard syntax can be used as a way of modifying pattern matches, and yes, a simple variable *is* a pattern, albeit one that matches everything, whereas an `if ... then ... else ...` can appear anywhere in an expression. This is similar to the distinction we saw with `which` vs. `let`.

Exercise 3.1 *Consider the* collatz *function defined as follows:*

$$\text{collatz}(n) = \begin{cases} 1, & \textit{if } n = 1 \\ 1 + \text{collatz}(n/2), & \textit{if } n \textit{ is even} \\ 1 + \text{collatz}(3n+1), & \textit{otherwise} \end{cases}$$

Give Haskell definitions of `collatz` *using guards, and* `collatz'` *using* `if ... then ... else ...`*. [We can use apostrophes in variable names!] Note that you should use* `div` *rather than* `(/)` *to divide integral values.*

Another simple ADT is `Ordering`,

```
data Ordering = LT | EQ | GT
    deriving (Eq,Ord,Show)
```

Which is used as the return type of the `compare :: Ord a => a -> a -> Ordering` function of the `Ord` type class.

A next step in complexity is polymorphic ADTs. These are ADTs that take one or more type variables, e.g., the pair type, which could be written as

```
-- | The Pair data type

data Pair a b = Pair a b
```

But since tuples arise frequently in programming practice, it is convenient to have a special, terse and familiar notation for them, specifically, `(a,b)`, and Haskell supports this. Somewhat counter-intuitively, this leads to the following definition:

```
-- | The 2-tuple type

data (,) a b = (,) a b
```

This is notationally a bit opaque, but `(,)` is (by a mild abuse of notation) just the prefix form of pairing operator "`,`" (the abuse being in the fact that `,` isn't an operator, but in fact is special syntax...). As a type, this abstracts a value that contains values of two other types. This is often useful, e.g., when we want to return multiple results from a single function, e.g., the `divMod` returns a pair consisting of the quotient and remainder of a division.

```
> :t divMod
divMod :: Integral a => a -> a -> (a, a)
> divMod 10 3
(3,1)
```

There are 3-tuples, 4-tuples, all the way through 62-tuples. Note that tuples support `Eq`, `Ord`, and `Show` instances when their constituent types do, via deriving instances.

Exercise 3.2 *The* `Prelude` *provides some simple functions for dealing with pairs,* `fst` *and* `snd` *for extracting components, and* `curry` *and* `uncurry` *for swizzling between functions that expect two arguments, either separately, or packaged together via a pair.*

Unfortunately, analogous functions do not exist for 3-tuples, etc. Code a Haskell module `Triple.hs` *which provides analogous functions* `fstOf3`, `sndOf3`, `thirdOf3`, `curry3`, *and* `uncurry3`.

Polymorphic ADTs

Let's consider a simple example of a polymorphic type:

```
data Maybe a
    = Nothing
    | Just a
    deriving (Eq,Ord,Show)
```

We can think of `Maybe a` as a type that contains `0` or `1` `a` values. "`Maybe` wrapped types" often come up in the context of error handling, e.g., we might use `Just a` to denote a computation that successfully completes with the value `a`, and `Nothing` to denote a computation that encountered some sort of error.

Let consider a fairly complex, but informative example. Consider the `Num` type class. Its instances have to provide `()+`, `(-)`, and `(*)` (and a few other functions), but not `(/)`. We might ask, why is `(/)` different? There are at least a couple of answers. One is that we expect division to work a bit differently with floating-point rather than integral types, and indeed Haskell provides three different functions, `(/)`, `div`, and quo for division, the first for ordinary floating point division, the other two for integer division with different rounding behavior. But there's another reason, which is that `(/)` isn't total. We have to worry about division by zero.

By default, division by zero generates an *exception*, which is an ugly, heavy-weight control structure in Haskell, and beyond this class. But we can approach the problem differently using `Maybe`, using `Nothing` as a `NaN`, a value that is not a valid number. With some processor configurations, divisions by zero in floating point arithmetic result in `NaN` (not a number), but there is no native `NaN` for integral types.

As we'll see, we can use `Maybe` wrapped integral types to "compute" an entire expression, and then check the result for an error. If the result is `Nothing`, then some sort of error occurred, whereas if it is `Just a`, then the computation succeeded with the value `a`, and we can use ordinary code (e.g., a pattern match, rather than an exception handler) to distinguish between the two. To that end, let's consider some code that is much more complicated than we've seen before:

```
-- | Wrapping a number in Maybe

module MaybeNum where

-- | Derived instance definition for Num (Maybe n) given Num n.

instance Num n => Num (Maybe n) where
    Just a + Just b = Just $ a + b
    _ + _ = Nothing

    Just a - Just b = Just $ a - b
```

```
    _ - _ = Nothing

    Just a * Just b = Just $ a * b
    _ * _ = Nothing

    negate (Just a) = Just $ negate a
    negate _ = Nothing

    abs (Just a) = Just $ abs a
    abs _ = Nothing

    signum (Just a) = Just (signum a)
    signum _ = Nothing

    fromInteger i = Just $ fromInteger i
```

The `instance` construct will make a `Maybe n` an instance of the `Num` type class, so long as `n` is an instance of `Num` itself. The `Num` type class has quite a few functions we have to implement. The `$` operator is just application, albeit right associative and of low precedence, as opposed to the usual implicit space, which is left associative and of highest precedence. Using `$` as we do here spares us a few parentheses. This is a "deriving instance," as it defines instances for a whole lot of types and not just one. This is something that you can't do with a Java `interface`.

This seems peculiar, but it works, e.g.,

```
> 1 + 3 * 3 :: Maybe Integer
Just 10
```

Why might we want such an instance declaration? What does it gain us? Patience.

```
-- | safe division

infixl 7 //

(//) :: (Eq n, Integral n) => Maybe n -> Maybe n -> Maybe n
Just a // Just b
    | b == 0 = Nothing
    | otherwise = Just $ div a b
```

First we define a new infix name, `(//)`, which will have the same fixity and precedence as ordinary division. Then, we implement our new `(//)` operator as integer division, including a denominator check which returns `Nothing` if the denominator equals zero. The `Integral` type class adds `div`, and the `Eq` type class added `(==)`.

```
> 1 + 2 // 0
```

```
Nothing
```

This is what is gained: We've now have modified types that allow for errors to propagate seamlessly through the usual process of arithmetic expression evaluation, and so don't require exception handling or other exotic flow control to deal with if they occur. If you've programmed in Swift, this may remind you of Swift's optional types, and the way they can propagate errors through call chains. Of course, Swift appropriated the concept from Haskell (somewhat unusually, with credit given), and not the other way around.

Exercise 3.3 *We could have approached this example by creating a deriving instance* `Integral n => Integral (Maybe n)`, *as* `div` *is part of the* `Integral` *type class. But this would involve implementing several* other *type classes. Explore the documentation, to determine what type classes are involved, and what functions they contain.*

Recursive ADTs

You may have guessed that the list data type from last lecture is itself just an ADT provided by the `Prelude`. We can imagine that the list data type is defined as follows:

```
infixr 5 :

data [] a
    = []
    | a : [] a
```

The prefix use of `[]` as a type constructor is a bit unusual, but follows a pattern that we saw with tuples. We've seen the usual syntax Haskell uses for list types in the last lecture, i.e.,, the more familiar `[a]`, but it's important to be aware of this more primitive form.

Part of what is significant about this definition is that it is *recursive*, i.e., the data-type is defined in terms of itself. This is a very useful facility, and as we've seen, one that can give rise to natural recursions in the functions that manipulate data of this type.

As with tuples, lists belong to the `Eq`, `Ord` and `Show` type classes when their underlying types do.

Lists are a widely used in Haskell to build a variety of types, e.g.,

```
type String = [Char]
```

The `type` declaration, somewhat counter-intuitively, doesn't introduce a new type (that's what `data` is for), but instead a *type alias*, i.e., another name for a type. This particular representation choice for `String` means that we can use ordinary list-based functions for working with strings, e.g.,

```
> length "foo"
```

```
3
```

or

```
> "foo" ++ "bar"
"foobar"
```

We haven't looked at the (++) function before, and this is a natural time to do so:

```
-- | Append two lists.

infixr 5 ++

(++) :: [a] -> [a] -> [a]
[] ++ bs = bs
(a:as) ++ bs = a : (as ++ bs)
```

Note here that both (++) and (:) are `infixr 5`, and so the parentheses on the right hand side of the last equation in this definition are not strictly speaking necessary, but this depends on the *associativity* of the operators, and not merely their precedence, and so is more fragile than usual. Adding the parentheses here seems prudent: it doesn't detract from clarity, and doesn't make what seem like unreasonably optimistic assumptions about the reader's ability to remember the minutiae of precedence *and* associativity.

A particularly useful data structure, built out of tuples and lists, is an *association list*, `[(a,b)]`. Association lists are often used as a simple implementation of *dictionaries*, where `a` is the type of the key (or entry or definiendum) and `b` is the type of the value (or gloss or definiens).

The following function is defined in the `Prelude`, and it shows how several of the language features we've been using work together:

```
-- | Look up a value by key from an association list, returning a Maybe wrapped result.

lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup _ [] = Nothing
lookup a ((k,v):ps)
    | a == k    = Just v
    | otherwise = lookup a ps
```

The design choice here, using `Maybe` to wrap a result so that we have a way to deal with the "key not present" problem isn't the only choice possible. Other possibilities involve returning an unwrapped value of type `b`, and deal with the possibility that the key isn't present in a different way:

- If we're confident that the key will always be there, we could throw an exception if it's missing. This gives us

the simplest code at the call site, but involves some programming risk.

- We could provide a default value to be returned as an argument to a modified `lookup` function.
- We could modify the representation of a dictionary, providing a default value.

Let's implement all three. For the sake of simplicity, we'll use our base `lookup` function in all three cases.

```
-- | Look up a value by key from an association list, throwing an error if the key is missing.

lookupWithError :: Eq a => a -> [(a,b)] -> b
lookupWithError a dict = case lookup a ps of
    Nothing => error "key not found"
    Just v => v

-- | Lookup a value by key from an association list, returning a default value on missing key.

lookupWithDefault :: Eq a => a -> b -> [(a,b)] -> b
lookupWithDefault a b dict = case lookup a dict of
    Nothing => b
    Just v => v

-- | A dictionary with default value:

data Dictionary a b = Dictionary b [(a,b)]

-- | Lookup a value by key from a Dictionary, returning the Dictionary's default value
--   on missing key.

lookupInDictionary :: Eq a => a -> Dictionary a b -> b
lookupInDictionary a (Dictionary default dict) = case lookup a dict of
    Nothing => default
    Just value => value
```

Note that `case` is an expression that enables pattern matching. Although we don't use it here, guarded equations can be used with the patterns in a `case` statement just as they are used in definitions.

Exercise 3.4 *Implement the three functions* `lookupWithError`, `lookupWithDefault`, *and* `lookupInDictionary` *by direct recursions, i.e., without calling* `lookup`.

Exercise 3.5 *A common data structure is a* rose tree. *This is a kind of tree in which each node holds a value of a particular type. The actual declarations are a bit different (they rely on Haskell's* record *syntax, which we'll see in due course), but they amount to:*

```
-- | A rose tree.
```

```
data Tree a = Node a (Forest a)
type Forest a = [Tree a]
```

Note that recursion can be mutual, *and need not be direct.*

A tree consists of a node, which has two constituents: the value of type `a`, *and a list of children.*

Rose trees are often used to represent semi-structured data, e.g., an outline, or an XML infoset.

Write a function `preorder :: Tree a -> [a]` *which returns the values contained in a* `Tree` *as a list, based on a preorder traversal (i.e., the value at a node comes before the values at its children). It may be helpful to know about the function* `concat :: [[a]] -> [a]`, *which flattens a list of lists into a simple list. (Note that the actual type of* `concat` *is just a bit more general than this.)*

Chapter 4

Case Study: Peano Arithmetic

Today, we're going to jump into the deep end of the pool, Haskell wise. Don't be put off if you don't understand everything. Today's lecture is mostly literature, i.e., your task as a student is to read and expand your mind. I'll expect a bit of mimcry at first, real comprehension will come in time.

Our subject for today is the Peano-Dedekind axioms for arithmetic. Yes, we're going to redo third-grade math, but as it might be done by a graduate student studying mathematical logic. The underlying structure is the natural numbers, `0`, 1, 2, ...

The Peano-Dedekind axioms posit the existence of a constant 0 and function s (the *successor* function) with the following properties:

- 0 is a natural number,
- s is a *one-one* function from the natural numbers to the natural numbers, i.e., for all natural numbers a, $s(a)$ is also a natural number, moreover, for all natural numbers a and b, if $s(a) = s(b)$ then $a = b$,
- `0` is not a successor, i.e., for all a, $s(a) \neq 0$.
- every natural number is either `0` or a successor, i.e., for all a, $a = 0$ or there exists b such that $a = s(b)$.

We'll start with a data definition:

```
data NaturalNumber = Zero | S NaturalNumber
    deriving (Show)
```

This introduces a new *type*. Haskell's type system is a central part of the language, and our ability to introduce new types that exploit built-in facilities of the language is both useful and powerful. The type system partitions the universe of *values* and *expressions* into equivalence classes, and limits how the elements of these equivalence classes can be combined. Types are denoted by *type definitions* (as above) and *type expressions* (of which more later).

The definition above expresses the intent that natural numbers come in two distinct forms (it is helpful here to read `|` as "or"): `Zero`, and `S x` where `x` is also a `NaturalNumber`. Thus, `Zero`, `S Zero`, `S (S Zero)` are values (and

expressions) of type `NaturalNumber`, which we ordinarily know by the names `zero`, `one`, and `two` respectively. Note here that `S` is not just a function, it is a *constructor*, which can be thought of as a kind of labeled box that holds another value of type `NaturalNumber`. Haskell uses a simple, mandatory notational convention here: variables have names that begin with lower-case letters (including _), while constructors begin with an upper-case letter. The `deriving (Show)` instructs Haskell to print natural number values via constructor-based expressions.

Let's define a few names for common natural numbers:

```
-- common names for small natural numbers

zero  = Zero
one   = S zero
two   = S one
three = S two
four  = S three
five  = S four
six   = S five
seven = S six
eight = S seven
nine  = S eight
ten   = S nine
```

There are several things to notice here. The first is a simple comment, which is introduced by a double hyphen, `--`. This, and any text that follows on the same line, is ignored by Haskell. This text is there to support *our* understanding. The next is the sequence of *definitions*, in which *variables* are *bound* to *values* denoted by *expressions*. Note that in Haskell, like many programming languages, we use `=` to indicate binding. This essentially creates equality by fiat, but it is distinct from the equality predicate, which we'll see next.

```
$ ghci NaturalNumbers.hs
> three
S (S (S Zero))
```

Equality and Ordering

First, we'll write code that makes the `NaturalNumber` type an *instance* of the `Eq` typeclass.

```
instance Eq NaturalNumber where
    Zero == Zero = True
    Zero == S y  = False
    S x  == Zero = False
    S x  == S y  = x == y
```

We can make a tiny code improvement here: note that the `x` and `y` on the left-hand side of the bindings in the 2nd and 3rd line of the definition are not used on the right hand side. This means that we don't actually need to name a variable:

```
instance Eq NaturalNumber where
    Zero == Zero = True
    Zero == S _  = False
    S _  == Zero = False
    S x  == S y  = x == y
```

In Haskell, types are organized into *typeclasses*. This is a very different meaning of "class" than you might have encountered in a language like Java. It's actually closer to Java's notion of an `interface`, but more powerful. The significance of a typeclass is that all types that belong to a given typeclass define certain common names. In the case of the `Eq` typeclass, instances must implement one or both of `==` (the equality predicate) or `/=` (the inequality predicate), and having done so, will have both defined for them. In this case, we're defining the equality predicate `==`, based on the Peano-Dedekind axioms: `Zero == Zero` is a special case of reflexivity, `Zero` does not equal any successor, and two successors are equal if and only if their predecessors are equal (this follows from s being *one-to-one*).

There's no essential difference between defining an ordinary function (one that is applied in prefix to zero or more argument/s), and defining an infix binary function like `==`: we bind patterns to expressions to expressions in both cases. Indeed, from Haskell's point of view, infix binary functions are just syntactic sugar for ordinary (curried) binary functions. Having said that, the definition of equality *is* interesting, because it's *recursive*, i.e, we are defining `==` *in terms of itself* when we write `S x == S y = x == y`. Being able to define a function by recursion is powerful, but it's not free. Such a definition is going to drive a sequence of substitutions (as we saw in Lecture 1), a sequence that might or might not terminate. In this case, `==` will terminate on grounded elements of the natural numbers, because we defined it without recursion on base cases (`Zero`), and the recursive definition in constructed cases involved recursive calls of the function being defined on proper substructures of the argument. We will call recursions like this *natural*, or *structural* recursions, and the proof of termination follows from ordinary mathematical induction.

```
> ten == ten
True
> ten == nine
False
> ten /= nine
True
```

Let's understand how this works on a small example:

```
two == one
==> S one == S zero
==> one == zero
==> S zero == Zero
==> False
```

How many of you noticed the weasel-word "grounded" up above?! This is actually important, and it gets to one of the ways that Haskell is *not* like other languages that you might have experienced. Let's talk about this particular case. `Zero` is grounded. This is a specific case of a general principle: all of the nullary constructors (i.e., *constants*) of an algebraic type are grounded. Also, if `n` is grounded, then `S n` is also grounded. Again, this is a specific case of a general principle: applying a k-ary constructor to k grounded values, results in grounded value. At this point, you're probably asking yourself, "Are there any ungrounded values?!" *Yes*. Consider

```
infinity = S infinity
```

What is `infinity`? Note `infinity` is defined recursively, but not *naturally*, because the right-hand side of the recursion mentions the left-hand side (rather than a proper substructure thereof). Therefore this is a potentially *dangerous* definition. Note that there's nothing special about Haskell in allowing you to make such a definition – you can do something analogous in any general purpose programming language. The difference is that in Haskell, such definitions can be *useful*, which is to say, they can be used in terminating computations. Of course, you wouldn't want to simply evaluate `infinity`:

```
> infinity
(S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S ...
^-C
Interrupted.
>
```

But this works:

```
> infinity == one
False
>
```

Why?! Let's trace it...

```
infinity == one
==> S infinity == S zero
==> infinity == zero
==> S infinity == Zero
==> False
```

You're all invited to try this in Java. Get back to me when it finishes :-).

Whether the inclusion of objects like `infinity` in our `NaturalNumber` type seems like a good idea or not, it is an inevitability, and this is one of the ways that computational systems can fail to exactly correspond to mathematical systems. Objects like `infinity` are explicitly excluded from Peano-Dedekind Arithmetic by the induction axiom, which essentially says that all (Peano-Dedekind) natural numbers are grounded.

Adding `infinity` has other costs, e.g., computed equality is no longer reflexive, and this can be a trap in naïve reasoning about code.

```
infinity == infinity
==> S infinity == S infinity
==> infinity == infinity
...
```

This leads to a high-speed twiddling of the CPUs thumbs, but rather than termination with the result `True`. A general rule of thumb is that natural recursions are safe, i.e., they terminate, because all recursive calls are made to *simpler* arguments, and therefore a termination proof can be obtained by induction over the definition of the set of grounded values. The problem is that the grounded values of the domain aren't exhaustive.

Exercise 4.1 *Haskell contains an* `Ord` *typeclass, consisting of types with a natural trichotomous ordering. Provide an* `instance` *declaration that adds* `NaturalNumber` *to the* `Ord` *typeclass.*

The documentation for the `Ord` *typeclass can be found here, as well as on your system (assuming you've properly installed the Haskell Platform). The most satisfactory way to do this is by implementing the* `compare` *function, which takes two arguments, and returns* `LT`, `EQ`, *or* `GT`, *according as to whether the first argument is less than, equal to, or great than, the second argument, respectively.*

To do this, complete the following definition

```
instance Ord NaturalNumber where
    compare Zero Zero    = EQ
    ...
```

Once you've done this, you should be able to do simple comparisons, e.g.,

```
> ten >= nine
True
```

Nonterminating computations are inevitable in any general-purpose programming language. The problem is not just `infinity`, which is in some sense a natural "limit point" in the natural numbers, but also definitions that twiddle their thumbs without creating new constructors, e.g.,

```
loop :: NaturalNumber
loop = loop
```

Here, we provide a type ascription to `loop` because there's not enough information otherwise for the compiler to figure it out. This may seem like a silly example, but it's easy to achieve the same effect accidentally, and often

difficult to sort out when you do. Oddly enough, `loop` can be used to define new natural numbers, and we can even compute with them:

```
> S loop == Zero
False
```

but

```
> loop == loop
...
```

never terminates. It might seem to be a defect of Haskell that we can't weed out "unintentional" values of type `NaturalNumber` like the values of `loop` and `infinity`, but if so, it's a defect that all current *and future* programming languages must share, a consequence of the undecidability of the halting problem, one of the most fundamental theorems of computability theory, in essence a restatement of the Gödel's celebrated incompleteness theorem for Peano Arithmetic, which oddly enough is the subject of *this* lecture.

Addition and Multiplication

The principal operations of Peano-Dedekind arithmetic are addition and multiplication, which are recursively defined. This is most naturally accomplished by adding `NaturalNumber` to Haskell's `Num` typeclass. Doing this correctly requires implementing a number of functions, `+`, `*`, `-` (or `negate`), `abs`, `signnum`, and `fromInteger`. We'll get the definitions of `+` and `*` directly from the Peano-Dedekind axioms.

```
instance Num NaturalNumber where
    x + Zero = x
    x + S y  = S (x + y)

    x * Zero = Zero
    x * S y  = x + x * y
```

Haskell will permit us to provide partial implementations of typeclasses, although not without complaint. Even so, this is a useful way to proceed, because it allows us to test our code incrementally.

```
> two * three + one
S (S (S (S (S (S (S Zero))))))
```

We'll next add an implementation of `fromInteger`, along with a simple helper function `nat`:

```
instance Num NaturalNumber where
    ...
```

```
    fromInteger n
        | n > 0  = S (fromInteger (n-1))
        | n == 0 = Zero


nat :: NaturalNumber -> NaturalNumber
nat = id
```

The definition of fromInteger involves a programming new construct, *guarded equations*. The idea here is that we'll define a function by cases through a sequence of predicates (tests) and equations. The reduction of a term will evaluate each of the predicates in turn, until a predicate evaluates to True. Once this happens, the corresponding equation is used. The point to this definition is that it consciously avoids defining fromInteger on negative inputs, as this is meaningless in the context of the natural numbers.

The nat function is just the identity, but with a restricted type. We can use nat to force the interpretation of a literal like 1234 as a NaturalNumber via the fromInteger function.

The significance of these definitions is that they enable us to take advantage of Haskell's input processing. Consider a simple expression like:

```
2
```

What is the type of 2? The answer is that it can belong to any type that belongs to the Num typeclass, although it defaults to Integer (an "infinite-precision" integral type). By adding fromInteger, we make expressions like this meaningful:

```
two + 27
```

In effect, the two, which can only be a NaturalNumber, forces both + and 27 to be interpreted as a binary operator on NaturalNumber and a NaturalNumber respectively. The function nat does the same thing. Here we see both a type declaration (of which much more later), and a definition. Using nat enables us to force the interpretation of a numeric literal to be a NaturalNumber. Thus, we can write

```
> nat (2 * 3 + 4)
...
```

As we'll see later, this forces the expression 2 * 3 4+ to have the type NaturalNumber, and this in turn will ultimately force 2, 3, and 4 to have type NaturalNumber too.

```
...
S (S (S (S (S (S (S (S (S (S Zero))))))))))
```

We can write this with one fewer character via

```
> nat $ 2 * 3 + 4
S (S (S (S (S (S (S (S (S (S Zero)))))))))
```

Here, `$` is a low precedence *right* associative infix operator for function application. It is a friend, a good friend, which can save us from the "lots of irritating single parentheses" that is sometimes used in a pejorative etymology of "Lisp."

Exercise 4.2 *Complete the definition of* `instance Num NaturalNumber` *by implementing* `-`, `abs` *and* `signnum` *in as meaningful a manner as possible. Note that for natural numbers, subtraction is truncated at* `Zero`*, i.e., evaluating* `one - ten` *should return* `Zero`.

Exercise 4.3 *It's easy to use pattern matching to write simple number theoretic functions using pattern matching like this:*

```
even Zero = True
even (S Zero) = False
even (S (S n)) = even n
```

but it's much more convenient to take advantage of the machinery we've built up, and write it like this:

```
even n
    | n == 0 = True
    | n == 1 = False
    | otherwise = even (n-2)
```

and indeed, I tend to think of the first form as a translation of the second into the particular representation we used for `NaturalNumber`*, whereas the later is "more generic," and because it doesn't make assumptions about representations, works perfectly well for other representations.*

Write such representation-independent implementations of `odd` *and* `remainder`*. Note that because* `even` *and* `odd` *are defined in the Haskell Prelude, it's necessary to begin the Haskell source file that includes your definitions with an import statement that explicitly hides them:*

```
import Prelude hiding (even,odd)
```

Laziness

Thus far, we've been modeling Haskell's evaluation mechanism via term rewriting. This is not exactly correct, but it will do for now.

Now, my friend Robby, an exquisite Scheme programmer, would instinctively re-write the definition of + as follows:

```
x + Zero = x
x + S y  = S x + y
```

Remember here that application binds more tightly than infix operations, so this change is from the original `S (x + y)` to `(S x) + y` in Robby's code.

If you ask Robby why, he'd talk about "tail recursion," and about why the second form is tail recursive (and so runs in constant stack space), whereas the first form is not (and so involves putting a bunch of pending "apply `S` to the result, and return" frames on the stack). And he'd be perfectly right. In Scheme. But Haskell is not Scheme, and Robby's definition involves a "loss of laziness" if carried over unreflectively from Scheme to Haskell. Let's see how that works.

Suppose we want to evaluate `ten + ten == one`. If we use the original definition, we'd have

```
ten + ten == one
==> ten + S nine == S zero
==> S (ten + nine) == S zero
==> ten + nine == zero
==> ten + S eight == Zero
==> S (ten + eight) == Zero
==> False
```

In effect, we only needed to apply the successor case of the addition rule twice to prove that `ten + ten /= one`. Whereas, with Robby's implementation, we'd have to do something like this:

```
ten + ten == one
==> ten + S nine == S zero
==> S ten + nine == S zero
==> S ten + S eight == S zero
==> S (S ten) + eight == S zero
==> S (S ten) + S seven == S zero
==> S (S (S ten)) + seven == S zero
...  -- 13 steps elided
==> S (S (S (S (S (S (S (S (S (S ten))))))))) + Zero == S zero
==> S (S (S (S (S (S (S (S (S (S ten))))))))) == S zero
==> S (S (S (S (S (S (S (S (S ten)))))))) == zero
==> S (S (S (S (S (S (S (S (S ten)))))))) == Zero
==> False
```

Note that we got the same answer both times, but that Robby's implementation required that we completely reduce the addition, whereas the original implementation did not. This point can be driven even further. With the first definition, we can evaluate `nine + infinity > ten`, and promptly obtain the result `True`. With Robby's definition,

this would run forever without returning a result.

Exercise 4.4 *The axiom for the successor case of multiplication in the Wikipedia article is not what I remembered from graduate school. The crux of the matter is whether you consider* `S x = x + 1`, *or* `S x = 1 + x`, *and the Wikipedia article assumes the second. If we assume the first, as I remember it, the natural definition for the successor case of multiplication is subtly different:*

```
x * S y = x * y + x
```

The form that I remember is better Haskell, in the sense that it's lazier. Illustrate this by a simple example, as above.

A complete version of today's program (and a reasonable starting point for today's exercises) is here: `NaturalNumber.hs`.

Chapter 5

Functions

Lambda

Consider our earlier definitions of `sumf`:

```
-- | Given a function f, compute the sum of the images
--   under f of the elements of a list

sumf :: Num n => (a -> n) -> [a] -> n
sumf f [] = 0
sumf f (c:cs) = f c + sumf f cs
```

The `sumf` function is higher-order, because it takes a function as an argument. This can be inconvenient, as in our implementation of `sumSquares`, as we needed to define a top-level `square` function to pass to `sumf`.

```
-- | Square a number

square :: Num n => n -> n
square x = x^2

-- | Compute the sum of the squares of the numbers in a list.

sumSquares :: Num n => [n] -> n
sumSquares xs = sumf square xs
```

But there is another way. One of Haskell's sources of mathematical inspiration is Alonzo Church's λ-calculus (λ is the Greek letter "lambda"), which included a notion of *abstraction* that allowed the *definition* of a function *apart* from the *naming* of that function. In abstraction, we consider an expression of our language, e.g., `x^2`, and turn it into a function by *abstracting* over the variable `x`, thus:

```
-- | Compute the sum of the squares of the numbers in a list.

sumSquares :: Num n => [n] -> n
sumSquares xs = sumf (\x -> x^2) xs
```

Note that the ascii backslash \ is used to recall the memory of the form of the λ character.

Indeed, our ordinary means of defining a function combines two distinct and more primitive ideas—the definition of a function *per se*, and the *binding* of a name to that value—into a single useful idiom. We can use abstractions to tease these apart, i.e.,

```
-- | Square a number

square :: Num n => n -> n
square = \x -> x^2
```

Indeed, behind the scenes, the compiler does this for us.

Haskell has a couple of more useful tricks involving λ. One of them deals with what we ordinarily think of as multi-argument functions, e.g., consider the useful `zipWith` function, which combines two lists (not necessarily of the same type) to produce a list, via a combining function:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith f _ _ = []
```

Let's consider a simple problem. We have a `[String]`, representing keys, and an `[Int]` representing values. We want to produce a table, e.g.,

```
> formatKeysAndValues ["one","two","three"] [1,2,3]
one: 1
two: 2
three: 3
```

We'll use the `unlines :: [String] -> String` function to convert a list of strings into a multi-line string, and `zipWith` to combine the keys and values to produce a line of output, e.g.,

```
-- | format a list of key and a list of values

formatKeysAndValues :: Show s => [String] -> [s] -> String
formatKeysAndValues ks vs = unlines (zipWith formatLine ks vs) where
```

```
    formatLine k v = k ++ ": " ++ show v
```

We can use λs to rewrite the local definition:

```
formatLine = \k -> \v -> k ++ ": " ++ show v
```

and then replace the name by its definition:

```
formatKeyAndValues ks vs =
    unlines (zipWith (\k -> \v -> k ++ ": " ++ show v) ks vs)
```

Haskell allows us to collapse the λs, and use `\k v -> k ++ ": " ++ show v` instead.

A second trick is that the argument need not be a variable, but it can be a pattern enclosed in parentheses. Thus, e.g.,

```
> map (\(a,b) -> (b,a)) [(1,2),(3,4)]
[(2,1),(4,3)]
```

This facility should be used with caution, as a match failure results in an exception. But `(,)` has only a single data constructor, so this pattern should be safe.

Prefix/Infix

The syntax of the λ-calculus consists of application and abstraction. There is no infix notation in the λ-calculus. But mathematicians use infix notation all the time, and find it to be a considerable convenience. To make this work, they've found three ideas to be useful: *precedence*, which is to say, which operators are applied first, *associativity*, which determines the order of operations of the same precedence, and *parentheses*, which are used to override precedence and associativity.

For example, Haskell has the following fixity declarations in the `Prelude`:

```
infixl 6 +
infixl 6 -
infixl 7 *
infixl 7 /
infixr 8 ^
```

Consider the expression

```
1 - 2 * 3 ^ 2 ^ 3 / 7 + 1 * 4
```

The highest fixity operator is `(^)`, and it associates to the right. So the subexpression `3 ^ 2 ^ 2` is grouped first, as `(3 ^ (2 ^ 2))`, resulting in

```
1 - 2 * (3 ^ (2 ^ 3)) / 7 + 1 * 4
```

We next group the operators at precedence level 7, associating to the left per their fixity declarations:

```
1 - ((2 * (3 ^ (2 ^ 3))) / 7) + (1 * 4)
```

Finally, the operators at fixity level 6 are grouped, associating again to the left per their fixity declaration.

```
((1 - ((2 * (3 ^ (2 ^ 3))) / 7)) + (1 * 4))
```

At this point the expression is fully parenthesized. Let's check...

```
> 1 - 2 * (3 ^ (2 ^ 3)) / 7 + 1 * 4
-1869.5714285714287
> ((1 - ((2 * (3 ^ (2 ^ 3))) / 7)) + (1 * 4))
-1869.5714285714287
```

Whew!

Note that if operators are given the same precedence, but different associativity, a syntax error can occur, e.g., if we have declarations and definitions

```
infixr 4 <+>
infixl 4 <->

(<+>) :: Int -> Int -> Int
(<+>) = (+)

(<->) :: Int -> Int -> Int
(<->) = (-)
```

i.e., versions of the ordinary addition and subtraction specialized in type to `Int -> Int -> Int`, and with (non-standard) precedences and associativity, the expression `1 <+> 2 <-> 3` will result in a parsing error.

We can build ordinary infix functions out strings built out of most of the non-letter, non-digit characters that aren't reserved for some other role: `!#$%&*+./<=>?@\^|-~`. Most of the short names are already taken, and we'll encounter a fair number of these operators in future lectures.

We've already seen how an infix operator can be converted into a prefix function by parentheses, e.g., `1 + 2` and `(+) 1 2` are essentially the same. We can do the converse too, converting an ordinary prefix function into an infix function by surrounding it with back-ticks. Thus, `div 10 3` can be written as ``10 `div` 3``, and this is sometimes very useful. Ordinary prefix functions can be subject of fixity declarations, e.g., ``infixl 6 `div` ``. Note the back-tics.

Haskell uses both λ-calculus style prefix notation, and ordinary mathematical operators. Understanding how these mix is crucial. The rule is simple: application binds more tightly than any infix operation. Thus, a precedence analysis of `1 + div 2 3 + 4` proceeds as follows. First we group the application `div 2 3`, resulting in

```
1 + (div 2 3) + 4
```

The resulting expression has two additions at the top level, and because addition associates to the left, the left most top level occurrence of + is grouped next,

```
(1 + (div 2 3)) + 4
```

As a general rule, you'll want to avoid the use of unnecessary parentheses, and rely on precedence and associative to fix the order of operations in a Haskell expression. There are exceptions, but even in exceptional cases, a "parenthesis light," if not "parenthesis minimal" style is generally more readable than a "parenthesis heavy" style.

Sections

One of the nice features of Haskell is partial application, where we build a function out of a binary (or higher arity) function by providing some but not all of the arguments. For example,

```
> map ((*) 2) [1,2,3,4]
[2,4,6,8]
```

A problem with this is that it only enables us to provide the *first* argument. Thus, for example, we can compute 17 modulo various numbers,

```
> map (mod 17) [1..4]
[0,1,2,1]
```

but this simple form of partial application doesn't enable us to fix the second argument, e.g., to compute various numbers mod 5. Instead, we'd have to do something like this:

```
> map (\x -> mod x 5) [1..5]
[1,2,3,4,0]
```

Haskell has a simple syntax that makes it easy to build partial applications out of infix binary functions, a remarkably common case, called *sections*. Just parenthesize an expression, with one argument omitted, e.g.,

```
> map (2 *) [1..4]
[2,4,6,8]
> map (`mod` 5) [1..5]
[1,2,3,4,0]
```

The use of sections provides a very terse solution to the `allp` problem from a couple of lectures ago,

```
allp :: [a -> Bool] -> a -> Bool
allp ps a = and $ map ($ a) ps
```

as we build a section that applies a function to a fixed argument. Sweet!

Compositions, η-reductions, and Point-Free Programming

We will often use the basic machinery provided for us by λ-terms. But Haskell is a language that provides higher-order abstractions, and in particular allows us to define functions that define functions. Doing so comes with great advantages: we can eliminate common boilerplate, and instead focus on the parts of the code that vary, the parts that contain our creative work as programmers.

One such function is *composition*, which is implemented in Haskell as the `(.)` operator:

```
infixr 9 .

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

We can almost show that composition is associative by a sequence of forward and backward substitutions, i.e.,

```
((f . g) . h) x = (f . g) (h x)
                = f (g (h x))
                = f ((g . h) x)
                = (f . (g . h)) x
```

But this argument isn't quite as compelling as we'd like. The problem is that we haven't actually showed that composition is associative, but rather that it is associative in application. This is a small distinction, but we have to consider at least the possibility that it is a significant one. To that end, we'll revisit our discussion of extensionality from Lecture 2.

We start with a basic question. What does it mean to say that two functions are equal? In classical mathematics, functions are just sets of ordered pairs, and so the meaning of equality of functions is inherited from the definition of equality for sets. In particular, if f and g are mathematical functions, such that $\text{dom}(f) = \text{dom}(g)$, and $\forall x \in \text{dom}(f), f(x) = g(x)$, then $f = g$, because these are the same sets. This is sometimes called "equality in extensionality."

We'll adopt the result, if not the argument, to our notion of functions in Haskell: if `f x = g x`, then `f = g`, because they produce the same results when applied to an arbitrary argument, and so are extensionally equal. Note here that we're assuming that there are no occurrences `x` other than the two explicit ones. In the λ-calculus, this is captured by the idea of η-reduction, which is formally written as

$$(\lambda x, Mx) \triangleright_\eta M$$

i.e., `\x => M x` η-reduces to `M`, so long as x has no free occurrences in M. This justifies a "right cancellation rule" in equational reasoning, which we can apply to the argument above, obtaining

```
(f . g) . h = f . (g . h)
```

the full form of associativity, from the preceding argument.

η-Reduction is often a beneficial transformation, as it eliminates the use of a variable (and so there are fewer local notions to understand), and as it often results from minor refactorings of the code that result in greater clarity and flexibility. The introduction of compositions is often a set-up for η-reductions.

For example, let's revisit a problem from Lecture 2, and write a function that sums the square of the even numbers that occur in a list. We have

```
sumOfSquaresOfEvens :: Num n => [n] -> n
sumOfSquaresOfEvens ns = sum (map square (filter even ns))
```

This hints as the possibility of η-reduction, as there is only a single occurrence of `ns` on both sides of the equality, and indeed it is already in the right-most position. The problem is that that this isn't quite good enough, and as we need the equations to take the form of a function applied to an argument.

We can rewrite this was the composition three functions (indicated by hyphens below))

```
sum (map square (filter even ns)) = (sum . map square . filter even) ns
---  ----------  -----------
```

Thus

```
sumOfSquaresOfEvens ns = (sum . map square . filter even) ns
```

which we η-reduce to

```
sumOfSquaresOfEvens :: Num n => [n] -> n
sumOfSquaresOfEvens = sum . map square . filter even
```

This is terse, but easy to understand and modify with practice. Moreover, expressing this as a composition is a kind of literal factoring of the problem that decomposes the problem into simple, list-based building blocks.

The resulting style of programming is sometimes called `point-free`, as we're not defining functions point-wise.

Exercise 5.1 *Consider the* `even` *predicate:*

```
even :: Integral n => n -> Bool
even n = mod n 2 == 0
```

It is possible to transform `even` *into a composition of two sections. Do so. This sort of thing is often "encountered in the wild."*

The descent into silliness

There is a `Prelude` function `flip`, defined as follows:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f b a = f a b
```

This function is often used when we want to specialize a binary function at its second argument, e.g., rather than writing ``(`mod` 3)`` for the function that computes `\x -> mod x 3`, we could reason equationally as follows:

```
\x -> mod x 3
\x -> flip mod 3 x
flip mod 3
```

using an explicit η-reduction (and the associativity of application) to justify the last step. There's some danger in a little knowledge. Using `flip` (and a few other tricks), we can usually contrive to set up an η-reduction whenever there is only a single occurrence of inner-most (right-most) bound variable on the right-hand side of a defining equation. For example, consider the definition of `allp` that we developed earlier in the lecture:

```
allp :: [a -> Bool] -> a -> Bool
allp ps a = and $ map ($ a) ps
```

There's only a single `a` on the right hand side. Can we get at it?

```
allp ps a = and $ map ($ a) ps
          = ($) and (map ($ a) ps)
          = ($) and (flip map ps ($ a))
          = ($) and (flip map ps (flip ($) a))
          = (($) and . flip map ps . flip ($)) a
```

whence

```
allp ps = ($) and . flip map ps . flip ($)
```

And as incomprehensible as this is, we now note that there's only a single `ps` on the right hand side, and so we can indulge in more of the same trickery

```
allp ps = ($) and . flip map ps . flip ($)
        = (.) (($) and) (flip map ps . flip ($))
        = (.) (($) and) ((.) (flip map ps) (flip ($)))
        = (.) (($) and) (flip (.) (flip ($)) (flip map ps))
        = ((.) (($) and) . flip (.) (flip ($)) . flip map) ps
```

and so

```
allp :: [a -> Bool] -> a -> Bool
allp = (.) (($) and) . flip (.) (flip ($)) . flip map
```

This is entirely non-intuitive code, but it works. What is most impressive about this is that we arrived at this form by a lawful series of code transformations. There is a deep algebraic structure to Haskell code, reflective of its roots in the λ-calculus. There's a power here, although whether we use it for good or for evil is entirely up to us.

Oddly enough, (and terrifyingly enough) though, we're not done. Consider the operator `($)`, which appears explicitly as such above.

```
($) f x = f x
        = id f x
```

Where `id = \x -> x` is Haskell's identity function. We can η-reduce this last equation, twice!, resulting in `($) = id`. This enable us to rewrite the code above as

```
allp = (.) (($) and) . flip (.) (flip ($)) . flip map
     = (.) (id and) . flip (.) (flip id) . flip map
     = (.) and . flip (.) (flip id) . flip map
```

Next, we can reintroduce sections to eliminate both `(.)` and `flip (.)` as follows:

```
allp = (.) and . flip (.) (flip id) . flip map
     = (and .) . (. (flip id)) . flip map
```

At this point, we note that

```
flip id x y = id y x
            = y x
```

So that `flip id` is just application written backwards! Naturally enough, this comes up in practice, and a bit of searching reveals the `(&)` operator in `Data.Function` as reverse application. Thus:

```
import Data.Function

allp :: [a -> Bool] -> a -> Bool
allp = (and .) . (. (&)) . flip map
```

It may not be comprehensible, but at the end of a shock-and-awe series of transformations, we're left with code that has a zen succinctness.

Exercise 5.2 *Our starting point,*

```
allp ps a = and $ map ($ a) ps
```

did us no particular favors. Give a slightly simpler derivation of the final form of `allp` *based on*

```
allp ps a = and (map ($ a) ps)
```

There's no point titrating crazy

As wild as this is, we're not done yet. Let's go back to our original definition of `allp`:

```
allp :: [a -> Bool] -> a -> Bool
allp ps a = and $ map ($ a) ps
```

What made the analysis above tricky is that we began our attempt to produce a point-free definition by using an η-reduction to eliminate `a`. But this is hard, since `a` is buried in the right hand side. It would be much easier to eliminate `ps` from the right hand side. So let's expose it on the left!

```
flip allp a ps = and (map ($ a) ps)
```

This creates a problem for us, as we'll have to some how eliminate the `flip` on the left, but let's suspend disbelief and carry on...

