Haskell Basics

CIS 194 Week 1
14 January 2013

Suggested reading:

- Learn You a Haskell for Great Good, chapter 2
- Real World Haskell, chapters 1 and 2

What is Haskell?

Haskell is a *lazy, functional* programming language created in the late 1980's by a committee of academics. There were a plethora of lazy functional languages around, everyone had their favorite, and it was hard to communicate ideas. So a bunch of people got together and designed a new language, taking some of the best ideas from existing languages (and a few new ideas of their own). Haskell was born.

So what is Haskell like? Haskell is:

Functional

There is no precise, accepted meaning for the term "functional". But when we say that Haskell is a *functional* language, we usually have in mind two things:

- Functions are *first-class*, that is, functions are values which can be used in exactly the same ways as any other sort of value.
- The meaning of Haskell programs is centered around *evaluating expressions* rather than *executing instructions*.

Taken together, these result in an entirely different way of thinking about programming. Much of our time this semester will be spent exploring this way of thinking.

Pure

Haskell expressions are always *referentially transparent*, that is:

- No mutation! Everything (variables, data structures…) is *immutable*.
- Expressions never have "side effects" (like updating global variables or printing to the screen).
- Calling the same function with the same arguments results in the same output every time.

This may sound crazy at this point. How is it even possible to get anything done without mutation or side effects? Well, it certainly requires a shift in thinking (if you're used to an imperative or object-oriented paradigm). But once you've made the shift, there are a number of wonderful benefits:

- *Equational reasoning and refactoring*: In Haskell one can always "replace equals by equals", just like you learned in algebra class.
- *Parallelism*: Evaluating expressions in parallel is easy when they are guaranteed not to affect one another.
- *Fewer headaches*: Simply put, unrestricted effects and action-at-a-distance makes for programs that are hard to debug, maintain, and reason about.

Lazy

In Haskell, expressions are *not evaluated until their results are actually needed*. This is a simple decision with far-reaching consequences, which we will explore throughout the semester. Some of the consequences include:

- It is easy to define a new *control structure* just by defining a function.
- It is possible to define and work with *infinite data structures*.
- It enables a more compositional programming style (see *wholemeal programming* below).
- One major downside, however, is that reasoning about time and space usage becomes much more complicated!

Statically typed

Every Haskell expression has a type, and types are all checked at *compile-time*. Programs with type errors will not even compile, much less run.

Themes

Throughout this course, we will focus on three main themes.

Types

Static type systems can seem annoying. In fact, in languages like C++ and Java, they *are* annoying. But this isn't because static type systems *per se* are annoying; it's because C++ and Java's type systems are insufficiently expressive! This semester we'll take a close look at Haskell's type system, which

- *Helps clarify thinking and express program structure*

  The first step in writing a Haskell program is usually to *write down all the types*. Because Haskell's type system is so expressive, this is a non-trivial design step and is an immense help in clarifying one's thinking about the program.

- *Serves as a form of documentation*

  Given an expressive type system, just looking at a function's type tells you a lot about what the function might do and how it can be used, even before you have read a single word of written documentation.

- *Turns run-time errors into compile-time errors*

  It's much better to be able to fix errors up front than to just test a lot and hope for the best. "If it compiles, it must be correct" is mostly facetious (it's still quite possible to have errors in logic even in a type-correct program), but it happens in Haskell much more than in other languages.

Abstraction

"Don't Repeat Yourself" is a mantra often heard in the world of programming. Also known as the "Abstraction Principle", the idea is that nothing should be duplicated: every idea, algorithm, and piece of data should occur exactly once in your code. Taking similar pieces of code and factoring out their commonality is known as the process of *abstraction*.

Haskell is very good at abstraction: features like parametric polymorphism, higher-order functions, and type classes all aid in the fight against repetition. Our journey through Haskell this semester will in large part be a journey from the specific to the abstract.

Wholemeal programming

Another theme we will explore is *wholemeal programming*. A quote from Ralf Hinze:

> "Functional languages excel at wholemeal programming, a term coined by Geraint Jones. Wholemeal programming means to think big: work with an entire list, rather than a sequence of elements; develop a solution space, rather than an individual solution; imagine a graph, rather than a single path. The wholemeal approach often offers new insights or provides new perspectives on a given problem. It is nicely complemented by the idea of projective programming: first solve a more general problem, then extract the interesting bits and pieces by transforming the general program into more specialised ones."

For example, consider this pseudocode in a C/Java-ish sort of language:

```
int acc = 0;
for ( int i = 0; i < lst.length; i++ ) {
  acc = acc + 3 * lst[i];
}
```

This code suffers from what Richard Bird refers to as "indexitis": it has to worry about the low-level details of iterating over an array by keeping track of a current index. It also mixes together what can more usefully be thought of as two separate operations: multiplying every item in a list by 3, and summing the results.

In Haskell, we can just write

```
sum (map (3*) lst)
```

This semester we'll explore the shift in thinking represented by this way of programming, and examine how and why Haskell makes it possible.

Literate Haskell

This file is a "literate Haskell document": only lines preceded by > and a space (see below) are code; everything else (like this paragraph) is a comment. Your programming assignments do not have to be literate Haskell, although they may be if you like. Literate Haskell documents have an extension of `.lhs`, whereas non-literate Haskell source files use `.hs`.

Declarations and variables

Here is some Haskell code:

```
x :: Int
x = 3

-- Note that normal (non-literate) comments are preceded by two hyphens
{- or enclosed
   in curly brace/hyphen pairs. -}
```

The above code declares a variable `x` with type `Int` (`::` is pronounced "has type") and declares the value of `x` to be `3`. Note that *this will be the value of* `x` *forever* (at least, in this particular program). The value of `x` cannot be changed later.

Try uncommenting the line below; it will generate an error saying something like `Multiple declarations of `x'`.

```
-- x = 4
```

In Haskell, *variables are not mutable boxes*; they are just names for values!

Put another way, `=` does *not* denote "assignment" like it does in many other languages. Instead, `=` denotes *definition*, like it does in mathematics. That is, `x = 4` should not be read as "`x` gets `4`" or "assign `4` to `x`", but as "`x` is *defined to be* `4`".

What do you think this code means?

```
y :: Int
y = y + 1
```

Basic Types

```
-- Machine-sized integers
i :: Int
i = -78
```

`Int`s are guaranteed by the Haskell language standard to accommodate values at least up to \(\pm 2^{29}\), but the exact size depends on your architecture. For example, on my 64-bit machine the range is \(\pm 2^{63}\). You can find the range on your machine by evaluating the following:

```
biggestInt, smallestInt :: Int
biggestInt  = maxBound
smallestInt = minBound
```

(Note that idiomatic Haskell uses `camelCase` for identifier names. If you don't like it, tough luck.)

The `Integer` type, on the other hand, is limited only by the amount of memory on your machine.

```
-- Arbitrary-precision integers
n :: Integer
n = 123456789098765432198734098233498734987234987453 4

reallyBig :: Integer
reallyBig = 2^(2^(2^(2^2)))

numDigits :: Int
numDigits = length (show reallyBig)
```

For floating-point numbers, there is `Double`:

```
-- Double-precision floating point
d1, d2 :: Double
d1 = 4.5387
d2 = 6.2831e-4
```

There is also a single-precision floating point number type, `Float`.