```
flip allp a ps = (and . (map ($ a)) ps
```

which we η-reduce

```
flip allp a = and . (map ($ a))
```

At this point, we can rely on the analysis above, which showed

```
($ a) = (&) a
```

and so

```
flip allp a = and . (map ((&) a))
            = (.) and (map ((&) a))
            = ((.) and . map . (&)) a
```

which we immediately η-reduce to

```
flip allp = (.) and . map . (&)
          = (and .) . map . (&)
```

But what do we do with about the `flip`? Intuitively, we should just flip it back!, which we can justify by the reasoning

```
flip (flip f) x y = (flip f) y x
                  = flip f y x
                  = f x y
```

So

```
allp = flip $ (and .) . map . (&)
```

Oddly enough, if you think this through, it makes sense!

Chapter 6

Type Classes

Haskell, as a programming language, has a somewhat unusual history. Most programming language designs are driven by small groups, or even single individuals, at their formative stages, and driven by a fairly focused set of computational problems. Lisp was driven by John McCarthy's misunderstandings of Church's λ-calculus, and his desire for a programming language in which programs could naturally be represented as data objects in the language. FORTRAN was developed by a small team of programming language experts at IBM working under the direction of John Backus, and focussed on scientific computing. COBOL was largely the work of Admiral Grace Mary Hopper, and was driven by the data processing needs (at the time, mostly business/logistical needs) of the US Navy. SNOBOL was largely the work of Ralph Griswold, and was intended to facilitate quick-and-dirty manipulations of string data, much as Larry Wall's Perl does today. And so it goes.

There have been a few languages that were designed by committees, and even fewer of these that have been successful. Algol 60 and Haskell are exemplars. In the case of Haskell, there were a large number of programming researchers in the early 1980's all working on the problem of lazy evaluation, and they each tended to have their own language, a few of which are still in use. A very real problem was that none of these languages was gaining the kind of wide-spread adoption necessary to gain attention outside of a relatively narrow set of programming language specialists. And so the process that lead to Haskell was driven by a desire to create a shared language that would attract a critical mass of users, and moreover, which would allow a direct comparison of some of the implementation ideas then current. And so a Haskell committee was formed to create a common tongue for this work.

There is an excellent history of Haskell, A History of Haskell: Being Lazy With Class that was written by a few core committee members.

One of the goals of the project was a fairly conservative design, which relied on well-tested ideas. Type classes, which are widely viewed as one of the most distinctive and interesting aspects of Haskell's design, represented an almost casual breach of this goal! The basic design came out of a conversation between Phil Wadler and Joe Fasel, as a way of dealing with the problem of numeric types and operator overloading. Wadler suggested the addition of type classes in an email message to the committee's email list, and which adopted it without much discussion or reflection. Amazing!

Defining and Instantiating Type Classes

Type classes are defined by the `class` keyword, and consist of a sequence of declarations, associated with optional default implementations. E.g., the `Eq` type class is defined as follows:

```
class Eq a where
    (==), (/=)           :: a -> a -> Bool

    x /= y               = not (x == y)
    x == y               = not (x /= y)
```

In this definition, the variable `a` represents the type of an instance of the `Eq` class. Instances of `Eq` are required to provide both the `(==)` and `(/=)` functions. This type class definition also includes default definitions for the two operators, which have the effect that a programmer needs only provide an instance of `(==)` or of `(/=)`. If one of the definitions is omitted, the default definition will be used instead.

For example, let's consider a simple key-value pair,

```
data Entry = Entry String String -- key, value
```

as might be used in a dictionary. We can make `Entry` an instance of `Eq` by providing an *instance definition*, e.g.,

```
instance Eq Entry where
    Entry key1 value1 == Entry key2 value2 = key1 == key2 && value1 == value2
```

In this case, the omitted definition will be provided by default.

Haskell also allows for *deriving instances*, which allow type class instances to propagate through type hierarchies, e.g.,

```
data Pair a b = Pair a b

instance (Eq a, Eq b) => Eq (Pair a b) where
    Pair a1 b1 == Pair a2 b2 = a1 == a2 && b1 == b2
```

A similar idea allows Haskell to derive the `Eq` type class for a new ADT whenever the constituent types are also instances of `Eq`, and we can invoke this automatic machinery by the `deriving` clause in the definition of an ADT.

Part of what makes type classes interesting in Haskell is both the analogy and dis-analogy with the classes of (probably more familiar) object-oriented language. In both cases, we're providing a common vocabulary and somewhat similar type checking behavior, and so the programming use cases are often remarkably similar. But there is an important difference: Typical object-oriented languages rely on run-time (dynamic) selection of code that corresponds to a particular member/message, typically through pointers, stored in the object itself, to type-specific dispatch tables.

Haskell is able to use its static type analysis to select the appropriate instance at compile time, saving itself both time and space at runtime.

A Few Standard Type Classes

`Eq`

As we've just seen, `Eq` is a type class which includes types with a meaningful notion of equality.

Type classes interact with the type system via *constraints* on the types of free variables in a type expression. For example, consider the `Data.List` function `elem`, which determines whether a value occurs in a list:

```
> 3 `elem` [1,2,3,2,3,4]
True
```

It's easy to implement `elem` via a natural recursion:

```
elem _ [] = False
elem x (a:as)
    | x == a = True
    | otherwise = elem x as
```

or, more concisely

```
elem _ [] = False
elem x (a:as) = (x == a) || elem x as
```

relying on the left-biased definition of (||).

It should be clear that the definition of `elem` relies on the list elements supporting an equality test, and this reflected in the type of `elem`, via a type constraint:

```
elem :: Eq a => a -> [a] -> Bool
```

In recent versions of Haskell, `elem` has an even more general type, which we'll get to later in the lecture.

`Ord`

Haskell's `Prelude` defines an `Ord` type class:

```
class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a

    ...
```

The documentation (cf. Data.Ord on Hackage) identifies `compare` as *a minimal complete definition*, meaning that default implementations are set up in a way such that all other type class functions have default definitions that are ultimately grounded in `compare`. Minimal complete definitions need not be unique, although setting up multiple minimal complete definitions is tricky. Note that <= is also minimally complete for `Ord`.

This definition comes with a constraint: All instances of type class `Ord` must also be instances of type class `Eq`.

Exercise 6.1 *Read the documentation on* `Ord`*, and then provide a parsimonious instance of* `Ord` *for suitably type-constrained instances of* `Pair`.

Number Types

Numeric types are described by the `Num` class, as we've seen before.

```
class Num a where
    (+), (-), (*) :: a -> a -> a

    negate :: a -> a
    abs :: a -> a
    signum :: a -> a

    ...
```

`Num` is the basis of a hierarchy of various numeric types. For example, `Int` and `Integer` are instances of the `Integral` subclass of `Num`, and `Float` and `Double` are instances of the `Fractional` subclass of `Num`.

We will not delve much into this numerical tower, but you can learn more by reading the documentation, this tutorial, or by using the `:info` command in `ghci` to poke around:

```
> :info Num
...

> :info Fractional
...
```

```
> :info Int
...
```

Enum

Haskell provides concise syntax for enumerating values of enumerable types:

```
> [1..10]
[1,2,3,4,5,6,7,8,9,10]

> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"

> [2,4..10]
[2,4,6,8,10]

> ['a','e'..'u']
"aeimqu"
```

The `Enum` type class describes types that can be enumerated (i.e. indexed from `0` to `n-1`):

```
class Enum a where
    toEnum   :: Int -> a
    fromEnum :: a -> Int
    ...
```

Show

The `Show` class describes types whose values can be converted to `String`s:

```
class Show a where
    show :: a -> String
    ...
```

As we have seen, Haskell can auto-generate `Show` instances:

```
data Pair x y = Pair x y
    deriving (Eq, Show)
```

```
> Pair "Hello" 161
Pair "Hello" 161
```

Exercise 6.2 *Instead of deriving the default* `Show` *instance for* `Pair`*, define one that produces the following:*

```
> Pair "Hello" 161
( "Hello" , 161 )
```

`Read`

The `Read` class is complementary to `Show`:

```
class Read a where
    ...

read :: Read a => String -> a
```

The `read` function *parses* a `String` to produce a value of the given type. Together, `read`ing and `show`ing values allow programs to interact with the outside world (e.g. `String`s from user input, the file system, and network requests). We will have much more to say about parsing and the outside world later.

`Foldable`

The `Foldable` type class is interesting, and it has an interesting history. Let's consider `[]`. We know that `[]` has two data constructors, `[]` and `(:)`. We can define a list function by a recursion on this type, which typically involves a definition like the one we gave for `enum` earlier in the lecture:

```
-- | A predicate for list membership

elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (a:as)
    | x == a = True
    | otherwise = elem x as
```

Historically, one of the significant list-based functions was `foldr`, which we can think of as a function for defining functions by recursion over lists:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr combine base [] = base
foldr combine base (c:cs) = combine c (foldr combine base cs)
```

We can then reimplement many of our list based functions in terms of calls to `foldr`, thinking of it as a function that takes two arguments and produces a function as a result, e.g.,

```
sum :: Num n => [n] -> n
sum = foldr (+) 0

product :: Num n => [n] -> n
product = foldr (*) 1
```

Or even

```
elem :: Eq a => a -> [a] -> Bool
elem a = foldr (\c cs -> a == c || cs) False
```

Having written this, the code fairy (we'll get to know the code fairy and her ways through the rest of this course), taps imperiously on our shoulder. Since I know her, I know that she's telling me that we can do better. Indeed, we can:

```
\c cs -> a == c || cs = \c cs -> (||) (a == c) cs
                      = \c -> (||) (a == c)
                      = \c -> (||) ((a ==) c)
                      = \c -> ((||) . (a ==)) c
                      = (||) . (a ==)
```

so

```
elem a = foldr ((||) . (a ==)) False
```

We'll pass on eliminating the a.

The revelation comes from the observation that there are other data types we might want to fold into a single summary value. A good example is kind of binary tree that holds its values at its inner nodes:

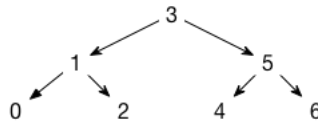
```
data BinaryTree a
    = EmptyTree
    | Node a (BinaryTree a) (BinaryTree a) -- value leftChild rightChild
```

```
        deriving (Show)
```

We can think of a binary tree as consisting of a sequence of values, ordered in infix, i.e., where all of the values in the left child of a node come before the value it carries, and these are followed by the values held by the right children. We can made this explicit via a function that produces a list with this order, i.e.,

```
visit :: BinaryTree a -> [a]
visit EmptyTree = []
visit (Node a left right) = visit left ++ [a] ++ visit right
```

Thus, for example if we convert the following



into a `BinaryTree`

```
tree :: BinaryTree Integer
tree = Node 3
        (Node 1
            (Node 0 EmptyTree EmptyTree)
            (Node 2 EmptyTree EmptyTree))
        (Node 5
            (Node 4 EmptyTree EmptyTree)
            (Node 6 EmptyTree EmptyTree))
```

We'd have

```
> visit tree
[0,1,2,3,4,5,6]
```

as expected.

Interestingly enough, we can implement `foldr` for `BinaryTree` as well, but managing the traversal is more subtle. The definition for traversing the EmptyTree is straightforward:

```
foldr :: (a -> b -> b) -> b -> BinaryTree a -> b
foldr combiner base EmptyTree = base
foldr combiner base (Node a left right) = ?
```

The definition for a `Node` is trickier. It's natural to begin by calling `foldr combiner base` recursively on left. The conceptual problem is that when we're done with `left`, we don't want to end with `base`, but instead the result of processing the rest of the tree, i.e., a and `right`. But we can do this! Thus,

```
foldr combiner base (Node a left right) =
    foldr combiner (combiner a (foldr combiner base right)) left
```

And this completes the definition. With this, we can define

```
visit :: BinaryTree a -> [a]
visit = foldr (:) []
```

Indeed, even this isn't necessary. If we read the definition of `Foldable` more closely, we'll discover the `toList` function, which is simply a polymorphic version of `visit` which we don't have to write.

Exercise 6.3 *Write a* `reverseTree` *function for* `BinaryTree a` *which creates a mirror-image of the original tree, and verify that*

```
> toList (reverseTree tree) == reverse (toList tree)
True
```

Note that the `toList` *function is not in the* `Prelude`, *but it is in* `Data.Foldable`, *and so you'll need to either*

```
import Data.Foldable
```

in your source, or

```
> :m +Data.Foldable
```

in `ghci`.

Chapter 7

A Brief Introduction to Haskell I/O

So far, we've lived in the interpreter, and built little snippets of code. This is useful, but it limits the mode of interaction. Programs intended for end-users (often ourselves) often want more control over how they interact with the user. Moreover, programs intended for end users present themselves as complete—we don't want Grandma to have to install `ghc` and master Haskell to enjoy the fruits of our labors this quarter.

This will, of present necessity, be a very incomplete introduction. A thorough understanding of Haskell I/O will come later. Mimicry and practice though can form a foundation for later understanding, so I'm asking you to suspend disbelief for a bit. It's time to get on the bike, to start pedaling, and to believe that when Dad lets go, you'll keep going.

Output

So let's start with the old "Hello, world!" chestnut:

```
module Main where

main :: IO ()
main = do
    putStrLn "Hello, world!"
```

We'll ignore the actual content of the file for just a bit. Let's suppose we put this in a file called `hello.hs`. We can produce an executable (binary) file by compiling this using ghc (*not* ghci):

```
$ ghc hello.hs
[1 of 1] Compiling Main             ( Hello.hs, Hello.o )
Linking hello ...
$ ./hello
```

```
Hello, world!
$
```

If we're clever enough to have a `~/bin` directory, and to have it on our `PATH`, we can simplify this further:

```
$ cp hello ~/bin/hello
$ hello
Hello, world!
$
```

This is good enough for simple programs, but more complicated situations (including where there's non-trivial configuration and/or testing involved) are better handled through `cabal` (or `stack`), as you'll be learning in the lab.

There's a fair bit to explain here, and actually a fair bit that isn't necessary for this program, but will be essential soon enough.

We'll start at the top. Haskell programs are typically divided into modules. A module is a related collection of declarations and definitions. Modules have simple alphanumeric names, and may be structured hierarchically, using the period (.) symbol as a separator. The declaration

```
module Main where
```

indicates that the code in this file will be in the `Main` module. Evaluation of compiled code is driven by *performing* the *IO action* `main` from the `Main` module.

Next, we have the type declaration `main :: IO ()`. This looks odd, a bit of advanced technology indistinguishable from magic. It will seem less magical in time. Types of the form `IO a` are *IO actions*, which when *performed* return a value of type `a`. The type `()` is Haskell's *unit* type, which contains a single defined value, also denoted by `()`, as we've seen before. We use `()` as a simple, concrete, placeholder in situations where Haskell requires a type, but we don't aren't going to do anything subsequently that discriminates between values of that type.

The definition of `main` consists of a `do` construct, which is used to combine a sequence of IO actions into a single IO action. In this case, there is only one action, so we could get by without the `do`-wrapping, i.e.

```
main = putStrLn "Hello, world!"
```

but that doesn't generalize to the more complicated examples we're going to see soon, and it's often useful to use a `do` to sequence a single IO action, when we expect that we may be adding other IO actions to the sequence later.

Finally, `putStrLn :: String -> IO ()` is a function that takes a `String` as an argument, and produces as a *value* an IO action, which when *performed* prints its argument to the standard output. Note here that we've been carefully using separate words: an expression may have a *value* in `IO a`, but evaluation doesn't cause an IO action to be performed. Only *performing* it does. The following code may make the distinction a bit clearer:

```
module Main where

naNaNaNa :: IO ()
naNaNaNa = putStrLn "Na, na, na, na"

heyHeyGoodbye :: IO ()
heyHeyGoodbye = putStrLn "Hey, hey, Goodbye!"

main :: IO ()
main = do
    naNaNaNa
    naNaNaNa
    heyHeyGoodbye
```

We define several IO actions here, notably `naNaNaNa` and `heyHeyGoodbye`. *Defining* these values doesn't cause the IO actions they describe to be performed, but when `main` itself is performed, they in turn are performed. This seems clear enough when these IO actions are defined globally, but the same holds true if they are defined locally, e.g.,

```
module Main where

main :: IO ()
main = do
    let naNaNaNa = putStrLn "Na, na, na, na"
        heyHeyGoodbye = putStrLn "Hey, hey, Goodbye!"
    naNaNaNa
    naNaNaNa
    heyHeyGoodbye
```

The output is as before:

```
$ ./goodbye
Na, na, na, na
Na, na, na, na
Hey, hey, Goodbye!
$
```

Defining is not performing. Performing is performing. Note here that top-level `let` bindings within a `do` have a scope that consists of the binding (allowing mutually recursive definitions) and the rest of the `do` body, so the keyword `in` is not used, and a level of indentation is saved. Note (as here) that a single `let` may be used to define multiple names. This may ring a bell—we don't use the `in` keyword when binding values in the interpreter. This isn't different: for all practical purposes, the interpreter's read loop is the body of a `do` block in the `IO` context.

Input

We've learned how to use `putStrLn` to produce output, and you'll not be surprised to learn that there are many more output-oriented functions in Haskell, or that we'll encounter some of them later, but `putStrLn` is enough to get us started on output. But what about input?

The complement to `putStrLn` is `getLine :: IO String`, a function that reads a line of text from standard input (for now, the terminal), up to the next newline or the EOF (end-of-file), and returns a `String` value (conveniently omitting the newline).

```
module Main where

main :: IO ()
main = do
    putStrLn "Hello. I am a HAL 9000 series computer."
    putStrLn "Who are you?"
    name <- getLine
    putStrLn ("Good morning, " ++ name ++ ".")
```

Note the *binding* syntax here, in which an IO action is performed, and the value it returns is bound to a variable (in this case, `name`). People learning Haskell often struggle at first with the distinction between `let` and `<-`, in that both bind names, and so seem to do similar things. The difference is that with a `let`, the defining expressing is *evaluated*, and the name is bound to the resulting value; whereas, with a binding, the expression is *performed*, and the name is bound to the result returned by the action.

One bit of trickiness is in the final line, where we concatenate several strings together using the `(++)` operator, and apply `putStrLn` to the result. It's good Haskell style to omit unnecessary parenthesis, and this often tempts beginners into dropping them from the last line,

```
putStrLn 'Good morning, ' ++ name ++ '.'
```

This unfortunately doesn't work, because function application binds more tightly than application, so the syntax says to apply the function `putStrLn` to the string `"Good morning, "`, and then to use `(++)` to combine the result (of type `IO ()`) with `name`, which has type `String`. The resulting error message says this, but it's a bit intimidating at first. Experienced Haskell programmers will often use the `($)` form of application here::

```
putStrLn $ "Good morning, " ++ name ++ "."
```

which also supports a nice syntax when things get long, as they sometimes do:

```
putStrLn $ "Good morning, "
         ++ name
```

```
            ++ "."
```

This program runs pretty much as any Space Odyssey aficionado would anticipate:

```
$ ./hal
Hello. I am a HAL 9000 series computer.
Who are you?
\emph{Dave}
Good morning, Dave.
$
```

Of course, input is available from places other than standard input, e.g., the command line, files, network sockets, and the environment. The latter is a simple key-value list associated with each process, and is typically used for communication between processes. One of the environment variables is `USER`, which is initialized by the log-in process to contain the account name, often the user's personal name. We can easily rewrite this program so that it uses the `USER` environment variable, rather than interrogating the user:

```
module Main where

import System.Environment

main :: IO ()
main = do
    putStrLn "Hello. I am a HAL 9000 series computer."
    name <- getEnv "USER"
    putStrLn $ "Good morning, " ++ name ++ "."
```

Here, we've used the function `getEnv :: String -> IO String`, which takes a key as argument, and when performed returns the corresponding value. An addition bit of complexity comes from the fact that `getEnv` isn't exported by the standard Haskell `Prelude`, but instead is exported by the `System.Environment` module, which we import here.

```
$ ./hal
Hello. I am a HAL 9000 series computer.
Good morning, stuart.
$
```

OK, this is hitting a little too close to home.

Exercise 7.1 *Modify the second (*`getEnv`*-based)* `hal` *program so that it capitalizes the user's name in greeting them. Compile and run your program, and provide a sample interaction. You may find the function* `Data.Char.toUpper` *to be helpful.*

Before moving on, note that Windows, as is so often the case, gratuitously varies from the standards it co-opts, and uses `USERNAME` instead. Caveat emptor. We could easily re-write this example so that it worked with Windows, but we might like to write a *portable* version that can deal with either convention. This turns out to be more difficult than we might expect, and we'll need to develop some new tools first.

The `Read` and `Show` type classes.

So far, we've dealt with simple problems of IO, in particular, sending strings to standard output, and retrieving them from standard input. But Haskell is a type-strict language, and this raises the question of how do we get other kinds of information in and out of our program. To that end, Haskell's `Show` and `Read` type classes are very convenient. Most of the types exported by the `Prelude` have both `Read` and `Show` instances, and Haskell supports derived instances of both, using the same syntax as Haskell programs.

Consider a simple program that generates binomial coefficients:

```
module Main where

import System.Environment

binomial :: Int -> [Integer]
binomial n
    | n > 0  = let bs = binomial (n-1) in zipWith (+) ([0]++bs) (bs++[0])
    | n == 0 = [1]
    | otherwise = error "domain error: negative argument to binomial"

main :: IO ()
main = do
    [nstr] <- getArgs
    putStrLn . unwords . map show . binomial . read $ nstr
```

The particular algorithm whereby we compute lists of binomial coefficients isn't important here—but if you're familiar with Pascal's triangle, you should be able to see it here. Our current concern is the content of `main`, which includes IO actions, `read`, and `show`.

We start with

```
[nstr] <- getArgs
```

Which reads the command line arguments (*not* including the program name), and assuming that there is just one, binds it to the variable `nstr :: String`. The composition

```
putStrLn . unwords . map show . binomial . read
```

uses `read` to convert a `String` to an `Int`, then uses the binomial function to produce an `[Integer]`. We map `show` across this, resulting in an `[String]`. These strings are concatenated together, with separating spaces, using the `Prelude` function `unwords :: [String] -> String`, and finally, the resulting string is the argument to `putStrLn`, creating an `IO ()`, which when performed, produced the desired output:

```
$ ./binomial 6
1 6 15 20 15 6 1
$
```

Let's focus first on the wonderful and seemly mysterious `read :: Read a => String -> a`. Students familiar with other programming languages and their conventions might be a bit perplexed here. How does the compiler know what type `read` is supposed to return, if it always takes the same type of argument? But this involves projecting an understanding of overloading and type resolution (as is done in more traditional programming languages like `C++` and `Java`) onto Haskell, and that's a mistake. Haskell's type inference can consider the *return* type, as well as the *argument* types, and since we're composing `read :: Read s => String -> s` with `binomial :: Int -> Integer`, the fully grounded type of `read` in this context must be `String -> Int`, and this allows Haskell to select the correct instance of `Read` in binding `read`. Trust me, you'll come to rely on this, and you'll miss it in other languages.

Note here that our error-handling strategy is naïve, and there are lots of things that can go wrong at runtime: there might be too many command line arguments, or too few. The argument may not have the format of an `Integer`, or it might have the format of a negative integer. We'll learn how to avoid and/or handle such errors later, but for now, they'll raise an uncaught exception, and terminate the program with an error. Still, it's better to crash and burn visibly than to silently produce nonsense.

Best Coding Practice and Complex Contexts

The binomial program introduces an important idea. IO code is different from ordinary, "pure" code: it is more difficult to reason about, and more difficult to test. Therefore, we want to structure our code so as to move complexity out of IO code. Introducing `binomial` as a separate function, outside of the `IO` context, renders the hard core of the program in pure code, simplifying reasoning, testing, and debugging.

Part of the distinctive character of programming in Haskell comes from this split, and it really is a good-news/bad-news deal for the programmer. The good news is that you can really put your algebraic thinking hat on, and perform some nifty transformations that increase readability, concision, and efficiency, with confidence, when dealing with pure code. The bad news is that there are times when a real-world programmer wants to sprinkle some IO in the middle of what is otherwise pure code, e.g., to facilitate "wolf-fence" debugging. The division between pure and impure code in Haskell makes it impossible to do this without dragging the code in question into the impure world, conceding the very advantages we worked so hard to obtain.

As is so often the case in Haskell, IO contexts are a special case of a much more general phenomenon. We can imagine the existence of other kinds of other contexts `m`, and the general problem of writing functions of the form `g :: m a -> m b`. In such circumstances, it is often possible to write a function `f :: a -> b`, and then to somehow *lift* `f` into a `m`'s context. Lest this seem hopeless abstract, review our `binomial` program again, where the function `binomial` is pure, and is subsequently employed at the core of `main`, which lives in the `IO` context. This

exemplifies a simple, yet profound, engineering reality: the easiest and most elegant way to build reliable systems is by composing reliable components.

Haskell encourages this kind of factoring, and Haskell programmers look for opportunities to refactor their code as much as possible along a pure/lift axis. Look for it. Expect it. Do it.

Exercise 7.2 *Read the Rot-13 chapter (Chapter 8), and do the exercises there.*

Chapter 8

Case Study: Rot-13

Our next program is a simple filter, not in the Haskell sense, but in the Unix sense of a program that reads from standard input, and writes to standard output.

In the days before Facebook and blogs, there was a distributed netnews system that consisted of a large collection of newsgroups. The most popular of these newsgroups, was "rec.humor.funny." This was a moderated news group, which sent out a few jokes every day, most of which were indeed funny. But humor is a tricky thing, and some humor can cause offense. So the moderator adopted a simple strategy. Jokes that were potentially offensive were rot-13'd, i.e., the characters were remapped so that 'a' → 'n', 'b' → 'o', etc. People who wanted to read possibly offensive jokes would then rot-13 it again (since there are 26 letters, rotating twice by 13 takes you back to where you started). What made this work was that people were clearly warned that rot-13'ing was at your own risk. The social contract was that if you rot-13'ed, you surrendered the right to complain.

Let's start by thinking about this as a problem in pure code. How do we rot-13 a character? For the sake of simplicity, we'll assume that the underlying text is drawn from the ASCII subset of Unicode.

```
import Data.Char

rotChar :: Char -> Char
rotChar c
    | isLower c = chr (ord 'a' + (ord c - ord 'a' + 13) `mod` 26)
    | isUpper c = chr (ord 'A' + (ord c - ord 'A' + 13) `mod` 26)
    | otherwise = c
```

This code uses the functions `chr :: Int -> Char` and `ord :: Char -> Int` from `Data.Char`, which translate between Unicode characters and their associated `Int` code points, and relies on the fact that the each case of letters is associated with an ascending, sequential set of code points. This is true of ASCII, but is not true of all character encodings. One minor mystery in the code has to do with the relative precedences of `+` and `` `mod` ``. Intuitively, `` `mod` `` is related to division, and so might be assumed to have the same precedence as `*`, `/`, etc. This is indeed so, and can be checked using the `:i` (info) command in `ghci`:

```
Prelude> :i mod
class (Real a, Enum a) => Integral a where
  ...
  mod :: a -> a -> a
  ...
    -- Defined in GHC.Real
infixl 7 mod
Prelude> :i (*)
class Num a where
  ...
  (*) :: a -> a -> a
  ...
    -- Defined in `GHC.Num'
infixl 7 *
Prelude>
```

Here we see that `mod`, when used as an infix operator, associates to the left and at precedence level 7, which is exactly the same as `*` and its friends.

There are programmers (and sometimes I'm one) who would complain about the duplication of code in `rotChar`. We can easily eliminate the duplication by introducing a helper function, which is built out of the duplicated code, and abstracts out the part that varies:

```
rotChar :: Char -> Char
rotChar c
    | isLower c = rotCase 'a' c
    | isUpper c = rotCase 'A' c
    | otherwise = c

rotCase :: Char -> Char -> Char
rotCase base char = chr (ord base + (ord char - ord base + 13) `mod` 26)
```

But our problem wasn't to rotate individual characters, it was to rotate the entire input String. We need to *lift* the function

```
rotChar :: Char -> Char
```

to the function

```
rot :: String -> String
```

This is ridiculously easy, remembering that `String` is a synonym for `[Char]`.

```
rot :: String -> String
rot = map rotChar
```

Indeed, we'll soon enough come to understand `map` in exactly these terms—as a particular instance of a generalized "lifting" function.

At this point, it makes sense to revisit our use of local definitions. The function that we care about is `rot`—the functions `rotChar` and `rotCase` are simply there to help us define `rot`. It makes sense to tidy our namespace up a bit, and encapsulate the definitions of these helper functions within the definition of `rot`:

```
rot :: String -> String
rot = map rotChar where
    rotChar c
        | isLower c = rotCase 'a' c
        | isUpper c = rotCase 'A' c
        | otherwise = c
    rotCase base char = chr (ord base + (ord char - ord base + 13) `mod` 26)
```

Our full program is now

```
module Main where

import Data.Char

rot :: String -> String
rot = map rotChar where
    rotChar c
        | isLower c = rotCase 'a' c
        | isUpper c = rotCase 'A' c
        | otherwise = c
    rotCase base c = chr (ord base + (ord c - ord base + 13)  `mod` 26)

main :: IO ()
main = do
    input <- getContents
    putStr $ rot input
```

The function `getContents :: IO String` is an IO action that packages the program's standard input stream as a Haskell `String`. The function `putStr` writes a `String` to standard output, but without a terminating newline. In this case, the desired terminating newline would have been present in the input stream, and would have survived our mapping, so there's no need to add another.

We can now test this, and what better input source than our source?!

```
$ ./rot < rot.hs
zbqhyr Znva jurer

vzcbeg Qngn.Pune

ebg :: Fgevat -> Fgevat
ebg = znc ebgPune jurer
    ebgPune p
        | vfYbjre p = ebgPnfr 'n' p
        | vfHccre p = ebgPnfr 'N' p
        | bgurejvfr = p
    ebgPnfr onfr p = pue (beq onfr + (beq p - beq onfr + 13)  `zbq` 26)

znva :: VB ()
znva = qb
    vachg <- trgPbagragf
    chgFge $ ebg vachg
```

Hopefully, this makes less sense to you than the cleartext version. Anyway, if we rot twice, we get back to where we started.

Let's play with this just a bit more. Suppose we want to implement other ciphers. For example, the Caesar cipher is rot 3, and would be decoded by rot 23 (or rot -3!).

To do this in a uniform way, we'll design our program so that if it is called without any command-line arguments, it does a standard rot-13, but if a single command-line argument is provided, it will be interpreted as an integer that gives the desired rotation. This exemplifies the design pattern of making the common case (rot-13) simple.

To do this, we'll use a `case` to pattern match on the result of performing `getArgs`:

```
module Main where

import ...

rot :: Int -> String -> String
rot n = ...

main :: IO ()
main = do
    args <- getArgs
    case args of
        [] -> ...
        [x] -> ...
        _ -> ...
```

If there are no arguments, we want to write read standard input as a string, rotate it by the default of 13, and write the rotated string to standard output:

```
            [] -> do
                input <- getContents
                let output = rot 13 input
                putStr output
```

Let's step through this. Since we're doing IO, we know that the expressions in the do construct must be IO actions, so the alternatives of the case must also be IO actions. We'll usually need to combine several distinct IO actions (reading from standard input, writing to standard output) into a single IO action, hence the inner do's. We read standard input by the IO action `getContents :: IO String`, which when performed, returns the contents of standard input as a `String`. Next, we see the use of a `let` to bind the result of a pure computation. Finally, we write the resulting output string to standard out using `putStr :: String -> IO ()`.

On to the second case... . A first cut at this might look like this:

```
            [x] -> do
                input <- getContents
                let output = rot (read x) input
                putStr output
```

There's one thing to be grumpy about here, and one really big thing to worry about.

We're grumpy, of course, about the code duplication between these two cases. Let's identify a common abstraction and eliminate the code duplication:

```
rotStdin :: Int ->  IO ()
rotStdin n = do
    input <- getContents
    let output = rot n input
    putStr output

main :: IO ()
main = do
    args <- getArgs
    case args of
        []  -> rotStdin 13
        [x] -> rotStdin (read x)
        _   -> ...
```

The thing to worry about is the wonderful world of user error. The code as written makes a call to `read` on unchecked user input. What if the user (maybe ourselves, in a few months), supplies an invalid argument? Let's test...

```
$ ./rot foo < rot.hs
rot: Prelude.read: no parse
$
```

Grandma is *not* going to be pleased. What's happened here is that `read` wasn't able to make sense of `"foo"` as the representation of an `Int`, so it gave up, and *threw an exception*. That exception wasn't caught by our code, but instead by the runtime system, which simply printed a user error, and terminated the program. In a sense, we're going to have to do much the same, but maybe we can print a more informative error. The question is *how*. We have two plausible approaches: (1) catch the exception that `read` throws ourselves, or (2) *preflight* the argument, making sure that it's in a form that `read` can correctly handle. It turns out that catching the exception is a bit complicated (we don't actually cover exceptions in this course...), so we'll preflight. An initial attempt might be:

```
import System.Exit
...

        [x]
            | all isDigit x -> rotStdin (read x)
            | otherwise     -> do
                  progname <- getProgName
                  hPutStrLn stderr $ "usage: " ++ progname ++ " [n]"
                  exitWith $ ExitFailure 255
```

Let's step through this, before we rip it apart. The `all :: (a -> Bool) -> [a] -> Bool` function is defined in the `Prelude`, and it simply makes sure that every element of its argument list satisfies the argument predicate. So we're checking to make sure we have a sequence of digits. If the preflight passes, we'll do the `read`, otherwise, we'll print an error message and terminate program execution in the Unix standard way: printing a usage message that includes our program name, and terminating with an error code.

Looking ahead a bit, we'll see that the last case is going to look a lot like the first, so let's abstract out the `usage` action, and finish up `main`:

```
usage :: IO ()
usage = do
    progname <- getProgName
    hPutStrLn stderr $ "usage: " ++ progname ++ " [n]"
    exitWith $ ExitFailure 255

main :: IO ()
main = do
    args <- getArgs
    case args of
        []  -> rotStdin 13
        [x]
```

```
                | all isDigit x -> rotStdin (read x)
                | otherwise     -> usage
            _   -> usage
```

At this point, we're pretty close to being done, but there are a couple of issues, both related to the preflighting of `read`.

The first is that our preflighting isn't *quite* strong enough, although the problematic case is unlikely to arise by accident:

```
$ ./rot "" < rot.hs
rot: Prelude.read: no parse
$
```

Maybe you didn't see that one coming: an *empty* command line argument, as distinct from an omitted argument. We can deal with that by tightening up the test:

```
                | x /= "" && all isDigit x -> rotStdin (read x)
```

The next is to notice that it would be really nice to be able to accept a leading minus sign, because then we could decode `rot 4` with `rot -4`. We can do this naïvely, e.g.,

```
validateInt :: String -> Bool
validateInt "" = False
validateInt "-" = False
validateInt (c:cs) = (isDigit c || c == '-') && all isDigit cs

...

                | validateInt x -> rotStdin (read x)
```

or, we make use of one of Haskell's regular expression libraries:

```
import Text.Regex.Posix

...

main :: IO ()
main = do
    args <- getArgs
    case args of
        []  -> rotStdin 13
```

```
        [x]
            | x =~ "^-?[0-9]+$" -> rotStdin (read x)
            | otherwise         -> usage
        _   -> usage
```

Note that the =~ operator has a lot of potential return types, and here we're using it in a context where its return type is Bool, in which case, s =~ pat will be True when the String s matches the regular expression pat, i.e., contains a matching *substring*. In this case, the regular expression matches strings that consist of the beginning of string anchor (^), a optional minus sign (-?), followed by one or more digits ([0-9]+), followed by the end of string anchor ($). The presence of the anchors forces the match of the entire string, and not merely a substring thereof.

Finally, it might occur to us that writing IO actions that filter stdin via some function to stdout is a pretty common case, and perhaps there is an easier way. And of course, there is, interact :: (String -> String) -> IO ().

```
rotStdin :: Int -> IO ()
rotStdin n = interact (rot n)
```

whence

```
rotStdin = interact . rot
```

Exercise 8.1 *Complete the implementation of* rot*, compile, and run* ./rot 3 < rot.hs*. While you're at it, deal in a principled way with input texts that don't consist solely of ASCII characters by exiting with an error. Turn in your program, both in cleartext and in cyphertext.*

The Vigenère cipher

The simple rot ciphers are good enough to hide messages from people who don't want to read them—which was kind of the point to rot-13—but they're trivial to crack as there are only 25 viable keys. A simple modification to the Caesar cipher is the Vigenère cipher, which uses as password to describe a sequence of rotations, and then enciphers a message by rotating through the sequence.

For example, if we encrypt the sentence "Haskell is fun!" using the password "stuart" we'd get "Ztmkved cs ymg!" Ymg indeed!

Exercise 8.2 *Extend the functionality of your* rot *program by implementing the Vigenère cipher. You should use the Vigenère cipher if your program is passed an argument that consists of a password, i.e., a non-empty sequence of lower-case letters. You should enable decryption through the use of an optional minus prefix on the password. Aim for elegance, which includes avoiding code duplication.*

Chapter 9

A Few More Things to Comprehend

(Chapter with contributions from RC)

To wrap up "Part 1" of the course, we will take a look at three more features in Haskell that often come in handy: records, newtype, and list comprehensions.

9.1 Records

Data constructors (and tuples) can have many component values. Consider the following:

```
data Person
  = Student String String String String Int [(String, (Int, Int))]
  | Teacher String String String [(Int, Int)]
```

The intention of `Student` is to carry a first name, last name, identification string, major (i.e. home department), College year, and list of enrolled courses. A course comprises a department, course number, and section. The intention of `Teacher` is to carry a first name, last name, home department, and list of courses (within the department) currently being taught.

A couple aspects of this datatype definition are not ideal. First, it is easy to confuse which components of the data values are meant to represent what. Second, we may have to write tedious pattern matching functions to extract components from `Person` values, such as the following:

```
lastName :: Person -> String
lastName (Student _ last _ _ _ _) = last
lastName (Teacher _ last _ _)     = last
```

What are we to do? For the former concern, one option is to write comments to declare our intentions (we should

be doing this anyway!). Another is to introduce type aliases to emphasize our intent, as follows:

```
type FirstName = String
type LastName = String
type Id = String
type Department = String
type Year = Int
type CourseNum = (Int, Int)

data Person
  = Student FirstName LastName Id Department Year [(Department, CourseNum)]
  | Teacher FirstName LastName Department [CourseNum]
```

But this doesn't actually prevent us from mixing up, for example, `FirstName`s and `LastName`s because they are just synonyms for `String`. Alternatively, we could introduce wrapper types so that the type system would prevent us from mixing up different types:

```
data FirstName = FirstName String
data LastName = LastName String
data Id = Id String
data Department = Department String
data Year = Year Int
data CourseNum = CourseNum (Int, Int)
```

Although this mitigates the first concern, it exacerbates the second because now even more patterns must be written to get at the data. This may not scalable in large programs with many datatypes and data representation needs to balance.

Fortunately, Haskell provides a feature that can be used to simultaneously address the two concerns above, namely, *records*. Consider the following type definition, where the components, or *fields*, of each `Course` value are named `department`, `num`, and `section`, respectively:

```
data Course = Course { department :: String, num :: Int, section :: Int }
  deriving (Eq, Show)
```

`Course` values can be constructed with record syntax, where field names alleviate the need to remember the intended purpose of each positional component. Notice how the order of fields does not matter.

```
> let c1 = Course { department = "CMSC", num = 161, section = 1 }
> let c2 = Course { num = 161, department = "CMSC", section = 1 }
> c1 == c2
True
```

`Course` can also be used as an ordinary (non-record) data constructor.

```
> :t Course
Course :: String -> Int -> Int -> Course

> let c3 = Course "CMSC" 161 1
> c1 == c2 && c2 == c3
True
```

Based on the record declaration, Haskell automatically generates functions to *project*, or unwrap, the fields of `Course` values:

```
> :t department
department :: Course -> String

> :t num
num :: Course -> Int

> :t section
section :: Course -> Int

> department c1
"CMSC"

> (num c1, section c1)
(161, 1)
```

In addition, one can use ordinary data constructor patterns, where components are matched by position:

```
numAndSection (Course _ num section) = (num, section)
```

Alternatively, record patterns allow field names to be used:

```
numAndSection (Course { department = _, num = i, section = j }) = (i, j)
```

As with record construction, the order of fields in record patterns does not matter. Furthermore, fields can be omitted if their values are not needed within the scope of the pattern:

```
numAndSection (Course { section = j, num = i }) = (i, j)
```

Records can be defined (or not) for multiple data constructors of a type. For example:

```
data ABCD
  = A { foo :: String, bar :: Int }
  | B { foo :: String, baz :: () }
  | C Int
  | D
```

This definition exhibits several noteworthy features. First, a field name can be used for components of different data constructors within a type. Second, not all data constructors need to be defined with record syntax.

Exercise 9.1 *Based on the definition of* `ABCD`*, what are the types and behaviors of the functions* `foo`*,* `bar`*, and* `baz`*? Think about it, and then test them out.*

Exercise 9.2 *Should the following definition be acceptable? Think about it, and then try it out.*

```
data Data = One { data :: Int } | Two { data :: Bool }
```

Now, using records, we can define `Person` as follows:

```
data Person
  = Student
      { firstName :: String
      , lastName :: String
      , id :: String
      , major :: String
      , year :: Int
      , courses_enrolled :: [(String, (Int, Int))]
      }
  | Teacher
      { firstName :: String
      , lastName :: String
      , dept :: String
      , courses_teaching :: [(Int, Int)]
      }
```

9.2 `newtype`

Data constructors tag, or label, the values they carry in order to distinguish them from values created with different data constructors for the same type. As we have seen, it is sometimes useful to define a new datatype even with only one data constructor. In such cases, tagging and untagging (or constructing and destructing, or boxing and unboxing)

values is useful for enforcing invariants while programming, but these operations add unnecessary run-time overhead: there is only one kind of value, so they ought to be free of labels.

Haskell allows datatypes with exactly one unary constructor to be defined with the keyword `newtype` in place of `data`, such as

```
newtype Identity a = Identity a
```

or, if we were using a record,

```
newtype Identity a = Identity { runIdentity :: a }
```

The choice of `Identity` may see a bit odd, but it will make more sense later.

For the purposes of programming, using `newtype` is almost exactly the same as using `data`. But it tells the compiler to optimize the generated code by not including explicit `Identity` labels at run-time. We will get into the habit of using `newtype` whenever we define a datatype with one unary constructor, without delving into the subtle differences between using `newtype` and `data`.

As we meet more of the type classes that are central to Haskell's design, we will often create wrapper types (i.e. with one data constructor). Hence, we will use `newtype`. Furthermore, we will often write expressions of the form

```
Identity . doSomething . runIdentity
```

to unwrap (`runIdentity`), transform (`doSomething`), and rewrap (`Identity`) values. Hence, we will use records so that unwrapping functions are generated automatically. At least, until we find a better way.