Finally, there are booleans, characters, and strings:

```
-- Booleans
b1, b2 :: Bool
b1 = True
```

```haskell
b2 = False

-- Unicode characters
c1, c2, c3 :: Char
c1 = 'x'
c2 = '∅'
c3 = ' '

-- Strings are lists of characters with special syntax
s :: String
s = "Hello, Haskell!"
```

GHCi

GHCi is an interactive Haskell REPL (Read-Eval-Print-Loop) that comes with GHC. At the GHCi prompt, you can evaluate expressions, load Haskell files with `:load` (`:l`) (and reload them with `:reload` (`:r`)), ask for the type of an expression with `:type` (`:t`), and many other things (try `:?` for a list of commands).

Arithmetic

Try evaluating each of the following expressions in GHCi:

```haskell
ex01 = 3 + 2
ex02 = 19 - 27
ex03 = 2.35 * 8.6
ex04 = 8.7 / 3.1
ex05 = mod 19 3
ex06 = 19 `mod` 3
ex07 = 7 ^ 222
exNN = (-3) * (-7)
```

Note how ‘backticks‘ make a function name into an infix operator. Note also that negative numbers must often be surrounded by parentheses, to avoid having the negation sign parsed as subtraction. (Yes, this is ugly. I’m sorry.)

This, however, gives an error:

```haskell
-- badArith1 = i + n
```

Addition is only between values of the same numeric type, and Haskell does not do implicit conversion. You must explicitly convert with:

- `fromIntegral`: converts from any integral type (`Int` or `Integer`) to any other numeric type.
- `round`, `floor`, `ceiling`: convert floating-point numbers to `Int` or `Integer`.

Now try this:

```haskell
-- badArith2 = i / i
```

This is an error since `/` performs floating-point division only. For integer division we can use `div`.

```
ex08 = i `div` i
ex09 = 12 `div` 5
```

If you are used to other languages which do implicit conversion of numeric types, this can all seem rather prudish and annoying at first. However, I promise you'll get used to it—and in time you may even come to appreciate it. Implicit numeric conversion encourages sloppy thinking about numeric code.

Boolean logic

As you would expect, Boolean values can be combined with `(&&)` (logical and), `(||)` (logical or), and `not`. For example,

```
ex10 = True && False
ex11 = not (False || True)
```

Things can be compared for equality with `(==)` and `(/=)`, or compared for order using `(<)`, `(>)`, `(<=)`, and `(>=)`.

```
ex12 = ('a' == 'a')
ex13 = (16 /= 3)
ex14 = (5 > 3) && ('p' <= 'q')
ex15 = "Haskell" > "C++"
```

Haskell also has `if`-expressions: `if b then t else f` is an expression which evaluates to `t` if the Boolean expression `b` evaluates to `True`, and `f` if `b` evaluates to `False`. Notice that `if`-*expressions* are very different than `if`-*statements*. For example, with an if-statement, the `else` part can be optional; an omitted `else` clause means "if the test evaluates to `False` then do nothing". With an `if`-expression, on the other hand, the `else` part is required, since the `if`-expression must result in some value.

Idiomatic Haskell does not use `if` expressions very much, often using pattern-matching or *guards* instead (see the next section).

Defining basic functions

We can write functions on integers by cases.

```
-- Compute the sum of the integers from 1 to n.
sumtorial :: Integer -> Integer
sumtorial 0 = 0
sumtorial n = n + sumtorial (n-1)
```

Note the syntax for the type of a function: `sumtorial :: Integer -> Integer` says that `sumtorial` is a function which takes an `Integer` as input and yields another `Integer` as output.

Each clause is checked in order from top to bottom, and the first matching clause is chosen. For example, `sumtorial 0` evaluates to `0`, since the first clause is matched. `sumtorial 3` does not match the first clause (`3` is not `0`), so the second clause is tried. A variable like `n` matches anything, so the second clause matches and `sumtorial 3` evaluates to `3 + sumtorial (3-1)` (which can then be evaluated further).

Choices can also be made based on arbitrary Boolean expressions using *guards*. For example:

```
hailstone :: Integer -> Integer
hailstone n
  | n `mod` 2 == 0 = n `div` 2
  | otherwise      = 3*n + 1
```

Any number of guards can be associated with each clause of a function definition, each of which is a Boolean expression. If the clause's patterns match, the guards are evaluated in order from top to bottom, and the first one which evaluates to `True` is chosen. If none of the guards evaluate to `True`, matching continues with the next clause.

For example, suppose we evaluate `hailstone 3`. First, `3` is matched against `n`, which succeeds (since a variable matches anything). Next, `n `mod` 2 == 0` is evaluated; it is `False` since `n = 3` does not result in a remainder of `0` when divided by `2`. `otherwise` is just an convenient synonym for `True`, so the second guard is chosen, and the result of `hailstone 3` is thus `3*3 + 1 = 10`.

As a more complex (but more contrived) example:

```
foo :: Integer -> Integer
foo 0 = 16
foo 1
  | "Haskell" > "C++" = 3
  | otherwise         = 4
foo n
  | n < 0            = 0
  | n `mod` 17 == 2  = -43
  | otherwise        = n + 3
```

What is `foo (-3)`? `foo 0`? `foo 1`? `foo 36`? `foo 38`?

As a final note about Boolean expressions and guards, suppose we wanted to abstract out the test of evenness used in defining `hailstone`. A first attempt is shown below:

```
isEven :: Integer -> Bool
isEven n
  | n `mod` 2 == 0 = True
  | otherwise      = False
```

This *works*, but it is much too complicated. Can you see why?

Pairs

We can pair things together like so:

```
p :: (Int, Char)
p = (3, 'x')
```

Notice that the `(x,y)` notation is used both for the *type* of a pair and a pair *value*.

The elements of a pair can be extracted again with *pattern matching*:

```
sumPair :: (Int,Int) -> Int
sumPair (x,y) = x + y
```

Haskell also has triples, quadruples, … but you should never use them. As we'll see next week, there are much better ways to package three or more pieces of information together.

Using functions, and multiple arguments

To apply a function to some arguments, just list the arguments after the function, separated by spaces, like this:

```
f :: Int -> Int -> Int -> Int
f x y z = x + y + z
exFF = f 3 17 8
```

The above example applies the function `f` to the three arguments `3`, `17`, and `8`. Note also the syntax for the type of a function with multiple arguments, like `Arg1Type -> Arg2Type -> ... -> ResultType`. This might seem strange to you (and it should!). Why all the arrows? Wouldn't it make more sense for the type of `f` to be something like `Int Int Int -> Int`? Actually, the syntax is no accident: it is the way it is for a very deep and beautiful reason, which we'll learn about in a few weeks; for now you just have to take my word for it!

Note that **function application has higher precedence than any infix operators**. So it would be incorrect to write

`f 3 n+1 7`

if you intend to pass `n+1` as the second argument to `f`, because this parses as

`(f 3 n) + (1 7)`.

Instead, one must write

`f 3 (n+1) 7`.

Lists

Lists are one of the most basic data types in Haskell.

```
nums, range, range2 :: [Integer]
nums   = [1,2,3,19]
range  = [1..100]
range2 = [2,4..100]
```

Haskell (like Python) also has *list comprehensions*; you can read about them in LYAH.

Strings are just lists of characters. That is, `String` is just an abbreviation for `[Char]`, and string literal syntax (text surrounded by double quotes) is just an abbreviation for a list of `Char` literals.

```
-- hello1 and hello2 are exactly the same.

hello1 :: [Char]
hello1 = ['h', 'e', 'l', 'l', 'o']

hello2 :: String
hello2 = "hello"

helloSame = hello1 == hello2
```

This means that all the standard library functions for processing lists can also be used to process `String`s.