One of the common uses of `newtype` is to provide alternative implementations of various type classes. E.g., if for some program-specific reason, we wanted to order pairs based on second-element first, we might consider

```
module BackwardsPair where

import Data.Ord

newtype BackwardsPair a b = BackwardsPair (a,b)
    deriving Show

instance (Eq a,Eq b) => Eq (BackwardsPair a b) where
    BackwardsPair p1 == BackwardsPair p2 = p1 == p2

instance (Ord a, Ord b) => Ord (BackwardsPair a b) where
    compare = comparing (\(BackwardsPair (a,b)) -> (b,a))
```

The function `comparing :: Ord a => (b -> a) -> b -> b -> Ordering` is very useful for defining one ordering in terms of another. This is a slightly silly example, but we'll see more substantial examples soon.

9.3 List Comprehensions

Mathematicians have a concise notation for describing a set, which typically involves describing an initial set, a predicate, and a form for elements of that set, e.g., $\{x^2 \mid x \in \omega \text{ and } x \text{ is even}\}$, the squares of all even natural numbers. These are called *set comprehensions*. Haskell provides a similar notation for lists:

```
> [x^2 | x <- [1..10], even x]
[4,16,36,64,100]
```

It is possible in list comprehensions to have multiple generators, `let` bindings, and to interleave generations, tests, and bindings. As a simple example, let's generate all of the pythagorean triples that where the hypotenuse is less than a given number:

```
pythagoreanTriples n =
    [ (a,b,c)
    | a <- [1..n]
    , b <- [a+1..n]
    , c <- [b+1..n]
    , a^2 + b^2 == c^2
    ]
```

```
> pythagoreanTriples 20
[(3,4,5),(5,12,13),(6,8,10),(8,15,17),(9,12,15),(12,16,20)]
```

Note that the results are sorted by `a`, because the generation runs like this, for each `a` from `1` to `n`, generate `b` from `a+1` to `n`, and for each such `a` and `b`, generate `c` from `b+1` to `n`...

<!– Also note our use of *tuples*. Tuples look like lists, but they're very different. We'll talk about them more next lecture, but for now, they're a convenient way to package up a few values. –>

But notice that this list contains a few *non-primitive* triples, e.g., `(6,8,10)`, `(9,12,15)`, and `(12,16,20)`, which are all multiples of `(3,4,5)`. Suppose we wanted to restrict ourselves to *primitive* triples, i.e., tuples that are relatively prime to one onother. How might we do that? A simple approach would be to filter out those triples where `a` and `b` have a non-trivial common divisor, i.e.,

```
primitiveTriples n =
    [ (a,b,c)
    | a <- [1..n]
```

```
, b <- [a+1..n]
, gcd a b == 1
, c <- [b+1..n]
, a^2 + b^2 == c^2
]
```

It is more efficient to do the gcd test *before* the generation of c, because otherwise we'd have to repeat the same test (on a and b alone) for each c; but it should be noted too that our basic computational plan emphases clarity over efficiency, and there are much more efficient ways to generate lists of pythagorean triples.

Exercise 9.3 *The file* `Records.hs` *defines two* `Teacher`*s,* `professorChugh` *and* `professorKurtz`*, and a "database" of* `Student`*s,* `allStudents`*.*

Implement the function

```
studentsOfTeacher_ :: [Person] -> Person -> [((Int, Int), [(String, String)])]
studentsOfTeacher_ students teacher = undefined
```

to return those `students`*, identified by lastname-firstname pairs, enrolled in each of the* `teacher`*'s courses. Your implementation can assume that* `students` *is a list of* `Student` *values and that* `teacher` *is a* `Teacher` *value.*

For example:

```
> studentsOfTeacher professorChugh
[((16100,1),[("Student","B"),("Student","STEAM")])]

> studentsOfTeacher professorKurtz
[((16100,2),[("Student","C")]),((28000,1),[("Student","D"),("Student","E")])]
```

Once you are done, consider how one might eliminate the assumption above (but don't submit this).

Part II

Core Type Classes and Instances

Chapter 10

Maybe Monad is Not So Scary

(Chapter contributed by RC)

[RC: See https://www.classes.cs.uchicago.edu/archive/2023/winter/22300-1/notes/maybe-monad/]

Chapter 11

Functors

In Wednesday's lecture, we developed the function `mapMaybe :: a -> b -> Maybe a -> Maybe b`, having earlier seen `map :: a -> b -> [a] -> [b]`. There's a similar pattern of type use here, in which we take a *pure* function of type `a -> b`, and *lift* it to a function of type `f a -> f b`, where `f` is either `Maybe` or `[]`. Having seen this pattern twice, it's tempting to generalize it, and that is done in the standard library via

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

The word *functor* comes from Category Theory, where a functor is a homomorphism between two categories. We're not going to define categories here, but *homomorphisms* are simply structure-preserving functions from one mathematical structure to another. The relevant structure to be preserved here is type-specialized instances of the identity function `id`, and composition `(.)`, and so we require

- `fmap id = id`
- `fmap (f . g) = fmap f . fmap g`

These equations are not checked by the compiler, but all of the `Functor` instances of the standard Haskell libraries satisfy these functor equations, and all your instances should, too. Advanced Haskell compilers are permitted to *assume* that the functor laws (and similar laws encountered in related type classes) are valid, and to make code transformations (a.k.a., *optimizations*) on that basis.

To a programmer's first approximation, a `Functor` is a parameterized type that can be "mapped over." Intuition is gained through use.

The `Functor` type class is defined in the `Prelude`, which also defines `(<$>)`, a commonly used infix version of `fmap`.

Review: Identity

We saw the identity parameterized type in Lecture 7:

```
newtype Identity a = Identity { runIdentity :: a }
    deriving (Show)
```

A value of `Identity a` type can be thought of as a value of type `a` contained in "an Emperor's clothes box", i.e., a virtual box that isn't really there, but which the type system believes it can see, and insists that we see too. The functor instance is extremely simple: take something out of the (virtual) box, hit it with the function, and put the result back in a (virtual) box:

```
instance Functor Identity where
    fmap f (Identity x) = Identity (f x)
```

alternatively, we could have written

```
instance Functor Identity where
    fmap f = Identity . f . runIdentity
```

Keeping in mind that `Identity` and `runIdentity` are just `id` at runtime, this means that `fmap` is just `id` too. This works pretty much as you'd expect:

```
> (1+) <$> Identity 2
Identity 3
```

Note our use of the infix version `(<$>)` of `fmap`.

It would seem that there's not much to say about `Identity`, nor that it's very useful. Neither of these are true. `Identity` plays an important role in the theory *and practice* of monad transformers, a topic for later in the quarter.

Review: Maybe

The `Functor Maybe` instance is pretty simple:

```
instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

These are just the defining equations of `mapMaybe`, which is revealed as a `Maybe`-specific version of `fmap`.

A Digression: Kinds, and the Types of Types

In the discussion above, we've referred to `Identity`, `Maybe`, `[]`, and even the parameter `f` from the class definition of `Functor` as *parameterized types*. These are types in the nomenclature of Haskell, but they're clearly different from types like `Int`, `Double`, etc. It's useful at this point to digress briefly to the language of *kinds*, which are nothing more or less than the types of types. Fortunately, Haskell's kind system is very simple.

Familiar types like `Int`, `Bool`, and `Double` have kind `*`. These are the ground types of Haskell, and are also the types that can have values.

`Identity`, `Maybe`, and `[]` all have kind `* -> *`. Intuitively, these are types that can be applied to types of kind `*`, resulting in a type of kind `*`. The pairing type constructor `(,)` is more complicated still, as it expects two argument types, both of kind `*`, and results in a value that is a type of kind `*`. Unsurprisingly, `(,)` has kind `* -> * -> *`.

Kinds are relevant to our discussion here because all of the instance types of a fixed type class must have the same kind. Thus, all instances of `Functor` have kind `* -> *`. What may not be as obvious is that those instances might arise via *type expressions*.

This language of kinds can also help us understand one of the category theoretic structures that is peeking out from the shadows of this course. The category **Hask** has Haskell types of kind `*` as its objects, and ordinary Haskell functions as its arrows. (Note that there are some technical issues here over divergence, which we're glossing over.) From this point of view, a type of kind `* -> *` is an *endomorphism* (a function with the same domain and range) on the objects of **Hask**. Thus a `Functor f` is just a *homomorphism* on **Hask**, where `f` itself is the homomorphism's endomorphism on objects, and `fmap` is the corresponding endomorphism on arrows.

Either

The `Either` type is a standard Prelude type similar to the `Maybe` class, but we allow the `Nothing`-like alternative to carry along arbitrary information. After all, when an error occurs, we usually want a bit more information.

```
data Either a b = Left a | Right b
    deriving (Eq, Ord)

instance Functor (Either a) where
    fmap _ (Left x) = Left x
    fmap f (Right y) = Right (f y)
```

Here we think of `Left` as the `Nothing`-like alternative, and `Right` (which functions as a pun [and what's a pun but a type error coerced into a joke?]: positionally as regards the declaration, and normatively in the "non-error value" sense) is analogous to `Just`. As the grandfather of a southpaw, I find this to be chirally incorrect, but one can only fight so many battles at one time. Note that `Either` has kind `* -> * -> *`, but for any type `s` of kind `*`, `Either a` is a type of kind `* -> *`, which is the kind of `Functor` instances.

A temptation here is to rewrite the `Left` case of the instance definition of `fmap` above, so as to avoid the pattern match and the (apparent) reconstruction of `Left x`. A hard moment's reflection, however, reveals that `Left x` has

different type on the LHS than the RHS, and so this can't be avoided. They're different `Left`'s.

Review: Lists

Lists are one of our motivating examples, and so it's not surprising that

```
instance Functor [] where
    fmap = map
```

and that is exactly how it *is* defined in `GHC.Base`.

Exercise 11.1 *Create an appropriate* `Functor` *instance for* `BinaryTree`*:*

```
data BinaryTree a
    = EmptyTree
    | Node a (BinaryTree a) (BinaryTree a)
    deriving (Show)
```

Pairs

Remember that we can write the ordered pair type `(a,b)` as `(,) a b`. This allows us to make ordered pairs an instance of functor, by having `fmap` act on the second coordinate:

```
instance Functor ((,) a) where
    fmap f (a,b) = (a,f b)
```

For example

```
> (+1) <$> ("one",1)
("one",2)
```

Exercise 11.2 *The* `Functor` *instance for pairs might make the first components of pairs jealous: why should the second components get all the attention?*

We might try to be clever and use our knowledge of type aliases to rearrange the type variables and provide the following additional instance declaration:

```
type Pair a b = (,) b a
```

```
instance Functor (Pair b) where
    fmap f (x, y) = (f x, y)
```

This makes Haskell very unhappy. What error does Haskell report and why?

Exercise 11.3 *An alternative approach to dealing to the problem of Exercise 9.2 is via the* `Bifunctor` *type class defined in* `Data.Bifunctor`*, which has the class definition:*

```
class Bifunctor p where
    bimap :: (a -> b) -> (c -> d) -> p a c -> p b d
    first :: (a -> b) -> p a c -> p b c
    second :: (c -> d) -> p a c -> p a d
```

A type of class `Bifunctor` *must have kind* `* -> * -> *`*. Provide instances of* `Bifunctor` *for* `(,)` *and* `Either`.

Functions

We can think of functions `g :: a -> b` as being containers of `b`'s, indexed by `a`'s. As such it is easy to turn ordinary functions into functors.

The idea is that if we `fmap` via `f` across such a container `g`, and look up the value of that container at `a`, we have

```
(fmap f g) a = f (g a)
             = (f . g) a
```

which we η-reduce to

```
fmap f g = f . g
         = (.) f g
```

which can be η-reduced twice more, giving us

```
instance Functor ((->) a) where
    fmap = (.)
```

This is surprising, but in a way, argues for the naturalness of what we're doing, as the `fmap` function can be thought of as a simultaneous generalization of `map` and `(.)`.

Exercise 11.4 *What is the type of* `fmap` *in the* `((->) a)` *instance? (Hint: it may help to first write out the type of* `fmap` *in the other instances above.)*

Association Lists

Association lists are lists of key-value pairs, and have long been used in functional programming languages to represent finite functions. In Haskell, we can represent association lists very cleanly:

```
type Assoc a b = [(a,b)]
```

Or, perhaps more suggestively for our current purposes,

```
type Assoc a b = [] ((,) a b)
```

If this were ordinary code, we'd be tempted to write

```
type Assoc a b = [] ((,) a b)
               = ([] . (,) a) b
```

and then η-reduce to

```
type Assoc a = [] . (,) a
```

This sort of thing is possible (albeit, not with this syntax), but our point here isn't to repurpose our code rewriting techniques as type rewriting techniques, but instead to note that `Assoc a` can be thought of as involving the composition of two functors, `[]` and `(,) a`. This is the more significant observation. Recall that we earlier identified functors with homomorphisms of categories. Compositions of homomorphisms are homomorphisms, and therefore compositions of functors must also be functors. The question is how to take advantage of this?

Our goal is to make `Assoc a` (where `Assoc a b` is a type alias for `[(a,b)]`) into a functor, much as `(->) a` is a functor (remembering now that `Assoc` is often used to represent finite functions). One problem is that we can't turn a type *synonym* into a functor, only a *type*, optionally applied to type variables. Another problem is that the outer type constructor `[]` is already a `Functor`, and types can belong to type classes in only one way.

A first approach to this solution would be to wrap our aliased type, and define

```
newtype Assoc a b = Assoc { getPairs :: [(a,b)] }

instance Functor (Assoc a) where
    fmap f assoc = Assoc [(a,f b) | (a,b) <- getPairs assoc]
```

This works well. An alternative might be to stick with the original definition of `Assoc` as a type alias, and introduce a new name for the `fmap` function associated with the composed functors, i.e.,

```
amap :: (b -> c) -> Assoc a b -> Assoc a c
```

When we do this, and rework the code, something very striking happens...

```
amap f ps = fmap (fmap f) ps
          = (fmap . fmap) f ps
```

which η-reduces to

```
amap = fmap . fmap
```

This is easy enough, and after a while intuitive enough, that Haskell programmers typically don't wrap composed functor instances, nor do they introduce new names associated with viewing a composed functor instance as a functor directly. Instead, they just use `(fmap . fmap)` to lift a function through two functors.

The really striking thing is that this generalizes! Let's consider a type that involves a three-level composition of functors, e.g.,

```
type AssocMap a b c = [(a,b->c)]
```

This is a type alias that composes the functors `[]`, `(,) a`, and `(->) b`, applying the result to `c`. If we want to produce a function that does a remapping three levels down:

```
mmap :: (c -> d) -> AssocMap a b c -> AssocMap a b d
```

We'll eventually derive `mmap = fmap . fmap . fmap`! So we can just use `fmap . fmap . fmap`. The first time I saw this idiom in live code, my mind was blown. The first time you see it, your mind will probably be blown too, but at least it will be something you've seen before.

As if that weren't enough, this idiom can be generalized to a few other type classes, but that's a story for another day.

Chapter 12

Applicative

Functors are very nice. They give us a uniform mechanism for "lifting" functions defined at a simple type to act on complicated types that are parameterized by that type, For example, if `F` is a Functor, and `f :: A -> B`, then `fmap f :: F A -> F B`.

But what happens if `f` isn't unary? Consider a hypothetical `f :: A -> B -> C`. Intuitively, we might hope `fmap f :: F A -> F B -> F C`, but it doesn't work that way. We've already defined `fmap` so that `fmap f :: F A -> F (B -> C`, i.e., the result of the first application is to produce a function in an `F` box, rather than a function that takes boxed values, and returns a boxed value. We could work with this if `F` had a mechanism for applying boxed functions to boxed arguments, returning boxed results.

The `Applicative` type class addresses this issue.

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
    ...
```

A type `f` is `Applicative` if it is a `Functor` that, additionally, comes with two members: an infix operator `(<*>)` (pronounced "ap") that takes a boxed function of type `f (a -> b)`, applies it to a boxed argument of type `f a`, and produces a boxed result of type `f b`; and a function `pure` of type `a -> f a` that "lifts" simple values into boxed ones. There are more functions in the `Applicative` type class than these, but the pair `pure` and `(<*>)` are both minimally complete for, and the most commonly used functions of, the `Applicative` type class.

Once we have this, we can "factor" `fmap`:

```
fmap f appA = pure f <*> appA
```

Or, for those who like to play with η-reduction:

```haskell
fmap = (<*>) . pure
```

Recall that `(<$>)` is an often used infix version of `fmap`. Indeed, `(<$>)` is especially useful in applicative contexts, as we saw in Lecture 5.

In addition to the `Functor` laws involving `fmap`, every `Applicative` must satisfy the following laws:

- `pure id <*> v = v`
- `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- `pure f <*> pure x = pure (f x)`
- `u <*> pure y = pure ($ y) <*> u`

We're not going to dwell on these laws now, but in time, they'll seem obvious.

Maybe

Our handling of `Maybe` in Lecture 5 was intended to anticipate and motivate our discussion of `Applicative` and other category theoretic type classes. Recall that we defined

```haskell
justMaybe :: a -> Maybe a
justMaybe = Just

applyMaybe :: Maybe (a -> b) -> Maybe a -> Maybe b
applyMaybe (Just f) (Just a) = Just $ f a
applyMaybe _ _ = Nothing
```

These functions had exactly the types and roles that we need for an `Applicative` instance of `Maybe`, and so,

```haskell
instance Applicative Maybe where
    pure = Just

    (Just f) <*> (Just x) = Just (f x)
    _ <*> _ = Nothing
```

The definition of `(<*>)` in `Control.Applicative` is a bit different, but produces the same values.

The effect of these definitions is to enable a very general mechanism for dealing with computations that may result in errors, in which the usual evaluation mechanism is adjusted to allow errors to propagate through the usual evaluation process, and so requires no other special handling.

Recall, for the moment, our `Maybe` arithmetic, not from Lecture 5, but from Lecture 3, where we created a derived instance

```
-- | Derived instance definition for Num (Maybe n) given Num n.

instance Num n => Num (Maybe n) where
    Just a + Just b = Just $ a + b
    _ + _ = Nothing

    Just a - Just b = Just $ a - b
    _ - _ = Nothing

    Just a * Just b = Just $ a * b
    _ * _ = Nothing

    negate (Just a) = Just $ negate a
    negate _ = Nothing

    abs (Just a) = Just $ abs a
    abs _ = Nothing

    signum (Just a) = Just (signum a)
    signum _ = Nothing

    fromInteger i = Just $ fromInteger i
```

We can use `Applicative` to substantially simplify this definition. First, we introduce the function `liftA2`, which is actually a part of the full definition of the `Applicative` type class, but you have to include `Control.Applicative` to get it:

```
class Functor f => Applicative f where
    ...
    liftA2 :: (a -> b -> c) -> f a -> f b -> f c
    liftA2 f a b = f <$> a <*> b
```

With this in hand, we can write:

```
instance Num n => Num (Maybe n) where
    (+) = liftA2 (+)
    (-) = liftA2 (-)
    (*) = liftA2 (*)
    negate = fmap negate
    signum = fmap signum
    abs = fmap abs
```

```haskell
        fromInteger = pure . fromInteger
```

Which is both amazingly more concise, but also, clearer once you gestalt/grok `Applicative`. Note that `fmap` can be thought of as `liftA1`, and indeed, there is a legacy `liftA` function from the dark days of 2015 and earlier, before all instances of `Applicative` were required to be instances of `Functor`, that is exactly that. There's also a `liftA3`, but surprisingly, no `liftA4`. If you need it, you'll have to write it. But returning to `liftA1`/`liftA`: an emerging Haskell convention that when essentially the same function is defined in two related type classes, to prefer the name associated with the more general type class, hence `fmap` above.

Exercise 12.1 *In the spirit of the* `liftA*` *functions, implement the following to lift an unboxed function and apply it to a boxed list of arguments.*

```haskell
liftAN :: Applicative f => ([a] -> b) -> f [a] -> f b
```

How useful is this function?

Exercise 12.2 *Unsurprisingly, there is also an instance of* `Applicative` *for* `Either a`. *Provide an instance definition, and compare it to the definition in the Haskell sources.*

Exercise 12.3 *Perhaps surprisingly, given the foregoing, there is* not *an* `Applicative` *instance for* `(,) a`. *Why not?*

List

It is sometimes useful to think of a function `f :: A -> [B]` as a non-deterministic function of type `f :: A -> B`, i.e., a function that can have zero or more return values. In this case, it may help to think of the `[]` type constructor as a "computational context" rather than a "box" of values. From such a perspective, the effect of "applying" a list of functions to a list of arguments ought to be to non-deterministically select a function, and apply it to a non-deterministically selected argument, i.e., to form the list of all the ways we can apply functions to arguments. Thus,

```haskell
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

Thus,

```
> [] <*> [2,3] <*> [4,5]
[]
> [(+)] <*> [2,3] <*> [4,5]
[6,7,7,8]
> [(+),(*)] <*> [2,3] <*> [4,5]
[6,7,7,8,8,10,12,15]
```

Note that the resulting list for the last expression above is `[2+4,2+5,3+4,3+5,2*4,2*5,3*4,3*5]`, in precisely this order.

Exercise 12.4 *Write an expression in applicative style that computes the same result as:*

```
[ (x,y,z) | x <- [1..3], y <- [1..3], z <- [1..3] ]
```

ZipList

It turns out that there's a second, natural way to implement a list as an `Applicative`. The idea is to represent parallel (rather than non-deterministic) computation, i.e., that the i-th element of the result list comes from applying the i-th operation to the i-th operand. As we are well aware by now, however, a type can be an instance of a type class in only one way, and therefore we need to use `newtype` to create a (virtual) distinct type for the purpose of driving type class instance selection:

```
newtype ZipList a = ZipList { getZipList :: [a] }
```

Thus, a `ZipList a` is just a `[a]` inside a (virtual) `ZipList` box.

Of course, we do this to provide a distinctive `Applicative` instance, but we have to provide a `Functor` instance as well. We'll use what is essentially the standard definition of `fmap` for lists, acting within the box:

```
instance Functor Ziplist where
    fmap f (Ziplist xs) = ZipList (fmap f xs)
```

Note here that the `fmap` on the right hand side of the definition is `[]`'s `fmap`, i.e., our old friend `map`. We now define the following:

```
instance Applicative ZipList where
    (ZipList fs) <*> (ZipList xs) = ZipList (zipWith id fs xs)
```

This requires a *bit* of explanation, because it's probably not what you'd expect. Certainly, *I'd* expect something like a binary function that performs application (e.g., `\f x -> f x`). But...

```
\f x -> f x = \f -> \x -> f x
            = \f -> (\x -> f x)
            = \f -> f
            = id
```

Or, to put this same pun differently,

```
id f x = (id f) x
       = f x
```

So we didn't need to "roll up" a special purpose binary application function. We already had one, in the identity function. Weird.

The alert reader/listener will have noticed that I haven't yet provided a definition of `pure` for `Applicative ZipList`. This takes a bit of thought... Let's think about what we want:

```
> id <$> ZipList [1]
ZipList {getZipList = [1]}
> id <$> ZipList [1,2]
ZipList {getZipList = [1,2]}
> id <$> ZipList [1,2,3]
ZipList {getZipList = [1,2,3]}
```

Hmm. So `pure id` has to be a function that contains `id` in every coordinate of a list of indeterminate length. Haskell's laziness bails us out here. Lists are not necessarily finite, and we can perform useful computations using infinite lists (as long as finiteness comes from somewhere else)

A standard Haskell function is

```
repeat :: a -> [a]
repeat a = a :: repeat a
```

In effect, `repeat` describes a computational process for building a potentially infinite list.

```
instance Applicative ZipList where
    pure x = ZipList (repeat x)
```

Of course, this definition of `pure` has implications for *arguments* as well as functions, cf.

```
> (+) <$> pure 3 <*> ZipList [1..4]
ZipList {getZipList = [4,5,6,7]}
```

Exercise 12.5 *Consider the following two, very similar looking calculations:*

```
> [(+),(*)] <*> pure 2 <*> pure 3
[5,6]
```

```
> ZipList [(+),(*)] <*> pure 2 <*> pure 3
ZipList {getZipList = [5,6]}
```

The results of these computations (modulo syntactic noise around `ZipList`*) are identical, but the computational patterns that produce these results are quite different. Explain the difference.*

Exercise 12.6 *There are several additional operators defined to improve readability when writing programs in applicative style:*

```
(<$)   :: Functor f => a -> f b -> f a
(*>)   :: Applicative f => f a -> f b -> f b
(<*)   :: Applicative f => f a -> f b -> f a
(<**>) :: Applicative f => f a -> f (a -> b) -> f b
```

We won't often use them in our examples. But, similar to our discussion of `foldMap` *and* `foldr` *last time, it can be helpful to think about how to implement such polymorphic functions based only on their types and what we know about the type classes that are mentioned in their constraints.*

Try implementing these functions before peeking at them in the libraries.

Functions

We saw last time an `Functor` instance for `(->) a`, i.e., functions that have domain type `a`. There is also an `Applicative` instance, and it's worth working through.

We begin by considering `pure`. Let's suppose that `b :: tb`, and consider `pure b`. This has to be a function of type `a -> tb`, which leaves us with the conundrum of what to do with the argument, and what to provide as a result. The only plausible answers are to (a) ignore the argument, and (b) to return `b`, as it's the only value of type `tb` available to us! Thus,

```
pure b = \a -> b
```

This is actually a predefined function,

```
const :: a -> b -> a
const a _ = a
```

Next, we need to implement `(<*>)`, which in this context will have type `(a -> b -> c) -> (a -> b) -> (a -> c)`. There's pretty much only one thing we can do:

```
f <*> b = \a -> f a (b a)
```

We can apply both sides to a, reduce, and get

```
(<*>) f b a = f a (b a)
```

Thus,

```
instance Applicative ((->) a) where
    pure = const
    (<*>) f b a = f a (b a)
```

Surprisingly, this sort of thing can be useful. Let's suppose, for the sake of argument, we want to compute the sum of all of the integers from 1 to 100 which are divisible by either 2 or 3. We can do this:

```
divisibleBy :: Int -> Int -> Bool
divisibleBy d n = n `rem` d == 0

result1
    = sum
    . filter (\x -> divisibleBy 2 x || divisibleBy 3 x)
    $ [1..100]
```

That's not terrible, but at some point in your development as a programmer, you'll decide that the expression `\x -> divisibleBy 2 x` | divisibleBy 3 x| is too low level an approach to building the "or" of two predicates. It should be possible to take the "or" more directly. At this point, you might define

```
orf :: (a -> Bool) -> (a -> Bool) -> (a -> Bool)
orf f g a = f a || g a

result2
    = sum
    . filter (divisibleBy 2 `orf` divisibleBy 3)
    $ [1..100]
```

After all, all you're doing is lifting or. At some point, this language will remind you that there are already type classes for dealing with lifted types, and a light goes on. You didn't need to define `orf` at all. Like Dorothy and the ruby red slippers, you already had what you needed:

```
result3
    = sum
    . filter ((||) <$> divisibleBy 2 <*> divisibleBy 3)
    $ [1..100]
```

or,

```
result4
    = sum
    . filter (liftA2 (||) (divisibleBy 2) (divisibleBy 3))
    $ [1..100]
```

There's no place like home.

Chapter 13

Monads

Monads

Let's review the bidding. We've seen two category theory inspired type classes:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

and

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

These type classes reflect common patterns of type usage, and a language that is we find helpful is that of *plain* types like `a`, vs. *fancy* types like `f a`. Thus, a `Functor` must implement a function `fmap` that enables us to transform a function on plain types to a function on fancy types. The `Applicative` function `pure` enables us to promote a plain value to a fancy value. Finally, `(<*>)` can be thought of as "apply for fancy things," or, if we think of it as a unary function in the same way we think of `fmap` as a unary function, a means to distribute fanciness over `(->)`, converting a fancified function into a plain function whose domain and range are fancy values.

Our next type pattern is `Monad`, boogie-man of Haskell. But as we'll see, monads are are just instances of a pattern of type use that occurs a lot in programming. We'll start with an abridged definition:

```
class Applicative m => Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
```

This looks a bit odd, but we can see in this a kind of "application written backwards," where we take a fancy value, and a *fancifying* function that maps plain values to fancy values, and some how "apply" the latter to the former, obtaining a fancy value. There are reasons why these things are the way they are.

These peculiar functions are sometimes called "actions," and `(>>=)` serves as a sequencing mechanism. It is natural, in such contexts, to describe a sequence of actions (beginning with a start value) as a pipeline, in which the actions take place in the order they're written. The operation `(>>=)` is called *bind*, because in a common notation for describing sequenced actions in which the `(>>=)` operator is cleverly elided, it's effect will be to establish the binding of a name to plain values extracted from fancy values.

There are other functions that belong to the `Monad` type class, but they have default definitions, and we can defer their consideration for the time being.

`Identity`

The `Identity` instance of bind is very simple: given a fancy value, we simply remove it from its virtual box, obtaining a plain value, and apply our fancifying function to the result.

```
instance Monad Identity where
    Identity a >>= f = f a
```

You can see that it's the fancifying function's responsibility to box its output.

`Maybe`

The `Maybe` instance of bind is also very simple: a `Just` value is handled like an `Identity` value, whereas there's no plain value to be extracted from a `Nothing` argument, and so we have a `Nothing` result:

```
instance Monad Maybe where
    Just a >>= f = f a
    Nothing >>= f = Nothing
```

I think you'll agree, we haven't seen anything scary yet.

A Bit of History

Monads solved a crucial problem in the design of pure functional languages like Haskell, i.e., how to deal with IO, which is intrinsically impure. Monads provide an interface for defining a kind of "isolation ward" in which impure actions could be defined and performed, apart from the pure core of the language, but in contact with it. This is part of why monads became the boogie-man of Haskell, because a full accounting of something as simple as the standard "Hello, world!" program required this digression through category theory, and a very abstract type class.

As the language was first developed, `Monad` was a stand-alone type class, i.e., it wasn't necessarily the case that instances `m` of the `Monad` type class belonged to `Functor`, let alone `Applicative`. In order to make the `Monad` useful, other functions were included in the standard interface:

```
instance Monad m where
    ...
    return :: a -> m a
    fail :: String -> m a
```

We recognize that `return` has the same type pattern as `pure`, and indeed `Monad` instances are required to satisfy the law `return = pure`, and a default definition of `return` is provided so that this is so. This results in a bit of confusion over lexicon: which should you use? The sense of the community is evolving, but we believe it will converge on the simple solution of always preferring `pure`, and that's the style we'll teach. We have a *lot* of code that we'll need to convert. In any event, if you ever encounter a `return` in the wild, read `pure`, and code on!

The inclusion of `fail` is more controversial. The monads of category theory do not have a `fail` function, and there are many types which would otherwise be monads for which no sensible `fail` function can be written. Indeed, the most common implementation of `fail` is `fail = error`, i.e., to raise an exception. This is just one of those cases where the motivation for including the monad abstraction (dealing with IO) resulted in a bad design decision. IO can fail, and there needs to be a way to deal with IO failures, but including `fail` in `Monad` was not the right choice.

The status quo is that a new type class has been defined,

```
instance Monad m => MonadFail m where
    fail :: String -> m a
```

and that the `fail` function of `Monad` is now deprecated, and will be removed in the future. *Caveat emptor!*

A Bit of Law

There are several axioms that describe how `pure` and `(>>=)` interact, and Haskell makes optimizations based on the assumption that they do:

- **left identity:** `pure a >>= f` is equivalent to `f a`
- **right identity:** `ma >>= pure` is equivalent to `ma`
- **associativity:** `(ma >>= f) >>= g` is equivalent to `ma >>= (\x -> f x >>= g)`

These laws (especially associativity) seem a bit odd. There's a simple transformation that makes them more comprehensible. Category theorists describe functions of type `a -> m b` where `m` is a monad as *Kleisli arrows* (pronounced "KLAI-slee"). We will think of them as machines that take plain values as input and produce fancy values as output. A key notion is *Kleisli composition*, which is a means for composing this sort of machine. It's useful to directly compare the type patterns involved:

```
(.)   ::            (b ->   c) -> (a ->   b) -> a ->   c
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
```

It is, by the way, quite easy for us to define `(<=<)`, and worth considering in parallel with the definition of `(.)`. We'll use `(=<<)` here, an infrequently used flipped bind, to accentuate the analogy between the definition of `(.)` and `(<=<)`:

```
f  .  g = \x -> f  $  g x
f <=< g = \x -> f =<< g x
```

In category theory, if we have a monad over a category, we can build the *Kleisli category* in which the objects are the objects of base category, the arrows from `a` to `b` are the Kleisli arrows of type `a -> m b`, and the identity is `pure`, and composition is `(<=<)`. Thus, the ordinary axioms of category theory for this category are:

- **left identity:** `pure <=< f` is equivalent to `f`
- **right identity:** `f <=< pure` is equivalent to `f`
- **associativity:** `(f <=< g) <=< h` is equivalent to `f <=< (g <=< h)`

These are equivalent to the laws above (although the careful reader will note that the chirality of the identity laws is flipped), but much more intuitive.

As with the other category theoretic type classes, Haskell compilers are allowed to assume that the monad laws are satisfied by any `Monad`, and to transform code accordingly. Naturally, all of the `Monad` instances defined in the Haskell libraries do.

Lists

The `[]` is also an instance of `Monad`, but in a more interesting way than `Identity` or `Maybe`. It is useful to start with a brief digression. Folks who are familiar with category theory via mathematics may be a bit perplexed by our presentation of monads. In the standard approach, the critical function is `join :: Monad m => m (m a) -> m a`. The way we like to think about this is that the plain/fancy hierarchy really only has two levels: plain and fancy. If we try to create an "extra fancy" value, we can demote it to ordinary fanciness by simply applying `join` to it.

The `join` function is often fairly easy to understand. If we have a double layered structure, we can often delayer it, e.g.,

```
join :: Identity (Identity a) -> Identity a
join (Identity (Identity a)) = Identity a

join :: Maybe (Maybe a) -> Maybe a
```

```
join (Just (Just a)) = Just a
join _ = Nothing
```

If we consider `join` on lists, we have:

```
join :: [[a]] -> [a]
```

There is already a very natural list function that has this type pattern:

```
concat :: [[a]] -> [a]
```

In this particular case, `join` essentially "flattens" our list structure, forgetting the boundaries between sublists. There are a lot of monads which `join` works this way, "erasing" interior boundaries.

There is something unreasonable about what we've just done, taking the type of something we're looking for, and then reaching into our magic hat, and pulling out a more familiar object that just happens to have the same type, and then claiming that that's what we've been looking for. What is surprising, in both mathematics and in type-driven development, is that it's often the case that the pieces fit together in essentially only one way. This has the perfectly unreasonable consequence that if we're looking for something that has a particular type, and there is only one way to build that object, then getting the types right is all we need to do.

We can use this idea to "discover" the relationship between `join` and `(>>=)`. Let's suppose we want to build `(>>=)` out of `join`. Our goal is to get to the type:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

Recall `fmap`:

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

If we supply `fmap` with an argument of type `a -> m b` where `m` is a monad, and we apply it to an argument of type `f a`, we'll have

```
fmap :: (Monad m) => (a -> m b) -> m a -> m (m b)
```

which is pretty close. The arguments are in the wrong order, and have to somehow get from `m (m b)` to `m b`, but that's what `join` is for. Thus, we guess

```
ma >>= f = join (f <$> ma)
```

It's a good guess.

Exercise 13.1 *Show that* `(>>= id) :: Monad m => m (m a) -> m a`*, and therefore that* `join = (>>= id)` *is a plausible definition for* `join` *in a* `Monad`*. It is, in fact, the definition.*

Putting all of this inspired, insane, guess work together, we have

```
instance Monad [] where
    xs >>= f = concat (f <$> xs)
```

The actual definition of `(>>=)` is equivalent, and perhaps even a bit more intuitive.

```
instance Monad [] where
    xs >>= f = [y | x <- xs, y <- f x]
```

Thus, `(>>=)` on lists is actually fairly simple. We consider the elements `x` of the argument list `xs` one at a time, for each such element we form `f x`, a list, which constitutes a part of the value we're constructing. We then combine these sublists by concatenation.

One caution in this: the `join` function is an ordinary function, and not part of the `Monad` type class. This is unfortunate, as it means there is no default definition of `(>>=)` in terms of `join`, but also that our approach of re-defining `join` will often entail explicit namespace specifications.

Making Monads Work for Us

We've invested some effort into monads, their definition, and a few of their instances. But we haven't yet used them to solve any programming problems. Let's take a look at a bit of code:

```
pure 1 >>= \x ->
pure (x + x) >>= \x ->
pure (x * 2) >>= \x ->
pure x
```

This can be evaluated (if you give the type checker enough information to know that you're working in with `Identity` as the monad), and returns `Identity 4`.

This is a bit confusing, especially as to why anyone would want to do something so convoluted, but this has a lot to do with the fact that the glue code associated with the `Identity` monad is trivial. More complicated examples are coming, as are some meaningful syntactic simplifications. In interpreting this, it is important to understand that >>= has low precedence, so this is really:

```
pure 1 >>= (\x ->
  pure (x + x) >>= \(x ->
    pure (x * 2) >>= (\x ->
      pure x)))
```

I.e., each lambda's body extends all the way to the bottom. We've laid this out so that each line represents a deeper level of lexical nesting. This syntax is certainly awkward, but keep in mind we're building up a machinery for contexts that are more interesting than simple transparent boxes. Moreover, Haskell has a special syntax for working with monads, the semi-familiar `do`, which is nothing more than syntactic sugar for expressions built out of `(>>=)`. Consider the following, which is just a `do`-sugared variant of the expressions above:

```
do
    x <- pure $ 1
    x <- pure $ x + x
    x <- pure $ x * 2
    return x
```

This makes our expression look a lot like assignment-based procedural code that is so familiar to C and Java programmers, with just a bit of extra syntactic noise. And we can think about it that way, too, although that's not what actually is happening. We're not making a sequence of assignments to a single variable `x`, instead we're establishing a "new" `x` on each line, whose scope extends to the next `x`, and we're binding a value to it.

Thus, it is possible to write code that looks imperative, but with a functional meaning and interpretation, in which sequencing is a metaphor for lexical nesting, and so makes it easy for us to use and reason about deeper levels of lexical nesting than we'd otherwise attempt.

The structure of `do` notation is fairly straight forward. Let's suppose that the value we're constructing has type `m a` for some monad `m`. Then every line of the `do` will be comprised of a value of type `m b` for some `b`, optionally including the binding of a name via the `<-` syntax. Consecutive lines are combined using `(>>=)` when there is an explicit binding, and `(>>)` when there is not, where

```
(>>) :: Monad m => m a -> m b -> m b
ma >> mb = ma >>= \_ -> mb
```

is just a binding that doesn't retain the value of its argument.

Let's use this with `[]`. Consider the word "Mississippi." Kids love this word, because of the repetition of letters, and the rhythm of spelling it out. We might even think of using this repetition as a way of compressing the spelling, e.g., representing it by a value of type `[(Int,Char)]` where each pair is a count together with a letter. Thus,

```
mississippiCompressed :: [(Int,Char)]
mississippiCompressed
```

```
    = [(1,'M'),(1,'i'),(2,'s'),(1,'i'),(2,'s'),(1,'i'),(2,'p'),(1,'i')]
```

Consider now the job of decompressing such a representation, of recovering the original string.

```
decompress :: [(Int,a)] -> [a]
decompress ps = do
    (i,x) <- ps
    a <- replicate i x
    pure a
```

This uses the `replicate :: Int -> a -> [a]` function from `Data.List`, which is also in the prelude.

We could have approached this as a list comprehension problem, and there's a clear relationship between list comprehension and the list monad

```
decompress ps = [a | (i,x) <- ps, a <- replicate i x]
```

But one advantage of the monadic approach is that we'll soon develop a number of skills for working with monadic definitions, and we can apply them even here. One such observation is that constructions like

```
do
    ...
    a <- expr
    pure a
```

Can be rewritten as

```
do
    ...
    expr
```

This is just a consequence of the right identity law for monads, after applying an η-reduction on `\a -> pure a`. Thus,

```
decompress ps = do
    (i,x) <- ps
    replicate i x
```

Next, we can recognize that `\(i,x) -> replicate i x` is just `uncurry replicate`, which allows us to write:

```
decompress ps = ps >>= uncurry replicate
```

or even

```
decompress = (>>= uncurry replicate)
```

Next, let's consider an earlier exercise that we solved with list comprehension, generating all of the primitive Pythagorean triples up to a particular hypotenuse:

```
primitiveTriples n =
    [ (a,b,c)
    | a <- [1..n]
    , b <- [a+1..n]
    , gcd a b == 1
    , c <- [b+1..n]
    , a^2 + b^2 == c^2
    ]
```

To translate this to `do` notation, we simply replicate each of expressions to the right of the bar as a line in the `do`, and move the expression to the left to the bottom:

```
primTriples n = do
    a <- [1..n]
    b <- [a+1..n]
    gcd a b == 1
    c <- [b+1..n]
    a^2 + b^2 == c^2
    pure (a,b,c)
```

This doesn't quite work, because the lines that correspond to tests don't have a monadic value, but rather a boolean value. We want to write a function `guard :: Bool -> [()]` that allows us to complete the translation as:

```
primTriples n = do
    a <- [1..n]
    b <- [a+1..n]
    guard $ gcd a b == 1
    c <- [b+1..n]
    guard $ a^2 + b^2 == c^2
    pure (a,b,c)
```

Writing `guard` will test our understanding of the list monad actually works. There is no explicit binding on the guarded lines, and we know that we're returning a list of `()`'s. Recalling the definition of `(>>)`, the body below the guarded line will be run once for each element of that list. Thus, we want the guarded line to have the value `[()]` if the predicate we're testing is true, and `[]` otherwise. Thus,

```
guard :: Bool -> [()]
guard True = [()]
guard False = []
```

There's a cute trick, known as the Fokker trick after the programmer who invented it, where

```
guard b = [() | b]
```

In this case, there are no generators, but the effect of the definitions is to produce a single `()` if `b` is `True`, and none otherwise. The `[]` type belongs to many other type classes, including `Alternative`, which we'll meet in a future lecture. There is a function `guard :: Alternative f => Bool -> f ()` of which our `guard` is just a type-restricted special case.

Exercise 13.2 *Consider the trivial two-element list* `[(),()]`. *Because this is an element of the list monad, we can include it on any line of list-defining* `do` *expression. Consider the two statements:*

```
do
    [(),()]
    x <- [1,2,3]
    pure x
```

vs

```
do
    x <- [1,2,3]
    [(),()]
    pure x
```

These produce very different values. Explain the difference. The first expression can be re-written, using the techniques described above, into a particularly simple form that does not involve `do`. *Do so. Optionally: if you're feeling especially brave, the second form can be re-written in the same way, albeit not quite so simply. Do so.*

Exercise 13.3 *Write out an instance definition for* `Monad Either`. *Yeah, you can look it up in the source code, but do it yourself.*

Exercise 13.4 *We saw earlier that every* `Applicative` *must be a* `Functor`, *and indeed that the link between* `Applicative`

and `Functor` *is so strong that if we can define* `pure` *and* `(<*>)` *for a type* `M` *without mentioning* `fmap`*, then we implement the* `Functor` *instance "for free" via:*

```
instance Functor M where
    fmap f x = pure f <*> x
```

or even

```
instance Functor M where
    fmap = (<*>) . pure
```

in much the same way, if we can implement `return` *and* `(>>=)` *without using* `pure` *or* `(<*>)`*, there's also a "free" instance implementation of* `Applicative` *available. Write it. Once you've done so, take a look at* `Control.Monad.ap` *and its implementation.*

Chapter 14

The IO Monad

Warmup

Let's start with a simple IO program. This is a Haskell version of a program one of Professor Kurtz's roommates encountered very late at night while working on a CDC 6700 in 1978... . Recall that a value of type `IO a` is an *IO-action*, which, when *performed*, results in the production of a value of type `a`, and this value can be captured by the `(>>=)` operator (which appears as `<-` in the commonly used, syntactically sugared `do` notation).