Constructing lists

The simplest possible list is the empty list:

```
emptyList = []
```

Other lists are built up from the empty list using the *cons* operator, `(:)`. Cons takes an element and a list, and produces a new list with the element prepended to the front.

```
ex17 = 1 : []
ex18 = 3 : (1 : [])
ex19 = 2 : 3 : 4 : []

ex20 = [2,3,4] == 2 : 3 : 4 : []
```

We can see that `[2,3,4]` notation is just convenient shorthand for `2 : 3 : 4 : []`. Note also that these are really *singly linked lists*, NOT arrays.

```
-- Generate the sequence of hailstone iterations from a starting number.
hailstoneSeq :: Integer -> [Integer]
hailstoneSeq 1 = [1]
hailstoneSeq n = n : hailstoneSeq (hailstone n)
```

We stop the hailstone sequence when we reach 1. The hailstone sequence for a general `n` consists of `n` itself, followed by the hailstone sequence for `hailstone n`, that is, the number obtained by applying the hailstone transformation once to `n`.

Functions on lists

We can write functions on lists using *pattern matching*.

```
-- Compute the length of a list of Integers.
intListLength :: [Integer] -> Integer
intListLength []     = 0
intListLength (x:xs) = 1 + intListLength xs
```

The first clause says that the length of an empty list is 0. The second clause says that if the input list looks like `(x:xs)`, that is, a first element `x` consed onto a remaining list `xs`, then the length is one more than the length of `xs`.

Since we don't use `x` at all we could also replace it by an underscore: `intListLength (_:xs) = 1 + intListLength xs`.

We can also use nested patterns:

```
sumEveryTwo :: [Integer] -> [Integer]
sumEveryTwo []         = []     -- Do nothing to the empty list
sumEveryTwo (x:[])     = [x]    -- Do nothing to lists with a single element
sumEveryTwo (x:(y:zs)) = (x + y) : sumEveryTwo zs
```

Note how the last clause matches a list starting with `x` and followed by… a list starting with `y` and followed by the list `zs`. We don't actually need the extra parentheses, so `sumEveryTwo (x:y:zs) = ...` would be equivalent.

Combining functions

It's good Haskell style to build up more complex functions by combining many simple ones.

```
-- The number of hailstone steps needed to reach 1 from a starting
-- number.
hailstoneLen :: Integer -> Integer
hailstoneLen n = intListLength (hailstoneSeq n) - 1
```

This may seem inefficient to you: it generates the entire hailstone sequence first and then finds its length, which wastes lots of memory… doesn't it? Actually, it doesn't! Because of Haskell's lazy evaluation, each element of the sequence is only generated as needed, so the sequence generation and list length calculation are interleaved. The whole computation uses only O(1) memory, no matter how long the sequence. (Actually, this is a tiny white lie, but explaining why (and how to fix it) will have to wait a few weeks.)

We'll learn more about Haskell's lazy evaluation strategy in a few weeks. For now, the take-home message is: don't be afraid to write small functions that transform whole data structures, and combine them to produce more complex functions. It may feel unnatural at first, but it's the way to write idiomatic (and efficient) Haskell, and is actually a rather pleasant way to write programs once you get used to it.

A word about error messages

Actually, six:

Don't be scared of error messages!

GHC's error messages can be rather long and (seemingly) scary. However, usually they're long not because they are obscure, but because they contain a lot of useful information! Here's an example:

```
Prelude> 'x' ++ "foo"

<interactive>:1:1:
    Couldn't match expected type `[a0]' with actual type `Char'
    In the first argument of `(++)', namely 'x'
    In the expression: 'x' ++ "foo"
    In an equation for `it': it = 'x' ++ "foo"
```

First we are told "Couldn't match expected type `[a0]` with actual type `Char`". This means that *something* was expected to have a list type, but actually had type `Char`. What something? The next line tells us: it's the first argument of `(++)` which is at fault, namely, `'x'`. The next lines go on to give us a bit more context. Now we can see what the problem is: clearly `'x'` has type `Char`, as the first line said. Why would it be expected to have a list type? Well, because it is used as an argument to `(++)`, which takes a list as its first argument.

When you get a huge error message, resist your initial impulse to run away; take a deep breath; and read it carefully. You won't necessarily understand the entire thing, but you will probably learn a lot, and you may just get enough information to figure out what the problem is.

Algebraic data types

CIS 194 Week 2
21 January 2013

Suggested reading:

- Real World Haskell, chapters 2 and 3

Enumeration types

Like many programming languages, Haskell allows programmers to create their own *enumeration* types. Here's a simple example:

```
data Thing = Shoe
           | Ship
           | SealingWax
           | Cabbage
           | King
  deriving Show
```

This declares a new type called `Thing` with five *data constructors* `Shoe`, `Ship`, etc. which are the (only) values of type `Thing`. (The `deriving Show` is a magical incantation which tells GHC to automatically generate default code for converting `Thing`s to `String`s. This is what `ghci` uses when printing the value of an expression of type `Thing`.)

```
shoe :: Thing
shoe = Shoe

listO'Things :: [Thing]
listO'Things = [Shoe, SealingWax, King, Cabbage, King]
```

We can write functions on `Thing`s by *pattern-matching*.

```
isSmall :: Thing -> Bool
isSmall Shoe       = True
isSmall Ship       = False
isSmall SealingWax = True
isSmall Cabbage    = True
isSmall King       = False
```

Recalling how function clauses are tried in order from top to bottom, we could also make the definition of `isSmall` a bit shorter like so:

```
isSmall2 :: Thing -> Bool
isSmall2 Ship = False
isSmall2 King = False
isSmall2 _    = True
```

Beyond enumerations

`Thing` is an *enumeration type*, similar to those provided by other languages such as Java or C++. However, enumerations are actually only a special case of Haskell's more general *algebraic data types*. As a first example of a data type which is not just an enumeration, consider the definition of `FailableDouble`:

```
data FailableDouble = Failure
                    | OK Double
  deriving Show
```

This says that the `FailableDouble` type has two data constructors. The first one, `Failure`, takes no arguments, so `Failure` by itself is a value of type `FailableDouble`. The second one, `OK`, takes an argument of type `Double`. So `OK` by itself is not a value of type `FailableDouble`; we need to give it a `Double`. For example, `OK 3.4` is a value of type `FailableDouble`.

```
exD1 = Failure
exD2 = OK 3.4
```

Thought exercise: what is the type of `OK`?

```
safeDiv :: Double -> Double -> FailableDouble
safeDiv _ 0 = Failure
safeDiv x y = OK (x / y)
```

More pattern-matching! Notice how in the `OK` case we can give a name to the `Double` that comes along with it.

```
failureToZero :: FailableDouble -> Double
failureToZero Failure = 0
failureToZero (OK d)  = d
```

Data constructors can have more than one argument.

```
-- Store a person's name, age, and favourite Thing.
data Person = Person String Int Thing
  deriving Show

brent :: Person
brent = Person "Brent" 31 SealingWax

stan :: Person
stan  = Person "Stan" 94 Cabbage

getAge :: Person -> Int
getAge (Person _ a _) = a
```

Notice how the type constructor and data constructor are both named `Person`, but they inhabit different namespaces and are different things. This idiom (giving the type and data constructor of a one-constructor type the same name) is common, but can be confusing until you get used to it.

Algebraic data types in general

In general, an algebraic data type has one or more data constructors, and each data constructor can have zero or more arguments.

```
data AlgDataType = Constr1 Type11 Type12
                 | Constr2 Type21
                 | Constr3 Type31 Type32 Type33
                 | Constr4
```

This specifies that a value of type `AlgDataType` can be constructed in one of four ways: using `Constr1`, `Constr2`, `Constr3`, or `Constr4`. Depending on the constructor used, an `AlgDataType` value may contain some other values. For example, if it was constructed using `Constr1`, then it comes along with two values, one of type `Type11` and one of type `Type12`.

One final note: type and data constructor names must always start with a capital letter; variables (including names of functions) must always start with a lowercase letter. (Otherwise, Haskell parsers would have quite a difficult job figuring out which names represent variables and which represent constructors).

Pattern-matching

We've seen pattern-matching in a few specific cases, but let's see how pattern-matching works in general. Fundamentally, pattern-matching is about taking apart a value by *finding out which constructor* it was built with. This information can be used as the basis for deciding what to do—indeed, in Haskell, this is the *only* way to make a decision.

For example, to decide what to do with a value of type `AlgDataType` (the made-up type defined in the previous section), we could write something like

```
foo (Constr1 a b)   = ...
foo (Constr2 a)     = ...
foo (Constr3 a b c) = ...
foo Constr4         = ...
```

Note how we also get to give names to the values that come along with each constructor. Note also that parentheses are required around patterns consisting of more than just a single constructor.

This is the main idea behind patterns, but there are a few more things to note.

1. An underscore `_` can be used as a "wildcard pattern" which matches anything.
2. A pattern of the form `x@pat` can be used to match a value against the pattern `pat`, but *also* give the name `x` to the entire value being matched. For example:

   ```
   baz :: Person -> String
   baz p@(Person n _ _) = "The name field of (" ++ show p ++ ") is " ++ n

   *Main> baz brent
   "The name field of (Person \"Brent\" 31 SealingWax) is Brent"
   ```

3. Patterns can be *nested*. For example:

   ```
   checkFav :: Person -> String
   checkFav (Person n _ SealingWax) = n ++ ", you're my kind of person!"
   checkFav (Person n _ _)          = n ++ ", your favorite thing is lame."

   *Main> checkFav brent
   "Brent, you're my kind of person!"
   *Main> checkFav stan
   "Stan, your favorite thing is lame."
   ```

   Note how we nest the pattern `SealingWax` inside the pattern for `Person`.

In general, the following grammar defines what can be used as a pattern:

```
pat ::= _
     |  var
     |  var @ ( pat )
     |  ( Constructor pat1 pat2 ... patn )
```

The first line says that an underscore is a pattern. The second line says that a variable by itself is a pattern: such a pattern matches anything, and "binds" the given variable name to the matched value. The third line specifies `@`-patterns. The last line says that a constructor name followed by a sequence of patterns is itself a pattern: such a pattern matches a value if that value was constructed using the given constructor, *and* `pat1` through `patn` all match the values contained by the constructor, recursively.

(In actual fact, the full grammar of patterns includes yet more features still, but the rest would take us too far afield for now.)

Note that literal values like `2` or `'c'` can be thought of as constructors with no arguments. It is as if the types `Int` and `Char` were defined like

```
data Int  = 0 | 1 | -1 | 2 | -2 | ...
data Char = 'a' | 'b' | 'c' | ...
```

which means that we can pattern-match against literal values. (Of course, `Int` and `Char` are not *actually* defined this way.)

Case expressions

The fundamental construct for doing pattern-matching in Haskell is the `case` expression. In general, a `case` expression looks like

```
case exp of
  pat1 -> exp1
  pat2 -> exp2
  ...
```

When evaluated, the expression `exp` is matched against each of the patterns `pat1`, `pat2`, … in turn. The first matching pattern is chosen, and the entire `case` expression evaluates to the expression corresponding to the matching pattern. For example,

```
exCase = case "Hello" of
           []      -> 3
           ('H':s) -> length s
           _       -> 7
```

evaluates to `4` (the second pattern is chosen; the third pattern matches too, of course, but it is never reached).

In fact, the syntax for defining functions we have seen is really just convenient syntax sugar for defining a `case` expression. For example, the definition of `failureToZero` given previously can equivalently be written as

```
failureToZero' :: FailableDouble -> Double
failureToZero' x = case x of
                     Failure -> 0
                     OK d    -> d
```

Recursive data types

Data types can be *recursive*, that is, defined in terms of themselves. In fact, we have already seen a recursive type—the type of lists. A list is either empty, or a single element followed by a remaining list. We could define our own list type like so:

```
data IntList = Empty | Cons Int IntList
```

Haskell's own built-in lists are quite similar; they just get to use special built-in syntax (`[]` and `:`). (Of course, they also work for any type of elements instead of just `Int`s; more on this next week.)

We often use recursive functions to process recursive data types:

```
intListProd :: IntList -> Int
intListProd Empty      = 1
intListProd (Cons x l) = x * intListProd l
```

As another simple example, we can define a type of binary trees with an `Int` value stored at each internal node, and a `Char` stored at each leaf:

```
data Tree = Leaf Char
          | Node Tree Int Tree
  deriving Show
```

(Don't ask me what you would use such a tree for; it's an example, OK?) For example,

tree :: Tree tree = Node (Leaf 'x') 1 (Node (Leaf 'y') 2 (Leaf 'z'))

Recursion patterns, polymorphism, and the Prelude

CIS 194 Week 3
28 January 2013

While completing HW 2, you probably spent a lot of time writing explicitly recursive functions. At this point, you might think that's what Haskell programmers spend most of their time doing. In fact, experienced Haskell programmers *hardly ever* write recursive functions!

How is this possible? The key is to notice that although recursive functions can theoretically do pretty much anything, in practice there are certain common patterns that come up over and over again. By abstracting out these patterns into library functions, programmers can leave the low-level details of actually doing recursion to these functions, and think about problems at a higher level—that's the goal of *wholemeal programming*.

Recursion patterns

Recall our simple definition of lists of `Int` values:

```
data IntList = Empty | Cons Int IntList
  deriving Show
```

What sorts of things might we want to do with an `IntList`? Here are a few common possibilities:

- Perform some operation on every element of the list
- Keep only some elements of the list, and throw others away, based on a test
- "Summarize" the elements of the list somehow (find their sum, product, maximum…).
- You can probably think of others!

Map

Let's think about the first one ("perform some operation on every element of the list"). For example, we could add one to every element in a list:

Or we could ensure that every element in a list is nonnegative by taking the absolute value:

```
absAll :: IntList -> IntList
absAll Empty       = Empty
absAll (Cons x xs) = Cons (abs x) (absAll xs)
```

Or we could square every element:

```
squareAll :: IntList -> IntList
squareAll Empty       = Empty
squareAll (Cons x xs) = Cons (x*x) (squareAll xs)
```

At this point, big flashing red lights and warning bells should be going off in your head. These three functions look way too similar. There ought to be some way to abstract out the commonality so we don't have to repeat ourselves!

There is indeed a way—can you figure it out? Which parts are the same in all three examples and which parts change?

The thing that changes, of course, is the operation we want to perform on each element of the list. We can specify this operation as a *function* of type `Int -> Int`. Here is where we begin to see how incredibly useful it is to be able to pass functions as inputs to other functions!

We can now use `mapIntList` to implement `addOneToAll`, `absAll`, and `squareAll`:

```
exampleList = Cons (-1) (Cons 2 (Cons (-6) Empty))

addOne x = x + 1
square x = x * x

mapIntList addOne exampleList
mapIntList abs    exampleList
mapIntList square exampleList
```

Filter

Another common pattern is when we want to keep only some elements of a list, and throw others away, based on a test. For example, we might want to keep only the positive numbers:

Or only the even ones:

```
keepOnlyEven :: IntList -> IntList
keepOnlyEven Empty = Empty
keepOnlyEven (Cons x xs)
  | even x    = Cons x (keepOnlyEven xs)
  | otherwise = keepOnlyEven xs
```

How can we generalize this pattern? What stays the same, and what do we need to abstract out?