```
-- | The annoying "frog" program.

module Main where

import System.IO

-- | Produce a String containing n lines, each of consisting of "frog."

manyFrogs :: Int -> String
manyFrogs i = unlines $ replicate i "frog"

-- | The main act...

main :: IO ()
main = do
    putStr "How many frogs? "
    hFlush stdout
    nstr <- getLine 'How many frogs? '
    let n = read nstr
    putStr $ manyFrogs n
    main
```

You could imagine doing this with the "ninety-nine bottles of beer on the wall" song, but that would have been *really* annoying!

This is a pretty typical first-pass at a program like this. We've tried to factor out the pure part of the code, cf., `manyFrogs`, while almost every line of `main` involves an IO action.

Note the use of binding to extract information from `getLine :: IO String`, and the use of `let` to bind a variable based on a pure computation. One minor surprise is the `hFlush stdout` line, which deals with a buffering issue. For efficiency reasons, compiled Haskell code line buffers output, which means that output is send to `stdout` when a newline is added to the buffer. This is a problem if we want to read the answer to a question on the same line as the prompt. The solution, naturally enough, is to flush our output buffer. Finally, note the recursive call to `main` as the last line of `main`. This has the effect of creating an infinite loop. It's a common idiom, but there is a better way.

Of course, this code works correctly the first time we write it, but as Haskell programmers, we're not satisfied until we've worked on the code a bit.

The code fairy insists that we η-reduce `manyFrogs`, so we do:

```
manyFrogs = unlines . (`replicate` "frog")
```

We quickly realize that the sequence of actions that consist of writing out a prompt string, flushing it, and reading a response, is something that we're likely to want to do again. So we create a new IO action that captures this common interaction:

```
-- | Prompt for a line of input

prompt :: String -> IO String
prompt msg = do
    putStr msg
    hFlush stdout
    answer <- getLine
    pure answer
```

Of course, we recognize the opportunity to eliminate a binding followed by `pure`, resulting in

```
-- | Prompt for input

prompt :: String -> IO String
prompt msg = do
    putStr msg
    hFlush stdout
    getLine
```

Optionally, we might decide this is simple enough to desugar into a one-liner:

```
prompt msg = putStr msg >> hFlush stdout >> getLine
```

Note that `(>>)` doesn't actually use the value from the left-hand argument, and so it more naturally lives in the land of `Applicative`, where it is found as `(*>)`. So a purist might take this to

```
prompt msg = putStr msg *> hFlush stdout *> getLine
```

on the theory that reducing our lexicon by eliminating `(>>)` is a good thing. We're purists.

Factoring out `prompt` enables us to simplify `main`:

```
main = do
    nstr <- prompt "How many frogs? "
    let n = read nstr
    putStr (manyFrogs n)
    main
```

Next, we realize that we're saving our input `String` in the variable `nstr`, only to immediately convert it via `read` into `n`. This makes us hopeful that we can eliminate `nstr` altogether, and eliminating ephemeral variables often improves functional code. A first thought is that we can fold this into `prompt`, producing a specialized version of prompt that results in binding an `Int`. But there's a better way. `IO` is not just a `Monad`, it's also a `Functor`, and we can use `(<$>)` to "adjust" the result of an IO-action. Thus,

```
main = do
    n <- read <$> prompt "How many frogs? "
    putStr (manyFrogs n)
    main
```

Finally, there's a nice function `forever :: Applicative f => f a -> f b` that repeats `f a` forever, so we can make this:

```
main = forever $ do
    n <- read <$> prompt "How many frogs? "
    putStr (manyFrogs n)
```

At this point, we could call it done, and probably should, but what the heck... As earlier, we might look at that `n` with some skepticism. It is nothing more than an ephemeral data carrier, and so it should be possible to eliminate it. A simple intermediate step is to try to simplify the call to `putStr` so that it consists of a single variable. We can do this by moving `manyFrogs` up to the preceding line, in effect making it part of the input processing.

```
main = forever $ do
    frogs <- manyFrogs . read <$> prompt "How many frogs? "
    putStr frogs
```

Which can be immediately transformed to

```
main = forever $
    manyFrogs . read <$> prompt "How many frogs? " >>= putStr
```

This works, but it's hard to read because the `($)` and `(.)` want to be read from right-to-left, while the `(>>=)` wants to be read from left-to-right. There are several approaches to dealing with this, e.g., we could use `(=<<)`, the backward version of `(>>=)` but it's more natural to read processing pipelines from left-to-right, and so we'll pursue that approach.

First, let's tackle `<$>`. We want an operator version of `flip <$>`, and after a bit of searching, we find `(<&>)` in `Data.Functor`. Very nicely, `(<&>)` has the following fixity declaration

```
infixl 1 <&>
```

This is just right because `$` has fixity `infixr 0`, and so operates at a lower precedence than our pipeline operators, and `(>>=)` has fixity `infixl 1`, the same as `<&>`, which means they mix and match naturally. This gets us to

```
main = forever $
    prompt "How many frogs? " <&> manyFrogs . read >>= putStr
```

Which is better, but there's still that reversal of order via `(.)`. This is not a perfect world. Ideally, there'd be a flipped composition that had fixity greater than 1 lying around somewhere in the Haskell libraries, but this does not seem to be the case. Instead, there's a flipped composition `(&)` in `Data.Function`, but it has fixity `infixr 0` which means we'd need to parenthesize it

```
prompt "How many frogs? " <&> (read & manyFrogs) >>= putStr
```

or, we could remember our functor laws, and apply them in the less often used direction, we arrive at the pleasing form

```
prompt "How many frogs? "
<&> read
<&> manyFrogs >>= putStr
```

The module `Data.Function` also defines `(&)` as simple backward application, with fixity `infixr 0`, so it could be used in a processing pipeline at the cost of an extra set of parentheses using `(>>=)` and `<&>`. We can summarize this in a very simple way. If you're building a pipeline around values in a monad `m`, and you want to bolt another machine onto the pipeline, select the correct left-to-right compositional operator based on the type of that machine:

Type	Operator
`m a -> m b`	`(&)`
`a -> b`	`(<&>)`
`a -> m b`	`(>>=)`

Understand, you'll probably need to import `Data.Function` and `Data.Functor`, but this is a small price to pay.

Exercise 14.1 *Write a Haskell program* `enumerate` *which processes standard input, adding line numbers. E.g., if you have a file* `numbers.txt` *containing:*

```
one
two
three
four
five
six
seven
eight
nine
ten
```

then

```
$ ./enumerate < numbers.txt
```

produces:

```
1. one
2. two
3. three
4. four
5. five
6. six
7. seven
8. eight
9. nine
10. ten
```

Hint: look at the `lines` *and* `getContents` *functions in the* `Prelude`*.*

For extra credit, add the minimum number of spaces before each letter so that the decimal points line up, i.e.,

```
 1. one
 2. two
 3. three
 4. four
 5. five
 6. six
 7. seven
 8. eight
 9. nine
10. ten
```

Standard IO

We start by considering a simple programming task: reading a file, and converting it to upper case. We call this AOL-ification, in honor of the old pre-internet AOL community, whose internal email system was UPPERCASE ONLY. When the internet was opened up to the public, AOL became an ISP, i.e., an internet service provider. This enabled AOL users to send email to a much larger community, albeit in UPPERCASE ONLY. Annoying, but it does provide us a simple programming task.

The UNIX operating system introduced a simplified mechanism for dealing with IO, the notion of *standard IO*. This consisted of a predefined standard input, standard output, and standard error output, which were available to any command-line program. These standard inputs and outputs could be associated with files by *redirection*, and programs that processed standard input, producing standard output, were known as filters, and could be composed at the command-line level by pipes. These ideas have been borrowed by almost all subsequent operating systems.

Haskell provides a number of functions for reading from standard input, e.g., `getChar`, `getLine`, and `getContents`, which read progressively larger chunks of standard input, and `putChar` and `putStr` for output. We can write a standard IO based AOLify program very simply:

```
-- | AOLify -- read stdin, capitalize, and write to stdout

module Main where

import Data.Char (toUpper)

main :: IO ()
main = do
    input <- getContents
    let output = map toUpper input
```

```
    putStr output
```

This code seems almost too simple to be worth simplifying, but a few moments reflection suggests that we should be able to eliminate `input` and `output`, arriving at something like this:

```
main = getContents <&> map toUpper >>= putStr
```

which certainly reduces the task to its essentials. But there's an even better way. The task of writing a UNIX-style filter is common, and so the `Prelude` defines a function `interact :: (String -> String) -> IO ()` that reduces the problem of writing a UNIX-style filter to the problem of defining a (pure) function that maps input strings to output strings. Using this, we can simply write

```
main = interact $ map toUpper
```

and be done with it.

Simple IO

Of course, there's more to IO than user interaction. There are also files on disk, network connections, etc. Haskell's `Prelude` has a number of functions for dealing with common file interactions, specifically

```
type FilePath = String

readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

The use of `FilePath` doesn't provide any type safety, but it does help us understand our intentions.

As a practical illustration, we'll do a simple version of UNIX's `cat` utility. Our goal is to write a program so that a command like

```
$ ./cat foo.txt bar.txt
```

Will result in the contents of `foo.txt` and `bar.txt` being written on standard output. In this, we will only use `readFile`, but the other functions have similar use. Here's a first, naïve, solution:

```
-- | A simple version of the UNIX 'cat' program
```

```
module Main where

import System.Environment

-- | Process a list of files, writing the contents of each to standard output.

outputFiles :: [FilePath] -> IO ()
outputFiles [] = pure ()
outputFiles (f:fs) = do
    content <- readFile f
    putStr content
    outputFiles fs

main :: IO ()
main = do
    args <- getArgs
    outputFiles args
```

We use `getArgs` to extract the argument list, and then the processing of that list is done by the (recursive) `outputFiles` function. This works exactly as expected.

We'll play with this a bit, as is our practice. We can eliminate the ephemeral values `content` and `args` by introducing bindings, and then η-reducing. This gives us:

```
outputFiles (f:fs) = do
    readFile f >>= putStr
    outputFiles fs

main = getArgs >>= outputFiles
```

which is pretty simple.

But at some level, we're asking `outputFiles` to do two different things: one is to output a file given its name, the other is to process a list. It would be nice to factor this, so that

```
outputFile :: FilePath -> IO ()
outputFile path = readFile path >>= putStr
```

and then to use `outputFile`, and then to rely on standard functions to process the list. The standard `map` function almost works, except that `map outputFile :: [IO ()]`, which isn't the type we're looking for. There's a nice function `mapM :: Monad m => (a -> m b) -> [a] -> m [b]` (the actual type is slightly more general) that can be thought of as a monadic version of `map`. The following doesn't quite work:

```
main = do
```

```
        args <- getArgs
        mapM outputFile args
```

But only because the last line has type `[()]` rather than `()`. We can add `pure ()`, or use `mapM_ :: Monad m => (a -> m b) -> [a` which does the job for us. Finally, we can do our usual trick for eliminating the ephemeral variable `args`, resulting in

```
main = getArgs >>= mapM_ outputFile
```

Which is remarkably terse.

Handle Based IO

The preponderance of your IO needs can be handled with the high-level IO functions and concepts we've seen so far. This is quite different from other languages, in which IO is typically done via lower-level interfaces. Such interfaces exist for Haskell, and you should be aware of them, as they are sometimes essential.

A basic concept is that of a `Handle`, which is a Haskell type that represents a file, or file like-object (e.g., one of the Standard IO streams). There are a large number of functions for handle-based IO. Some of these are simply handle-based versions of functions we've seen before, e.g.,

```
hGetContents, hGetLine :: Handle -> IO String
hPutStr, hPutStrLn :: Handle -> String -> IO ()
```

In addition to these functions, there are constants that represent the three Standard IO streams:

```
stdin, stdout, stderr :: Handle
```

We can create a `Handle` associated with a file and a particular IO mode by using:

```
data IOMode
    = ReadMode
    | WriteMode
    | AppendMode
    | ReadWriteMode

openFile :: FilePath -> IOMode -> IO Handle
```

Once we're done with a handle, it should be closed using

```
hClose :: Handle -> IO ()
```

Closing a file will write out the buffer, and release the kernel resources associated with the file. There are typically finite limits on how many files can be opened at any one time, both on a per-process and per-machine basis. These limits used to be small, but they're now quite large. Even so, it's better to develop the right programming disciplines from the beginning. One of the nice things about Haskell, and its IO monad, is that it's possible to write functions like:

```
withFile :: FilePath -> IOMode -> (Handle -> IO r) -> IO r
```

This is a file that opens a file obtaining a handle, performs the result of applying the action function to the handle, closes the handle, and then returns the result of the action. We don't need to implement `withFile`, because it's already done in `System.IO`, but it would be easy enough to write ourselves. We'll see a similar function a bit later in the lecture.

One reason to use handle-based IO is that there are no special-purpose functions for dealing with `stderr`, although it's easy enough to just write them yourself:

```
putErrStr :: String -> IO ()
putErrStr = hPutStr stderr
```

Another reason is that you want control over the buffering choices that are being made. We've already seen `hFlush :: Handle -> IO ()` used to force out buffered output. But we'll also often want finer control of `stdin`. The `System.IO` module defines:

```
data BufferMode
     = NoBuffering
     | LineBuffering
     | BlockBuffering (Maybe Int)
```

In `NoBuffering` mode, input is made available immediate, and output is written immediately. Thus, for example, you might want `NoBuffering` if you're writing an interactive game, and want access to player input in real time. Alternatively, `LineBuffering` waits until a full line of input is available, and so makes the usual line-editing functionality available. On output, `LineBuffering` buffers until a line-feed character, whereas `BlockBuffering` allows for larger buffers, and so offers greater performance, but isn't suitable for interactive use.

Buffering is manipulated using the functions:

```
hGetBuffering :: Handle -> IO Buffering
hSetBuffering :: Handle -> BufferMode -> IO ()
```

It is often convenient to have ”stack-oriented” functions to manage resources, and we can think of the buffering mode as a kind of resource that we want to acquire and release. To that end,

```
withBuffering :: Handle -> BufferMode -> IO a -> IO a
withBuffering handle mode action = do
    savedMode <- hGetBuffering handle
    hSetBuffering handle mode
    result <- action
    hSetBuffering handle savedMode
    pure result
```

is often very useful. This can be especially important when using the interpreter to debug your code. It's very frustrating to exit the program you're running, only to find that the interpreter is in an unexpected buffering mode.

Another reason for dealing with handle-based IO is to deal with a feature of Haskell that is sometimes a bug. Some of Haskell's IO functions, notably `getContents`, `hGetContents`, `readFile` are implicitly *lazy*, i.e., they return immediately with a `String`, but the `String` is built and IO performed on an as-used basis. This can be really great: often very naïve programs will be able to process huge files without using much memory. But it can be a problem in that IO is not as atomic as might be wished. This might be an issue, e.g., if you read a preferences file, and write it back out. If the writing is happening concurrently with the reading, the file can easily get corrupted, resulting in hard to diagnose crashes. Lower-level routines (even if that only means using `getLine` and friends can avoid this problem.

Chapter 15

Case Study: The Animal Game

In our lectures, we often focus on programming ideas and techniques, but not on programs *per se*. It's sometimes nice to see the programming ideas applied in more substantial examples than we can work out in class. In this supplemental lecture, we develop a complete program for the Animal Game.

The Game

The animal game is a two player game, in which the answering player thinks of an animal, and the questioning player asks a series of yes/no questions about the animal that the questioning player is thinking of. At some point, the questioning player believes they have enough information to make a guess. If the guess is correct, the questioning player wins; if the guess is incorrect the answering player wins.

The animal game is a standard example in computer science. We write a program that plays the questioning player against a human answering player. What makes this interesting is that the questioning player *learns* through repeated plays of the game, and so becomes progressively more difficult for the answering player to beat.

Data Representation

We will represent the questioner's knowledge base of the animal kingdom by a binary search tree, in which the internal nodes contain a yes/no question, and two subtrees corresponding to yes and no answers to that question, and leaf nodes that correspond to guesses, i.e., animals.

```
data YesNoTree
    = Question
        { query :: String
        , yes, no :: YesNoTree
        }
    | Answer
```

```
            { final :: String }
        deriving (Show, Read)
```

One of the trickier aspects to this program is that we need to be able to create a mutated tree that incorporates new knowledge into an existing `YesNoTree`. To that end, we introduce an auxiliary data structure, a `YesNoView`, which is an instance of Gerárd Huet's zipper design pattern to the simple case of a binary tree. (You should be able to download the paper if you're browsing from the `uchicago.edu` domain, but be aware that it's written in ML rather than Haskell.)

```
data YesNo = Yes | No

data YesNoView = YesNoView
    { current :: YesNoTree
    , context :: [(YesNo,String,YesNoTree)]
    }
```

The idea is that a `YesNoView` encodes both a current node of the tree, and the information needed to rebuild the tree from that node. Each of the `(YesNo,String,YesNoTree)` triples in the context indicated the direction in which we descended into the tree, i.e., by the 'yes' alternative, or the 'no' alternative, the query at that node, and the other child.

We navigate through such a tree in one of three directions: `up`, to our parent; `downYes`, to our 'yes' child; and `downNo`, to our 'no' child.

```
downYes :: YesNoView -> YesNoView
downYes (YesNoView (Question query_ yes_ no_) ctx) =
    YesNoView yes_ ((Yes,query_,no_) : ctx)
downYes _ =
    error "call to downYes on an answer node of a YesNoTree."

downNo :: YesNoView -> YesNoView
downNo (YesNoView (Question query_ yes_ no_) ctx) =
    YesNoView no_ ((No,query_,yes_) : ctx)
downNo _ =
    error "call to downNo on an answer node of a YesNoTree."

up :: YesNoView -> YesNoView
up (YesNoView focus ((yn,query_,saved) : ctx)) = case yn of
    Yes -> YesNoView (Question query_ focus saved) ctx
    No  -> YesNoView (Question query_ saved focus) ctx
up _ =
    error "call to up on an YesNoView with empty context."
```

Note that we "mutate" the tree by replacing the `current` field of a `YesNoView` with another value.

We have `enter :: YesNoTree -> YesNoView`, which allow us to initialize a `YesNoView` from a `YesNoTree`, with the root as the current node,

```
enter :: YesNoTree -> YesNoView
enter tree = YesNoView tree []
```

and `exit :: YesNoView -> YesNoTree`, which returns the full tree from a view:

```
exit :: YesNoView -> YesNoTree
exit (YesNoView tree []) = tree
exit ynv = exit (up ynv)
```

IO Interaction

We've already seen a couple of functions that will help us manage IO interaction, `withBuffering :: Handle -> BufferMode -> I` which enables us to perform an IO action within within a context that sets a buffering mode on a particular `Handle`, and `prompt :: String -> IO String`.

```
withBuffering :: Handle -> BufferMode -> IO a -> IO a
withBuffering handle mode action = do
    savedMode <- hGetBuffering handle
    hSetBuffering handle mode
    result <- action
    hSetBuffering handle savedMode
    pure result

prompt :: String -> IO String
prompt msg = do
    putStr msg
    hFlush stdout
    withBuffering stdin LineBuffering getLine
```

Note that this version of the `prompt` command ensures that the inner `getLine` call occurs in a context where the buffering mode is set to `LineBuffering`.

We're also going to be asking a lot of questions that have yes/no answers, indeed, this is going to be the dominant mode of user interaction. As such, we want it to be as frictionless as possible. This is going to result in a different buffering approach. Once we prompt for an answer, we'll read a single character in `NoBuffering` mode, so that it is immediately available to us. If we get an answer we can't interpret, we'll re-prompt.

```
yesNo :: String -> IO YesNo
```

```
yesNo msg = do
    putStr $ msg ++ " [y,n] "
    hFlush stdout
    answer <- withBuffering stdin NoBuffering getChar
    when (answer /= '\n') $ putChar '\n'
    case toLower answer of
        'y' -> pure Yes
        'n' -> pure No
        '\n' -> yesNo msg
        _   -> do
            putStrLn 'Please answer y for yes, or n for no.'
            yesNo msg
```

This kind of code is often quite delicate, and it's worth understanding why each line is present. Note in particular that because we get the character immediately, there's been no line feed (unless, of course, the character is itself the newline character), so we have to supply this ourselves.

Game Mechanics

A basic building block is the `playOneGame :: YesNoView -> IO YesNoView` function, which as its name and type suggests, represent the play of a single instance of the animal game, returning a possibly updated game tree.

```
playOneGame :: YesNoView -> IO YesNoView
playOneGame ynv = case current ynv of
    question@Question {} -> yesNo (query question) >>= \case
        Yes -> playOneGame (downYes ynv)
        No  -> playOneGame (downNo ynv)
    answer -> yesNo ('Is your animal ' ++ final answer ++ '?') >>= \case
        Yes -> do
            putStrLn 'You lose.'
            pure ynv
        No  ->  do
            putStrLn 'You win!'
            newAnimal <- prompt 'Your animal is > '
            newQuestion <- prompt $
                'Please state a question that is true of '
                ++ newAnimal
                ++ ' but false of ' ++ final answer ++ ' > '
            let newNode =
                    Question newQuestion (Answer newAnimal) answer
            pure $ ynv { current = newNode }
```

One reason for this choice of type signature is that it enables `playOneGame` to call itself recursively, which it does

as at each `Question` node. A common pattern in this code is to extract a value from `IO`, and then do an immediate pattern match on that variable. Using just standard Haskell syntax, we'd have to write a lot of code like

```
ans <- yesNoQuestion "Yes or no?"
case ans of
    Yes -> ...
    No -> ...
```

or

```
yesNoQuestion "Yes or no?" >>= \ans -> case ans of
    Yes -> ...
    No -> ...
```

Either way, we had to introduce the variable `ans`, and we can't get rid of it. By adding the declaration

```
{-# LANGUAGE LambdaCase #-}
```

at the top of the module, we introduce the `LambdaCase` extension, which allows us to write:

```
yesNoQuestion "Yes or no?" >>= \case
    Yes -> ...
    No -> ...
```

It's a small thing, but we do so much of it in this program that it seems worthwhile to do it well.

To play multiple games, we have `playGames :: YesNoTree -> IO YesNoTree`. Note that this function traffics in `YesNoTree`, not `YesNoView`.

```
playGames :: YesNoTree -> IO YesNoTree
playGames start =  do
    restart <- exit <$> playOneGame (enter start)
    yesNo "Would you like to play again?" >>= \case
        Yes -> playGames restart
        No  -> pure restart
```

Note that `playGames` essentially implements a "bottom-test loop," which is an iterative programming construct that is guaranteed to run its body once.

The swizzling back and forth between `YesNoTree` and `YesNoView` may seem wasteful, but it's important to keep in mind that `enter . exit` is *not* the identity on `YesNoTree`, as it resets the current node to the root. Stated metaphorically, it leaves us with the zipper zipped-up.

Serialization

While a program that learns is interesting, we can do this one better by *persisting* in our learning. Thus, if you play the animal game, quit, and then come back to it later, the knowledge it gained in the first play is still available. The game get harder.

To that end, we use a simple serialization strategy. We use the file `.animals` in the user's home directory to store a text representation of the `YesNoTree`. Using default `Read` and `Show` instances makes this very easy.

```
gameDBPath :: IO FilePath
gameDBPath = do
    home <- getHomeDirectory
    pure $ home </> ".animals"


loadGameDB :: IO YesNoTree
loadGameDB = do
    dbPath <- gameDBPath
    doesFileExist dbPath >>= \case
        True  -> readMaybe <$> S.readFile dbPath >>= \case
            Just db -> pure db
            Nothing -> do
                putStrLn 'Stored database corrupt, using default.'
                pure startTree
        False -> pure startTree


saveGameDB :: YesNoTree -> IO ()
saveGameDB tree = do
    dbPath <- gameDBPath
    writeFile dbPath (show tree)
```

One bit of trickiness here is that Haskell's `readFile` function is lazy. We use the `readFile` function from the module `System.IO.Strict`. This module isn't a part of the Haskell Platform distribution, and has to be installed using the cabal tool:

```
$ cabal install strict
```

One further nuance is that we use `readMaybe` rather than `read`. This function (which is found in `Text.Read`) indicates a parsing failure by returning `Nothing`, allowing for more graceful error handling.

Exceptions, and Main

We could finish this code by

```
main :: IO ()
main = loadGameDB >>= playGames >>= saveGameDB
```

a particularly simple and pleasing end to the program, but it's possible for the user to cause an ugly crash, by typing `^-D` in response to `prompt`. This causes `getLine` to throw an exception. We won't cover exceptions in this course, but there are a couple of things to know: (1) exceptions are caught in the IO monad, but an of a number of functions, including `catch`, `handle`, and `try`; (2) these functions rely on an `Exception` type class, and their use requires somehow specifying a specific type that implements this type class. Oddly enough, this is often the hard part of the exercise. We use

```
catchIO :: IO a -> (IOException -> IO a) -> IO a
catchIO = catch
```

a type-restricted version of `catch`, which catches the `IOException` type.

```
main :: IO ()
main = do
    putStrLn "Welcome to the Animal Game!\n"
    catchIO (loadGameDB >>= playGames >>= saveGameDB) $ \_ ->
        putStrLn "\nIO exception occurred, database not saved."
    putStrLn "Goodbye."
```

- The Animal Program (Haskell source)

Chapter 16

Monoids

(Chapter contributed by RC)

A semigroup is a pair $(S, op : S \times S \to S)$, where the binary operator op is associative.

A monoid is triple $(S, op : S \times S \to S, id \in S)$, where the binary operator op is associative and where id serves as left and right identity operands. We will see three Haskell typeclasses — `Monoid`, `Alternative`, and `MonadPlus` — that encapsulate this notion.

Semigroup and Monoid

We have seen how the `foldr` function provides a general way to "crunch" a list of values down to a single result. A few simple examples:

```
> foldr (+) 0 [1,2,3,4,5]
15
> foldr (*) 1 [1,2,3,4,5]
120
> foldr (:) [] [1,2,3,4,5]
[1,2,3,4,5]
> foldr (++) [] [[1,1],[2,2],[3,3],[4,4],[5,5]]
[1,1,2,2,3,3,4,4,5,5]
> foldr (&&) True [True,True,True,False]
False
> foldr (||) False [True,True,True,False]
True
> foldr (||) True [True,True,True,False]
True
```

We will now consider Haskell typeclasses that describe values that can result from this crunching process. We will start with the latter. In a subsequent lecture (Chapter 17), we will describe types, beyond just lists, that can be crunched down to a single result.

The Semigroup typeclass (defined in Data.Semigroup) consists of types that have an associative binary operation,

```
class Semigroup s where
    (<>)           :: s -> s -> s

    sconcat        :: [s] -> s
    sconcat []      =  undefined
    sconcat [x]     =  x
    sconcat (x:xs) =  x <> sconcat xs
```

where (<>) is the associative binary operator. Note that (<>) is minimal complete for Semigroup, and that sconcat, while formally a part of the Semigroup class, isn't declared in the Prelude, but in Data.Semigroup.

The Monoid typeclass (defined in Data.Monoid) consists of types that have an associative binary operation with identity,

```
class Semigroup m => Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mappend = (<>)

    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

where mempty is the identity, and mappend is the associative binary operator. An instance of Monoid must define mempty; mappend defaults to (<>). The reason that mappend appears at all Monoid is historical; in pre-8.4 versions of Haskell, Semigroup was not a superclass of Monoid. When the change was made, the mappend was kept around for backward compatibility with existing code. (Just in case you come across some older code examples, back then, (<>) = mappend was defined as an infix operator in Data.Monoid.) Note that mempty is mimimal complete for Monoid.

Notice that because the member signatures refer to the variable m in places where values are required (and, furthermore, because only ground types are inhabited by values), the kind of m is *. That is, only ground types (as opposed to type operators) can be Monoids.

The names mempty and sappend/mappend work well for the List instance, which we will see below, but not as well for many other Monoids whose identities and operators have little to with "emptiness" or "appending." Nevertheless, these are the names and we will learn to live with them (or, in the case of the latter, forgo in favor of (<>)).

The monoid laws can be expressed as, for all values a, b, and c of a monoid m,

- `a <> (b <> c) == (a <> b) <> c` (associativity),
- `a <> mempty == a` (right identity), and
- `mempty <> a == a` (left identity).

Exercise 16.1 *Among the seven example calls to* `foldr` *above, which binary operators and identities (the first and second arguments to* `foldr`*, respectively) do* not *constitute monoids, according to the definitions and laws just discussed?*

List

The List instance is straightforward and illustrates why the `Monoid` methods were so named:

```
instance Semigroup [a] where
    (<>) = (++)

instance Monoid [a] where
    mempty = []
 -- mappend = (++)
```

Notice how, based on the surrounding context, Haskell infers what the types of `mempty` and `(<>)` should be and retrieves the implementations from the List instance appropriately:

```
> [1,2] <> mempty <> [3,4,5] <> [6]
[1,2,3,4,5,6]
> foldr mappend mempty [[1,2],[],[3,4,5],[6]]
[1,2,3,4,5,6]
> mconcat [[1,2],[],[3,4,5],[6]]
[1,2,3,4,5,6]
```

Sum and Product

There are two useful ways of defining a monoid on numbers: `(+)` paired with the identity element `0` and `(*)` paired with the identity element `1`. However, if we were to define an instance of the form

```
instance Num a => Semigroup (Num a)
    ...

instance Num a => Monoid (Num a)
    ...
```

we could represent only one of these two monoids. To get around this obstacle, `Data.Monoid` defines two wrapper types for numbers, called `Sum` and `Product`, which capture the different monoids on numbers:

```
newtype Sum a = Sum { getSum :: a } deriving (...)

instance Num a => Semigroup (Sum a) where
    Sum x <> Sum y          = Sum (x + y)

instance Num a => Monoid (Sum a) where
    mempty                  = Sum 0

newtype Product a = Product { getProduct :: a } deriving (...)

instance Num a => Semigroup (Product a) where
    Product x <> Product y  = Product (x * y)

instance Num a => Monoid (Product a) where
    mempty                  = Product 1
```

We can now choose between the two `Monoid`s by explicitly wrapping and unwrapping numbers (without any additional run-time overhead, due to the use of `newtype` in the type definitions):

```
> getSum . mconcat . map Sum $ [1..6]
21
> getProduct . mconcat . map Product $ [1..6]
720
```

Any and All

Similarly to numbers, there are two `Monoid`s on booleans, which are defined by way of two wrapper types:

```
newtype All = All { getAll :: Bool } deriving (...)

instance Semigroup All where
    All x <> All y  = All (x && y)

instance Monoid All where
    mempty          = All True

newtype Any = Any { getAny :: Bool } deriving (...)

instance Semigroup Any where
    Any x <> Any y  = Any (x || y)
```

```
instance Monoid Any where
    mempty          = Any False
```

The following examples exhibit the same functionality as calling the `all id` and `any id` functions from the `Prelude`:

```
> getAll . mconcat . map All $ []
True
> getAny . mconcat . map Any $ []
False
> getAll . mconcat . map All $ [True,True,True,False]
False
> getAny . mconcat . map Any $ [True,True,True,False]
True
```

Maybe, First, and Last

There are many other useful `Monoids`, besides the common ones on lists, numbers, and booleans above. For example, the following derived instance "lifts" all `Monoids` over `a` to `Monoids` over `Maybe a`:

```
instance Semigroup a => Semigroup (Maybe a) where
    Nothing <> m        =  m
    m       <> Nothing  =  m
    Just m1 <> Just m2  =  Just (m1 <> m2)

instance Monoid a => Monoid (Maybe a) where
    mempty              =  Nothing
```

Furthermore, it is often useful to work with alternative versions of `Maybe` `Monoids`, namely, where the binary operator returns the first and last non-`Nothing` values, if any, in a list of `Maybe` values. The `Monoid` instance for `Maybe` is already "taken," so the `First` and `Last` wrapper types are defined to provide each of these two choices, and can be used as follows:

```
> getFirst . mconcat . map First $ [Just 1, Nothing, Just 3]
Just 1
> getLast . mconcat . map Last $ [Just 1, Nothing, Just 3]
Just 3
```

Notice that these two instances make sense even when the underlying type is not a `Monoid`

Exercise 16.2 *Without first peeking at the implementations in* `Data.Monoid`*, fill in the definitions below.*

```
newtype First a = First { getFirst :: Maybe a } deriving (Show)
newtype Last a  = Last  { getLast  :: Maybe a } deriving (Show)

instance Semigroup (First a) where
    ...

instance Monoid (First a) where
    ...

instance Semigroup (Last a) where
    ...

instance Monoid (Last a) where
    ...
```

Exercise 16.3 *Note that without constraints on the underlying types* `s` *and* `t`, `Either s t`, *unlike* `Maybe a`, *is* not *an element of the* `Monoid` *typeclass. Explain the obstruction.*

Endo

And one more `Monoid` for now, endomorphisms under composition:

```
newtype Endo a = Endo { appEndo :: a -> a }

instance Semigroup (Endo a) where
    Endo f <> Endo g = Endo (f . g)

instance Monoid (Endo a) where
    mempty           = Endo id
```

Alternative

The `Monoid` typeclass is useful but describes only ground types. What about type operators that also exhibit monoidal structure? For example, recall that we defined a wrapper type for `Maybe` called `First` that constituted a `Monoid` with an `mappend` operator that returns the first (leftmost) non-`Nothing` value.

```
> getFirst $ First (Just 1) <> First (Just 2)
Just 1
> getFirst $ First (Just 1) <> First Nothing
```

```
Just 1
> getFirst $ First Nothing <> First (Just 2)
Just 2
```

We might like to describe this monoidal structure directly for `Maybe` even though it is not a ground type. The `Alternative` class describes `Applicative` functors `f` that also exhibit monoidal structure.

```
class Applicative f => Alternative f where
    empty :: f a
    (<|>) :: f a -> f a -> f a
```

Notice the correspondence between `empty` and `(<>)`| in `Applicative` to `mempty` and `(<>)` in `Monoid`, respectively. Unsurprisingly, an `Alternative` type `f` ought to satisfy all of the same laws as `Monoids`... in addition to the ones for `Applicative`... in addition to the ones for `Functor`!

The `Maybe` instance for `Applicative` is quite like the `First a` instance of `Monoid`:

```
instance Alternative Maybe where
    empty         = Nothing
    Nothing <|> r = r
    l       <|> _ = l
```

And now we can write the previous examples directly in terms of the `Maybe` monoid (`Alternative`):

```
> Just 1 <|> Just 2
Just 1
> Just 1 <|> Nothing
Just 1
> Nothing <|> Just 2
Just 2
```

There are many other applicative functors with monoidal structure. For example, `Alternative` types are quite handy when writing parsers, as we shall see later in the course.

There's a handy library function in `Control.Monad` that generalizes the `guardMaybe` function we saw several lectures ago:

```
guard :: Alternative f => Bool -> f ()
guard True = pure ()
guard False = empty
```

Exercise 16.4 *The function* `mconcat :: Monoid a => [a] -> a` *combines a list of* `Monoid`*s. For example:*

```
> getFirst . mconcat . map First $ [Just 1, Just 2]
Just 1
> getFirst . mconcat . map First $ [Just 1, Nothing]
Just 1
> getFirst . mconcat . map First $ [Nothing, Just 2]
Just 2
```

Implement a function

```
altconcat :: Alternative f => [f a] -> f a
```

that combines a list of `Alternative`*s. Once defined, you will be able to use* `altconcat` *as follows:*

```
> altconcat [Just 1, Just 2]
Just 1
> altconcat [Just 1, Nothing]
Just 1
> altconcat [Nothing, Just 2]
Just 2
```

MonadPlus

There are some `Monad` types that are also `Alternatives`, a combination captured in the `MonadPlus` typeclass:

```
class (Alternative m, Monad m) => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

which the documentation describes as "Monads that also support choice and failure," but which we might prefer to think of as a monoidal monad. Indeed, notice that the class constraints require that a `MonadPlus` be a `Monad` plus an `Alternative` (recall that the `Alternative` type class describes single-argument type constructors that are monoids, as opposed to `Monoid` which describes ground types that are monoids).

As the documentation states, the minimal completion definition for `MonadPlus` is nothing! Indeed, the default definitions say it all:

```
    mzero = empty
```

```
    mplus = (<|>)
```

So, to make our friend `Maybe` an instance of `MonadPlus`, the instance declaration is simpler than trivial, it's empty:

```
instance MonadPlus Maybe
```

Notice the lack of the `where` keyword.

This is the first time we've seen a type class that doesn't require *any* methods to be implemented. So, what's the point? Well, one explanation is that writing both constraints `(Alternative m, Monad m)` could get tedious after a while, and so defining the `MonadPlus` class serves as a useful shorthand. This isn't a bad idea, but it turns out the real explanation is rather more incidental. Remember the history of the standard libary pre-7.10? Because `Applicative` was not a superclass of `Monad`, the constraints `(Alternative m, Monad m)` would not have been sufficient a "monad-plus-monoid." Instead, the constraints needed to be `(Applicative m, Alternative m, Monad m)`, which quickly becomes unwieldy. Instead, `MonadPlus` was defined with the single class constraint `Monad`, disconnected from the `Applicative`/`Alternative` geneology. With the restructuring of `Monad` in 7.10, `(Alternative m, Monad m)` suffices to describe a "monad-plus-monoid" and so the definitions come for free.

Recap

`Monoid` and `Alternative` both describe monoids, the former for types of kind `*` and the latter for those types of kind `* -> *`. Note that the latter is also defined to be a subclass of `Applicative`, because the combination of these two classes is often useful. `MonadPlus` simply combines `Monad` and `Alternative` and is a relic of older versions of the language.

```
                        Semigroup m
                         |  (<>) :: m -> m -> m
                         |
Functor f               Monoid m
 |  fmap                    mempty :: m
 |                          mappend = (<>) :: m -> m -> m
 |
Applicative f --------- Alternative f
 |  (<*>)                |  empty :: f a
 |  pure                 |  (<|>) :: f a -> f a -> f a
 |                       |
Monad f --------------- MonadPlus f
    (>>=)                   mzero = empty :: f a
    return = pure           mplus = (<|>) :: f a -> f a -> f a
```

Chapter 17

Foldable

(Chapter with contributions from RC)

We've been using the `foldr` function to crunch lists, as if its type was

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

This is great, but there are other collection classes that order their elements, and so could be processed in conceptually the same way. To that end, we have the `Foldable` type:

```
class Foldable t where
    foldr   :: (a -> b -> b) -> b -> t a -> b
    ...
```

which is where `foldr` actually lives. Let's do a simple example another type that orders its elements, the `BinaryTree` type we've seen before:

```
data BinaryTree a
    = Empty
    | Node (BinaryTree a) a (BinaryTree a)
    deriving Show
```

We can write `foldr` for this type:

```
instance Foldable BinaryTree where
    foldr f acc Empty = acc
    foldr f acc (Node left a right) =
```

```
            foldr f (f a (foldr f acc right)) left
```

The equation for `foldr f acc (Node left a right)` takes a little thinking, but this form is easy to derive if you understand that `foldr` builds its result value from right-to-left.

The `Foldable` type class defines a number of other useful functions, e.g.,

```
class Foldable t where
    foldr  :: (a -> b -> b) -> b -> t a -> b
    foldr1 :: (a -> a -> a) -> t a -> a
    foldl  :: (b -> a -> b) -> b -> t a -> b
    foldl1 :: (a -> a -> a) -> t a -> a
    fold   :: Monoid m => t m -> m
    foldMap :: Monoid m => (a -> m) -> t a -> m
    toList :: t a -> [a]
    ...
```

The `foldl` function does a left-to-right crunching of the list, and is often useful in processing very large data structures that are built lazily. The variants `foldr1` and `foldl1` deal with simple, monoid-like folds over non-empty lists (obviating the need for an explicit identity).

The `Foldable` type class includes many other functions we've been thinking of as list-specific, e.g., `length`, `elem`, `minimum`, `maximum`, `sum`, and `product`. A moment's reflection will reveal that each can easily be written in terms of `foldr`, and so their greater generality should not be a surprise.

The `fold` and `foldMap` functions deal with the case where the combining function has type `m -> m -> m` for some `Monoid m`. Default definitions exist so that a minimal complete definition of a `Foldable` instance must define *either* `foldr` or `foldMap`. What is perhaps even more surprising is that it is often easier to write the instance definition using `foldMap`, e.g.

```
instance Foldable BinaryTree where
    foldMap _ Empty = mempty
    foldMap f (Node left a right) =
        foldMap f left <> f a <> foldMap f right
```

This is very natural: we process the recursive parts recursively, and combine the pieces using `(<>)`.

What may be surprising in all of this is that functions like `foldr` and `foldl` may have different execution characteristics, but because `foldr` is complete for `Foldable`, it is somehow possible to write `foldl` in terms of `foldr`. How can this be? It is useful to understand the definition of `foldMap` in terms of `foldr`, and conversely.

To get `foldMap` from `foldr` is relatively straightforward:

```
foldMap f container = foldr (\a b -> f a <> b) mempty container
```

Exercise 17.1 *The code fairy would like you to η-reduce the definition above. She doesn't want you to eliminate* `f`, *but she thinks you should be able to eliminate* `a`, `b`, *and* `container` *without difficulty.*

What is perhaps surprising is the other direction. The `foldMap` function requires an argument whose domain is a `Monoid`, but there's no such restriction on the combining function of `foldr`. Obviously, there's some sort of trick. A crucial observation is that if `f` is the combining function for `foldr`, then providing a single argument to `f` results in a function of type `b -> b`, and this may remind you of the `Endo` type:

```
newtype Endo a = Endo { appEndo :: a -> a }
```

The type `Endo a` is a monoid instance, where `(<>)` is just composition under the wrapper, and `mempty` is `Endo id`. Thus,

```
foldr f acc container = appEndo (foldMap (\a -> Endo (f a)) container) acc
```

i.e., we use `foldMap` to build a function, which, when applied to `acc`, reduces to `foldr f acc container`.

Exercise 17.2 *It's the code fairy again. Do I need to tell you what to do?*

This may (should?) remind you of the difference list approach to implementing the `Doc` type from the Scalability supplementary lecture (Chapter 23).

Chapter 18

Traversable

The `Traverable` type class includes types that we can ”walk across.” It is a simultaneous generalization of `Functor` and `Foldable`, and as we'll see, it's especially natural and easy to implement when the underlying type is also an `Applicative` instance.