Fold

The final pattern we mentioned was to "summarize" the elements of the list; this is also variously known as a "fold" or "reduce" operation. We'll come back to this next week. In the meantime, you might want to think about how to abstract out this pattern!

Polymorphism

We've now written some nice, general functions for mapping and filtering over lists of `Int`s. But we're not done generalizing! What if we wanted to filter lists of `Integer`s? or `Bool`s? Or lists of lists of trees of stacks of `String`s? We'd have to make a new data type and a new function for each of these cases. Even worse, the *code would be exactly the same*; the only thing that would be different is the *type signatures*. Can't Haskell help us out here?

Of course it can! Haskell supports *polymorphism* for both data types and functions. The word "polymorphic" comes from Greek ( ) and means "having many forms": something which is polymorphic works for multiple types.

Polymorphic data types

First, let's see how to declare a polymorphic data type.

```
data List t = E | C t (List t)
```

(We can't reuse `Empty` and `Cons` since we already used those for the constructors of `IntList`, so we'll use `E` and `C` instead.) Whereas before we had `data IntList = ...`, we now have `data List t = ...` The `t` is a *type variable* which can stand for any type. (Type variables must start with a lowercase letter, whereas types must start with uppercase.) `data List t = ...` means that the `List` type is *parameterized* by a type, in much the same way that a function can be parameterized by some input.

Given a type `t`, a `(List t)` consists of either the constructor `E`, or the constructor `C` along with a value of type `t` and another `(List t)`. Here are some examples:

```
lst1 :: List Int
lst1 = C 3 (C 5 (C 2 E))

lst2 :: List Char
lst2 = C 'x' (C 'y' (C 'z' E))

lst3 :: List Bool
lst3 = C True (C False E)
```

Polymorphic functions

Now, let's generalize `filterIntList` to work over our new polymorphic `List`s. We can just take code of `filterIntList` and replace `Empty` by `E` and `Cons` by `C`:

```
filterList _ E = E
filterList p (C x xs)
  | p x       = C x (filterList p xs)
  | otherwise = filterList p xs
```

Now, what is the type of `filterList`? Let's see what type `ghci` infers for it:

```
*Main> :t filterList
filterList :: (t -> Bool) -> List t -> List t
```

We can read this as: "for any type `t`, `filterList` takes a function from `t` to `Bool`, and a list of `t`'s, and returns a list of `t`'s."

What about generalizing `mapIntList`? What type should we give to a function `mapList` that applies a function to every element in a `List t`?

Our first idea might be to give it the type

```
mapList :: (t -> t) -> List t -> List t
```

This works, but it means that when applying `mapList`, we always get a list with the same type of elements as the list we started with. This is overly restrictive: we'd like to be able to do things like `mapList show` in order to convert, say, a list of `Int`s into a list of `String`s. Here, then, is the most general possible type for `mapList`, along with an implementation:

```
mapList :: (a -> b) -> List a -> List b
mapList _ E        = E
mapList f (C x xs) = C (f x) (mapList f xs)
```

One important thing to remember about polymorphic functions is that **the caller gets to pick the types**. When you write a polymorphic function, it must work for every possible input type. This—together with the fact that Haskell has no way to directly make make decisions based on what type something is—has some interesting implications which we'll explore later.

The Prelude

The `Prelude` is a module with a bunch of standard definitions that gets implicitly imported into every Haskell program. It's worth spending some time skimming through its documentation to familiarize oneself with the tools that are available.

Of course, polymorphic lists are defined in the `Prelude`, along with many useful polymorphic functions for working with them. For example, `filter` and `map` are the counterparts to our `filterList` and `mapList`. In fact, the `Data.List` module contains many more list functions still.

Another useful polymorphic type to know is `Maybe`, defined as

```
data Maybe a = Nothing | Just a
```

A value of type `Maybe a` either contains a value of type `a` (wrapped in the `Just` constructor), or it is `Nothing` (representing some sort of failure or error). The `Data.Maybe` module has functions for working with `Maybe` values.

Total and partial functions

Consider this polymorphic type:

```
[a] -> a
```

What functions could have such a type? The type says that given a list of things of type `a`, the function must produce some value of type `a`. For example, the Prelude function `head` has this type.

…But what happens if `head` is given an empty list as input? Let's look at the source code for `head`…

It crashes! There's nothing else it possibly could do, since it must work for *all* types. There's no way to make up an element of an arbitrary type out of thin air.

`head` is what is known as a *partial function*: there are certain inputs for which `head` will crash. Functions which have certain inputs that will make them recurse infinitely are also called partial. Functions which are well-defined on all possible inputs are known as *total functions*.

It is good Haskell practice to avoid partial functions as much as possible. Actually, avoiding partial functions is good practice in *any* programming language—but in most of them it's ridiculously annoying. Haskell tends to make it quite easy and sensible.

`head` is a mistake! It should not be in the `Prelude`. Other partial `Prelude` functions you should almost never use include `tail`, `init`, `last`, and `(!!)`. From this point on, using one of these functions on a homework assignment will lose style points!

What to do instead?

Replacing partial functions

Often partial functions like `head`, `tail`, and so on can be replaced by pattern-matching. Consider the following two definitions:

```
doStuff1 :: [Int] -> Int
doStuff1 []  = 0
doStuff1 [_] = 0
doStuff1 xs  = head xs + (head (tail xs))

doStuff2 :: [Int] -> Int
doStuff2 []        = 0
doStuff2 [_]       = 0
doStuff2 (x1:x2:_) = x1 + x2
```

These functions compute exactly the same result, and they are both total. But only the second one is *obviously* total, and it is much easier to read anyway.

Writing partial functions

What if you find yourself *writing* a partial functions? There are two approaches to take. The first is to change the output type of the function to indicate the possible failure. Recall the definition of `Maybe`:

```
data Maybe a = Nothing | Just a
```

Now, suppose we were writing `head`. We could rewrite it safely like this:

```
safeHead :: [a] -> Maybe a
safeHead []    = Nothing
safeHead (x:_) = Just x
```

Indeed, there is exactly such a function defined in the `safe` package.

Why is this a good idea?

1. `safeHead` will never crash.

2. The type of `safeHead` makes it obvious that it may fail for some inputs.
3. The type system ensures that users of `safeHead` must appropriately check the return value of `safeHead` to see whether they got a value or `Nothing`.

In some sense, `safeHead` is still "partial"; but we have reflected the partiality in the type system, so it is now safe. The goal is to have the types tell us as much as possible about the behavior of functions.

OK, but what if we know that we will only use `head` in situations where we are *guaranteed* to have a non-empty list? In such a situation, it is really annoying to get back a `Maybe a`, since we have to expend effort dealing with a case which we "know" cannot actually happen.

The answer is that if some condition is really *guaranteed*, then the types ought to reflect the guarantee! Then the compiler can enforce your guarantees for you. For example:

```
data NonEmptyList a = NEL a [a]

nelToList :: NonEmptyList a -> [a]
nelToList (NEL x xs) = x:xs

listToNel :: [a] -> Maybe (NonEmptyList a)
listToNel []     = Nothing
listToNel (x:xs) = Just $ NEL x xs

headNEL :: NonEmptyList a -> a
headNEL (NEL a _) = a

tailNEL :: NonEmptyList a -> [a]
tailNEL (NEL _ as) = as
```

You might think doing such things is only for chumps who are not coding super-geniuses like you. Of course, *you* would never make a mistake like passing an empty list to a function which expects only non-empty ones. Right? Well, there's definitely a chump involved, but it's not who you think.

Higher-order programming and type inference

CIS 194 Week 4
4 February 2013

Suggested reading:

- *Learn You a Haskell for Great Good* chapter "Higher-Order Functions" (Chapter 5 in the printed book; Chapter 6 online)

Anonymous functions

Suppose we want to write a function

```
greaterThan100 :: [Integer] -> [Integer]
```

which keeps only those `Integers` from the input list which are greater than 100. For example,

`greaterThan100 [1,9,349,6,907,98,105] = [349,907,105].`

By now, we know a nice way to do this:

```
gt100 :: Integer -> Bool
gt100 x = x > 100

greaterThan100 :: [Integer] -> [Integer]
greaterThan100 xs = filter gt100 xs
```

But it's annoying to give `gt100` a name, since we are probably never going to use it again. Instead, we can use an *anonymous function*, also known as a *lambda abstraction*:

```
greaterThan100_2 :: [Integer] -> [Integer]
greaterThan100_2 xs = filter (\x -> x > 100) xs
```

`\x -> x > 100` (the backslash is supposed to look kind of like a lambda with the short leg missing) is the function which takes a single argument `x` and outputs whether `x` is greater than 100.

Lambda abstractions can also have multiple arguments. For example:

```
Prelude> (\x y z -> [x,2*y,3*z]) 5 6 3
[5,12,9]
```

However, in the particular case of `greaterThan100`, there's an even better way to write it, without a lambda abstraction:

```
greaterThan100_3 :: [Integer] -> [Integer]
greaterThan100_3 xs = filter (>100) xs
```

`(>100)` is an *operator section*: if `?` is an operator, then `(?y)` is equivalent to the function `\x -> x ? y`, and `(y?)` is equivalent to `\x -> y ? x`. In other words, using an operator section allows us to *partially apply* an operator to one of its two arguments. What we get is a function of a single argument. Here are some examples:

```
Prelude> (>100) 102
True
Prelude> (100>) 102
False
Prelude> map (*6) [1..5]
[6,12,18,24,30]
```

Function composition

Before reading on, can you write down a function whose type is

```
(b -> c) -> (a -> b) -> (a -> c)
```

?

Let's try. It has to take two arguments, both of which are functions, and output a function.

```
foo f g = ...
```

In the place of the `...` we need to write a function of type `a -> c`. Well, we can create a function using a lambda abstraction:

```
foo f g = \x -> ...
```

`x` will have type `a`, and now in the `...` we need to write an expression of type `c`. Well, we have a function `g` which can turn an `a` into a `b`, and a function `f` which can turn a `b` into a `c`, so this ought to work:

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

(Quick quiz: why do we need the parentheses around `g x`?)

OK, so what was the point of that? Does `foo` actually do anything useful or was that just a silly exercise in working with types?

As it turns out, `foo` is really called `(.)`, and represents *function composition*. That is, if `f` and `g` are functions, then `f . g` is the function which does first `g` and then `f`.

Function composition can be quite useful in writing concise, elegant code. It fits well in a "wholemeal" style where we think about composing together successive high-level transformations of a data structure.

As an example, consider the following function:

```
myTest :: [Integer] -> Bool
myTest xs = even (length (greaterThan100 xs))
```

We can rewrite this as:

```
myTest' :: [Integer] -> Bool
myTest' = even . length . greaterThan100
```

This version makes much clearer what is really going on: `myTest'` is just a "pipeline" composed of three smaller functions. This example also demonstrates why function composition seems "backwards": it's because function application is backwards! Since we read from left to right, it would make sense to think of values as also flowing from left to right. But in that case we should write \((x)f \) to denote giving the value \(x\) as an input to the function \(f\). But no thanks to Alexis Claude Clairaut and Euler, we have been stuck with the backwards notation since 1734.

Let's take a closer look at the type of `(.)`. If we ask `ghci` for its type, we get

```
Prelude> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Wait a minute. What's going on here? What happened to the parentheses around `(a -> c)`?

Currying and partial application

Remember how the types of multi-argument functions look weird, like they have "extra" arrows in them? For example, consider the function

```
f :: Int -> Int -> Int
f x y = 2*x + y
```

I promised before that there is a beautiful, deep reason for this, and now it's finally time to reveal it: *all functions in Haskell take only one argument*. Say what?! But doesn't the function `f` shown above take two arguments? No, actually, it doesn't: it takes one argument (an `Int`) and *outputs a function* (of type `Int -> Int`); that function takes one argument and returns the final answer. In fact, we can equivalently write `f`'s type like this:

```
f' :: Int -> (Int -> Int)
f' x y = 2*x + y
```

In particular, note that function arrows *associate to the right*, that is, `W -> X -> Y -> Z` is equivalent to `W -> (X -> (Y -> Z))`. We can always add or remove parentheses around the rightmost top-level arrow in a type.

Function application, in turn, is *left*-associative. That is, `f 3 2` is really shorthand for `(f 3) 2`. This makes sense given what we said previously about `f` actually taking one argument and returning a function: we apply `f` to an argument `3`, which returns a function of type `Int -> Int`, namely, a function which takes an `Int` and adds 6 to it. We then apply that function to the argument `2` by writing `(f 3) 2`, which gives us an `Int`. Since function application associates to the left, however, we can abbreviate `(f 3) 2` as `f 3 2`, giving us a nice notation for `f` as a "multi-argument" function.

The "multi-argument" lambda abstraction

```
\x y z -> ...
```

is really just syntax sugar for

```
\x -> (\y -> (\z -> ...)).
```

Likewise, the function definition

```
f x y z = ...
```

is syntax sugar for

```
f = \x -> (\y -> (\z -> ...)).
```

Note, for example, that we can rewrite our composition function from above by moving the `\x -> ...` from the right-hand side of the `=` to the left-hand side:

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

This idea of representing multi-argument functions as one-argument functions returning functions is known as *currying*, named for the British mathematician and logician Haskell Curry. (His first name might sound familiar; yes, it's the same guy.) Curry lived from 1900-1982 and spent much of his life at Penn State—but he also helped work on ENIAC at UPenn. The idea of representing multi-argument functions as one-argument functions returning functions was actually first discovered by Moses Schönfinkel, so we probably ought to call it *schönfinkeling*. Curry himself attributed the idea to Schönfinkel, but others had already started calling it "currying" and it was too late.

If we want to actually represent a function of two arguments we can use a single argument which is a tuple. That is, the function

```
f'' :: (Int,Int) -> Int
f'' (x,y) = 2*x + y
```

can also be thought of as taking "two arguments", although in another sense it really only takes one argument which happens to be a pair. In order to convert between the two representations of a two-argument function, the standard library defines functions called `curry` and `uncurry`, defined like this (except with different names):

```
schönfinkel :: ((a,b) -> c) -> a -> b -> c
schönfinkel f x y = f (x,y)

unschönfinkel :: (a -> b -> c) -> (a,b) -> c
unschönfinkel f (x,y) = f x y
```

`uncurry` in particular can be useful when you have a pair and want to apply a function to it. For example:

```
Prelude> uncurry (+) (2,3)
5
```

Partial application

The fact that functions in Haskell are curried makes *partial application* particularly easy. The idea of partial application is that we can take a function of multiple arguments and apply it to just *some* of its arguments, and get out a function of the remaining arguments. But as we've just seen, in Haskell there *are no* functions of multiple arguments! Every function can be "partially applied" to its first (and only) argument, resulting in a function of the remaining arguments.

Note that Haskell doesn't make it easy to partially apply to an argument other than the first. The one exception is infix operators, which as we've seen, can be partially applied to either of their two arguments using an operator section. In practice this is not that big of a restriction. There is an art to deciding the order of arguments to a function to make partial applications of it as useful as possible: the arguments should be ordered from from "least to greatest variation", that is, arguments which will often be the same should be listed first, and arguments which will often be different should come last.

Wholemeal programming

Let's put some of the things we've just learned together in an example that also shows the power of a "wholemeal" style of programming. Consider the function `foobar`, defined as follows:

```
foobar :: [Integer] -> Integer
foobar []     = 0
foobar (x:xs)
  | x > 3     = (7*x + 2) + foobar xs
  | otherwise = foobar xs
```

This seems straightforward enough, but it is not good Haskell style. The problem is that it is

- doing too much at once; and
- working at too low of a level.

Instead of thinking about what we want to do with each element, we can instead think about making incremental transformations to the entire input, using the existing recursion patterns that we know of. Here's a much more idiomatic implementation of `foobar`:

```
foobar' :: [Integer] -> Integer
foobar' = sum . map (\x -> 7*x + 2) . filter (>3)
```

This defines `foobar'` as a "pipeline" of three functions: first, we throw away all elements from the list which are not greater than three; next, we apply an arithmetic operation to every element of the remaining list; finally, we sum the results.

Notice that in the above example, `map` and `filter` have been partially applied. For example, the type of `filter` is

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

Applying it to `(>3)` (which has type `Integer -> Bool`) results in a function of type `[Integer] -> [Integer]`, which is exactly the right sort of thing to compose with another function on `[Integer]`.

This style of coding in which we define a function without reference to its arguments—in some sense saying what a function *is* rather than what it *does*—is known as "point-free" style. As we can see from the above example, it can be quite beautiful. Some people might even go so far as to say that you should always strive to use point-free style; but taken too far it can become extremely confusing. `lambdabot` in the `#haskell` IRC channel has a command `@pl` for turning functions into equivalent point-free expressions; here's an example:

```
@pl \f g x y -> f (x ++ g x) (g y)
join . ((flip . ((.) .)) .) . (. ap (++)) . (.)
```

This is clearly *not* an improvement!

Folds

We have one more recursion pattern on lists to talk about: folds. Here are a few functions on lists that follow a similar pattern: all of them somehow "combine" the elements of the list into a final answer.

```
sum' :: [Integer] -> Integer
sum' []     = 0
sum' (x:xs) = x + sum' xs

product' :: [Integer] -> Integer
product' [] = 1
product' (x:xs) = x * product' xs

length' :: [a] -> Int
length' []     = 0
length' (_:xs) = 1 + length' xs
```

What do these three functions have in common, and what is different? As usual, the idea will be to abstract out the parts that vary, aided by the ability to define higher-order functions.

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

Notice how `fold` essentially replaces `[]` with `z` and `(:)` with `f`, that is,

```
fold f z [a,b,c] == a `f` (b `f` (c `f` z))
```

(If you think about `fold` from this perspective, you may be able to figure out how to generalize `fold` to data types other than lists…)

Now let's rewrite `sum'`, `product'`, and `length'` in terms of `fold`:

```
sum''     = fold 0 (+)
product'' = fold 1 (*)
length''  = fold 0 (\_ s -> 1 + s)
```

(Instead of `(\_ s -> 1 + s)` we could also write `(\_ -> (1+))` or even `(const (1+))`.)

Of course, `fold` is already provided in the standard Prelude, under the name `foldr`. The arguments to `foldr` are in a slightly different order but it's the exact same function. Here are some Prelude functions which are defined in terms of `foldr`:

- `length :: [a] -> Int`
- `sum :: Num a => [a] -> a`
- `product :: Num a => [a] -> a`
- `and :: [Bool] -> Bool`
- `or :: [Bool] -> Bool`
- `any :: (a -> Bool) -> [a] -> Bool`
- `all :: (a -> Bool) -> [a] -> Bool`

There is also `foldl`, which folds "from the left". That is,

```
foldr f z [a,b,c] == a `f` (b `f` (c `f` z))
foldl f z [a,b,c] == ((z `f` a) `f` b) `f` c
```

In general, however, you should use `foldl'` from `Data.List` instead, which does the same thing as `foldl` but is more efficient.

More polymorphism and type classes

CIS 194 Week 5
11 February 2013

Haskell's particular brand of polymorphism is known as *parametric* polymorphism. Essentially, this means that polymorphic functions must work *uniformly* for any input type. This turns out to have some interesting implications for both programmers and users of polymorphic functions.

Parametricity

Consider the type

```
a -> a -> a
```

Remember that `a` is a *type variable* which can stand for any type. What sorts of functions have this type?

What about this:

```
f :: a -> a -> a
f x y = x && y
```

It turns out that this doesn't work. The syntax is valid, at least, but it does not type check. In particular we get this error message:

```
2012-02-09.lhs:37:16:
    Couldn't match type `a' with `Bool'
      `a' is a rigid type variable bound by
          the type signature for f :: a -> a -> a at 2012-02-09.lhs:37:3
    In the second argument of `(&&)', namely `y'
    In the expression: x && y
    In an equation for `f': f x y = x && y
```

The reason this doesn't work is that the *caller* of a polymorphic function gets to choose the type. Here we, the *implementors*, have tried to choose a specific type (namely, `Bool`), but we may be given `String`, or `Int`, or even some type defined by someone using `f`, which we can't possibly know about in advance. In other words, you can read the type

```
a -> a -> a
```

as a *promise* that a function with this type will work no matter what type the caller chooses.

Another implementation we could imagine is something like

```
f a1 a2 = case (typeOf a1) of
            Int  -> a1 + a2
            Bool -> a1 && a2
            _    -> a1
```

where `f` behaves in some specific ways for certain types. After all, we can certainly implement this in Java:

```
class AdHoc {

    public static Object f(Object a1, Object a2) {
        if (a1 instanceof Integer && a2 instanceof Integer) {
            return (Integer)a1 + (Integer)a2;
        } else if (a1 instanceof Boolean && a2 instanceof Boolean) {
            return (Boolean)a1 && (Boolean)a2;
        } else {
            return a1;
        }
```

```
    }

    public static void main (String[] args) {
        System.out.println(f(1,3));
        System.out.println(f(true, false));
        System.out.println(f("hello", "there"));
    }

}

[byorgey@LVN513-9:~/tmp]$ javac Adhoc.java && java AdHoc
4
false
hello
```

But it turns out there is no way to write this in Haskell. Haskell does not have anything like Java's `instanceof` operator: it is not possible to ask what type something is and decide what to do based on the answer. One reason for this is that Haskell types are *erased* by the compiler after being checked: at runtime, there is no type information around to query! However, as we will see, there are other good reasons too.

This style of polymorphism is known as *parametric polymorphism*. We say that a function like `f :: a -> a -> a` is *parametric* in the type `a`. Here "parametric" is just a fancy term for "works uniformly for any type chosen by the caller". In Java, this style of polymorphism is provided by *generics* (which, you guessed it, were inspired by Haskell: one of the original designers of Haskell, Philip Wadler, was later one of the key players in the development of Java generics).

So, what functions actually *could* have this type? Actually, there are only two!