Let's begin with a reconsideration of the `Applicative` type class, which we sometimes think of as a generalization of the `Functor` type class that enables us to apply pure functions to zero or more fancy values, where what ”apply” means depends on particular `Applicative` type. It is useful to think of `Applicative` types as defining a kind of *idiom*.

Consider, for example, the function `mapM :: Monad m => (a -> m b) -> [a] -> m [b]`. We can implement this function by ”walking along the list,” i.e., via a natural structural induction, as follows:

```
mapM f [] = pure []
mapM f (a:as) = pure (:) <*> f a <*> mapM as
```

McBride and Patterson, the authors of the original paper on `Applicative`, invented the idea of ”idiom brackets,” where

```
⟦ f a_1 ... a_n ⟧
```

represents

```
pure f <*> a_1 <*> ... <*> a_n
```

If we rewrite the defining of `mapM` using idiom brackets, we have

```
mapM f [] = [[ [] ]]
```

```
mapM f (a:as) = [[ (:) (f a) (mapM f as) ]]
```

Contrast this, for a moment, with the definition of map:

```
map f [] = []
map f (a:as) = f a : map f as
```

If we rewrite the last line in prefix notation, we get

```
map f [] = []
map f (a:as) = (:) (f a) (map f as)
```

Thus, the definition of mapM is just the definition of map, but embellished with idiom brackets on the right-hand side. Another way of describing this is that mapM is an effectful version of map, or alternatively, that map is a pure version of mapM.

This may seem like abstract nonsense, but it makes an interesting point: mapM has the wrong constraint! We didn't need Monad, only Applicative. We'll see in a bit why this hasn't been fixed.

Our next thought about generalizing mapM is that we might be able to replace the [] type with a less restrictive type class. This leads us to

```
class (Functor t, Foldable t) => Traversable t where
    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```

Note that traverse is mapM, but with just the right type constraints: we've moved from Monad to Applicative as the constraint on the type of f, and we're generalizing [] to t. In fact, the implementation of mapM in the Haskell sources is just mapM = traverse!

But the curious thing is that creating instances of Traversable follows the pattern we established above for mapM. We just "implement fmap," and then put the left hand side of the definitions in idiom brackets, i.e., add pure and (<*>) as needed.

Let's do an extended example, using the BinaryTree type as before. As we'll see, this involves an interesting and unexpected difficulty.

We have the Foldable instance above, and we've done Functor before:

```
instance Functor BinaryTree where
    fmap _ Empty = Empty
    fmap f (Node left a right) =
        Node (fmap f left) (f a) (fmap f right)
```

Let's remember that definition!

The `traverse` function is just an effectful `fmap`, intuitively,

```
instance Traversable BinaryTree where
    traverse _ Empty = [[ Empty ]]
    traverse f (Node left a right) =
        [[ Node (traverse f left) (f a) (traverse f right) ]]
```

I.e., just the definition of `fmap` for this type, but with idiom brackets on the left. Of course, idiom brackets aren't part of Haskell syntax, so we have to translate them out:

```
instance Traversable BinaryTree where
    traverse _ Empty = pure Empty
    traverse f (Node left a right) =
         Node <$> traverse f left <*> f a <*> traverse f right
```

Of course, it's one thing to have this toy, and another to know how to play with it. Let's consider a simple problem: labelling the items in a container with an integer. To that end, we'll use a little gadget in `State`:[1]

```
label :: a -> State Int (Int,a)
label a = do
    ix <- get
    modify (+1)
    pure (ix,a)
```

We can then add labels to the values held in a `Traversable` container:

```
addLabels :: (Traversable t) => t a -> t (Int,a)
addLabels ta = evalState (traverse label ta) 1
```

With this:

```
> testTree
Node (Node Empty
           1
           (Node Empty 2 Empty))
      3
      (Node Empty
            4
```

[1]RC: This forward reference to `State` (Chapter 20) is inessential. Skim the output below and proceed to the subsequent exercise, in which this "throwaway" use of `State` will be thrown away.

```
                (Node Empty 5 Empty))
> addLabels testTree
Node (Node Empty
           (1,1)
           (Node Empty (2,2) Empty))
     (3,3)
     (Node Empty
           (4,4)
           (Node Empty (5,5) Empty))
> toList (addLabels testTree)
[(1,1),(2,2),(3,3),(4,4),(5,5)]
```

exactly as you might expect.

Exercise 18.1 *The throwaway use of* `State` *here is a bit heavy-handed.* `Data.Traversable` *includes two functions,* `mapAccumL` *and* `mapAccumR` *that perform state-based traversals, albeit with an explicitly exposed state. Rewrite* `addLabels` *using one of the* `mapAccum{L,R}` *functions, rather than* `State`.

Exercise 18.2 *Let's consider a different tree-based container class:*

```
data MultiTree a
    = Node [MultiTree a]
    | Leaf a
```

This describes a tree, in which the values are contained in `Leaf` *values, and the* `Node` *values can contain an arbitrary number of sub-trees.*

Implement the usual type classes for `MultiTree`*:* `Eq`, `Ord`, `Functor`, `Applicative`, `Monad`, `Monoid` *(for* `MultiTree a`*),* `Alternative`, `MonadPlus`, `Foldable`, `Traversable`, *at suitable kinds and with suitable constraints.*

If you do this artfully, you'll see sub-expressions like `fmap . fmap` *in the definition of* `fmap`, *and* `traverse . traverse` *in the definition of* `traverse`. *These happen because* `[]` *is a* `Functor` *and a* `Traversable`, *and both* `Functor` *and a* `Traversable` *are closed under composition.*

The `Traversable` type class is also the homes of another familiar function, `sequence`, which until recently had the type:

```
sequence :: Monad m => [m a] -> m [a]
```

In `Traversable`, the specific use of lists is revised to account for any `Traversable` type `t`:

```
sequence :: Monad m => t (m a) -> m (t a)
```

but here, as in the case of `mapM`, the implementation doesn't require the full power of a `Monad` constraint, and so we also have:

```
sequenceA :: Applicative f => t (f a) -> f (t a)
```

with

```
sequence = sequenceA
```

being merely a less generally typed version of `sequenceA`.

A Bit of Wisdom

There is a programming truism, evidently due to Tony Hoare but often attributed to Donald Knuth that's well worth knowing: *Premature optimization is the root of all evil.* There is considerable wisdom in this, but also a history of it being applied foolishly.

For functional programmers, eschewing premature optimization often involves a natural preference for lists as an all-purpose container class, even when they're not entirely appropriate. Taking efficiency into consideration during the design phase of a program is *timely*, not premature, optimization, even if lower-level optimization is not. But we can usually get away with what otherwise would be a poor choice: by writing list-based code on a first pass, and then generalizing it to be type class dependent rather than specifically list dependent, gradually decoupling our program's code from its data representation choices, and so making it possible to revisit those choices late in program development, at a point when it would be positively painful in traditional languages.

The `traverse` function is a good example of this. Many a `traverse` has begun life as a `mapM` where the underlying monad is `IO`. There's a recurring pattern here: you build a list of values, and then you want to do some IO once for each value on the list. If the IO action is simple, there's no big deal: you use `mapM`, or `traverse` if you want to placate the code fairy and/or facilitate future changes to data organization. But if you want to do anything complicated, you pretty much have to define and name a function to pass as a first argument to either. There is a better way.

The `Data.Traversable` module has a variation of `traverse` called `for`, which takes its arguments in reversed order. This allows you to write code that looks like this:

```
for listOfValues $ \value -> do
    ...
```

This is natural enough that it soon becomes a programming idiom for processing the values in a container.

For example, consider the "Collecting Permutations" problem from Lab 4. If we wanted to tackle this with more advanced data structures, we might end up using `Data.Map` and `Data.Set` to build a map from sorted keys to the set of words that generate that sort via:

```
makePermutationMap :: [String] -> Map String (Set String)
makePermutationMap = Map.unionsWith Set.union . map makeEntry where
    makeEntry word = Map.singleton (sort word) (Set.singleton word)
```

We could then build main as follows:

```
main :: IO (Map String ())
main = do
    permMap <- makePermutationMap .  getWords <$> getContents
    for permMap $ \perms -> when (Set.size perms > 1) $ do
        putStrLn . intercalate ', ' . Set.toList $ perms
```

In this case, the `for` iterates over the values of the `Map`, which is what this problem requires.

Of course, as our programmer's eye develops, there's a tendency to view the call to `Set.toList` with a bit of skepticism. Why do we need to convert from one container class (`Set`) to another (`[]`)? In this case, it's because `intercalate` only works with lists. But we easily implement `intercalate` in terms of `foldr1`, and then take advantage of the fact that `foldr1 :: Foldable t => (a -> a -> a) -> t a -> a` is based on `Foldable` rather than `[]`, mooting the need for translating the container from one form to the other:

```
    for permMap $ \perms -> when (Set.size perms > 1) $ do
        putStrLn . foldr1 (\x acc -> x ++ ", " ++ acc) $ perms
```

A Bit of Whimsy

One of the standard arguments for preferring `Applicative` over `Monad` whenever possible is that the composition of `Applicative` types are also `Applicative` in a natural way, whereas the composition of `Monad` types is not necessarily a `Monad`.

Recall that category-theoretic monads are defined in terms of `join :: (Monad m) => m (m a) -> m a`. The type of `sequenceA` hints at the fundamental difficulty. If we had a natural operator `commuteM :: (Monad m, Monad n) => m (n a) ->` then we'd have a natural definition of `join` for the composed monads, as follows.

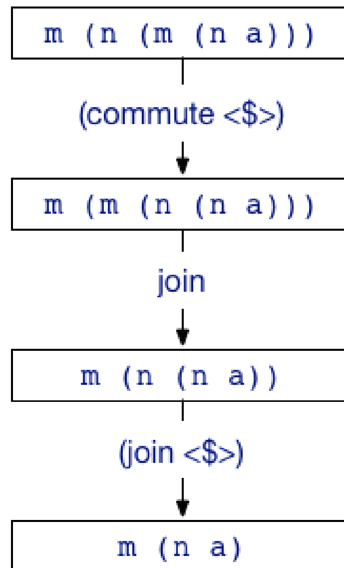
We start by undefining what we can't define (am I the only one who recalls the disassociated press version of the Gettysburg Address?!):

```
commuteM :: (Monad m, Monad n) => m (n a) -> n (m a)
commuteM = undefined
```

This may seem pointless, but we want to show that this is the only obstruction, which is remarkable given the

parallels with `sequenceA`: all we're missing is the `Traversable` instance for `m`!

Our goal is to show the we can start with `m (n (m (n a)))`, and end with `m (n a)`.



This sweeps under the rug a bit of the type traffic (we use `Data.Functor.Compose`'s `Compose` type to represent the composition, and this requires unwrapping and re-wrapping), but gives the main idea.

- CommuteM

Chapter 19

Writer, Reader

Background

We often think of monads as a mechanism for adding "effects" to our computations. E.g., if we want to add non-determinism as an effect, we can do this by considering lists of results. Working within the list monad enables us to use binding as a means of representing non-deterministic choice, and so hides the complexity by allowing the programmer to focus on the non-deterministic choices made, and hides the mechanism for organizing those choices, and assembling the results.

A complication comes from when we try to layer effects, e.g., we want to combine non-deterministic computation with IO. The problem here is that, unlike functors and applicatives, monads don't compose, i.e., there is no natural way to define an instance

```
instance (Monad m, Monad n) => Monad (Compose m n)
```

A solution comes from the notion of *monad transformers*, a sophisticated approach that enables us to build layered effects. The monad transformer library (mtl) defines a number of monad transformers. As the kind of a monad is `* -> *`, it is unsurprising that the kind of a monad transformer is `(* -> *) -> * -> *`. For example, if we have some sort of effect "foo," we'll have a monad transformer `FooT`, such that whenever `m` is a monad, then `FooT m` will also be a monad, but one that layers the "foo" effect onto the effects of `m`.

Associated with each of the `FooT` transformers, is the class `Foo = FooT Identity`, which is a monad that has only the "foo" effect. Doing things this way saves the library implementor some work, but it does incur a cost on the ordinary programmer. If you make a type error in the context of a `Foo` monad, you're liking to see errors referring to `FooT Identity`, which can be disorienting, especially if you encounter them before you've seen monad transformers, or understand how the `Foo` type constructor is implemented "under the hood."

There's also a type class `MonadFoo` which contains all of the functions and constants that we associate with the "foo" effect, and naturally there's a `MonadFoo` instance of `FooT m`. But the mad genius of the monad transformer library is that for each of the other monad transformers (e.g., `BarT`), there's a derived instance

```
instance MonadFoo m => MonadFoo (BarT m)
```

This has the consequence that when we add a "foo" effect, our work is not undone by adding additional (compatible) effects. This matters! But this makes for a complicated library (briefly, the number of deriving instances definitions has to grow quadratically with the number of base transformers). Again, this is a case where details of the implementation can unexpectedly "leak out," as the functions we associate with `Foo` aren't to be found in the definition of `Foo`, but rather in the related `MonadFoo` type class.

The mechanics of making the monad transformer library work are not for the faint-of-heart, but fortunately, we don't need to learn them all at once. An important first step is to understand the kind of "effect" that each transformer adds. We can study these effects in isolation by considering `Foo = FooT Identity` for each of the monad transformers, and so learn the basic building blocks of the mtl.

A Motivating Example

The core of many programs are evaluators of some sort. If we want effects in our evaluation process, it's common to have evaluation monads. To make things easy we'll construct an `Expr n` type, which is based on `Num`. We start with

```
data UnaryOp
    = Abs
    | Signum
    deriving (Show)

data BinOp
    = Add | Subtract | Multiply
    deriving (Show)

data Expr n
    = Value n
    | ApplyBinary BinOp (Expr n) (Expr n)
    | ApplyUnary UnaryOp (Expr n)
    deriving (Show)
```

Next, we'll make `Expr n` an instance of `Num`:

```
instance Num n => Num (Expr n) where
    (+) = ApplyBinary Add
    (-) = ApplyBinary Subtract
    (*) = ApplyBinary Multiply
    abs = ApplyUnary Abs
    signum = ApplyUnary Signum
```

```
        fromInteger = Value . fromInteger
```

This is a bit surprising. We're not doing computational work here, but instead are using the `Num` instance as a vehicle for translating from ordinary notation for simple expressions to values of our particular data type, and we're deferring the actual evaluation of the operations until later. A simple test shows what we've done.

```
> 1 + 2 * 3 :: Expr Int
ApplyBinary Add (Value 1) (ApplyBinary Multiply (Value 2) (Value 3))
```

We're going to use a monad for evaluation, but because we have no effects, we'll keep the monad as simple as possible:

```
type Eval = Identity
```

Our goal is to write a function `eval :: (Num n) => Expr n -> Eval n`. We'll abstract out the patterns of applying binary and unary operators via corresponding helper functions:

```
    -- | Evaluate an expression in the Eval monad.


    eval :: (Num n,Show n) => Expr n -> Eval n
    eval (Value n) = pure n
    eval (ApplyBinary Add e1 e2) = applyBinary (+) e1 e2
    eval (ApplyBinary Subtract e1 e2) = applyBinary (-) e1 e2
    eval (ApplyBinary Multiply e1 e2) = applyBinary (*) e1 e2
    eval (ApplyUnary Abs expr) = applyUnary abs expr
    eval (ApplyUnary Signum expr) = applyUnary signum expr

-- | Evaluate the result of applying a binary operator to a pair of expressions, in the
--   Eval monad.

    applyBinary :: (Num n,Show n) => (n -> n -> n) -> Expr n -> Expr n -> Eval n
    applyBinary op e1 e2 = do
        v1 <- eval e1
        v2 <- eval e2
        let result = v1 `op` v2
        pure result

-- | Evaluate the result of applying a unary operator to an expression, in the
--   Eval monad.

    applyUnary :: (Num n,Show n) => (n -> n) -> Expr n -> Eval n
    applyUnary op expr = do
        v1 <- eval expr
```

```
        let result = op v1
        pure result
```

These helper functions could be η-reduced, but let's leave them as is for now.

With this machinery, we're set up to do simple evaluation:

```
> runIdentity . eval $ 1 + 2 * 3
7
```

Exercise 19.1 *The functions* `applyBinary` *and* `applyUnary` *are implemented in a very naïve way (albeit for a reason that will become clear later). Show that they can be implemented as* `Applicative` *one-liners.*

Writer

Let's consider a simple effect we might want to add to our evaluator. We'd like to add a logging capability, so that we construct a record of computational work as it's being done.

The `Writer w a` represents a type where `w` is the type of the values we're writing, and `a` is the type of the value we're computing.

For the purposes of this lecture, we'll write our own `Writer` module, intended as a work-alike replacement for `Control.Monad.Writer` for dealing with simple `Writer` types. If we do our work well (and of course we will), we'll be able to build our example program using our `Writer` module, and then simply replace the import of `Writer` with `Control.Monad.Writer`, and have everything work, even though the implementation of `Writer` in `Control.Monad.Writer` actually goes via `WriterT`. Values of `Writer w a` have to encode both a message of type `w`, and a value of type `a`.

```
newtype Writer w a = Writer { runWriter :: (a,w) }
```

The "twist" in the order of `w` and `a` is a bit annoying, and will require constant attention as we code.

We start by providing a `Functor` instance, and it's easy:

```
instance Functor (Writer w) where
    fmap f (Writer (a,w)) = Writer (f a, w)
```

The `Applicative` instance hints at things to come. To implement `<*>`, we'll take two values of this type, containing both values and messages, and have to combine them. Combining the values is straightforward: one of the values is a function of type `a -> b`, and the other is a value of type `a`. We simply apply the function to the value, and we're set. But what about the messages? We need a way to combine messages. Moreover, to implement `pure`, we need

a way to conjure up a message from thin air. To make both possible, we'll add the constraint that `w` be a `Monoid`. Thus

```
instance Monoid w => Applicative (Writer w) where
    pure a = Writer (a,mempty)
    (Writer (fa,fw)) <*> (Writer (xa,xw)) = Writer (fa xa, fw <> xw)
```

Exercise 19.2 *Write a suitable* `Monad` *instance for* `Writer w`.

Associated with this monad are a number of useful functions. We'll use two in our example, the rest can be discovered by reading the code and documentation.

```
-- | Construct a writer action with a given message.

tell :: w -> Writer w ()
tell w = Writer ((),w)

-- | An fmap-like function that acts on the message component of a writer action.

censor :: Monoid w => (w -> w) -> Writer w a -> Writer w a
censor f (Writer (a,w)) = Writer (a,f w)
```

Our completed `Writer` module (modulo the definition of `(>>=)`) is `Writer.hs`.

Example: `WriterEval`

We want to add logging to our evaluator. This requires a few changes, but perhaps less than you'd think. First, we have to define `Eval` so that it takes into account writer effects.

```
type Eval = Writer [(String,String)]
```

The type of our message is a list of pairs. Lists are nice monoids, and the idea is that each of the pairs will correspond to a line of output. Each pair encodes an operation that was applied, and a brief description. We've done this for a reason...

It is often the case that we build a little private language on top of our effects, and we find it useful to have the following:

```
-- | Log a message that an operation was performed.
```

```
note :: String -> [String] -> Eval ()
note op ws = tell [(op,unwords ws)]
```

The `note` function uses `tell` to append a pair (with our intended interpretation) onto the message that's being built.

Our `eval` function now has to be reoriented a bit to account for logging:

```
eval :: (Num n,Show n) => Expr n -> Eval n
eval (Value n) = pure n
eval (ApplyBinary Add e1 e2) = applyBinary "+" (+) e1 e2
...
eval (ApplyUnary Abs expr) = applyUnary "abs" abs expr
...
```

The difference is that we have to print out values in our log, and so our underlying numeric type needs to be an instance of `Show`, and our "apply" functions are going to need a printable name for the function that's being applied. Note that would could have handled this in other ways, but this is simple.

```
applyBinary :: (Num n,Show n) => String -> (n -> n -> n) -> Expr n -> Expr n -> Eval n
applyBinary name op e1 e2 = do
    v1 <- eval e1
    v2 <- eval e2
    let result = v1 `op` v2
    note name [show v1,name,show v2,'=',show result]
    pure result
```

Our implementation of `applyBinary` has to handle that extra string argument, produce an appropriate message, which we do by adding the `note` line. The changes to `applyUnary` are similar.

We're now set up to *use* this machinery, via the `showWork` function:

```
-- | Perform an evaluation in our Eval monad, formatting our message log,
--   and sending it to standard output.

showWork :: (Num n,Show n) => Expr n -> IO ()
showWork expr = do
    let (result,output) = runWriter . eval $ expr
    putStr . unlines . map snd $ output
    putStrLn $ "The final answer is " ++ show result ++ "."
```

Now,

```
> showWork $ (2+3) * (4+5)
```

```
2 + 3 = 5
4 + 5 = 9
5 * 9 = 45
The final answer is 45.
```

If only we'd had this in third grade. Of course, we don't spend our *entire* life in third grade, but must in due course move along to fourth. As you know, "showing your work" never meant to show the trivial steps, just the hard/interesting ones. By fourth grade, it's assumed you can do addition and subtraction, but you still have to show your multiplication steps. We accomplish this with a simple change to the `showWork` function, slipping in a function that filters the message list, retaining only the hard steps:

```
-- | Perform an evaluation in our Eval monad, filtering our message log down to the
--   hard steps, formatting it, and sending it to standard output.

showHardWork :: (Num n,Show n) => Expr n -> IO ()
showHardWork expr = do
    let (result,output) = runWriter . censor (filter isHard) . eval $ expr
    putStr . unlines . map snd $ output
    putStrLn $ "The final answer is " ++ show result ++ "."
    where
        isHard (op,_) = op == "*"
```

Now,

```
> showHardWork $ 10 * 12 + 11 * 11
10 * 12 = 120
11 * 11 = 121
The final answer is 241.
```

We're making good progress, and are now ready for fifth grade. Our example code is `WriterEval.hs`.

Reader

Let's consider a different kind of effect. We'd like to beef up our evaluation system a little bit, allowing ourselves to save and re-use computed values. To that end, we need to introduce variables and binding expressions to our `Expr` type:

```
data Expr n
    = ...
    | Variable String
    | Let [(String,Expr n)] (Expr n)
```

Evaluation now needs to take place in a context that can maintain a list of bindings. For this, we introduce the `Reader` type. The idea is that the type `Reader e a` will wrap a function of type `e -> a`. We use `e`, because we're thinking of this as the *environment* of the computation. As before, we'll build a work-alike module `Reader` that implements the `Reader` type directly, rather than via `ReaderT`.

```
newtype Reader e a = Reader { runReader :: e -> a }
```

We can recognize this as a wrapped version of `(->) e`, and simply lift the `Functor`, `Applicative`, and `Monad` instances from there. As this involves nothing new (although some of the type swizzling is usually annoying), we'll elide it, but the definitions are in the source file.

Exercise 19.3 *Write your own* `Functor`, `Applicative`, *and* `Monad` *for* `Reader e`, *and compare them with the definitions in* `Reader.hs`*.*

The standard `Reader` functions are

```
-- | Retrieve the environment.

ask :: Reader e e
ask = Reader id

-- | Run an action in a modified environment.

local :: (e -> e) -> Reader e a -> Reader e a
local f (Reader g) = Reader $ g . f

-- | Retrieve a function applied to the current environment.

asks  :: (e -> a) -> Reader e a
asks f = Reader f
```

These seem quite simple, but they are powerful building blocks.

Example: `ReaderEval`

Our goal now is to implement a more powerful evaluator, one that allows the use of variables and bindings. For now, our goal will be to evaluate

```
testExpr :: Expr Int
testExpr =
    let x = Variable "x"
        y = Variable "y"
```

```
    in Let [ ("y",10) ]
           $ Let [("x", y + y)]
                 (x * x - y * y)
```

First, we define our evaluation monad

```
type Eval n = Reader [(String,n)] n
```

Note that the 'environment' in this case is a simple association list.

To keep our code simple, we'll use a crashing lookup function,

```
-- | Look up a value in an association list, crashing if there is no such value.

lookup' :: String -> [(String,n)] -> n
lookup' var env = case lookup var env of
    Just val -> val
    Nothing -> error $ "Unknown variable " ++ var
```

This isn't elegant, but trying to combine error handling via `Maybe` with the implicit environment of `Reader` puts us into the position of trying to layer effects, and we're not quite ready for that.

With this, we can handle variables easily

```
eval (Variable v) = asks (lookup' v)
```

All this does is to encapsulate the action of looking up the name of a variable in the (implicit) current environment, which is exactly what we need.

Unsurprisingly, most of the work is in evaluating `Let` expressions. To do this we first extract the keys from the binding list, then we evaluate the expressions to be bound from the binding list, then we create an association list of the keys and values resulting from this evaluation, then we modify the existing environment to include the newly bound variables, and finally we evaluate the body of the `Let` in the resulting environment. This sounds like a lot, but it's not so bad:

```
eval (Let bindings expr) = do
    let ks = map fst bindings
    vs <- mapM (eval . snd) bindings
    local (zip ks vs ++) $ eval expr
```

Oddly enough, this is all we have to do. Add a couple of new lines to our `Expr` definition, set our evaluation monad to be the appropriate `Reader`, add a couple cases to our `eval` function, and we're pretty much set. Oh, and

evaluating `testExpr`? All we have to do is use `eval` to create a value in the evaluation monad, use `runReader` to extract the underlying function, and then apply it to an empty environment.

```
> runReader (eval testExpr) []
300
```

Profit!

`ReaderEval.hs`

Chapter 20

State

20.1 State, I

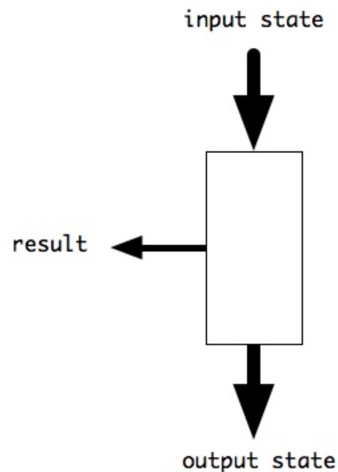
The `State` type enables us to build computations that manipulate (hidden) state. I think of this as "elephant in the room" programming, wherein we have a central, ubiquitous data structure that is so important that we can actually clarify our code by hiding it. We will follow our on-going strategy of building a work-alike version of `State`, as a stepping stone to `StateT`.

Haskell programmers will often describe `Reader` and `Writer` as complementary halves of `State`, it is a correspondence that's worth keeping in mind.

We define the `State` type as:

```
newtype State s a = State { runState :: s -> (a,s) }
```

The intuition here is that `s` is the type of the "state" of the computation, a value of which may be used and altered over the course of a sub-computation that produces a value of type `a`. This is realized through a function that takes a state value as an argument, and which returns a pair consisting of a value of the advertised binding type `a`, together with the updated state. I find it helpful to think of this diagrammatically:



Our first order of business is to make `State s` an instance of `Functor`:

```
instance Functor (State s) where
    fmap f ma = State $ \s ->
        let (a,t) = runState ma s
        in (f a,t)
```

Here, the effect of the `fmap f` is to wrap up the old state transforming function in a new state transforming function that calls the old state transforming function, capturing and adjusting its binding value.

Exercise 20.1 *Show that the functor instance above can be re-written as*

```
fmap f ma = State $ (\(a,s) -> (f a,s)) . runState ma
```

via a sequence of principled transformations. Hint: Think about how the expression `let x = e1 in e2` *can be written using more basic expressions such as lambdas and function application.*

For an extra challenge, it can be further reduced to

```
instance Functor (State s) where
    fmap f ma = State $ uncurry ((,) . f) . runState ma
```

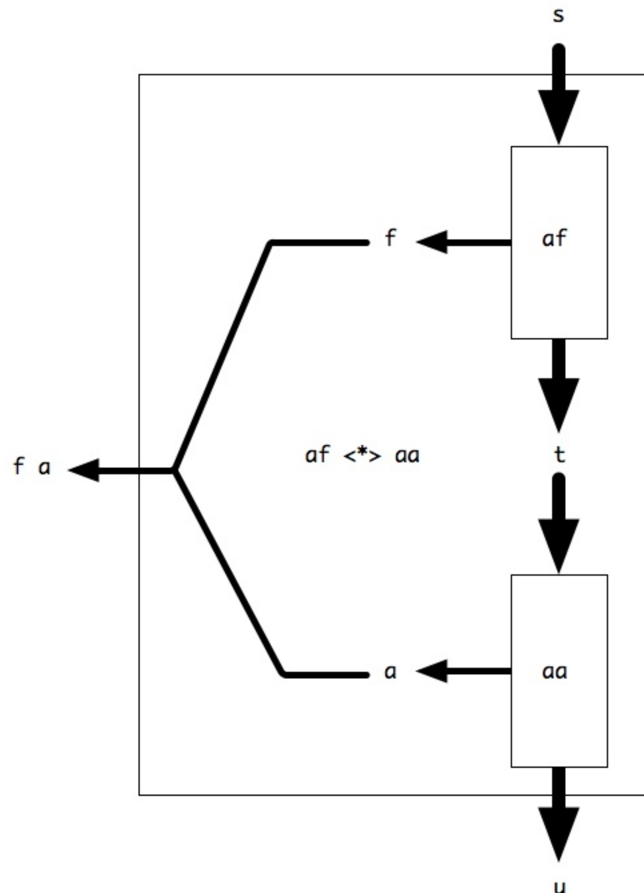
Next up in the type class hierarchy, we make `State s` an instance of `Applicative`:

```
instance Applicative (State s) where
    pure a = State $ \s -> (a,s)
```

```
    af <*> aa = State $ \s ->
        let (f,t) = runState af s
            (a,u) = runState aa t
        in (f a, u)
```

This is worth understanding. The `pure` instance produces a `State` function that doesn't use or change the state `s`, just passes it through, while making `a` available for binding through the first coordinate of the pair. The `(<*>)` action takes the input state, and obtains the function `f` and an updated state `t`, then passes the updated state `t` to `aa`, obtaining the argument `a` and an updated state `u`, and finally we package up the application `f a` and the final state `u` into a pair. It is important to understand that `(<*>)` specifies an *order* in which the state is used: left-hand (function) argument first, then the right-hand argument.

We can envision this process diagrammatically as follows:



Exercise 20.2 *Show that we can implement* `pure` *as* `pure a = State $ (,) a`.

A difficult challenge is to try to simplify the definition of `<*>` *to first eliminate the* `let` *construct, and then to eliminate any lambdas. It can be done!*

Finally (at least, finally in the first-pass sense), we make `State s` an instance of `Monad`:

```
instance Monad (State s) where
    ma >>= f = State $ \s ->
        let (a,t) = runState ma s
            mb = f a
            (b,u) = runState mb t
        in (b,u)
```

The definition of `(>>=)` is quite similar in spirit to `(<*>)`. As with `(<*>)`, the state will flow through the expression in left-to-right order. In the case of `(>>=)`, the argument `ma` is performed first, obtaining a result `a` and an updated state `t`, then the function `f` is applied to `a`, obtaining another monadic action `mb`, which is performed passing it the state `t`, obtaining a result `b` and a final state `u`, which are packaged into the result pair `(b,u)` as before.

Exercise 20.3 *Produce a diagram, analogous to the diagram for* `(<*>)` *above, that illustrates how* `(>>=)` *works.*

Exercise 20.4 *Show that the monad instance above can be re-written as*

```
instance Monad (State s) where
    ma >>= f = State $
        (\(a,s) -> runState (f a) s) . runState ma
```

via a sequence of principled transformations.

For extra-credit, reduce it to

```
instance Monad (State s) where
    ma >>= f = State $ uncurry (runState . f ) . runState ma
```

To use `State` effectively, we provide monadic functions that extract, inject, and modify the state, hence,

```
get :: State s s
get = State $ \s -> (s,s)

put :: s -> State s ()
put t = State $ \s -> ((),t)

modify :: (s -> s) -> State s ()
modify f = State $ \s -> ((),f s)
```

Note that the later two are essentially `pure` and `fmap` for the state component of a `State s a` value, while `get` simply exposes the state to where is can be extracted via `(>>=)`.

Exercise 20.5 *Unsurprisingly, the code transformation fairy isn't entirely satisfied with the definition of* `put` *and* `modify`*, and proposes*

```
put t = State $ const ((),t)
modify f = State $ (,) () . f
```

Verify the code fairy's work.

But now that we have this toy, what do we do with it? In a typical application, we'll consider `State s` for some application-specific type `s`, and we'll use `get` and `put` to write application-specific state accessors and mutators.

Example: Turning Haskell into a 1980s era calculator

The Hewlett-Packard corporation made a name for itself by producing high-quality electronic scientific (and later, financial) calculators which thoroughly disrupted the pre-existing economy of scientific calculation built on slide rules. A novel feature of the HP calculators was their use of reverse polish notation (RPN), which involved the use of an operand stack, together with operations that acted on that stack. This had the considerable virtue of simplifying input handling, thereby allowing silicon to be devoted to other tasks, and HP calculators were justly famous for their numerical precision and speed. With an RPN calculator, to add `1` to `2`, you'd hit, `1`, `Enter`, `2`, and then, finally, `+`. The more advanced models that started to appear in the 1980's were programmable, in the sense that certain keys could be programmed to execute a sequence of key strokes (possibly involving other programmable keys). With these calculators, the operand stack was arguably the most important abstraction, but paradoxically, it was hidden from view. Our goal is to write code that provides much of the feel of working with these early calculators, and doing so requires that we hide the operand stack.

You can find a javascript simulator of an HP-35 calculator—the very calculator Professor Kurtz purchased *used* for $100.00 in 1976 when he was an undergraduate.

This particular example will give us the opportunity to focus on the notion of *abstraction barriers*. These are a software engineering technique whereby we decompose the problem into subproblems, with each sub-solution specifying

a well-defined *interface* that completely determines the ways that other subproblems can interact with it. Haskell's module system provides excellent support for abstraction barriers, as we'll see. Let's start by taking the work that we've just done building the `State` monad, and encapsulate it in a module called `State` (in `State.hs`).

Our next module will be the `Calc` module (in `Calc.hs`), in which we'll define (and export) the basic functionality of our calculator. A particular convention of our calculator will be that all of the calculator operations we export will have the form `kOperation` for some operation.

We'll start by defining the type associated with a simple calculation:

```
type CalcState = State [Double]
```

`CalcState` is intended for internal use, whereas the more restricted

```
type Calculation = CalcState ()
```

is intended for exported values.

Next, we define a couple of monadic functions that enable us to push and pop values off of the stack. We can think of these as primitives, built on top of the still more primitive `get` and `put` functions:

```
pop :: CalcState Double
pop = do
    stk <- get
    case stk of
        [] -> pure 0.0
        x:xs -> do
            put xs
            pure x

push :: Double -> CalcState ()
push d = do
    stk <- get
    put (d:stk)
```

Note how our result is `pure 0.0` when we have an empty stack. Intuitively, this is equivalent to viewing the stack as being infinitely deep, and filled to the bottom with `0.0`'s. Standard code transformations can result in surprising concision:

```
push = modify . (:)
```

If we think in a somewhat more concrete way about how the `State` monad works, we can come up with the following much more succinct definition:

```
pop = state $ \s -> case s of
    [] -> (0.0,[])
    x:xs -> (x,xs)
```

At first, this feels a bit like both a typographical error and an abstraction barrier violation, but it's neither, because `state` is a standard part of the `State` interface:

```
state :: (s -> (a,s)) -> State s a
state = State
```

The `push` operation, which we think of as a stack primitive, is actually something we want to export, albeit under a different name:

```
kEnter :: Double -> Calculation
kEnter = push
```

Next, we provide the basic arithmetic functions:

```
binOp :: (Double -> Double -> Double) -> CalcState ()
binOp op = do
    y <- pop
    x <- pop
    push $ op x y

kAdd, kSub, kMul, kDiv :: Calculation
kAdd = binOp (+)
kSub = binOp (-)
kMul = binOp (*)
kDiv = binOp (/)
```

Notice how in `binOp` that if we've pushed the first operand first, that must mean that we'll pop the second operand first. Notice also the preference within a module for using the private `push` over the exported `kEnter`. This is not required, but it does reflect differences in how with think about the problem: outside of the abstraction barrier, we use the exported functionality, inside the abstraction barrier, we have unrestricted access to the functions defined at that level (or exported from a lower level still).

Next, we need a little bit of code to allow us to actually run a `Calculation`.

```
perform :: Calculation -> Double
perform ma = fst $ runState (ma >> pop) []
```

This is just a bit obscure. It might seem simpler conceptually to do this:

```
perform ma = head . snd $ runState ms []
```

The problem we solve with a final pop is that running `ma` might result in an empty stack, in which case the call to `head` would raise an exception. A final pop action takes advantage of the error handling we've built into pop, albeit while delivering the intended result as the value bound by performing the action, rather than as the top of the resulting state. We can simplify this a bit further through use of a couple of convenience functions that are a standard part of the `State` interface:

```
evalState :: State s a -> s -> a
evalState ma s = fst (runState ma s)

execState :: State s a -> s -> s
execState ma s = snd (runState ma s)
```

where `evalState` throws away the final state, evaluating to the final bound value, whereas `execState` throws away the final bound value (often `()`), evaluating to the final stack. With this, we write:

```
perform :: Calculation -> Double
perform ma = evalState (ma >> pop) []
```

With these, we can begin to write little programs that resemble old-school RPN calculations, e.g., here's how we'd compute `(1+2)*3`:

```
test :: Double
test = perform $ do
    kEnter 1
    kEnter 2
    kAdd
    kEnter 3
    kMul
```

Now, part of the game in trying to simulate an RPN calculator is to avoid the use of explicit bindings except in the definition of primitives in the `Calc` module. This is an issue if we want, e.g., to define a `hypotenuse` macro. The idea here is that `hypotenuse` should expect two arguments on the stack, and it should pop them off, leaving the hypotenuse of the corresponding right triangle pushed on the stack. To handle this, we introduce a couple of new primitives to `Calc`:

```
kSwap :: Calculation
kSwap = do
    y <- pop
```

```
    x <- pop
    push y
    push x

kDup :: Calculation
kDup = do
    x <- pop
    push x
    push x
```

together with the corresponding updates to `Calc`'s export list. At this point, purists might note that while the HP-35 had a swap key, it didn't have a dup key. That's because the Enter key did the work of both kEnter and kDup, depending on the input state, i.e., it had the effect of kEnter if we'd just typed a number in, and kDup if the last key-stroke was an operation or an Enter.

We're also going to need a square root function. To facilitate this, we'll add a private `unOp` function:

```
unOp :: (Double -> Double) -> CalcState ()
unOp op = do
    x &lt- pop
    push $ op x
```

and the keystroke function for square root:

```
kSqrt :: Calculation
kSqrt = unOp sqrt
```

We'll also take this opportunity to add the basic trigonometric functions,

```
kSin,kCos,kTan :: Calculation
kSin = unOp sin
kCos = unOp cos
kTan = unOp tan
```

This enables us to define (in the `CalcExample` module in `CalcExample.hs`)

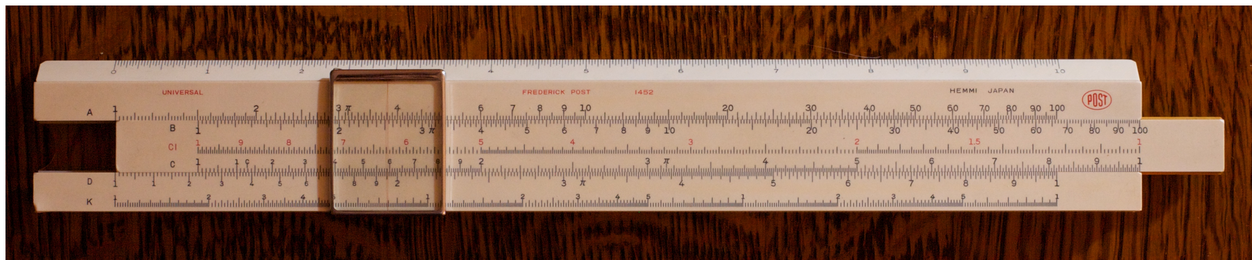
```
square :: Calculation
square = do
    kDup
    kMul

hypotenuse :: Calculation
```

```
hypotenuse = do
    square
    kSwap
    square
    kAdd
    kSqrt
```

With this in hand, we can compute the hypotenuse of a 3-4-? triangle:

```
> perform $ kEnter 3 >> kEnter 4 >> hypotenuse
5.0
```

This is all pretty straightforward, if a bit nerdy, and in Professor Kurtz's case, maybe even a bit maudlin. That HP-35 died decades ago, but oddly enough I still have the slide rule it replaced. There's probably a lesson in that. Life, and computing, go on.



But let's push this a bit further. HP calculators, in addition to the stack, also contained a memory location as a part of their state. Let's suppose we wanted to implement the store and recall functionality. This presents us with a conundrum. How? The answer is going to take us back to `Calc`, and a more complicated model of the state that's being manipulated:

```
data InternalState = InternalState
        { stack :: [Double]
        , memory :: Double
        }

type CalcState = State InternalState
```

This is obviously going to break things, but because of the abstraction barriers we've implemented, the breakage is limited to the `Calc` module, and indeed, because of an internal abstraction barrier within `Calc` to just the `push`, `pop` and `perform` functions, as they're the only functions that accessed the state directly:

```
pop :: CalcState Double
pop = state $ \st -> case stack st of
```

```
        [] -> (0.0,st)
        (x:xs) -> (x,st {stack = xs})

push :: Double -> CalcState ()
push d = modify $ \st -> st { stack = d : stack st }

perform :: Calculation -> Double
perform ma = evalState (ma >> pop) startState where
    startState = InternalState { stack = [], memory = 0.0 }
```

With this, our existing code works, leaving us only to implement access to the memory part of the internal state.

Exercise 20.6 *Add implementations of* `store` *and* `recall` *to the* `Calc` *module, along with exported definitions. The* `store` *action should copy the top of the stack into memory, while the* `recall` *action should push the memory onto the top of the stack.*

```
kSto :: Calculation
kSto = store

kRcl :: Calculation
kRcl = recall
```

and write an example program that illustrates their use.

Use the modules:

- *State.hs*
- *Calc.hs*
- *CalcExample.hs*

20.2 State, II

HTML

Our next example is a good deal more practical. We're going to use the `State` monad to write a small library for producing HTML. Note that we're moving to the "official" MTL implementation:

```
import Control.Monad.State

type Document = State String
```

The idea here is that our state is going to be a `String` that contains HTML, and our various operations are going to act on it. In the simplest case, we'll simply append some a `String` onto the state:

```
string :: String -> Document ()
string t = modify (\s -> s ++ t)
```

Which, after the usual transformations, we can write as

```
string = modify . flip (++)
```

Next, we need a bit of code (analogous to `perform` in the calculator example) to render an HTML value as a string:

```
render :: Document a -> String
render doc = execState doc ""
```

Note that this code is already borderline useful:

```
> render (string "foo" >> string "bar")
"foobar"
```

Thus, we can do simple string concatenation, with an alternative notion.

But the heart of HTML is its use of tags. We'll define `tag` to be a monadic function, which takes a tag name and a monadic argument, and writes a start tag, then performs the action of the argument monad (appending its output to the state), and then concludes by writing the end tag.

```
type HTML = Document ()

tag :: String -> HTML -> HTML
tag t html = do
    string $ '<' ++ t ++ '>'
    html
    string $ '</'++ t ++ '>'
```

We can then define a number of tagging functions:

```
html  = tag "html"
head  = tag "head"
title = tag "title"
body  = tag "body"
```

```
p     = tag "p"
i     = tag "i"
b     = tag "b"
h1    = tag "h1"
h2    = tag "h2"
h3    = tag "h3"
h4    = tag "h4"
ol    = tag "ol"
ul    = tag "ul"
table = tag "table"
tr    = tag "tr"
th    = tag "th"
td    = tag "td"
```

These functions can be used to give a nice structural definition of an HTML page in Haskell, e.g.,

```
doc :: HTML
doc =
        html $ do
        head $ do
            title $ string "Hello, world!"
        body $ do
            h1 $ string "Greetings"
            p $ string "Hello, world!"
```

We can render this, and write it as a file:

```
> writeFile "hello.html" $ render doc
>
```

This is all pretty enough, but how is it *useful*?

Let's consider a fairly typical minor problem in dealing with a web server: determining exactly what environmental variables are set. What we're going to do is write a little CGI (common gateway interface) program, which obtains the environmental bindings, and converts them to HTML. There is surprisingly little code required:

```
{- A program for rendering a CGI's environment as HTML -}

module Main where

import HTML
import Data.List (sort)
import System.Posix.Env
```

```
{- Create an HTML document based on a key/value list -}

makeDoc :: [(String,String)] -> HTML
makeDoc env =
    html $ do
        HTML.head . title . string $ 'Environment variables'
        body $ do
            h1 . string $ 'Environment variables:'
            ul . mapM_ makeEntry . sort $ env
    where

    makeEntry (key,value) = li . string $ key ++ ' = ' ++ encode value

    encode = concatMap encodeChar where
        encodeChar '<' = '<'
        encodeChar '&' = '&amp;'
        encodeChar '>' = '>'
        encodeChar c = [c]

{- The main act -}

main :: IO ()
main = do
    env <- getEnvironment
    putStr "Content-type: text/html\n\n"  {- the minimal required HTTP header -}
    putStr . render . makeDoc $ env
```

The heavy lifting here is done by the call to `mapM_`, which turns a list of binding pairs into an `HTML` value that sequences an appropriately formatted `li` element for each binding pair.

Most people don't write CGI programs in Haskell, I'm not most people, so I sometimes do, albeit usually using the functionality found in `Text.Blaze` and `Network.cgi`, but this small example shows how we can roll our own functionality.

Exercise 20.7 *When generating output in formats such as HTML, it is often desirable to* pretty print *the results, especially while debugging. The idea is to make judicious use of indentation, newlines, and other formatting choices to make the file more readable. In this problem, you will write a pretty-printing version of the HTML generator above. For example:*

```
> :load PrettyHTML.hs
> putStr $ render doc

<html>
  <head>
    <title>Hello, world!</title>
  </head>
  <body>
```

```
      <h1>Greetings</h1>
      <p>Hello, world!</p>
    </body>
  </html>
```

To implement this functionality, start with the new state representation

```
type Document = State (Int, String)
type HTML = Document ()
```

where the integer value represents the "depth" of the current tag in the HTML tree. When generating string output at depth `k`*, the string should typically be indented* `k` *"tabs" to the right. One tab should be four spaces.*

The file `PrettyHTML.hs` *provides a template for your solution, where* `undefined` *is used as a placeholder for the following definitions that you must complete:*

```
string :: String -> Document ()
newline :: Document ()
render :: Document a -> String
indent :: Document ()
exdent :: Document ()
```

The `newline` *function should append an indentation-aware newline (i.e., a newline followed by an appropriate number of tabs) to the current string state. The* `indent` *and* `exdent` *functions are used to increment (respectively decrement) the indentation level by one.*

Randomness

There is an important class of computer programs that use randomness (or more properly, as we'll see, pseudo-randomness), often to generate a "typical instance" based on a probabilistic model of some type. Our next program will do just that, through a re-implementation of Emacs's ludic "disassociated-press" command.

The idea behind disassociated-press is simple: The input is used to create a model of English prose, based on the frequency with which one word follows another, and then a random instance of that model is created. The result is best described as "English-like," often non-sensical, but sometimes disconcertingly sensical. It will be noted in passing that we had fewer sources of entertainment back in the day.

Our model is as follows:

```
type Model = (String,Map String [Maybe String])
```

A `Map` is simply a higher efficiency version of an association list.

Our model keeps track of the first word (which is used to kick-off generation), and a map which associates with each word of the text a list of following words. Now, this later is not exactly right, as we're going to use the map to deal with both word succession and termination – so the values are `[Maybe String]`, where a `Just w` element represents a succeeding word `w`, and `Nothing` represents the end of the text.

Building the model is something we do in pure code.

```
buildModel :: [String] -> Model
buildModel xs@(x:_) = (x,unionsWith (++) . transitions $ xs) where
    transitions (y:ys@(y':_)) = singleton y [Just y'] : transitions ys
    transitions [y] = [singleton y [Nothing]]
    transitions [] = error "Impossible error"
buildModel [] = error "Empty model"
```

The `Map` data type has a lot of existing functionality, including functionality for mutation, but it is generally more convenient to build maps out of simpler maps, as we've done here, providing an appropriate combining function.

Randomness enters into the program in generating an example text from the model. The central problem for us is to select a random element from a list, and herein enters the central problem of writing pure functional code that uses randomness. Most programming languages provide a function

```
rangen :: () -> Int
```

The idea here is that each call to `rangen ()` will produce a new, random result. But pure languages don't work that way: functions alway produce equal results on equal arguments. Haskell deals with this by defining

```
class RandomGen g where
   next     :: g -> (Int, g)
   ...
```

which should look like a familiar state transition function, because that's what it is.

The idea here is that a random number generator will produce both a random integer, *and* a new random number generator. Code that uses randomness then chains these random number generators through the various calls, and this can be a pain to keep straight. So we use the `State` monad to "hide" the random number generators.

```
import System.Random

type RandState = State StdGen
```

We can now write

```
roll :: Int -> RandState Int
roll n = state $ uniformR (1,n)
```

Which rolls an n sided dice, and

```
select :: [a] -> RandState a
select as = do
    i <-  roll . length $ as
    pure $ as !! (i-1)
```

Which we can express more succinctly as

```
select as = (as !!) . (subtract 1) <$> roll (length as)
```

The function `randomR (a,b)` will produce a random element in the range from `a` to `b` *inclusive*, which we'll use as an index into the list. Note that `randomR:: RandomGen g => (a, a) -> g -> (a, g)`, so we're going to use the `state` function to lift a pure function of type `RandomGen g => g -> (a,g)` into `RandState` as before.

This brings us to the actual generation of the list of words from the model. This starts with the first word, and we use each successive word to look up possible continuations.

```
runModel :: Model -> RandState [String]
runModel (start,wordmap) = iter start where
    iter word = do
        let successors = wordmap ! word
        succ <- select successors
        case succ of
            Just w -> do
                ws <- iter w
                pure (word:ws)
            Nothing -> pure [word]
```

Exercise 20.8 *Show how the code for* `runModel` *can be tightened up to the following:*

```
runModel :: Model -> RandState [String]
runModel (start,wordmap) = iter start where
    iter word = (word:) <$> do
        maybeNext <- select $ wordmap ! word
        case maybeNext of
            Just nextWord -> iter nextWord
```

```
                Nothing -> pure []
```

Of course, a list of words doesn't lend itself to nice output, so we'll write a little line-breaking function:

```
linefill :: Int -> [String] -> String
linefill _ [] = '\n'
linefill n (x:xs) = iter x xs where
    iter current (nextWord:ys)
        | length current + length nextWord + 1 > n = current ++ '\n' ++ linefill n (nextWord:ys)
        | otherwise                   = iter (current ++ ' ' ++ nextWord) ys
    iter current [] = current ++ '\n'
```

This leaves us with `main`:

```
main :: IO ()
main = do
    input <- getContents
    gen <- getStdGen
    let model = buildModel (words input)
        disassociatedPress = evalState (runModel model) gen
    putStr . linefill 72 $ disassociatedPress
```

All that remains is a good chunk of prose to test this on. We'll consider the Gettysburg address, and produce the following Gettysburg address-like word salad:

```
$ ./disassociated-press < gettysburg.txt
Four score and proper that nation might live. It is for us to the last
full measure of that we can not dedicate, we can never forget what we
can not consecrate, we can never forget what we can never forget what we
can long endure. We are created equal. Now we can not hallow this
ground. The world will little note, nor long remember what they who here
gave the unfinished work which they gave their lives that government of
that these honored dead we here to the unfinished work which they who
fought here gave the last full measure of that field, as a portion of
that war. We have thus far so nobly advanced. It is altogether fitting
and dead, who here have a final resting place for which they did here.
It is rather for the people, by the proposition that government of that
government of freedom-and that field, as a new birth of that we can long
remember what we can not perish from these honored dead we can not
hallow this continent a great task remaining before us-that from these
honored dead shall not have died in vain-that this continent a final
resting place for which they did here. It is for which they gave the
```

```
earth.
```

Exercise 20.9 *A problem with simple probabilistic text generators like the one above is that they can generate very large amounts of text. How great is the danger in this case? Rework the program to run the model 1,000 times (without printing!), and compute the lengths of the largest and smallest strings generated. (To be clear here, we're measuring length in characters, not words.) Hint:* `replicateM` *is really useful at running a monad a bunch of times.*

Use the module `disassociated-press.hs`*, and* `gettysburg.txt`*.*

[Note for 2019] As originally written, this problem could have been interpreted as generating the longest and shortest strings, and printing them, rather than their lengths. This is acceptable for 2019, but not after.

Chapter 21

Case Study: Functional Parsing

21.1 Introduction to Functional Parsing

A parser is a function/procedure that translates a `String` representing a value of type `a` into that value. It's tempting to propose the following as the definition of an abstract `Parser` type:

```
type Parser s = String -> s
```

The problem here is that if we're building a parser out of pieces, the pieces are going to be sub-parsers that consume part but not all of the input. This suggests the following:

```
type Parser s = String -> (s,String)
```

where we're returning a pair that consists of the result of parsing an initial segment of the input string, and the unparsed remainder. This may be reminiscent of the `State` monad, specialized to `String`. Such a definition would allow us to write code like this:

```
pairp :: Parser a -> Parser b -> Parser (a,b)
pairp ap bp s = ((a,b),u) where
    (a,t) = ap s
    (b,u) = bp t
```

or even, if we somehow monadify `Parser` along the lines of `State`,

```
pairp :: Parser a -> Parser b -> Parser (a,b)
pairp ap bp = do
    a <- ap
```

```
        b <- bp
        pure (a,b)
```

or, after putting our applicative thinking hats on, even

```
pairp = liftA2 (,)
```

both of which hint at things to come. Unfortunately, though, this definition of `Parser` isn't quite robust enough. Consider, e.g., the special case of trying to parse a simple arithmetic expression. Suppose we had

```
data Expression
    = Const Double
    | Add Expression Expression
    | Mul Expression Expression

expressionParse :: Parser Expression
```

What should `expressionParse "1+2*3"` return? Obviously, we're looking for the result

- `(Add (Const 1) (Mul (Const 2) (Const 3)), "")`

But what about

- `(Const 1,"+2*3")`, or
- `(Add (Const 1) (Const 2), "*3")`?

All are plausible, in that they meet the contract for `expressionParse`, even though the last one is a bit problematic. We don't just have one right answer, it seems we have three! It's useful to think this as a non-deterministic calculation. So let's return a list:

```
type Parser s = String -> [(s,String)]
```

Of course, we're going to want to make `Parser` an instance of various standard type classes, so we'll use `newtype` as before:

```
newtype Parser s = Parser { runParser :: String -> [(s,String)] }
```

The result looks like a mash-up of a `State` and a `[]`. This is an interesting and productive observation.

Following Joroen Fokker, who first wrote this sort of functional parser, we now build some simple parsers, e.g., `satisfy` "shifts" the first character of input if it satisfies the argument predicate:

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = Parser $ \s -> case s of
    [] -> []
    a:as
        | p a ->  [(a,as)]
        | otherwise -> []
```

Note here the spiffy use of guards within patterns within a case statement. There's a really cute way to clean up that the last little bit, which is called the Fokker trick:

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = Parser $ \s -> case s of
    [] -> []
    a:as -> [(a,as) | p a]
```

How does this work? If `p a` is false, we end up returning the `[]` type's `empty` which is just `[]`, so `[(a,as)  p a]|` is either `[(a,as)]` or `[]`, according to whether `p a` is `True` or `False`, respectively. It's a cute trick, and well worth remembering.

We can use `satisfy` to define a number of additional simple parsers

```
char :: Char -> Parser Char
char c = satisfy (c==)

alpha, digit, space :: Parser Char
alpha = satisfy isAlpha
digit = satisfy isDigit
space = satisfy isSpace
```

Exercise 21.1 *Show by a series of principled transformations that we can define:*

```
char :: Char -> Parser Char
char = satisfy . (==)
```

The character predicates from `Data.Char` all beg to be turned into simple parsers, in similar fashion.

Next, we have a simple parser that recognizes a string:

```
string :: String -> Parser String
string str = Parser $ \s -> [(t,u) | let (t,u) = splitAt (length str) s, str == t]
```

Here, again, we use the Fokker trick, creating either a zero- or one-element list. A remarkable fact is that this pretty much concludes our effort to build *primitive* parsers, further progress is going to take the form of adding `Parser` to various standard type classes, leveraging the power of these simple functions.

We'll start by considering the task of writing a `Parser Bool`. We can start by considering a couple of primitive parsers for recognizing the strings `"True"` and `"False"`.

```
parseTrue = string "True"
parseFalse = string "False"
```

An obvious first problem is that these parsers both have type `Parser String`, rather than `Parser Bool`. To get a `Parser Bool`, it's convenient to make `Parser` an instance of `Functor`.

```
instance Functor Parser where
    fmap f p = Parser $ \s ->
        [(f a,t) | (a,t) <- runParser p s]
```

With the `Functor` instance in hand, we can write:

```
parseTrue = (const True) <$> (string "True")
parseFalse = (const False) <$> (string "False")
```

This sort of thing (a combination of `const` and `(<$>)`) happens a lot, so unsurprisingly there's an operator `(<$) :: Functor f => a` in `Data.Functor` that does this.

and perhaps recognize a useful pattern here:

```
token :: String -> a -> Parser a
token s a = a <$ string s

parseTrue, parseFalse :: Parser Bool
parseTrue = token "True" True
parseFalse = token "False" False
```

With this,

```
> runParser parseTrue "True"
[(True,"")]
```

which isn't exactly what we want, but it's a solid step in the right direction.

Of course, we don't want to be able to parse `"True"` *or* parse `"False"`, we want to be able to parse a string that

contains *either*. Somehow, we want to combine the two parsers into one parser, merging their outputs. There are a number of ways to approach this. We could make `Parser s` an instance of `Monoid`, or we could make `Parser` an instance of `Alternative`. Indeed, we could do both. There is an apparent cost associated with making `Parser` an instance of `Alternative`, and that is that `Alternative` requires `Applicative`. But a moment's consideration should suggest that we're going to want to combine the outputs of multiple parsers, as hinted at with the `pairp` example, and we know that the easiest way to provide generalizations of `fmap` to functions of greater arity is through `Applicative` anyway. So...

```
instance Applicative Parser where
    pure a = Parser $ \s -> [(a,s)]
    af <*> aa = Parser $ \s ->
        [ (f a,u)
        | (f,t) <- runParser af s
        , (a,u) <- runParser aa t
        ]

instance Alternative Parser where
    empty = Parser $ \s -> []
    p1 <|> p2 = Parser $ \s ->
        runParser p1 s ++ runParser p2 s
```

With this in hand, not only does our `pairp` example work, we can finish the `Parser Bool`:

```
parseBool :: Parser Bool
parseBool = token "True" True <|> token "False" False
```

An alternative solution, well worth remembering in other contexts, is

```
parseBool = read <$> (string 'True' <|> string 'False')
```

At this point, it's standard practice to introduce parser combinators `many` and `some`, which given a `Parser a` return a `Parser [a]`, where the `many` version returns a list of zero or more successful parses, and `some` a list of one or more parses. It's natural to write these via mutual recursion, as

```
many, some :: Parser a -> Parser [a]
many p = some p <|> pure []
some p = liftA2 (:) p (many p)
```

Much to our surprise, attempting to do so results in an error: `many` and `some` are already defined in `Control.Applicative`, with the type signature

```
many, some :: Alternative f => f a -> f [a]
```

Moreover, a quick consideration of our implementation reveals that it only uses functions in the `Alternative` or `Applicative` type classes. Unsurprisingly, we've just reimplemented these standard functions! Likewise, it's often useful to have an `optional` parser combinator, which is used for elements that may be present, but it too is already available in `Control.Applicative`, simplifying our work considerably.

For example:

```
> runParser (many parseBool) "TrueTrue!"
[([True,True],"!"),([True],"True!"),([],"TrueTrue!")]
> runParser (some parseBool) "TrueTrue!"
[([True,True],"!"),([True],"True!")]
> runParser (optional parseBool) "TrueTrue!"
[(Just True,"True!"),(Nothing,"TrueTrue!")]
```

We can now consider a mildly non-trivial parsing problem. Consider the type

```
data IntV = IntV Int deriving (Show)
```

This is just an `Int` in a box. Can we parse this given its natural syntax?

```
parseInt = read <$> some digit
skipSpaces = const () <$> many space

parseIntV = liftA3 (\_ _ i -> IntV i) (string "IntV") skipSpaces parseInt
```

For example:

```
> runParser parseIntV "123!"
[]
> runParser parseIntV "IntV 123!"
[(IntV 123,"!"),(IntV 12,"3!"),(IntV 1,"23!")]
```

A nice consequence of writing `parseIntV` is that it enables us to add `IntV` to the `Read` class, via

```
instance Read IntV where
    readsPrec _ = runParser parseIntV
```

Which we can test with

```
> read "IntV 18" :: IntV
IntV 18
```

Sweet! Of course, we could have also achieved the ability to read `IntV` values like this by including `Read` in the `deriving` clause of the type definition, but explicitly defining the `Read IntV` instance allows us to choose a different syntax if desired. Note that the wildcard argument to `readsPrec` is a precedence level, which means that we can have different parsers associated with different parsing contexts.

There's a small gap here, in that the parsers we're building return multiple results, whereas `read` returns only a single result. The bridge is in terms of code that filters the result list for those pairs `(a,t)` where the unparsed string `t` consists only of whitespace, and then requires that there be only a single result of that form, e.g.,

```
parseWith :: Parser a -> String -> a
parseWith p s = case [a | (a,t) <- runParser p s, all isSpace t] of
    [a] -> a
    [] -> error "no parse"
    _ -> error "ambiguous parse"
```

It's reasonable to argue that this might have been better handled by returning a value of type `Either String a` than by throwing an exception via `error`. When you write your own language, be sure to do it that way.

Monadic Parsing

Historically, functional programmers approached parser combinators through `Monad` rather than through `Applicative` and friends. It's worth noting that we can make `Parser` into a monad, and this has the considerable advantage of making monadic `do`-notation available to us, although pending changes to GHC/Hackage will permit the use of restricted versions of `do` with `Applicative`, largely eliminating this advantage.

```
instance Monad Parser where
    p >>= g = Parser $ \s ->
        [ (b,u)
        | (a,t) <- runParser p s
        , (b,u) <- runParser (g a) t
        ]

instance MonadPlus Parser
```

With these definitions, we could have written

```
parseIntV :: Parser IntV
```

```
parseIntV = do
    string 'IntV'
    skipSpaces
    i <- parseInt
    pure $ IntV i
```

which has the advantage of conceptual simplicity. That said, contemporary practice favors using `Applicative` based constructors over `Monad` based constructors, and the results argue for themselves in terms of concision if not always clarity.

Exercise 21.2 *Consider the following type declaration:*

```
data ComplexInt = ComplexInt Int Int
    deriving (Show)
```

Use `Parser` *to implement* `Read ComplexInt`*, where you can accept either the simple integer syntax* `"12"` *for* `ComplexInt 12 0` *or* `"(1,2)"` *for* `ComplexInt 1 2`*, and illustrate that* `read` *works as expected (when its return type is specialized appropriately) on these examples. Don't worry (yet!) about the possibility of minus signs in the specification of natural numbers.*

This is cool stuff, and we can and will take this style of parsing very far. This isn't our father's Fortran, nor indeed our high school Java or Scheme.

With all of this work behind us, the end product is remarkably concise and powerful parser combinator library that requires less than a full page of code. Just to be clear here: we haven't written a *parser* in less than a full page of code, we've written a *module for writing parsers* in less than a full page of code. And we're not done yet.

- `Parser.hs`, as above.

Ἀπὸ μηχανῆς θεός

Remember back when we remarked that original definition of `Parser` looked like a mashup of a `State` monad and a `[]` monad? We can breathe life into this observation by noting that `Control.Monad.State` ultimately defines `State` in terms of the monad transformer `StateT`. Looking at the definition of `StateT`, we have

```
newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }
```

So, if we define

```
type Parser = StateT String []
```

we at least get the type right. But here's where the wild magic begins: if we look at the instances provided by `StateT`, we'll see:

```
Functor m => Functor (StateT s m)
(Functor m, Monad m) => Applicative (StateT s m)
(Functor m, MonadPlus m) => Alternative (StateT s m)
Monad m => Monad (StateT s m)
MonadPlus m => MonadPlus (StateT s m)
```

In the current context, `m` is `[]`, and we know that `[]` is a `Functor`, a `Monad`, and a `MonadPlus`, and as such, our `Parser` type will be an instance of `Functor`, `Applicative`, `Alternative`, `Monad`, and `MonadPlus` for free! Moreover, once we have these instances, we can use combinators for these type classes to simplify our base definitions, e.g.,

```
string :: String -> Parser String
string = mapM char
```

Pulling this all together, we have

- `ParserFinal.hs`, via Monad Transformers, in 22 lines.

Not a parser, but a powerful *module for writing parsers* . Try that in C++ or Java.

The sudden appearance of monad transformers here very much has the flavor of a *deus ex machina*, and you're all very much to be forgiven if you're not only saying to yourself, "I didn't see that coming," but "Is he really expecting that I'm going to be able to pull this kind of wild rabbit out of my hat?" No. Or at least, not yet. In truth, I gave a predecessor version of this lecture for three consecutive years without invoking this wild magic, only realizing in the summer before the fourth year that not only could the previously mysterious monad transformers be used here, but they resulted in a `Parser` class that was both more concise and more powerful. It was a programmer's epiphany—to paraphrase the great Hungarian combinatorialist Paul Erdös, I felt as if I'd been granted a peak at a page of code from God's own git repository.

But that said, you're unlikely to find what you're not looking for. We'll do some more examples that use monad transformers later in the quarter, and will even take a quick peek at some of the wild magic that the Monad Transformer Library (mlt) uses to simplify both the coding and the use of monad transformers.

Historical Note

The use of parser combinators to build backtracking parsers has a long history in functional programming. A particularly formative (and illustrative) article is Functional Parsers (PDF) by Joroen Fokker. Note that Fokker's code isn't actually Haskell—it's Gofer, a predecessor language. You shouldn't have any trouble with translation. Another good source for this material is Chapter 13 of Hutton (2nd edition).

21.2 Practical Functional Parsing

Text.ParserCombinators.ReadP

Having gone to all of the trouble of writing our own parser combinator library, we'll now set it aside and use a similar but much more developed and efficient combinator library that comes with the Haskell Platform: `Text.ParserCombinators.ReadP`. Note that there are several other parser libraries for Haskell, and that `ReadP` isn't necessarily the most popular, it's just the best pedagogical fit for us right now.

First, the good news. May familiar parser combinators from `Parser` are available to us in `ReadP`, notably, `satisfy`, `char`, and `string`, as well as many other conveniences. `ReadP` also belongs to all of the type classes that you'd expect: `Functor`, `Applicative`, `Alternative`, `Monad`, `MonadFail`, and `MonadPlus`.

The bad news is that when `ReadP` was written, functional parsing was usually referred to as *monadic* parsing, and many of the other type classes hadn't yet been formalized. So there is an occasional awkwardness, in that `ReadP` in some ways anticipated `Applicative`, `Alternative` and other type classes that didn't exist when it was written, and so in some ways is duplicative of them. In particular, `ReadP` defines `many` with the same meaning as `Applicative`'s `many`, but conflicting with it; and `many1` which is synonymous with `Applicative`'s `some`. `ReadP` defines `(+++)` which is synonymous with `<|>`, while `pfail` is synonymous with `empty`.

A mildly annoying difference is that the "run" function is `readP_to_S` rather than `runParser`. If it's sufficiently annoying, just define `runParser = readP_to_S`, otherwise live with it. We'll live with it.

`ReadP` was written with a concern for efficiency, and this leads us to consider a couple of new parser combinators.

```
(<++) :: ReadP a -> ReadP a -> ReadP a
```

is described as a left-biased alternative. The idea is that `pa <++ pb` is a parser that tries `pa` first. If it returns any results, those are the results, but if it fails, it next tries `pb`. We can imagine that `(<++)` is defined as

```
ap <++ bp = Parser $ \s ->
    case runParser ap s of
        [] -> runParser ab s
        rs -> rs
```

although the actual definition is quite different, as the underlying representations are more complex. Next, we have:

```
munch :: (Char -> Bool) -> ReadP String
```

This function returns the largest substring of the string to be parsed that satisfy the predicate. This is subtly different from `many (satisfy p)`, cf.

```
> readP_to_S (munch (=='a')) "aaabb"
```

```
[("aaa","bb")]
> readP_to_S (many $ satisfy (=='a')) "aaabb"
[("","aaabb"),("a","aabb"),("aa","abb"),("aaa","bb")]
```

This often doesn't make any difference at all (as only the longest version can be valid), but there can be a quadratic savings in not generating the shorter forms.

Note that there's a variant `munch1` of `munch` which succeeds only if there is at least one character in the result.

Becoming an Expert

The module `Text.ParserCombinators.ReadP` contains many functions beyond those that we implemented in our `Parser` module. A good strategy for building expertise in Haskell is to read through the documentation for modules like `Text.ParserCombinators.ReadP`, and try to implement the various functions it contains. Then, follow the links to the source code, and see how Haskell experts have implemented them. This will give you practice, the opportunity to read and learn from experts, and also a close acquaintance with the facilities the module provides.

Example: Duplication With Variation

It's easy to duplicate a file, that's what the Unix `cp` utility is for. But what if we want to produce a bunch of near-duplicates of a given text, i.e., we want those duplicates to vary somehow? If the variations are sufficiently simple, we can write a program that generates all of the variations, and produces the duplicated text. But often, we'll have some sort of database, and the variations will amount to the rows of one of the relations of that database. In this case, we'll want to be able to handle a file that contains our data, and CSV (comma separated values) with headers is a natural choice.

If we're only interested in duplicating a particular fixed base text, we incorporate that text into the logic of the program we're writing. But this is the sort of job where the text to be duplicated tends to vary over time too, and this leads to a preference to "move it out of the code." So we'll create a simple text format to describe the underlying text.

As an aside, the code we're studying is a simplified version of the `spam` program that Professor Kurtz has used to send out wait-list notifications and similar bulk emailings.

`DupV.Template`

Our templates will consist of ordinary text files, in which set-braces are used to indicate a placeholder to be filled in from our data file. For example, one of the test files for this program contains:

```
The Arabic numeral {arabic} and the Roman numeral {roman} both represent {english}.
```

The idea here is that `{arabic}`, `{roman}`, and `{english}` will all be filled in with data from our CSV file.

Our internal representation of a `Template` will be:

```
data Template = Template { items :: [TemplateItem] }

data TemplateItem
    = Literal String
    | Variable String
```

Our first programming task is to parse the input file, e.g., obtaining

```
Template {items =
    [ Literal "The Arabic numeral "
    , Variable "arabic"
    , Literal " and the Roman numeral "
    , Variable "roman"
    , Literal " both represent "
    , Variable "english"
    , Literal ".\n"
    ]}
```

We'll use `ReadP`:

```
parseTemplate :: ReadP Template
parseTemplate = Template <$> many parseTemplateItem where
    parseTemplateItem = parseLiteral <++ parseVariable
    parseVariable = Variable <$>
        (char '{' *> munch1 isAlphaNum <* char '}')
    parseLiteral  = Literal <$> munch1 (`notElem` '{}')
```

This requires a bit of explanation. We use `many` to parse a list of items. The items to be parsed are `TemplateItems`, which come in one of two forms. We write sub-parser for each, using `(<++)` to avoid a parse that must fail if the preceding parse succeeded. There's some subtly in both subparsers.

Our use of `munch1` in `parseLiteral` is important! If we just used `munch`, the parser would succeed in producing a `Literal ""` without reading any input, and in the context of `many` would result in infinitely many `Literal ""`'s at the end of the string (as well as at each transition to a variable). Pragmatists will note that parsers should always consume some input, otherwise bad things can happen!

Our definition of `parseVariable` uses two new applicative operators, `<*` and `*>`. These are sequencing operators, which return the value of the first, or second, argument respectively. Note that the relational operator "points towards" the applicative value we will keep. In the old days, we'd have written `parseVariable` in a monadic style, which is superficially quite different, but essentially the same:

```
parseVariable = Variable $ do
```

```
        char '"'
        result <- munch1 isAlphaNum
        char '"'
        pure result
```

It is perfectly reasonable to write parsers this way when you're trying to figure things out, but it's useful to keep in mind that if a bound variable never appears on the right-side of a binding, it should be possible to rewrite the code applicatively. With practice, the applicative forms come first.

We can use the `ReadP` parser to make `Template` an instance of the `Read` type class,

```
instance Read Template where
    readsPrec _ = readP_to_S parseTemplate
```

and then introduce the convenience function

```
loadTemplateFile :: FilePath -> IO Template
loadTemplateFile path = read <$> readFile path
```

Finally, we have a simple bit of code for instantiating a template, given a list of keys and a list of values:

```
instantiate :: Template -> [String] -> [String] -> String
instantiate template header record =
    let dict = Map.fromList $ zip header record
        fill (Literal s) = s
        fill (Variable v) = dict ! v
    in concat . map fill . items $ template
```

This is simple, yet powerful code. We build a `Map`, an efficient structure for manipulating key-value pairs, out of an association list built out of the keys and values. We define a `fill :: TemplateItem -> String` that "evaluates" a `TemplateItem` in the context of the `Map` we just built. Finally, we `map` the `fill` function across the `items` of the `template`, obtaining a `[]` of `String`, which we flatten using `concat`.

`DupV.SimpleCSV`

The CSV format is deceptively simple, and parsing any individual CSV file is usually straightforward. A CSV file consists of a sequence of newline terminated *records*, where each record is a sequence of comma separated *fields*, and each field is just a `String`. So, what's hard? CSV began life as an *informal* format. There is an Internet Engineering Task Force (IETF) Received for Comment (RFC, a.k.a., and internet standards document) RFC 4180 that describes CSV, but that "standard" is based on reverse engineering files that claimed to be CSV, so the cart of practice came before the horse of specification. And my experience of CSV includes files that don't meet the "standard" of RFC 4180, a very real *caveat* for anyone who *emptor*'s it. So writing a good, *general* CSV parser has real challenges.

We'll start by writing a simplified ReadP parser, and then deal with some of the complexities of CSV.

```
newtype CSV = CSV { content :: [[String]] }

instance Read CSV where
    readsPrec _ = readP_to_S parseSimpleCSV

parseSimpleCSV :: ReadP CSV
parseSimpleCSV = CSV <$> record `endBy` newline <* eof where
    newline = string "\n"
    record = field `sepBy` char ','
    field =  munch (`notElem` ",\n")
```

There's a bit of complexity here.

As before, we're going to take advantage of the `Read` type class, and so introduce use `newtype` to wrap the type we're interested in. But `read` can ignore white space at the end of its input, and in the case of CSV parsing, this can result in an ambiguity, so we use the `eof` parser from `ReadP` to ensure that the entire input string is read.

The `endBy` parser combinator builds lists of values from a parser for the values and their terminators. There is a similar `sepBy` parser combinator for building lists of values from a parser for the values and a parser for the separators. The expression ``record `endBy` newline`` is a parser for a list of records, each of which must be terminated by a newline, while ``field `sepBy` char ','`` is a parser for a list of fields separated by commas.

This would be good enough, if

- we were only going to run this code on Unix-like systems, and
- we never needed a comma or a newline within a field value.

Unfortunately, these are not assumptions we want to make. So there's a bit of work to do.

Newlines

Operating systems don't agree as to what precise sequence of characters constitutes a *newline*. Unix uses a bare linefeed (LF) `"\n"`, a.k.a. ASCII 012 (that's an octal codepoint). MacOS Classic used a bare carriage return (CR) `"\r"`, a.k.a., ASCII 015. MacOS X follows the Unix convention as befits its Unix foundations. Finally, Windows follows a convention that is as old as teletype machines, and relies on a CRLF pair, `"\r\n"`. And CSV is such a basic file format that it could have come from anywhere, and may have passed through many hands, so we can't even be entirely confident that the same newline convention will be used consistently *within* a single file, although we expect that exceptions to this will be rare.

We'll take a pragmatic point of view. We may have to deal with Windows, Linux, and MacOS X, but we're not going to have to deal with MacOS Classic. So we can define

```
newline = string "\n" <++ string "\r\n"
```

Yes, this is biased towards Unix. I like it that way.

Commas

To deal with the issue of commas and newlines within fields, CSV has the notion of a quoted field: this is a field that begins and ends with a `'"'`, and can contain anything but a `'"'` within it. To avoid any ambiguity, CSV forbids simple (non-quoted) fields from containing a `'"'`. Thus

```
field = quotedField <++ simpleField
simpleField = munch (`notElem` ",\n\"")
quotedField = char '"' *> munch (/='"') <* char '"'
```

Of course, this just trades one problem for another. We can now have commas in our fields, but we can't have double-quotes. To deal with this, CSV further allows a quoted double-quote within a quoted field. How do you quote a double-quote? In CSV, you repeat it. Thus, e.g.,

```
"He said, ""Foo!"""
```

To accommodate this, we'll replace the `munch (/='"')` within the definition of `quotedField` as follows:

```
quotedField = char '"' *> many quoteChar <* char '"'
quoteChar = satisfy (/= '"') <++ (string "\"\"" $> '"')
```

Note the useful `$>` operator from `Data.Functor`. The expression `(string "\"\"" $> '"')` means that if we successfully parse a repeated double-quote, the value we should return is a single double-quote.

Exercise 21.3 *The use of* `many` *in the definition of* `quotedField` *introduces the very problem that* `munch` *was written to solve. The problem is that we can't use* `munch`*, as it has the wrong type (it acts on a character predicate, rather than general parser). Write the function*

```
greedy :: ReadP a -> ReadP [a]
```

which greedily parses a sequence of values, returning only maximal sequences. Note that "maximal" in this context does not mean that list has the greatest possible length, nor that it ingests the maximum number of input characters, but rather that it cannot be extended. E.g.,

```
> readP_to_S (greedy (string "a" +++ string "ab")) "abaac"
[(["a"],"baac"),(["ab","a","a"],"c")]
```

Hint: the `(<++)` *parser combinator is very helpful!*

Main

Our `main` is

```
main :: IO ()
main = do
    args <- getArgs
    case args of
        [variationFile,templateFile] -> do
            template <- loadTemplateFile templateFile
            header:variations <- content <$> loadSimpleCSV variationFile
            putStr . concat . (`map` variations) $ \variation ->
                instantiate template header variation
        _ -> do
            hPutStrLn stderr 'Usage: spam variationFile templateFile'
            exitFailure
```

This is mostly straightforward.

Real Programming

There is a good overview of this process on the `HaskellWiki` at `haskell.org`, How to write a Haskell program. The name notwithstanding, this wiki page describes how to package and distribute a Haskell program or library.

I recommend using `git` for version control, `cabal` for builds, and `haddock` for documentation.

Real programs often try to move as much of their data into data files as possible (we've done this), and as much of their internal logic into modules organized around basic data structures an the algorithms that support them. This simplifies code reuse.

Real programs also require a means of distribution. For this particular program, the sources are on GitHub: stuartkurtz/DupV, and the program can easily be downloaded and installed from there.

Exercise 21.4 *One way this program can be made much more useful is to add a number of pre-defined variables, e.g., the time and date of processing. Modify the* `dupv` *program to include such a feature.*

Part III

Towards the Real World

Chapter 22

Monad Transformers

22.1 Maybe Monad Transformer is Not So Scary

(Chapter contributed by RC)

[RC: See https://www.classes.cs.uchicago.edu/archive/2023/winter/22300-1/notes/maybe-monad-transformer/]

22.2 StateT Example: Sudoku

[RC: See http://cmsc-16100.cs.uchicago.edu/2021-autumn/Lectures/26/sudoku.php]

22.3 MTL: Implementation

Today I'd like to review the monad transformer library (mtl) with an eye to its overall construction.[1] There is some very clever engineering here, and you can learn a lot from the careful study of the work of masters. But there is another reason. Some of the cleverness leaks out, in that the documentation of the unadorned `State` monad and similar monads can be confusing to the uninitiated. For starters, if you look where you expect in the documentation to be in `Control.Monad.State`, you get pointed to `Control.Monad.State.Lazy`, but it starts out by talking about the type class `MonadState`, which is something different than the `State` monad, although they seem somehow related. And then, the documentation for `State` and `StateT` doesn't contain links to source code, and that makes us unhappy. But then, if you dig around a bit more, you'll find `Control.Monad.Trans.State`, which points us to `Control.Monad.Trans.State.Lazy`, and *that* module contains what seem to be duplicate definitions of `State` and `StateT`, this time with source links, but `MonadState` is now nowhere to be found. But what about our imports? We've been importing `Control.Monad.State`, and it's been working for us. Should we keep doing that? Should we import `Control.Monad.Trans.State`, or maybe even `Control.Monad.Trans.State.Lazy`? Or should we just go home, have a good cry, and reconcile ourselves to a life of writing JavaScript? No, not *that*. The short answer is that we should import `Control.Monad.State`, but get our documentation from `Control.Monad.State.Lazy`. The rest of the lecture constitutes the long answer.

The fundamental idea behind the mtl is that derived types can preserve structure (i.e., capabilities of the types from which they're derived), and that preserved structure can be realized programmatically via automatically derived instances of typeclasses. This has the effect at library use time of considerable functionality "coming along for free," as we saw when we implemented `Parser.hs` using monad transformers: not only did we get put and friends (as you'd expect from something built using `StateT`), we got `Functor`, `Applicative`, `Monad`, and other typeclass instances.

`IdentityT`

There is, unsurprisingly, an `IdentityT` monad transformer. The definition of `IdentityT` can't be surprising:

```
newtype IdentityT m a = IdentityT { runIdentityT :: m a }
```

This is just an untransformed `m a` in a (virtual) `IdentityT` box. We'll look at `IdentityT` as a simple setting in which we can get our bearings. First of all, `IdentityT` is a monad transformer, and so is itself an instance of the `MonadTrans` class:

```
class MonadTrans t where
    lift :: Monad m => m a -> t m a
```

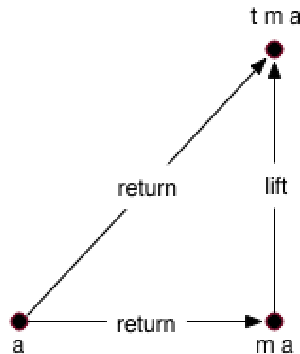
i.e., `IdentityT` possesses a `lift` function that maps the untransformed monad `m a` into the transformed monad `IdentityT m a`. Instances `t` of `MonadTrans` are expected to satisfy a couple of laws:

- `return = lift . return`

[1] RC: This chapter was last updated in 2016.

- `lift (m >>= f) = lift m >>= (lift . f)`

These look mysterious, in part because the types are hidden from us, but they're well motivated and so are easy to understand. E.g., if we start with a value object of type `a`, there are two natural ways to embed it into the transformed monad `t m a`. The first is directly via `t m`'s `return` function, the second is indirectly, from `a` to `m a` via `m`'s `return` function, and from there to `t m a` via `t m`'s `lift`. The first law says that two ways must produce the same value in `t m a`.



The second law says something similar. If we have a binding `m >>= f` in the untransformed `m a`, there are two natural ways that we can go about lifting the result into the transformed monad `t m a`. The first, as before, is directly via `t m a`'s `lift` function. The second is by lifting `(>>=)`'s arguments, resulting in `lift m` and `lift . f`, and applying `(>>=)` in the transformed monad `t m a`. Again, these two different natural ways of lifting a binding must produce the same value in `t m a`.

The instance definition for `MonadTrans IdentityT` is straightforward—we just take a value in `m a`, and drop it in a (virtual) `IdentityT` box.

```
instance MonadTrans IdentityT where
    lift = IdentityT
```

More complicated monad transformers will have more complicated definitions of `lift`.

Next up, we have instances (when appropriate) for `IdentityT m` of `Functor`, `Applicative`, `Foldable`, and of course `Monad`, `MonadPlus`, and a few other type classes we haven't met yet. These are all pretty simple, with all of the activity taking the form of traffic in (virtual) boxing and unboxing. To this end, the following helper functions are defined:

```
mapIdentity :: (m a -> n b) -> IdentityT m a -> IdentityT n a
mapIdentity f = IdentityT . f . runIdentityT

lift2IdentityT ::
```

```
    (m a -> n b -> p c) -> IdentityT m a -> IdentityT n b -> IdentityT p c
lift2IdentityT f a b = IdentityT (f (runIdentityT a) (runIdentityT b))
```

Note that `lift2IdentityT` is just a binary version of `mapIdentity`, it's impressive type notwithstanding, and both simply lift functions on untransformed monads to functions on transformed monads. We're now ready to define some instances:

```
instance (Functor m) => Functor (IdentityT m) where
    fmap f = mapIdentityT (fmap f)

instance (Foldable f) => Foldable (IdentityT f) where
    foldMap f (IdentityT a) = foldMap f a

instance (Applicative m) => Applicative (IdentityT m) where
    pure x = IdentityT (pure x)
    (<*>) = lift2IdentityT (<*>)

instance (Monad m) => Monad (IdentityT m) where
    return = IdentityT . return
    m >>= k = IdentityT $ runIdentityT . k =<< runIdentityT m
```

The most interesting thing here is the definition of `(>>=)` in the monad instance. Note the use of a right-to-left version `(=<<)` of bind, and how this mixes better with `(.)`. We solved the same problem earlier, but by resolving the follow of compositions and binds in the other direction, using `(>>>)`. The critical thing to notice is how the we're using the `(>>=)` from the untransformed monad to define `(>>=)` in the transformed monad. Our code is all about making sure that things get boxed or unboxed as needed. Now, let's look at a couple of more interesting monad transformers.

`StateT`

The `StateT` monad transformer has everything going for it so far as this lecture is concerned. It's powerful and even somewhat familiar. But there are a few interesting twists and turns in its implementation, and understanding it will go a long way to helping you understand how the monad transformer library as a whole is implemented, and *why*. We'll start by considering the definition:

```
newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }
```

This type isn't really surprising, the only question being "where do we stick the inner monad `m` into the state transformation function?" There are at least three plausible answers,

- `m (s -> (a,s))`
- `s -> m (a,s)`

- `s -> (m a,s)`

Perhaps unsurprisingly, the choice taken is one that can't be obtained by simply stacking untransformed monads. Of course, we use `StateT` as a way to define functions that we're latter going to want to run via `runStateT` or one of its friends. But there's a trickiness with any of these, because the result returned is going to be in terms of the untransformed monad `m`. This is explicit in `runStateT :: StateT s m a -> s -> m (a,s)`, but requires a bit more work with the others, e.g.,

```
evalStateT :: (Monad m) => StateT s m a -> s -> m a
evalStateT m s = do
    (a, _) <- runStateT m s
    return a
```

Note here that I'm pulling a few minor cheats on you, as I will throughout this lecture (and indeed, already had). This is the implementation of `evalStateT` from `Control.Monad.Trans.State.\textit{Strict}`, not `Lazy`. The difference is a minor technicality in pattern matching that's not important for this lecture. I'm making a few other pedagogical simplifications too, but nothing you can't puzzle out from the actual code.

Of course, we want types that result from applying `StateT s` to a monad `m` to be monads themselves, hence

```
instance Monad m => Monad (StateT s m) where
    return = lift . return
    m >>= f = StateT $ \s -> do
        (a,t) <- runStateT m s
        runStateT (f a) t
```

Here, the focus should be on the definition of `(>>=)`, in which the body of the lambda form `\s -> ...` is an object of type `m (a,s)`. As before, we rely on `(>>=)` in the untransformed monad `m (a,s)` (here hidden by `do` sugar) to extract the `(result,newState)` pair, which is then handed off to `runStateT` again. This is essentially the same as the naïve code for `(>>=)` in `State`, albeit with a monadic binding `(>>=)` replacing a let binding. We'll see this again.

Instances of `Functor`, `Applicative`, `Alternative`, and `MonadPlus` and similar type classes are also provided, when the untransformed monad has the corresponding instance. It's worth noting the that in the particular case of `Applicative`, it suffices that the inner type (not necessarily a monad) merely be a functor.

We make `StateT` into an instance of a `MonadTrans` by providing an appropriate `lift`:

```
instance MonadTrans (StateT s) where
    lift m = StateT $ \s -> do
        a <- m
        return (a,s)
```

Now that Haskell requires that every monad is a functor, we can have write `lift` more concisely as

```
instance MonadTrans StateT where
    lift m = StateT$ \s -> (`(,)` s) <*> m
```

The implementations of get, put and modify, are built on that of state:

```
state :: (s -> (a,s)) -> StateT s m a
state f = StateT $ return . f
```

i.e., state puts in a StateT box the result of applying the function f to a state, obtaining a (result,newState) pair, which is then lifted into the monad m (a,s) via return. With this:

```
get :: StateT s m s
get = state $ \s -> (s,s)

put :: s -> StateT s m ()
put t = state $ \s -> ((),t)

modify :: (s -> s) -> StateT s m ()
modify f = state $ \s -> ((),f s)
```

Now we encounter that extra layer of trickiness, alluded to at the beginning of the lecture. The class definition for StateT isn't in Control.Monad.State, or even Control.Monad.State.Lazy, it's in Control.Monad.Trans.State.Lazy, for two reasons. First of all, Control.Monad.State.Lazy relies on functional dependencies, an extension to Haskell implemented in GHC, but not in all Haskell systems. Non-GHC systems have to include Control.Monad.Trans.State, and lose the advantages of the new facilities in Control.Monad.State.Lazy. Second, for GHC, we're going to hide the definitions of put, get, and state from Control.Monad.Trans.State.Lazy, exposing instead the definitions that come from the MonadState type class:

```
class Monad m => MonadState s m | m -> s where
    get :: m s
    put :: s -> m ()
    state :: (s -> (a, s)) -> m a
```

Note the *functional dependency* (aka, fundep) m -> s| in the class declaration. This is indicates a commitment from the programmer to GHC that there can't be distinct s and t for which there are MonadState s m and MonadState t m instances.To define an instance of MonadState, it is necessary to define either get and put, or state. The instance for StateT s m is particularly easy:

```
import qualified Control.Monad.Trans.State.Lazy as Lazy (StateT, get, put, state)

instance Monad m => MonadState s (Lazy.StateT s m) where
```

```
        get = Lazy.get
        put = Lazy.put
        state = Lazy.state
```

But the virtue of MonadState becomes apparent if we take a look back at IdentityT. The punchline here is the following instance declaration:

```
    instance MonadState s m => MonadState s (IdentityT m) where
        get = lift get
        put = lift . put
        state = lift . state
```

The point here is that if m is a state monad with state s, then IdentityT m is also a state monad with state s. Or, to say the same thing in slightly different words, IdentityT preserves the MonadState s property. And this saves us from having to remember how many layers deep the StateT actually is, and therefore how many times we have to apply lift to get the state in or out of the monad. We can just use get, put, state, and their friends on the transformed monad, is if it were a plain old-fashioned state monad. That is very convenient.

WriterT

The Writer monad is usually used to implement logging. I think of a Writer as being half of a State, in that we're producing something like a state on output, we're just not consuming state on input. And like the State monad's state, the log we're building is hidden.

We'll do a quick example once we made some introductions:

```
    class Writer w a = { runWriter :: (a,w) }
```

Our first problem is "how do we make Writer into a monad." The hard part of the job is usually in defining (>>=). In the state monad, (>>=) handled the states by hooking up the output state of the first argument's state-transition function to the input state of the second monad, which we got by applying the second argument, a Kleisli arrow, to the value that came through the binding. That way, we ended up with one input state, and one output state, and all was well. This can't work the same way with writers, because there's no input on the second monad to attach the output of the first. The plumbing is just different. Instead, we'll combine the logs, and this requires some sort of binary operator, which is going to have to be associative if (>>=) is to be associative. The implementation of return will require a identity. So we're going to need to require that w is a Monoid, because that's precisely what monoids give us: an associative binary operator with identity.

```
    instance (Monoid w) => Monad (Writer w) where
        return a = Writer $ (a,mempty)
        m >>= f = Writer $ let
                               (a,w1) = runWriter m
```

```
                    (b,w2) = runWriter $ f a
                in (b,w1 <> w2)
```

The practical use of writer monads relies, much like state monads, on a scant handful of functions that enable us to interface with the monad, without dwelling on its implementation. The first of these is `tell`, which is similar to `put`, in that it writes its argument to the log:

```
tell :: (Monoid w) => w -> Writer w ()
tell w = Writer ((), w)
```

The `listen` function is analogous to `State`'s `get`, but different in that it takes a writer as an argument, and it returns the result returned by the argument monad along with its log:

```
listen :: (Monoid w) => Writer w a -> Writer w (a,w)
listen m = Writer $ let (a, w) = runWriter m
                    in return ((a, w), w)
```

The `censor` function is analogous to `State`'s `modify`, in much the same way that `listen` is analogous to `put`, in that it takes a writer monad as an argument, and applies the argument function to its argument's log, packaging the result.

```
censor :: (Monoid w, Monad m) => (w -> w) -> Writer w a -> Writer w a
censor f m = Writer $ let (a, w) = runWriter m
                      in return (a, f w)
```

The following, somewhat contrived, example illustrates the use of `Writer`:

```
import Control.Monad.Writer

type LogApply = Writer [String]

logFunction :: (Show a, Show b) => String -> (a -> b) -> a -> LogApply b
logFunction name f = \a -> do
    let b = f a
    tell $ ["Applying " ++ name ++ " to " ++ show a ++ " resulting in " ++ show b ++ "."]
    return b

logSquare, logDouble, logSuccessor, logPredecessor :: (Show n,Num n) => n -> LogApply n
logSquare = logFunction "square" (^2)
logDouble = logFunction "double" (*2)
logSuccessor = logFunction "successor" (+1)
logPredecessor = logFunction "predecessor" (subtract 1)
```

```
computation :: LogApply Integer
computation = return 10 >>= logPredecessor
                        >>= logSquare
                        >>= logSuccessor
                        >>= logDouble

main :: IO ()
main = do
    let log = execWriter computation
    putStr $ unlines log
```

When run, this produces the output:

```
Applying predecessor to 10 resulting in 9.
Applying square to 9 resulting in 81.
Applying successor to 81 resulting in 82.
Applying double to 82 resulting in 164.
```

This is, perhaps, not truly inspiring, but it does illustrate the sort of thing that `Writer` makes easy.

Of course, this whole discussion about `Writer` glosses over the same sort of complexities we saw with `State`. In reality, there's a `WriterT` monad transformer, where

```
newtype WriterT w m a = WriterT { runWriterT :: m (a,w) }
type Writer w = WriterT w Identity
```

In this case the contribution of `WriterT` isn't in the type (which we could arrange by ordinary stacking), but in the plumbing,

```
instance (Monad m, Monoid w) => Monad (WriterT w m) where
    return a = WriterT $ return (a,mempty)
    m >>= g = WriterT $ do
        (a,w1) <- runWriterT a
        (b,w2) <- runWriterT (g a)
        return (b,w1 <> w2)
```

Note how this adapts the definition we gave for `(>>=)` for an ordinary writer, by replacing the let bindings by monadic bindings, packaging our results with `return`.

Similar changes get made to `tell`, `listen` and `censor`, e.g.,

```
listen :: (Monad m, Monoid w) => WriterT w m a -> WriterT w m (a,w)
```

```
    listen m = Writer $ do
        (a, w) <- runWriter m
        return ((a, w), w)
```

As with `State`, these types and functions get defined in `Control.Monad.Trans.Writer.Lazy`. And in `Control.Monad.Writer.La`
contains the `MonadWriter` type class, and associated functions and instances:

```
class (Monoid w, Monad m) => MonadWriter w m | m -> w where
    writer :: (a,w) -> m a
    tell   :: w -> m ()
    listen :: m a -> m (a, w)

censor :: MonadWriter w m => (w -> w) -> m a -> m a
censor f m = do
    (a,w) <- runWriterT m
    return (a, f w)
```

Naturally, monads built by the `WriterT w` monad transformer are instances of `MonadWriter w`:

```
instance (Monoid w, Monad m) => MonadWriter w (Lazy.WriterT w m) where
    writer = Lazy.writer
    tell   = Lazy.tell
    listen = Lazy.listen
```

Likewise, any monad `m` with a `MonadWriter w m` instance remains a `MonadWriter w` when wrapped with `IdentityT` or a `StateT s`:

```
instance MonadWriter w m => MonadWriter w (IdentityT m) where
    writer = lift . writer
    tell   = lift . tell
    listen = Identity.mapIdentityT listen

instance MonadWriter w m => MonadWriter w (Lazy.StateT s m) where
    writer = lift . writer
    tell   = lift . tell
    listen = Lazy.liftListen listen
```

Likewise, any monad `m` with a `MonadState s m` instance remains a `MonadState s m` when wrapped with a `WriterT w`, and the associated code is found in `Control.Monad.State.Lazy`.

And that's the way the whole mlt is built. Associated with each monad transformer is a type class that defines the properties that transformer adds. For each pair of a transformer and another transformer's associated type class, there's an instance of that type class, when possible, that preserves that property in a transformed monad. The

exception to all of this is `IO`, as it so often is, but that's a puzzle for another day.

Chapter 23

Scalability

Programming requires attention to *correctness* and *efficiency*. The dominant focus of this class has been on correctness, and on using the type system of Haskell and the type classes of its library as aids to writing correct code. This reflects our ontology of programming: it's easier and more satisfactory to *tune* correct code so that it runs faster, than it is to *debug* fast code so that it runs correctly. Getting it right, in our view, is a *constraint*, it's something we must do. Making it fast is a *goal*, something we hope to do, and want to make progress towards.

In this, the `Document` type from our second `State` lecture fails the test of efficiency. It is a common failing, and one that's worth understanding. We build our document from front to back, in effect, building

```
(... ((s1 ++ s2) ++ s3) ... ++ sn)
```

Let's recall the implementation of `(++)`:

```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys
```

It is easy to show, by induction on the length of `xs`, that fully evaluating `x ++ y` requires $|x|$ new `(:)`'s. The cost of building these `(:)`'s is the dominant cost of evaluating the `(++)`.

Consider now the overall cost of building a string via `(++)`. For the sake of simplicity, we'll consider `(((s1 ++ s2) ++ s3) ++ s4)`.

operation	cost
`s1 ++ s2`	$\lvert s_1\rvert$
`(s1 ++ s2) ++ s3`	$\lvert s_1\rvert + \lvert s_2\rvert$
`(((s1 ++ s2) ++ s3) ++ s4)`	$\lvert s_1\rvert + \lvert s_2\rvert + \lvert s_3\rvert$

So the overall cost is the sum of these operation costs, i.e., $3|s_1|+2|s_2|+|s_3|$. Contrast this to the cost of building the same string via the same operations, but with the `(++)`'s associated to the right, i.e, `s1 ++ (s2 ++ (s3 ++ s4))`:

operation	cost
`s3 ++ s4`	$\|s_3\|$
`s2 ++ (s3 ++ s4)`	$\|s_2\|$
`s1 ++ (s2 ++ (s3 ++ s4))`	$\|s_1\|$

So the overall cost of building the string this way is $|s_1| + |s_2| + |s_3|$, because each `(:)` we build contributes to the final answer, whereas we build a *lot* of ephemeral `(:)`'s when the `(++)`'s associate to the left.

This can make a difference! Building a moderately complex web page might involve thousands of calls to `string`, meaning that we end up re-copying the `(:)`'s associated with early calls to `string` thousands of times.

Fortunately, we can address this problem rewriting only a little bit of code, as there's an abstraction barrier between `Document` and `HTML`, so we need only consider the definition of `Document`, and make sure that the `string` and `render` functions are aligned with it. We will explore several different solutions. For the sake of simplicity, we won't do our usual code transformations, but will just present natural first-pass code.

Let Simon Do It

An efficient solution to problems like this is to rely on the fact that others have encountered them before, resulting in carefully engineered, widely-deployed, and so well-tested solutions. In this case, it's well known that `String` is inefficient, not only in the cost of operations like `(++)`, but also in terms of storage space. The module `Data.Text` uses a packed utf-16 representation of Unicode, while the `Data.Text.Lazy` module adds to this by considering chunked representations (i.e., the text is a sequence of packed utf-16 representations). We can then define:

```
import Data.Text.Lazy

type Document = State Text

{- Append a String to a Document. -}

string :: String -> Document ()
string t = modify $ \s -> append s (pack t)

{- Render a Document a as a String -}

render :: Document a -> String
render doc = unpack $ execState doc empty
```

Note that the `pack` and `unpack` functions convert between `String` and `Text` values, and that `append` is used to append `Text` values.

It turns out that there are a fair number of alternative representations for what we think of as list-like data, including `Data.Array` which allows efficient access to lists of fixed length, and `Data.Sequence` which allows for efficient access to both the front and back of a list. The existence of these alternative representations explains the existence

of many of the standard type classes, e.g., `Functor`, `Foldable`, etc., all of which capture standard list-programming techniques. To the extend that we write our list-based code using the functionality of these type classes, we abstract away from specific representation decisions, making it easier to transition from simpler to more complex (and efficient) data representations later on.

The Composition Trick, a.k.a., Difference Lists

This is a technique that is used in several places in the Haskell library. We build up the efficient right-associative use of `(++)` via a composition of functions that associates to the left. There's something deeply sneaky about this, as we're trading off adding content to beginning, for a promise to content to the end!

```
type Document = State (String -> String)
```

The idea is that the function we're building tacks the `String` we've built onto its argument.

```
{- Append a String to a Document. -}

string :: String -> Document ()
string t = modify $ \f -> f . (t ++)

{- Render a Document a as a String -}

render :: Document a -> String
render doc = execState doc id ""
```

Note in particular our implementation of `render`. We start the ball rolling by passing the identity function to `execState`, which represents an empty document. When we get the final function out, we apply it to the empty string, as there is no "rest of the string" to build.

Be Lazier!

The idea here is to defer the `(++)`'s to the end. So we build a list of strings to combine, but we organize that list from back-to-front, i.e., reversed in order from the document we intend to produce. This means that the actual joining together of the lists happens with `render`, rather than with `string`:

```
type Document = State [String]

{- Append a String to a Document. -}

string :: String -> Document ()
```

```
string t = modify $ \s -> t : s

{- Render a Document a as a String -}

render :: Document a -> String
render doc = concat . reverse . execState doc $ []
```

Reflections

All of these changes will turn a program that had quadratic run-time to one that has linear run-time. The tuned programs *scale*. But what is perhaps most remarkable about this exercise is that in each case, the new code ran correctly the first time! This had much to do with the work we put into our initial list-based solution, and in particular, the simplicity of the `Document` abstraction, and how thoroughly the rest of the code was decoupled from the particular representation choice we made within `Document`.

Programs work at different scales. Inefficiencies can creep in at any of these scales, e.g., in the algorithms used, in the data representations used, in the efficiency with which those representations are implemented. Traditional languages try to snuggle up to the underlying hardware (this is especially so in C), and naturally draw the programmer's attention to improving the efficiency of the chosen implementation, which is not a bad thing, except to the extent that it draws their attention away from larger inefficiencies in the algorithms and representation choices, and thereby creates an investment in the higher-level choices that were made. Haskell seems in practice to encourage more of a top-down approach, and this tends to deliver greater fruit faster.

Chapter 24

Seq and All That

Evaluation: the Term Model.

Haskell is defined in terms of lazy evaluation, which we can describe briefly as a computational plan that evaluates only what is necessary, and when it evaluates something, it only evaluates it once. For the most part, we can get away with this sort of informal description, but there comes a point when we can't, and need a better approximation.

Let's start by considering a simple function:

```
mySumR :: [Int] -> Int
mySumR [] = 0
mySumR (n:ns) = n + mySumR ns
```

This is definition of summation via a natural recursion, with the terminal `R` in `mySumR` intended to recall `foldr`. This code works well, at least until we try to sum a long list:

```
main = print $ mySumR [1..10^8]
```

This computes correctly, but it takes a surprisingly long amount of time to do so (a bit over 4.5 seconds, even at `-O2`). Profiling this code produces a somewhat surprising and disconcerting output:

```
18,495,737,248 bytes allocated in the heap
     1,390,456 bytes copied during GC
 1,674,375,728 bytes maximum residency (13 sample(s))
        31,592 bytes maximum slop
          2458 MB total memory in use (38 MB lost due to fragmentation)
```

The thing to notice here is the memory footprint: At some point in the computation, we were using 1.67GB of heap, and the total memory footprint reached 2.46GB. That's a whole heck of a lot of memory for what seems like a fairly simple task. A few moments of introspection reveals a possible cause: as we work our way through the list, we build an arithmetic expression to evaluate. But that expression grows to be essentially the size of the list we're processing, and we can't make progress on reducing that expression until we've built all of it. Ugly, especially as we know that traditional languages can do this in constant space, cf.,

```
def sum(ns):
    acc = 0
    for n in ns:
        acc += n
    return acc

print(sum(range(1,10**8+1)))
```

and we'd like to think that Haskell could too.

A simple analysis of this program shows that `sum` is accumulating the sum of from the front of the list to the back, unlike `mySumR` which built up an expression for the sum from back to front. Indeed, the iterative structure uses a register variable to hold a partially evaluated sum, rather than building up the stack of deferred function calls implied by a recursive function. There are at this point two fairly natural ways to think about this. Former Scheme programmers would think about tail-recursion and register variables, and note that `mySumR` isn't tail recursive, and so would look for a tail recursive analog to the Python program. Cynical students, i.e., students who know me, might read something into my emphasis of the `foldr`-like structure of `mySumR`, and wonder about what a `foldl` solution might look like. Either way, we end up with essentially the same code:

```
mySumL :: [Int] -> Int
mySumL = iter 0 where
    iter r [] = r
    iter r (n:ns) = iter (r+n) ns
```

A few moments of reflection around the notion of laziness, and we might even hope that our original program, updated to use `mySumL` rather than `mySumR` might run in *constant* space. After all, if the list is being built up lazily and consumed as it's being built, there will never be more than a single `(:)` active at a time. We're surprised and, indeed, horrified, to discover that this hopeful change make our code perform much *worse*, requiring almost a half-minute of run time and

```
26,620,313,904 bytes allocated in the heap
38,121,812,496 bytes copied during GC
 7,934,690,960 bytes maximum residency (16 sample(s))
   190,450,032 bytes maximum slop
         17973 MB total memory in use (0 MB lost due to fragmentation)
```

Ouch!! That's 7.94GB of heap, and 26.6GB of total memory allocation! Don't try this on your laptop!

Anyway, let's try to understand what's going on, by "playing the computer," and working our way through some modest sized examples (summing `[1..4]`). We'll use a simple term re-writing model, in which both the traditional eager evaluation and Haskell's lazy evaluation share the same re-write rules, but apply them in a different order. A preliminary issue is that Haskell's range notation `[a..b]` is syntactic sugar for `enumFromTo a b`, which we'll assume for our present purposes is defined as:

```
enumFromTo a b = case compare a b of
    LT -> a : enumFromTo (succ a) b
    EQ -> [b]
    GT -> []
```

If we were doing the usual eager model of calculation, we have a pattern in which we first fully evaluate the argument to `mySumR`:

```
mySumR [1..4]
==> mySumR (enumFromTo 1 4)          -- desugar [..]
==> mySumR (1:enumFromTo 2 4)        -- expand enumFromTo
==> mySumR (1:2:enumFromTo 3 4)      -- expand enumFromTo
==> mySumR (1:2:3:enumFromTo 4 4)    -- expand enumFromTo
==> mySumR (1:2:3:4:[])              -- expand enumFromTo
```

Note here that we're eliding a few steps, e.g., around the evaluation of `enumFromTo` in order to focus on the evaluation of `mySumR`.

Once the argument to `mySumR` is evaluated, we can replace the application by its definition, i.e.,

```
==> 1 + mySumR (2:3:4:[])            -- expand mySumR
==> 1 + (2 + mySumR (3:4:[]))        -- expand mySumR
==> 1 + (2 + (3 + mySumR (4:[])))    -- expand mySumR
==> 1 + (2 + (3 + (4 + mySumR [])))  -- expand mySumR
==> 1 + (2 + (3 + (4 + 0)))          -- expand mySumR
```

Finally, we'd simplify the resulting expression:

```
1 + (2 + (3 + (4 + 0)))
==> 1 + (2 + (3 + 4))
==> 1 + (2 + 7)
==> 1 + 9
==> 10
```

You can clearly see in this process the building up of the complete list `[1,2,3,4]`, and the building up of the full arithmetic expression `1 + (2 + (3 + (4 + 0)))`. Now Haskell doesn't work quite this way. Haskell is *lazy*. So

let's work through a lazy evaluation. It starts out much the same way:

```
mySumR [1..4]
==> mySumR (enumFromTo 1 4)              -- desugar [..]
==> mySumR (1:enumFromTo 2 4)            -- expand enumFromTo
```

At this point, the argument to `mySumR` is sufficiently defined (it's a cons) to determine that we're in the cons case of the definition of `mySumR`:

```
==> 1 + mySumR (enumFromTo 2 4)          -- expand mySumR
```

At this point we can't make progress on the addition, because its second argument isn't sufficiently well-defined, and we can't make progress on `mySumR`, because *its* argument isn't sufficiently well-defined, but we can make progress on `enumFromTo`, concluding as follows:

```
==> 1 + (mySumR (2 : enumFromTo 3 4))         -- expand enumFromTo
==> 1 + (2 + mySumR (enumFromTo 3 4))         -- expand mySumR
==> 1 + (2 + mySumR (3 : enumFromTo 4 4))     -- expand enumFromTo
==> 1 + (2 + (3 + mySumR (enumFromTo 4 4)))   -- expand mySumR
==> 1 + (2 + (3 + mySumR (4:[])))             -- expand enumFromTo
==> 1 + (2 + (3 + (4 + mySumR [])))           -- expand mySumR
==> 1 + (2 + (3 + (4 + 0)))                   -- expand mySumR
...
==> 10
```

Notice that, as our intuition earlier suggested, there's only a single `(:)` present in any of terms of the reduction sequence. This means that we can garbage collect the list more-or-less as we construct it, and the list contributes only a constant amount to our maximum active memory. But we're not so fortunate with the arithmetic expression. Too bad.

One thing to note here is that the results of eager and lazy evaluation are the same. This isn't an accident, but instead is a consequence of the Church-Rosser Theorem of the λ-calculus, which states that any two sequences of reductions to a term t result in terms t_1 and t_2 that can be "brought back together" by additional reductions. This is sometimes called the *confluence property*. As normal forms can't be further reduced, this means that if two normal forms result from sequences of reductions applied to the same initial lambda-term, they must be equal. We'll have a bit more to say about this later.

So let's consider `mySumL`. Again, we'll first do this via eager evaluation, which begins much like the `mySumR` case by expanding the full list:

```
mySumL [1..4]
==> mySumL (enumFromTo 1 4)              -- desugar [..]
==> mySumL (1:enumFromTo 2 4)            -- expand enumFromTo
```

```
==> mySumL (1:2:enumFromTo 3 4)          -- expand enumFromTo
==> mySumL (1:2:3:enumFromTo 4 4)        -- expand enumFromTo
==> mySumL (1:2:3:4:[])                  -- expand enumFromTo
```

At this point, we expand `mySumL` in terms of its local `iter` procedure, which we'll indicate via `mySumL.iter`:

```
==> mySumL.iter 0 (1:2:3:4:[])           -- expand mySumL
==> mySumL.iter (0+1) (2:3:4:[])         -- expand mySumL.iter
```

Now, eager evaluation steps in at this point, and forces the full evaluation of the first argument to `mySumL.iter`:

```
==> mySumL.iter 1 (2:3:4:[])             -- simplify addition
```

The evaluation continues

```
==> mySumL.iter (1+2) (3:4:[])           -- expand mySumL.iter
==> mySumL.iter 3 (3:4:[])               -- simplify addition
==> mySumL.iter (3+3) (4:[])             -- expand mySumL.iter
==> mySumL.iter 6 (4:[])                 -- simplify addition
==> mySumL.iter (6+4) []                 -- expand mySumL.iter
==> mySumL.iter 10 []                    -- simplify addition
==> 10                                   -- expand mySumL.iter via the [] case
```

The thing to note here is that, even though eager evaluation forced us to build the full list, we never had more than a single pending addition. Eager evaluation enabled us not to spend space on the accumulating variable.

Let's consider lazy evaluation of the same expression:

```
mySumL [1..4]
==> mySumL (enumFromTo 1 4)              -- desugar [..]
```

At this point, we don't have to wait for any evaluation of the argument to `mySumL`, so we expand immediately:

```
==> mySumL.iter 0 (enumFromTo 1 4)       -- expand enumFromTo
```

To expand `mySumL.iter` further, we need the second argument to be more well-defined, so our evaluation continues there:

```
==> mySumL.iter 0 (1:enumFromTo 2 4)     -- expand enumFromTo
```

And now the second argument is sufficiently well-defined to permit the expansion of `mySumL.iter`:

```
==> mySumL.iter (0+1) (enumFromTo 2 4)    -- expand mySumL.iter
```

At this point, we'd like to continue (as we did in the eager case) with the evaluation of the sum, but `mySumL.iter` doesn't depend on the value of its first argument to determine which case to apply, so it invests its evaluation effort on the second argument. This is a very important point, and I want you to think about it until you understand why Haskell makes the choice it's making. The evaluation continues:

```
==> mySumL.iter (0+1) (2:enumFromTo 3 4)
==> mySumL.iter ((0+1)+2) (enumFromTo 3 4)
==> mySumL.iter ((0+1)+2) (3:enumFromTo 4 4)
==> mySumL.iter (((0+1)+2)+3) (enumFromTo 4 4)
==> mySumL.iter (((0+1)+2)+3) (4:[])
==> mySumL.iter ((((0+1)+2)+3)+4) []
==> ((((0+1)+2)+3)+4)
```

Exercise 24.1 *Annotate the reduction steps above in the style of the foregoing discussion.*

The evaluation now amounts to simplifying an algebraic expression. But the important thing to note here is that our lazy evaluation commits the complementary sin to eager evaluation: it doesn't require that the whole list ever be resident in memory, but it does require that the whole algebraic expression be resident in memory.

It seems that our search for a constant-space summation fails for one reason with eager evaluation, and for another with lazy evaluation. We can trace out an evaluation process that would do this...

Exercise 24.2 *Write down an evaluation sequence in the style of above, which never involves more than a single cons or addition.*

But can we make Haskell produce this trace?

Evaluation strategy vs. strictness

Before getting on with directing evaluation strategies, let's talk briefly about *strictness*. The Church-Rosser Theorem entails that two reduction sequences that begin with the same term, and end with normal forms, must end with the same normal form. But this does not say that if one evaluation strategy results in a normal form, then all evaluation strategies result in a normal form. Here's a simple example. Consider the simple infinite loop function:

```
loop :: a
loop = loop
```

and the evaluation of `const 0 loop`. If we use eager evaluation, we'll attempt to evaluate the `loop` argument, resulting in a high speed twiddling of our thumbs:

```
const 0 loop
==> const 0 loop                              -- expand loop
==> const 0 loop                              -- expand loop
==> const 0 loop                              -- expand loop
...
```

But if we use normal order, expanding the definition of `const` *before* attempting to evaluate `loop`:

```
const 0 loop                                  -- expand const, done
==> 0
```

We'll say that a function `f` is *strict* in one of its arguments with respect to an evaluation strategy if the divergence of that argument forces the divergence of the function when evaluated using that strategy. Since languages usually define evaluation strategies, we'll often speak of the strictness (or lack thereof) of a function without explicit reference to that language's defined evaluation strategy.

Eager (sometimes called call-by-value) is the strictest evaluation strategy: eager evaluation evaluates every subterm of a term, hence, if any sub-term diverges, so too with the top-level evaluation. Some consider this to be a bug, others consider it to be a feature. Lazy evaluation comes with the complementary promise: if any evaluation strategy succeeds in reducing a term to normal form, it will succeed. Again, some consider this to be a bug, others consider it to be a feature. But the basic value proposition of normal-order evaluation is that it will evaluate every sub-expression as often as it needs to, which may well be a lot of times, but it may also be zero times.

GHC and Lazy Evaluation

At this point, I feel a slight need to come clean. If you've been following along with `ghc`, you might have noticed that it's perfectly happy and reasonably efficient in evaluating `mySumL [1..10^8]`, even on laptops with modest memory. I'm tempted to ask, what do you believe? Your humble instructor who would never knowingly lead you astray, or your own lying eyes? But I probably wouldn't be happy with the answer.

The truth here involves a great, and even surprising subtlety. Haskell does not *require* that Haskell compilers/interpreters use lazy evaluation. It just requires that they'll always produce as well-defined a result as lazy evaluation would. Now the people who write `ghc` are very clever, and examples like the one above are embarrassing. So `ghc` does a *strictness* analysis of the code it compiles, even at normal optimization levels. I actually had to explicitly turn off strictness analysis to get the results above. This strictness analysis enables `ghc` to find the "golden path" evaluation that runs in constant space. This might lead you to conclude, "I'll just use `ghc` and not worry." While using `ghc` *is* a good idea, the problem with this is that the strictness analyzer isn't omnipotent, and exactly the issue described here can and does happen with real-world code. So we've wasted no effort in this discussion, we've just employed a pedagogical example that's handled differently (i.e., better) in practice.

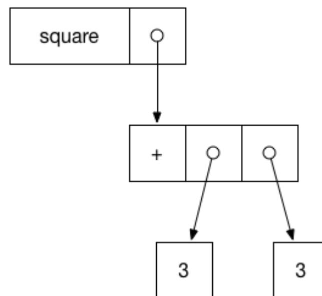
Graph Reduction

Our discussion of evaluation strategies assumed a term evaluation model. A fuller understanding of lazy evaluation (vs. the lambda-calculus's normal-order evaluation) requires a more sophisticated mental model of graph reduction. We'll give a brief (and as usual, incomplete) account of that, before returning to the problem of a constant space summing.

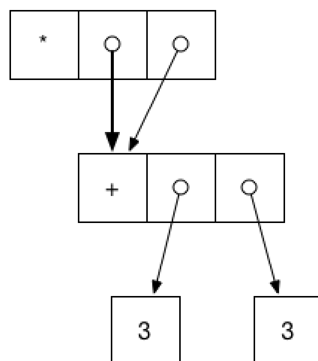
Let's consider a very simple program:

```
square x = x * x

main = print $ square (3+3)
```

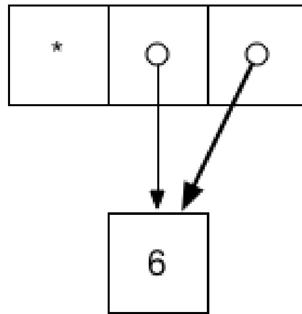
Again, the `print` function will drive the evaluation of `square (3+3)`. We'll start with the following graph structure:



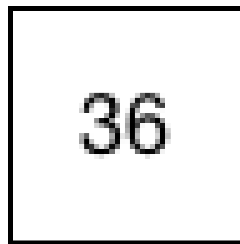
Our lazy evaluation begins by replacing the `square` node with a multiplication node, in which the argument to square ends up as both arguments to the resulting multiplication:



The next step of the evaluation looks at the first argument to `(*)`, indicated above by a darker line, to the `(+)` node, and since both of its arguments are already reduced, it is reduced to `6`.



It is crucially important to understand that because of the sharing involved, the effect of reducing `3+3` to `6` in the first argument to `(*)` also reduces it in the second, as shown above. Thus, when the evaluation of the arguments to the `(*)` moves to the second argument, the evaluator finds that node already evaluated, and needs to do no more work there. Instead, the top level `(*)` is ready to be evaluated, and its node is replaced by `36`:



This brief discussion hints at the complexity involved. For a slightly more complete discussion, consult HaskellWiki: Graph Reduction.

Returning to the main problem

Let's recall the space usage issue from the beginning of the lecture. We want to sum `[1..10^8]`, but without incurring excessive space use. We know that there is a reduction sequence that requires only constant space based on the definition of `mySumL`, but that both eager and lazy evaluation result in the use of huge amounts of memory.

An *ad hoc* solution.

We'll start by revisiting the lazy evaluation of `mySumL [1..4]`, and stop at the moment the train left the tracks:

```
mySumL [1..4]
==> mySumL (enumFromTo 1 4)              -- desugar [..]
```

```
==> mySumL.iter 0 (enumFromTo 1 4)         -- expand enumFromTo
==> mySumL.iter 0 (1:enumFromTo 2 4)       -- expand enumFromTo
==> mySumL.iter (0+1) (enumFromTo 2 4)     -- expand mySumL.iter
```

At this point, the next reduction step focusses on the second argument to `mySumL.iter`, because *it doesn't depend on the first*. What if, what if, what if, ... we could somehow *make* it depend on that first argument?

Consider the following, rather odd code:

```
mySumWithForce :: [Int] -> Int
mySumWithForce = iter 0 where
    iter r [] = r
    iter r (n:ns)
        | force r = iter (r+n) ns
    force n = n /= 0 || n == 0
```

The idea here is that the `force` function will always return `True`, but even the mighty `ghc` can't figure this out. So it has to run the test, and thereby evaluate `r` before it can expand `MySumWithForce.iter`. This keeps the argument simple, if not completely reduced, and allows the code to run in constant space. Oddly enough, that's what the profile says:

```
20,800,063,448 bytes allocated in the heap
     3,735,120 bytes copied during GC
        46,232 bytes maximum residency (2 sample(s))
        31,592 bytes maximum slop
             2 MB total memory in use (0 MB lost due to fragmentation)
```

The difference between 46K and 7.94GB should definitely get your attention! Factors on the order of 200,000 usually get mine.

Introducing seq

This brings us to one of the more mysterious functions of Haskell, `seq`.

```
seq :: a -> b -> b
```

Informally, `seq` *sequences* the reduction of its arguments to weak head normal form (i.e., just enough to do the weakest possible non-trivial pattern match). More precisely, evaluation in Haskell is built around forcing a thunk (a delayed evaluation) through just enough steps so that it is no longer simply a thunk, but a non-trivially defined partially evaluated expression. For algebraic types, this means that the top-level constructor has been determined. For atomic types like `Int`, it amounts to full evaluation. Function types don't get reduced at all (hence, "weak").

Using `seq` effectively is a difficult task, and there's many a Haskell programmer with a weak understanding of `seq` who tries to deal with space leaks by sprinkling `seq`'s throughout their code, and hoping for the best. This is rarely an effective strategy, and it's often counterproductive. Extra strictness is usually going to make your program run *slower*. But the code we've been working up sets us up for the proper use of `seq` as a replacement for `force`:

```
mySumWithSeq :: [Int] -> Int
mySumWithSeq = iter 0 where
    iter r [] = r
    iter r (n:ns) =
        let r' = r+n
        in  r' `seq` iter r' ns
```

This has the effect of reducing the accumulator argument in the call to `iter` to weak head normal form (i.e., fully evaluating it because it is an atomic type), *before* we make the call. This use of `seq` as an infix operator, together with a `let` binding of the arguments of a function, is fairly common.

Unsurprisingly, `mySumWithSeq` has constant space use.

```
12,800,064,600 bytes allocated in the heap
     1,181,848 bytes copied during GC
        46,232 bytes maximum residency (2 sample(s))
        31,592 bytes maximum slop
             2 MB total memory in use (0 MB lost due to fragmentation)
```

Bask in the efficiency of a well-managed evaluation process!

Bang!

Although `seq` is in some sense all you need, in practice many Haskell programmers find it difficult to use. So Haskell provides an alternative: strictness annotations on the components of algebraic data types, which have the effect of making constructors (when used as functions) strict, i.e., they evaluate their arguments to weak head normal form. Haskell programmers can then use these strictness-augmented type constructors to tune the use of strictness in their code. Let's consider a simple example:

```
data SPair a b = SPair !a b

mySumWithSPair :: [Int] -> Int
mySumWithSPair ns = iter (SPair 0 ns) where
    iter (SPair r []) = r
    iter (SPair r (n:ns)) = iter (SPair (r+n) ns)
```

Note the use of an exclamation point to indicate that the first argument (which we'll use to hold the register in the

computation) is strict.

The idea here is that the arguments to `mySumWithSPair.iter` are marshalled into an `SPair` value, whose strictness annotation forces the evaluation of the first argument, and thereby avoids building a large deferred calculation. This program results in slightly more than twice the overall allocation of space (those `SPairs` aren't free), but essentially the same maximum residency as `mySumWithSeq`.

Bang patterns

The problem with strictness annotations is that they seem to require the definition of a new `data` type for every function that benefits from additional strictness. GHC is experimenting with an alternative approach: the addition of strictness annotations to the *arguments* to a function. Consider:

```
mySumWithBang :: [Int] -> Int
mySumWithBang = iter 0 where
    iter r [] = r
    iter !r (n:ns) = iter (r+n) ns
```

This code is identical with that of `mySumL`, save for the exclamation point preceding `r` in the second clause of the definition of `mySumWithBang.iter`. The effect of the bang is to require the (partial) evaluation of the corresponding argument before a match can be made. Note that bang patterns are specific to GHC, and are not a feature of the Haskell definition, moreover, that their use requires the `{-# LANGUAGE BangPatterns #-}` pragma, and/or the equivalent command-line flag during compilation.

Conclusions

The lazy evaluation process of Haskell can sometimes result in shocking large space use, which in turn can result in poor runtime performance. Identifying the causes of excessive space use requires a more complete understanding of Haskell's evaluation process. Titrating a bit of strictness in at the right points can make a huge difference. But strictness isn't a magic bullet, as strictness in the wrong places can actually hurt overall performance.

In terms of time complexity, `mySumWithSeq` was by far the fastest, requiring about 1.3 seconds. The other space-efficient programs typically required about twice as much time, and the space inefficient programs could require up to a half a minute, so this matters. Oh, and that Python code? 5.8 seconds.

Chapter 25

Case Study: Propositional Logic

25.1 Mathematical Preliminaries

Propositional Logic is a simple, formal means for capturing certain kinds of rigorous argument. Facility in propositional (and later, quantificational) reasoning is a foundational skill for programming, as well as a nice domain area for us to play in as programmers.

We assume the existence of an infinite (usually countable) set of atomic *propositional variables* V. Propositional formulae are expressions built out of propositional variables, and the boolean constants true and false, using the boolean operators of negation/not ($\neg$), conjunction/and ($\wedge$), disjunction/or ($\vee$), and implication/implies ($\rightarrow$), e.g.

$$a \wedge (\neg b \vee c) \rightarrow d$$

The parsing priority for these connectives is (from highest to lowest): $\neg$, $\wedge$, $\vee$, and $\rightarrow$. It might be helpful to think of $\wedge$ as being analogous to multiplication, and $\vee$ to be analogous to addition. This analogy runs deeper than just in helping you understand how to parse boolean expressions: it also gives you some hints as to how to interpret them.

By convention, $\wedge$ and $\vee$ associate to the left (just like multiplication and addition), but since they're associative operators, it doesn't matter. Implies ($\rightarrow$) is not associative. By convention it associates to the right, as in Haskell.

A *valuation* is a function from the set of propositional variables to the set of boolean values True ($\top$) and False ($\bot$). We will often use the Greek letter ν (nu) to denote a valuation. We lift valuations to propositional formulae by describing *truth-tables* for the various propositional operators:

a	b	$\neg a$	$a \vee b$	$a \wedge b$	$a \rightarrow b$
$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$
$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$

Thus, e.g., if $\nu(a) = \top$, and $\nu(b) = \bot$, we can compute $\nu(a \vee b \to a \wedge b) = \bot$. This is often done via a table, where we write put a $\top$ or an $\bot$ under each connective, as appropriate, e.g.,

$$\begin{array}{cc|ccccccc}
a & b & a & \vee & b & \to & a & \wedge & b \\
\top & \bot & \top & & \bot & & \top & & \bot \\
 & & & \top & & & & \bot & \\
 & & & & & \bot & & &
\end{array}$$

or in a more compressed

$$\begin{array}{cc|ccccccc}
a & b & a & \vee & b & \to & a & \wedge & b \\
\top & \bot & \top & \top & \bot & \bot & \top & \bot & \bot
\end{array}$$

A propositional formula φ (phi) is a *tautology* if for all valuations ν, $\nu(\varphi) = \top$. A formula φ is *satisfiable* if there exists a valuation ν such that $\nu(\varphi) = \top$. Note that φ is a tautology if and only if $\neg\varphi$ is not satisfiable. The question of how difficult, in general, it is to determine whether or not a given propositional formula is satisfiable is a complete instance of the P vs. NP problem, one of the grand open problems of Computer Science.

Logicians care about the notion of a sound argument. In particular, let's suppose H is a set of hypotheses, and φ is a purported conclusion. Logicians want to define a notion of *proof,* which is a formal analog to the notion of a *sound argument* from rhetoric. They would write $H \vdash \varphi$ to mean that φ is a provable consequence of H.

The theory of provability for classical propositional logic has been completely worked out, although there are some very deep computational questions (in the plural) that lurk here.

Definition of Proof: Let H denote a set of propositional formulae, and φ denote a single propositional formula. An *H-proof* of φ is a sequence φ_1, φ_2, ..., φ_n of propositional formula such that

1. for each φ_i, either
   a φ_i is an element of H, in which case we say that φ_i is a *hypothesis* or *assumption*,
   b φ_i is a *tautology*, or
   c there exist $j, k < i$ such that $\varphi_j \equiv \varphi_k \to \varphi_i$, in which case, we say that φ_i follows by *modus ponens*;
2. $\varphi_n = \varphi$.

Note that in many presentations of the definition of a proof, item 1.b is further restricted to being a substitution instance of one of a finite set of tautological schemas.

Note that if $\vdash \varphi$, i.e., we can prove φ without hypotheses, then φ is a tautology. This follows from a simple induction on proof length.

Also note that what's happening with modus ponens looks a lot like a type calculation. This is no accident, although working this connection out in detail involves a digression into the intuitionistic propositional calculus, which we will forego, but hint at a lot.

One of the great virtues of functional programming languages is that they can deal with systems like this fairly easily, so it is no big deal to write a propositional proof-checker in a functional language like Haskell. It's a bit more of a challenge to write a proof-generator, though, which is one of the major intended applications of Haskell. But before we get into coding, let's work on developing some skill in proving theorems.

The Deduction Theorem is a tremendously useful meta-principle: if $H, \varphi \vdash \psi$, then $H \vdash \varphi \to \psi$. The proof of the Deduction Theorem is by induction on proof length, and is routinely covered in an introductory logic class. The Deduction Theorem often feels like cheating: not only do we end up with a simpler statement to prove, we have more hypotheses with which to prove it.

The converse of the Deduction Theorem, i.e., if $H \vdash \varphi \to \psi$ then $H, \varphi \vdash \psi$, follows immediately from modus ponens and from the monotonicity of $\vdash$ with respect to the hypothesis set.

Short example: $\vdash \alpha \to \beta \to \alpha$.

This is a tautology, but it's easier to prove than to verify:

1. $\alpha, \beta \vdash \alpha$, hypothesis
2. $\alpha \vdash \beta \to \alpha$, deduction theorem
3. $\vdash \alpha \to \beta \to \alpha$, deduction theorem

This style of reasoning, using just hypothesis, the deduction theorem, and modus ponens, is called *natural deduction*, and it is very powerful.

Now, it is not entirely irrelevant that `const :: a -> b -> a` in Haskell, and the type of the (pure) function has essentially the form of the tautology we've just proven. Indeed, $\alpha \to \beta \to \alpha$ is an intuitionistic/constructive tautology, and all intuitionistic tautologies are classical tautologies (but not conversely!!).

Exercise 25.1 *Give a natural deduction proof of* $(\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$. *What standard Haskell function has a similar type?*

A key problem in the definition of a proof given above is "how do we know that a formula φ is a tautology?" Basically, we have two choices at this point:

1. We can evaluate it for all valuations (note that we need only consider partial valuations, which are defined on all of the variables that occur within φ, a set has size 2^n if n distinct variables occur in φ).
2. We can prove it without hypotheses.

Quine's method is often a useful third alternative. In Quine's method, we chose a one variable x, and we create two new formulae $\varphi[x := \top]$ and $\varphi[x := \bot]$, substituting $\top$ and $\bot$ for x in φ. If both of these formulae are tautologies (facts that we can use Quine's method to verify), then the original formula is a tautology too. If either of this formulae is not a tautology, then the original formula isn't either.

What makes Quine's method so useful is that we can often greatly simplify the formulae $\varphi[x := \top]$ and $\varphi[x := \bot]$, using a few simple rules based on the truth tables above:

- $\neg\top \equiv \bot$,
- $\neg\bot \equiv \top$,
- $\top \wedge x \equiv x$,
- $x \wedge \top \equiv x$,
- $\bot \wedge x \equiv \bot$,
- $x \wedge \bot \equiv \bot$,
- $\top \vee x \equiv \top$,
- $x \vee \top \equiv \top$,
- $\bot \vee x \equiv x$,
- $x \vee \bot \equiv x$,
- $\top \to x \equiv x$,
- $x \to \top \equiv \top$,
- $\bot \to x \equiv \top$,
- $x \to \bot \equiv \neg x$.

We'll often add the rule

- $\neg\neg x \equiv x$

as well.

We begin with a simple example.

Example: Show $\varphi \equiv \alpha \to \beta \to \alpha$ is a tautology by Quine's method. We begin by splitting on α. This gives us two formulae: $\varphi[\alpha := \top] \equiv \top \to \beta \to \top$, and $\varphi[\alpha := \bot] \equiv \bot \to \beta \to \bot$. The first of these reduces to $\top \to \top$, and thence to $\top$; the second reduces directed directly to $\top$. Both are tautologies, and therefore $\alpha \to \beta \to \alpha$ is also a tautology.

Our next example is a bit more complicated.

Example: Show $\varphi \equiv (\alpha \vee \beta) \to (\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$ is a tautology by Quine's method.

We begin by splitting on α.

1. $\phi[\alpha := \top]$. We have

$$\begin{aligned} \phi[\alpha := \top] &\equiv (\top \vee \beta) \to (\top \to \gamma) \to (\beta \to \gamma) \to \gamma \\ &\equiv \top \to \gamma \to (\beta \to \gamma) \to \gamma \\ &\equiv \gamma \to (\beta \to \gamma) \to \gamma \end{aligned}$$

   This is a substitution instance of the known tautology $\alpha \to \beta \to \alpha$, and so must also be a tautology.

2. $\phi[\alpha := \bot]$. We have

$$\begin{aligned} \phi[\alpha := \bot] &\equiv (\bot \lor \beta) \to (\bot \to \gamma) \to (\beta \to \gamma) \to \gamma \\ &\equiv \beta \to \top \to (\beta \to \gamma) \to \gamma \\ &\equiv \beta \to (\beta \to \gamma) \to \gamma \end{aligned}$$

This is also a tautology, it's basically modus ponens expressed as a formula. But for the sake of completeness, let's go ahead and show that it's a tautology by splitting on β. Let $\psi = \beta \to (\beta \to \gamma) \to \gamma$

a $\psi[\beta := \top]$.

$$\begin{aligned} \psi[\beta := \bot] &\equiv \top \to (\top \to \gamma) \to \gamma \\ &\equiv \gamma \to \gamma \end{aligned}$$

and $\gamma \to \gamma$ is a familiar tautology.

b $\psi[\beta := \bot]$.

$$\begin{aligned} \psi[\beta := \bot] &\equiv \bot \to (\bot \to \gamma) \to \gamma \\ &\equiv \top \end{aligned}$$

and $\top$ is the ultimate tautology.

Exercise 25.2 *The formula we just proved, $\varphi \equiv (\alpha \lor \beta) \to (\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$, is also similar to the type of a Haskell Prelude function. What function is it? Hint: What standard type is most analogous to $\lor$?*

There are a number of standard tautologies which are worth committing to memory:

- The law of the excluded middle: $\alpha \lor \neg\alpha$,
- Disjunction elimination: $(\alpha \lor \beta) \to (\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$,
- Hypothetical syllogism: $(\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$,
- The Sherlock Holmes syllogism: $(\alpha \lor \beta) \to \neg\alpha \to \beta$,
- Reductio ab adsurdum: $(\alpha \to \bot) \to \neg\alpha$.

Exercise 25.3 *Prove the Sherlock Holmes syllogism by Quine's method.*

There is a crucial connection between provability and truth. We write $\Gamma \models \alpha$ if for all valuations ν, if $\nu(\gamma) = \top$ for all $\gamma \in \Gamma$, then $\nu(\alpha) = \top$ also, i.e., $\Gamma \models \alpha$ means that whenever the hypotheses are all true, then the conclusion is also true.

There are two crucial theorems, which together show that provability $\vdash$ and entailment $\models$ are equivalent:

- The Soundness Theorem: if $\Gamma \vdash \alpha$, then $\Gamma \models \alpha$, and
- The Completeness Theorem: if $\Gamma \models \alpha$, then $\Gamma \vdash \alpha$.

Proving these theorems is beyond this course (in the sense of subject, not preparation).

We write $\alpha \equiv \beta$, if α and β are semantically equivalent, i.e., if $\alpha \models \beta$ and $\beta \models \alpha$. By the equivalence of $\models$ and $\vdash$, this also means that α and β are provably equivalent, and so we'll often refer to this relation as simply "equivalent" henceforth.

Theorem (Referential Transparency): If $\alpha_i \equiv \beta_i$, for $i \in 1, 2, ..., k$, then

1. $\gamma[x_1 := \alpha_1, x_2 := \alpha_2, ..., x_k := \alpha_k] \equiv \gamma[x_1 := \beta_1, x_2 := \beta_2, ..., x_k := \beta_k]$, and
2. If $\Gamma \vdash \gamma$, then $\Gamma[x_1 := \alpha_1, x_2 := \alpha_2, ..., x_k := \alpha_k] \vdash \gamma[x_1 := \alpha_1, x_2 := \alpha_2, ..., x_k := \alpha_k]$.

This is often used in the case where Γ is empty, i.e., every substitution instance of a tautology is also a tautology. We saw this kind of reasoning before when we claimed $\gamma \to (\beta \to \gamma) \to \gamma$ is a substitution instance of $\alpha \to \beta \to \alpha$ (which we can now make precise via the substitution $[\alpha := \gamma, \beta := \beta \to \gamma]$).

This makes our life a *lot* easier, because it often enables us to work with complicated formulae by replacing various subformulae with equivalent subformulae.

There are a few equivalences that are crucially important to know, and often get used in applications of referential transparency.

1. Double negation elimination: $\alpha \equiv \neg\neg\alpha$,
2. de Morgan's laws:
    a $\alpha \vee \beta \equiv \neg(\neg\alpha \wedge \neg\beta)$, and
    b $\alpha \wedge \beta \equiv \neg(\neg\alpha \vee \neg\beta)$;
3. Contraposition: $\alpha \to \beta \equiv \neg\beta \to \neg\alpha$, and
4. The definition of implication: $\alpha \to \beta \equiv \neg\alpha \vee \beta$.

25.2 Parsing

Propositional logic is an important tool for reasoning about many things, including programs. But our immediate goal is to write programs that will help us in working with propositional formulae. Haskell is very well suited for this sort of work. We'll start by declaring the type of propositional formulae:

```
module SimpleProposition where

data Proposition
    = Var String
    | Boolean Bool
    | Not Proposition
    | And Proposition Proposition
    | Or Proposition Proposition
    | Implies Proposition Proposition
    deriving (Eq,Show)
```

Our major task for today is write a parser for propositional formulae. To that end, we'll first deal with the unfortunate fact that the notion system that we've been using isn't especially friendly for keyboard input, so we're going to replace the standard logical symbols for propositional connectives with keyboard friendly versions: conjunction/and (`&`), disjunction/or (|), implication/implies (`->`), and negation (`!`). We'll also use `T` for $\top$, and `F` for $\bot$. But, being defensive programmers, we'll define variables to hold these values, which makes it easy to change our minds:

```
impliesT,andT,orT,notT,trueT,falseT :: String
impliesT = "->"
andT = "&"
orT = "|"
notT= "!"
trueT = "T"
falseT = "F"
```

We'll start by writing a simple function `means :: a -> String -> ReadP a`, which associates a value with a `String`. The name is chosen because we intend to use it in infix form, e.g., ``impliesT `means` Implies``. We're going to want to ignore whitespace, and this functionality is going to be built into `token` as well:

```
means :: String -> a -> ReadP a
name `means` meaning = skipSpaces *> string name *> pure meaning
```

We'll use this first to define an atomic parser for boolean constants:

```
parseBool =  Boolean <$> (trueT `means` True
```

```
                                        <++ falseT `means` False)
```

Next, we'll consider `Variable`. Here, our representational choices for true and false present us with a minor conundrum. We could do something like this:

```
parseVar = skipSpaces
        *> (Var <$> ((:) <$> satisfy isLower
                         <*> munch isAlphaNum))
```

This, a variable begins with a lower-case letter, followed by zero or more alpha-numerics.

It's possible to push this further, but it's hard to argue that doing so improves clarity or efficiency, so we'll stop here.

Next, it will be useful for having a parser that handles parenthesized expressions: `parens :: ReadP a -> ReadP a`. We can define this naïvely via a monadic construct:

```
parens :: ReadP a -> ReadP a
parens p = do
    skipSpaces
    char '('
    result <- p
    skipSpaces
    char ')'
    pure result
```

But this involves both monadic code, and some twitchiness around our conventions regarding calls to `skipSpaces`. Fortunately, `Text.ParserCombinators.ReadP` already contains a function that's perfect for dealing with this issue:

```
between :: ReadP open -> ReadP close -> ReadP a -> ReadP a
```

What appears to be an odd order of arguments is actually a setup for an η-reduction, thus

```
parens :: ReadP a -> ReadP a
parens = between (skipSpaces *> char '(') (skipSpaces *> char ')')
```

At this point, there are two extremely useful parser combinators that come into play:

```
chainr1 :: ReadP a -> ReadP (a -> a -> a) -> ReadP a
chainl1 :: ReadP a -> ReadP (a -> a -> a) -> ReadP a
```

These are used to parse right- and left-associative operators, where we're guaranteed that at least one expression is present. This makes snapping together a precedence-based parser based on binary operations very easy. To that end, let's assume that implication is at precedence level 0, conjunction at precedence level 1, disjunction at precendence level 2, negation at precedence level 3, and the atomic formula at precedence level 4. We can use this to snap together a precedence-based parser with very little work (not here that we're punting on negation):

```
parseProposition :: ReadP Proposition
parseProposition = prec0 where
    prec0 = chainr1 prec1 (impliesT `means` Implies)
    prec1 = chainl1 prec2 (orT `means` Or)
    prec2 = chainl1 prec3 (andT  `means` And)
    prec3 = prefix  prec4 (notT `means` Not)
    prec4 = parseVar <++ parens prec0 <++ parseBool
```

This actually works (at least, on the negation free part of `Proposition`).

To deal with negation requires only a simple `prefix :: ReadP a -> ReadP (a -> a) -> ReadP a` function, which isn't in `Text.ParserCombinator.ReadP`. Fortunately, we can write this, but it's a bit tricky:

```
prefix :: ReadP a -> ReadP (a -> a) -> ReadP a
prefix p op = p <++ (op <*> prefix p op)
```

In this, the repetition of `prefix p op` is undesirable (it result in unnecessary copying), and we can eliminate it with a recursive definition:

```
prefix p op = result where
    result = p <++ (op <*> result)
```

For those who like golfing, we can go for the hole-in-one by adding the fixed point combinator `fix :: (a -> a) -> a` from `Data.Function`,

```
prefix p op = fix $ \result -> p <++ (op <*> result)
```

But if we do this, the Code Fairy is going to want to eliminate the lambda via η-reduction, resulting in the incomprehensibly zen:

```
prefix p op = fix $ (<++) p . (<*>) op
```

and that way lies madness. Or your instructors, who at this very moment are contemplating further η-reductions.

Anyway, at this point a bit of gentle packaging and code-reorganization results in

```
instance Read Proposition where
    readsPrec _ = readP_to_S prec0 where
        prec0 = chainr1 prec1 (impliesT `means` Implies)
        prec1 = chainl1 prec2 (orT `means` Or)
        prec2 = chainl1 prec3 (andT  `means` And)
        prec3 = prefix  prec4 (notT `means` Not)
        prec4 = parseVar <++ parens prec0 <++ parseBool
        parseVar = skipSpaces
                 *> (Var <$> ((:) <$> satisfy isLower
                                  <*> munch isAlphaNum))
        parseBool = Boolean <$> (trueT `means` True
                                <++ falseT `means` False)
```

which is shockingly to the point. It would not be unreasonable to be thinking of the calculator lab at this point, and certain simplifications that might be made to the code found therein.

Exercise 25.4 *Write a small desktop calculator, based the following data type:*

```
data Expression = DoubleValue Double
                | Sum Expression Expression
                | Difference Expression Expression
                | Product Expression Expression
                | Quotient Expression Expression
```

You should process input line-by-line, doing a proper precedence-based parse of each line, an evaluation of the resulting expression, and print the final result.

Don't worry about syntax errors, but try to make your calculator program as smooth as possible otherwise. Note also that you can define the parser directly, so there's no need for a `Proposition`*-like extra type layer.*

Note that generalizing the parsing strategy above requires dealing with multiple operators at the same precedence level, fortunately with the same associativity. This requires a more complicated "composing" parser than in the examples above, e.g.,

```
prec0 = chainl1 prec1 (token Sum "+" <++ token Difference "-")
```

At this point, we can test our parser.

We can approach this naively:

```
> read "a | b -> c -> d & e" :: Proposition
```

```
Implies (Or (Var "a") (Var "b")) (Implies (Var "c") (And (Var "d") (Var "e")))
```

or

```
> read "!!!a" :: Proposition
Not (Not (Not (Var "a")))
```

This is great for reading propositions, but it's not so great for printing them. We can go with something simple:

```
instance Show Proposition where
    show = showProp where
        showProp (Var v) = v
        showProp (Boolean True) = "T"
        showProp (Boolean False) = "F"
        showProp (Not p) = "!" ++ show p
        showProp (And p q) = "(" ++ show p ++ " & " ++ show q ++ ")"
        showProp (Or p q)  = "(" ++ show p ++ " | " ++ show q ++ ")"
        showProp (Implies p q) = "(" ++ show p ++ " -> " ++ show q ++ ")"
```

But the results are unsatisfactory. E.g.,

```
> read "a | b -> (a -> c) -> (b -> c) -> c" :: Proposition
((a | b) -> ((a -> c) -> ((b -> c) -> c)))
```

As you can see, too many parentheses detract from readability. To avoid this, we have to go to precedence based output. The idea here is that we have a precedence context. If we're in too deep (whatever that means), we need parentheses, but we get to reset our precedence context.

One nuance here is that we'll want to have direct access to the precedence level in deciding when to parenthesize, and so we'll make the precedence level an argument to a `showp` function (rather than having a bunch of `showpx` functions):

```
instance Show Proposition where
    show = showp (0 :: Int) where
        showp _ (Boolean True) = trueT
        showp _ (Boolean False) = falseT
        showp _ (Var v) = v
        showp _ (Not p) = notT ++ showp 3 p
        showp i (And s t) =
            paren 2 i $ unwords [ showp 2 s, andT, showp 2 t ]
        showp i (Or s t)      =
            paren 1 i $ unwords [ showp 1 s, orT, showp 1 t ]
```

```
showp i (Implies s t) =
    paren 0 i $ unwords [ showp 1 s, impliesT, showp 0 t]
paren cutoff precedence str
    | precedence > cutoff = "(" ++ str ++ ")"
    | otherwise           = str
```

Note here the particular trickiness around the right-associativity of implication.

We can use the parser/formatter pair to do something that is already a bit interesting. Consider `norm.hs`:

```
module Main where

import Control.Monad
import System.Environment
import Text.Read
import Proposition

process :: String -> IO ()
process arg = do
    case readMaybe arg of
        Nothing -> putStrLn $ "Could not parse \'" ++ arg ++ "\'."
        Just r -> print (r :: Proposition)

main :: IO ()
main = void $ getArgs >>= traverse process
```

This simply reads and writes a proposition.

Why is this interesting? Because beginning students are often confused about precedence and associatively, and will use *way* more parentheses than are actually necessary, e.g.

```
$ norm "(a&b) -> (a|b)"
a & b -> a | b
$
```

Exercise 25.5 *Note that the it is not necessarily the case that*

```
(read (show p)) == p
```

for all propositions `p`. *Why? Does this really matter, and if it doesn't how might we explain ourselves?*

Hint: Consider `"a` *(b | c)"*|.

Code

- `Proposition.hs`
- `Main.hs`

25.3 A Tautology Checker

Propositional logic is an important tool for reasoning about many things, including programs. But our immediate goal is to write programs that will help us in working with propositional formulae. Haskell is very well suited for this sort of work. We'll start by declaring the type of a propositional formula:

```
data Proposition
    = Var String
    | Boolean Bool
    | Not Proposition
    | And Proposition Proposition
    | Or Proposition Proposition
    | Implies Proposition Proposition
```

Exercise 25.6 *If you would like more practice with parsing, work through these Propositional Logic: Parsing (section 25.2). Below, we will assume the presence of the parser developed there.*

Our goal for today is to write a brute-force tautology checker. It greatly simplifies this program to have the following function-defining function for `Proposition`, which we'll add to the `Proposition` module:

```
abstractEval :: (Applicative m)
             => (String -> m b)    -- ^ Var
             -> (Bool -> m b)      -- ^ Boolean
             -> (b -> b)           -- ^ Not
             -> (b -> b -> b)      -- ^ And
             -> (b -> b -> b)      -- ^ Or
             -> (b -> b -> b)      -- ^ Implies
             -> Proposition
             -> m b
abstractEval varf boolf notf andf orf impliesf = eval where
    eval (Var a)       = varf a
    eval (Boolean b)   = boolf b
    eval (Not p)       = pure notf  <*> eval p
    eval (And p q)     = pure andf <*> eval p <*> eval q
    eval (Or p q)      = pure orf  <*> eval p <*> eval q
    eval (Implies p q) = pure impliesf <*> eval p <*> eval q
```

This looks scary, but it's actually quite simple, once you get the hang of it: we use `abstractEval` to define an evaluator of type `Proposition -> m a` by providing it with functions for processing variables, boolean constants, negations, compositions, disjunctions, and implications. We will use `abstractEval` to implement a couple of different evaluators in this program, with different evaluation contexts. The generality of this function isn't just showing off! We actually need it.

Our tautology checker will work as follows. Given a proposition p, it will:

1. Compute the set v of variables that occur in p.
2. Generate a list valuations consisting of all possible maps from v to Bool.
3. Evaluate the proposition p at each valuation in turn, producing a list of valuations at which the proposition is false
4. Generate output, either noting that the formula is a tautology, or providing a refuting refutation.

The first step uses abstractEval to build a function that extracts the set of names of the variables from a Proposition:

```
variables :: Proposition -> Set String
variables = runIdentity . abstractEval
          (Identity . Set.singleton)     -- Var
          (const (Identity Set.empty))   -- Boolean
          id                             -- Not
          Set.union                      -- And
          Set.union                      -- Or
          Set.union                      -- Implies
```

A key notion is that of a valuation, i.e., a representation of a function from variables to boolean values. We'll use Map String Bool to represent valuations. A key problem is to produce a list of all possible valuations. We started with code that looked like this:

```
valuations :: Set String -> [Map String Bool]
valuations = map Map.fromList
           . sequence
           . map (\v -> [(v,True),(v,False)])
           . Set.toList
```

A key twist in this is the use of sequence, acts on a list of lists by building the Cartesian product of the lists, i.e., the list of lists that consist of one element from each of the original lists. It's remarkable that sequence does this, but it does.

But our experience with the Foldable and Traversable type classes suggest that the sort of representation swizzling we're doing here (visible in the toList and fromList) functions should be unnecessary. This lead to a search for a better way. A first observation is that sequence has a more general type:

```
sequence :: (Traversable t, Monad m) => t (m a) -> m (t a)
```

As [] is both Traversable and a Monad, we can use it as if it's type were

```
sequence :: [[a]] -> [[a]]
```

But our desired end result has type `[Map String Bool]`. We can get there via `sequence` if take an argument of type `Map String [Bool]`, as `[]` is a `Monad`, and `Map String` is `Traversable`. This reduces the question (at least, at the level of types, if not values) to the problem of producing a function of type `Set String -> Map String [Bool]`. To this end, we can simplify things a bit, using the fact that `Map String` is a `Monoid` (its monoidal action is left-biased union), but that works for us, so we can form a function of the necessary type via

```
foldMap (\v -> Map.singleton v [True,False])
```

The code fairy suggests that we replace `sequence` with `sequenceA`, which has a more modern type, and that we do a bit of η-reduction (as usual), giving us

```
valuations = sequenceA . foldMap (`Map.singleton` [True,False])
```

This is remarkable! Fully 3/5ths of the list traversing we were doing was unnecessary. As it turns out, this codes is not only type-correct, it's value-correct. This does not come as a major surprise.

Next up, we have to evaluate the proposition at each valuation. Fortunately, we have `abstractEval` ready to serve us.

```
eval :: (Map String Bool)
     -> Proposition
     -> Bool
eval valuation prop = runReader (evalf prop) valuation where
    evalf = abstractEval (reader . flip (!))  -- Var
                         pure                 -- Boolean
                         not                  -- Not
                         (&&)                 -- And
                         (||)                 -- Or
                         ((||) . not)         -- Implies
```

This takes a bit of work. The local function

```
evalf :: Proposition -> Reader (Map String Bool) Bool
```

is key. The `abstractEval` function has no difficulty working in the context of `Reader (Map String Bool)`, and most of the code writes itself. The tricky bit is our variable-handling argument to `abstractEval`. Intuitive, we'll need

```
varf :: String -> Reader (Map String Bool) Bool
```

This involves lookup up the argument variable in the `Reader` context, which we can do by the `reader` function:

```
varf v = reader (\r -> r ! v)
```

Getting from here to

```
varf = reader . flip (!)
```

is just standard code fairy η-reduction work, but once we've gotten that far, we've mooted the need for a local definition.

There is something mildly suspect about `eval`. Since it relies on `(!)`, it can throw an exception if the proposition being evaluated has a variable that's not included in the valuation. That can't happen with this program, as we only call `eval` with a valuation that was generated from the proposition's variables, but it's worth thinking about how we might generalize `eval` so that an undefined variable can be handled more gracefully in the code itself, but that's a project for another day. But it's worth noticing that `abstractEval` sets us up for working with more complex evaluation contexts (e.g., contexts that handle errors).

Next, we generate a list of exceptional valuations, i.e., valuations for which a given proposition is not true:

```
refutations :: Proposition -> [Map String Bool]
refutations p =
    [ v
    | v <- valuations . variables $ p
    , not . eval v $ p
    ]
```

Finally, there's the `IO` code to make it all work. The `taut` program processes its command line arguments, reading each as a `Proposition`, and then reporting whether the resulting `Proposition` is or is not a tautology. There's nothing especially remarkable about the code involved:

```
process :: String -> IO ()
process arg =
    case readMaybe arg of
        Nothing -> putStrLn $ "Could not parse \'" ++ arg ++ "\'."
        Just p -> case refutations p of
            [] -> putStrLn $ show p ++ " is a tautology."
            r:_ -> do
                putStrLn $ show p ++ " is refutable, cf.,"
                void . (`Map.traverseWithKey` r) $ \k v ->
                    putStrLn $ printf "  %s := %s" k (show v)
```

Briefly, we use `Text.Read.readMaybe`, which parses in the `Maybe` context, returning `Nothing` for unparseable input. Assuming a successful parse, we then build the list of refutations of the resulting proposition. If that list is

empty, we have a tautology (and so report), and if it's non-empty, we have a refutable proposition, and report both the fact of refutability, and print out a witnessing refutation.

Our `main` action was originally written as

```
main :: IO ()
main = do
    args <- getArgs
    mapM_ processArg args
```

But again, the code fairy suggests that we should prefer the modern `traverse` over the legacy `mapM`, which results (after further transformations) in

```
main :: IO ()
main = void $ getArgs >>= traverse process
```

The `void` simply throws out the result type, and it's a useful function to know about, especially to quite warnings on non-binding lines in a do expression if you compile with `-Wall`, which is generally recommended.

A sample run looks like this:

```
$ taut 'a -> a' 'a -> b' '(a -> b) -> (b -> c) -> (a -> c)' 'a ->'
a -> a is a tautology.
a -> b is refutable, cf.,
  a := True
  b := False
(a -> b) -> (b -> c) -> a -> c is a tautology.
Could not parse 'a ->'.
```

On one hand, this is a fairly useful little program, but on another, it's a bit unsatisfying. If we process a refutable proposition, we're given a refuting valuation. But if the proposition is a tautology, all we get is a bare assertion that this is so. If we want to do more, we'll have to work harder.

Code

- `Proposition.hs`
- `Main.hs`

25.4 A Quine's Method Theorem Prover

Our goal for today is ambitious. We will write a program that will take as input a series of propositions (intended to be tautologies, but checked as such), and will produce a web page that consists of a series of Quine-style proofs, using the mathematical typesetting language TeX (strictly speaking, LaTeX) to render boolean formulae in a attractive way. I.e., we'll take input file that looks like this:

```
a -> a
a -> b -> a
a -> (a -> b) -> b
```

and produce web content that looks like this:

Propositional Tautologies

Proposition 1: $\alpha \to \alpha$ Quine Alternatives:

- $\alpha \to \alpha$ $[\alpha := \top]$ Simplification
  - $\top \to \top$
  - $\top$

  Reduced to $\top$.
- $\alpha \to \alpha$ $[\alpha := \bot]$ Simplification
  - $\bot \to \bot$
  - $\top$

  Reduced to $\top$.

☐ Proposition 1 **Proposition 2:** $\alpha \to \beta \to \alpha$ Quine Alternatives:

- $\alpha \to \beta \to \alpha$ $[\alpha := \top]$ Simplification
  - $\top \to \beta \to \top$
  - $\beta \to \top$
  - $\top$

  Reduced to $\top$.
- $\alpha \to \beta \to \alpha$ $[\alpha := \bot]$ Simplification
  - $\bot \to \beta \to \bot$
  - $\top$

Reduced to $\top$.

☐ Proposition 2 **Proposition 3:** $\alpha \to (\alpha \to \beta) \to \beta$ Quine Alternatives:

- $\alpha \to (\alpha \to \beta) \to \beta$ $[\alpha := \top]$
  - $\top \to (\top \to \beta) \to \beta$
  - $(\top \to \beta) \to \beta$
  - $\beta \to \beta$

  Substitution instance of Proposition $1[\alpha \to \alpha]$ at

  - $\alpha := \beta$
- $\alpha \to (\alpha \to \beta) \to \beta$ $[\alpha := \bot]$ Simplification
  - $\bot \to (\bot \to \beta) \to \beta$
  - $\top$

  Reduced to $\top$.

☐ Proposition 3

There are a lot of moving parts here!

Quine's method

Let's remember the basics of Quine's method. Given a proposition `p`, we simplify `p`, i.e., we produce a new proposition `p'` which logically equivalent to `p`, such that `p'` is either:

- `Boolean True`, in which case it is a tautology,
- `Boolean False`, in which case it is refutable, i.e., not a tautology, or
- there are no `Boolean` nodes remaining, in which case we
  1. select a variable `v` which occurs in `p`.
  2. create two new propositions `pt` and `pf`, by substituting `Boolean True` and `Boolean False` for `v` in the original proposition. Then `p` is a tautology if and only if both `pt` and `pf` are.

We'll start with code for simplifying a proposition:

```
simplify :: Proposition -> Proposition
simplify = simpleEval Var Boolean notf andf orf impliesf where
    notf p = case p of
        Boolean b -> Boolean (not b)
        Not p' -> p'                    -- eliminate double negations
```

```
            _ -> Not p
        andf p q = case (p,q) of
            (Boolean True,_) -> q
            (Boolean False,_) -> Boolean False
            (_,Boolean True) -> p
            (_,Boolean False) -> Boolean False
            _ -> And p q
        orf p q = case (p,q) of
            (Boolean True,_) -> Boolean True
            (Boolean False,_) -> q
            (_,Boolean True) -> Boolean True
            (_,Boolean False) -> p
            _ -> Or p q
        impliesf p q = case (p,q) of
            (Boolean True,_) -> q
            (Boolean False,_) -> Boolean True
            (_,Boolean True) -> Boolean True
            (_,Boolean False) -> Not p
            _ -> Implies p q
```

Much of the heavy lifting is done by `simpleEval`, which essentially eliminates the contextualizing type constructor of the last lecture's `abstractEval`.

The `simplify` function can be thought of as a partial-evaluator, i.e., it takes an existing expression, and partially evaluates it, resulting in a simpler expression that can be used more efficiently in lieu of the original in subsequent processing. Partial evaluators are important in many optimizations, and one of the nice things about Haskell is that it often makes writing partial evaluators easy, as it does here.

Next up, we have the job of creating substitution instances of our proposition, in which the selected variable is set to `True` and `False` respectively. First, we'll write a general purpose substitution routine:

```
type Substitution = Map String Proposition

substitute :: Substitution -> Proposition -> Proposition
substitute sub = simpleEval varf Boolean Not And Or Implies where
    varf v = Map.findWithDefault (Var v) v sub
```

The idea here is that `substitution` includes as keys only the names of variables that we're substituting out, so a missing key simply means a variable that we're not changing.

Our first cut at our theorem prover uses a simple type for representing a quine-style proof:

```
data QuineProof
    = Split
        Proposition     -- ^ the proposition to be proven
```

```
        String          -- ^ the name of the variable to split on
        QuineProof      -- ^ case where the split variable is True
        QuineProof      -- ^ case where the split variable is False
    | Trivial
    deriving (Show)

type Analysis = Maybe QuineProof
```

This will get more sophisticated, but it's a start. With this, we can do our first pass at our quine theorem prover.

```
analyze :: Proposition -> Analysis
analyze prop = case simplify prop of
    Boolean True -> pure Trivial
    Boolean False -> Nothing
    q -> Split q v <$> branch True <*> branch False where
        v = elemAt 0 . variables $ q
        branch b = analyze (substitute (Map.singleton v (Boolean b)) q)
```

This is just a simple realization of Quine's algorithm. <!– Note that at this point in the code, `variables :: Proposition -> [Stri` –> We will get more sophisticated soon. This code is good enough to to find our our bearings, but a little bit of exploration reveals a need to do more work:

```
> let runQuine = quine . read
> runQuine "a -> a"
Just (Split a -> a "a" Trivial Trivial)
> runQuine "a -> b"
Nothing
> runQuine "a | b -> (a -> c) -> (b -> c) -> c"
Just (Split a | b -> (a -> c) -> (b -> c) -> c "a" (Split c -> (b -> c) -> c "b" (Split c -> c ->
```

We're getting proofs out, and it's worth understanding that first.

Let's consider the analysis of `a -> a`, a familiar tautology. It doesn't simplify to a `Boolean True`, but rather to itself. We split on `"a"`, and substitute in `Boolean True` and `Boolean False` for `Var "a"` in the true and false branches of the proof, in both cases resulting in formula that simplify to `Boolean True`, and so have trivial proofs.

Next, we have `a -> b`, which isn't a tautology. Since the formula is refutable, we return `Nothing`, as there is no Quine proof to return.

Finally, we have the analysis of a   b -> (a -> c) -> (b -> c) -> c|. The proof that was returned isn't that easy to read, so it's helpful to format it a bit differently (note that this isn't valid Haskell)

```
Split
    a | b -> (a -> c) -> (b -> c) -> c
```

```
"a"
Split c -> (b -> c) -> c
     "b"
     Split c -> c -> c "c" Trivial Trivial
     Split c -> c "c" Trivial Trivial
Split b -> (b -> c) -> c
    "b"
    Split c -> c "c" Trivial Trivial
    Trivial
```

We see that the analysis of the original formula results in a splitting on `Var "a"`, resulting in two formula that simplify to `c -> (b -> c) -> c` and `b -> (b -> c) -> c` respectively. The analysis of the first of these does a split on `Var "b"`, which is unfortunate, because a split on `Var "c"` would result in trivial instances. Likewise, the analysis of `b -> (b -> c) -> c` splits on `Var "b"`, resulting in a non-trivial branch, whereas a split on `Var "c"` instead would have resulting in two trivial branches.

So it's a start, but there's a lot to be unhappy with if this particular theorem prover is to live up to its potential:

- The simplification process is abrupt. We only get to see the beginning and ending steps of the simplifications, and simplification is a nontrivial process. It would be helpful to illustrate a step-by-step sequence of simplifications.
- Our goal is to give a result that is human checkable, but we fail badly in the case of a refutable statement. Our prover is just returning the fact of refutability, rather than an explanation. Our brute-force tautology prover did better, and we should too.
- Our theorem prover forces us to pursue the process of splitting a formula all the way down to success (`Boolean True`) or failure (`Boolean False`). We can give more concise proofs by recognizing when we've reduced a formula to a (substitution instance of) a known tautology, and quit there.
- The proofs we're producing aren't particularly economical. If our goal is human readability, we should value concision, even at the cost of much more computation.
- The result we return is unnecessarily opaque. Accepting the default formatting of the result isn't a particularly friendly thing to do. We should format the final result better.

These changes involve a certain amount of additional book-keeping, and the difficulty of doing so isn't to be slighted. But the more essential difficulties are in writing the step-by-step simplifier and the instance checker.

We'll start with the step-by-step simplifier:

```
simplifyInSteps :: Proposition -> [Proposition]
simplifyInSteps = converge step where
    converge f = untilFixed . iterate f where
        untilFixed (p:qs@(q:_))
            | p == q = [p]
            | otherwise = p : untilFixed qs
```

```haskell
            untilFixed _ = error "untilFixed: impossible error"
        step prop = case prop of
            v@(Var _) -> v
            b@(Boolean _) -> b
            Not p -> case p of
                Boolean b -> Boolean (not b)
                Not p' -> p'
                _ -> Not $ step p
            And p q -> case (p,q) of
                (Boolean True,_) -> q
                (Boolean False,_) -> Boolean False
                (_,Boolean True) -> p
                (_,Boolean False) -> Boolean False
                _ -> And (step p) (step q)
            Or p q -> case (p,q) of
                (Boolean True,_) -> Boolean True
                (Boolean False,_) -> q
                (_,Boolean True) -> Boolean True
                (_,Boolean False) -> p
                _ -> Or (step p) (step q)
            Implies p q -> case (p,q) of
                (Boolean True,_) -> q
                (Boolean False,_) -> Boolean True
                (_,Boolean True) -> Boolean True
                (_,Boolean False) -> Not p
                _ -> Implies (step p) (step q)
```

This isn't a perfect solution to the simplify-in-steps problem (it may make multiple simplifications, but at least they'll be in distinct parts of the expression), but it's a bit of improvement for now. The key is the local `step` function, which only calls itself recursively in the cases where it can't make progress otherwise. This is driven by `converge`, which iterates `step` until it reaches a fixed point.

We now can tackle the issue of getting simplification steps into our proofs, by modifying the definition of `QuineProof` so that it includes the simplification steps, and not just the final result of simplification.

```haskell
data QuineProof
    = Split
        [Proposition]   -- ^ simplifications
        String          -- ^ variable to split on
        QuineProof      -- ^ case where split variable is true
        QuineProof      -- ^ case where split variable is false
    | Trivial
        [Proposition]   -- ^ simplifications
```

Because refutations are also more complex, we replace the use of the `Maybe` type with a type that can carry a value

in the failure case:

```
data Refutation = Refutation (Map String Bool)

type Analysis = Either Refutation QuineProof
```

But we're not done yet. We want to handle a sequence of propositions, as logic texts invariably do, precisely so that we can make our proofs ”shallower,” by cutting off our Quine-style proofs when we get a formula that's a substitution instance of an earlier formula. This results in an appreciably more complex type for our :

```
type Heuristic = Proposition -> Maybe String

analyzeWithHeuristic
    :: Heuristic                -- variable selection heuristic
    -> [(Int,Proposition)]      -- list of indexed tautologies
    -> Proposition              -- the Proposition to analyze
    -> Analysis
```

There are a couple of twists here. The `Heuristic` type is used to pick a variable for splitting on. We allow a `Heuristic` to return `Nothing` if its argument contains no variables. The second argument to `analyzeWithHeuristic` is a list of tautologies (already proven), paired with an `Int` which identifies the proposition which established it.

To determine whether or not a `Proposition` is a substitution instance of a given propositional formula, we use

```
instanceOf :: Proposition -> Proposition -> Maybe Substitution
instanceOf target pattern = case (target,pattern) of
    (t, Var v) -> Just $ Map.singleton v t
    (Boolean t,Boolean p) -> guard (t == p) >> pure Map.empty
    (Not t, Not p) -> instanceOf t p
    (And t1 t2, And p1 p2) -> process t1 t2 p1 p2
    (Or t1 t2, Or p1 p2) -> process t1 t2 p1 p2
    (Implies t1 t2, Implies p1 p2) -> process t1 t2 p1 p2
    _ -> Nothing
    where
        process t1 t2 p1 p2 = do
            m1 <- instanceOf t1 p1
            m2 <- instanceOf t2 p2
            if m1 `Map.intersection` m2 == m2 `Map.intersection` m1
                then pure $ Map.union m1 m2
                else Nothing
```

This tries to build substitutions, but the trickiness comes from combining them. We do this fairly simply, by relying on the left-bias of `Map` operators, in this case `intersection`, to insure that in the case we have to follow two branches, that we derive identical substitutions or fail.

There are a fair number of details being swept under the rug here, which you can find out by reading the code.

Producing HTML

The output side of this program will use `Text.Blaze.Html5` and related modules to render HTML. We'll pass over the specifics of generating TeX—as they're not all that different from generating the kind of text-based output we've used so far.

A key output handling procedure is

```
renderProof ix prop proof = do
    h2 $ "Proposition " >> toHtml ix >> ": " >> toHtml' prop
    format proof
    p ! class_ "box" $
        "$\\Box$ Proposition " >> toHtml ix
    where
    format prf = case prf of
        Split props var trueb falseb -> do
            when (length props > 1) $ do
                p "Simplification"
                ul . void $ traverse (li . toHtml') props
            let prop' = last props
            p "Quine Alternatives:"
            ul $ do
                li $ showBranch prop' var True trueb
                li $ showBranch prop' var False falseb
        Trivial props -> do
            p "Simplification"
            ul . void $ traverse (li . toHtml') props
            p $ "Reduced to " >> toHtml' (Boolean True) >> "."
        Reference props citation pattern bindings -> do
            ul . void . traverse (li . toHtml') $ props
            p $ "Substitution instance of Proposition "
                >> toHtml citation
                >> "$[" >> toTeX pattern >> "]$ at"
            ul . void . flip Map.traverseWithKey bindings
               $ \k v -> li $ "$" >> greekDict Map.! k >> " := "
                                  >> toTeX v >> "$"
    showBranch prop' var bool branch = p $ do
        toHtml' prop' >> " $[" >> greekDict Map.! var >> ":="
            >> boolDict Map.! bool >> "]$"
        format branch
```

Note here the additional `Reference` case to `QuineProof`. This is a fairly straightforward recursive processing of a proof tree, in as much as the structure of `QuineProof` type is mirrored in the structure of the resulting HTML.

There are a couple of things to notice here. We have the `{-# LANGUAGE OverloadedStrings #-}` pragma on, which allows literal strings in the source to be interpreted via the `IsString` type class, which implements a `fromString :: String -> a` function, similar to `fromInt`. There's also the `toHtml'` function, which parallels `Blaze`'s `toHtml`, but is specific to the `Proposition` type (a minor work-around necessary to avoid a orphaned instance warning).

The Full Program

- `Main.hs`
- `Proposition.hs`
- `Quine.hs`
- `QuineHtml.hs`