```
f1 :: a -> a -> a
f1 x y = x

f2 :: a -> a -> a
f2 x y = y
```

So it turns out that the type `a -> a -> a` really tells us quite a lot.

Let's play the parametricity game! Consider each of the following polymorphic types. For each type, determine what behavior(s) a function of that type could possibly have.

- `a -> a`
- `a -> b`
- `a -> b -> a`
- `[a] -> [a]`
- `(b -> c) -> (a -> b) -> (a -> c)`
- `(a -> a) -> a -> a`

Two views on parametricity

As an *implementor* of polymorphic functions, especially if you are used to a language with a construct like Java's `instanceof`, you might find these restrictions annoying. "What do you mean, I'm not allowed to do X?"

However, there is a dual point of view. As a *user* of polymorphic functions, parametricity corresponds not to *restrictions* but to *guarantees*. In general, it is much easier to use and reason about tools when those tools

give you strong guarantees as to how they will behave. Parametricity is part of the reason that just looking at the type of Haskell function can tell you so much about the function.

OK, fine, but sometimes it really is useful to be able to decide what to do based on types! For example, what about addition? We've already seen that addition is polymorphic (it works on `Int`, `Integer`, and `Double`, for example) but clearly it has to know what type of numbers it is adding to decide what to do: adding two `Integer`s works in a completely different way than adding two `Double`s. So how does it actually work? Is it just magical?

In fact, it isn't! And we *can* actually use Haskell to decide what to do based on types—just not in the way we were imagining before. Let's start by taking a look at the type of `(+)`:

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
```

Hmm, what's that `Num a =>` thingy doing there? In fact, `(+)` isn't the only standard function with a funny double-arrow thing in its type. Here are a few others:

```
(==) :: Eq a   => a -> a -> Bool
(<)  :: Ord a  => a -> a -> Bool
show :: Show a => a -> String
```

So what's going on here?

Type classes

`Num`, `Eq`, `Ord`, and `Show` are *type classes*, and we say that `(==)`, `(<)`, and `(+)` are "type-class polymorphic". Intuitively, type classes correspond to *sets of types* which have certain operations defined for them, and type class polymorphic functions work only for types which are instances of the type class(es) in question. As an example, let's look in detail at the `Eq` type class.

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

We can read this as follows: `Eq` is declared to be a type class with a single parameter, `a`. Any type `a` which wants to be an *instance* of `Eq` must define two functions, `(==)` and `(/=)`, with the indicated type signatures. For example, to make `Int` an instance of `Eq` we would have to define `(==) :: Int -> Int -> Bool` and `(/=) :: Int -> Int -> Bool`. (Of course, there's no need, since the standard Prelude already defines an `Int` instance of `Eq` for us.)

Let's look at the type of `(==)` again:

```
(==) :: Eq a => a -> a -> Bool
```

The `Eq a` that comes before the `=>` is a *type class constraint*. We can read this as saying that for any type `a`, *as long as* `a` *is an instance of* `Eq`, `(==)` can take two values of type `a` and return a `Bool`. It is a type error to call the function `(==)` on some type which is not an instance of `Eq`. If a normal polymorphic type is a promise that the function will work for whatever type the caller chooses, a type class polymorphic function is a *restricted* promise that the function will work for any type the caller chooses, *as long as* the chosen type is an instance of the required type class(es).

The important thing to note is that when `(==)` (or any type class method) is used, the compiler uses type inference to figure out *which implementation of `(==)` should be chosen*, based on the inferred types of its arguments. In other words, it is something like using an overloaded method in a language like Java.

To get a better handle on how this works in practice, let's make our own type and declare an instance of `Eq` for it.

```
data Foo = F Int | G Char

instance Eq Foo where
  (F i1) == (F i2) = i1 == i2
  (G c1) == (G c2) = c1 == c2
  _ == _ = False

  foo1 /= foo2 = not (foo1 == foo2)
```

It's a bit annoying that we have to define both `(==)` and `(/=)`. In fact, type classes can give *default implementations* of methods in terms of other methods, which should be used whenever an instance does not override the default definition with its own. So we could imagine declaring `Eq` like this:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

Now anyone declaring an instance of `Eq` only has to specify an implementation of `(==)`, and they will get `(/=)` for free. But if for some reason they want to override the default implementation of `(/=)` with their own, they can do that as well.

In fact, the `Eq` class is actually declared like this:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

This means that when we make an instance of `Eq`, we can define *either* `(==)` or `(/=)`, whichever is more convenient; the other one will be automatically defined in terms of the one we specify. (However, we have to be careful: if we don't specify either one, we get infinite recursion!)

As it turns out, `Eq` (along with a few other standard type classes) is special: GHC is able to automatically generate instances of `Eq` for us. Like so:

```
data Foo' = F' Int | G' Char
  deriving (Eq, Ord, Show)
```

This tells GHC to automatically derive instances of the `Eq`, `Ord`, and `Show` type classes for our data type `Foo`.

Type classes and Java interfaces

Type classes are quite similar to Java interfaces. Both define a set of types/classes which implement a specified list of operations. However, there are a couple of important ways in which type classes are more general than Java interfaces:

1. When a Java class is defined, any interfaces it implements must be declared. Type class instances, on the other hand, are declared separately from the declaration of the corresponding types, and can even be put in a separate module.

2. The types that can be specified for type class methods are more general and flexible than the signatures that can be given for Java interface methods, especially when *multi-parameter type classes* enter the picture. For example, consider a hypothetical type class

   ```
   class Blerg a b where
     blerg :: a -> b -> Bool
   ```

   Using `blerg` amounts to doing *multiple dispatch*: which implementation of `blerg` the compiler should choose depends on *both* the types `a` and `b`. There is no easy way to do this in Java.

   Haskell type classes can also easily handle binary (or ternary, or ...) methods, as in

   ```
   class Num a where
     (+) :: a -> a -> a
     ...
   ```

   There is no nice way to do this in Java: for one thing, one of the two arguments would have to be the "privileged" one which is actually getting the `(+)` method invoked on it, and this asymmetry is awkward. Furthermore, because of Java's subtyping, getting two arguments of a certain interface type does *not* guarantee that they are actually the same type, which makes implementing binary operators such as `(+)` awkward (usually requiring some runtime type checks).

Standard type classes

Here are some other standard type classes you should know about:

- Ord is for types whose elements can be *totally ordered*, that is, where any two elements can be compared to see which is less than the other. It provides comparison operations like `(<)` and `(<=)`, and also the `compare` function.
- Num is for "numeric" types, which support things like addition, subtraction, and multipication. One very important thing to note is that integer literals are actually type class polymorphic:

  ```
  Prelude> :t 5
  5 :: Num a => a
  ```

  This means that literals like `5` can be used as `Int`s, `Integer`s, `Double`s, or any other type which is an instance of `Num` (`Rational`, `Complex Double`, or even a type you define...)
- Show defines the method `show`, which is used to convert values into `String`s.
- Read is the dual of `Show`.
- Integral represents whole number types such as `Int` and `Integer`.

A type class example

As an example of making our own type class, consider the following:

```
class Listable a where
  toList :: a -> [Int]
```

We can think of `Listable` as the class of things which can be converted to a list of `Int`s. Look at the type of `toList`:

```
toList :: Listable a => a -> [Int]
```

Let's make some instances for `Listable`. First, an `Int` can be converted to an `[Int]` just by creating a singleton list, and `Bool` can be converted similarly, say, by translating `True` to `1` and `False` to `0`:

```
instance Listable Int where
  -- toList :: Int -> [Int]
  toList x = [x]

instance Listable Bool where
  toList True  = [1]
  toList False = [0]
```

We don't need to do any work to convert a list of `Int` to a list of `Int`:

```
instance Listable [Int] where
  toList = id
```

Finally, here's a binary tree type which we can convert to a list by flattening:

```
data Tree a = Empty | Node a (Tree a) (Tree a)

instance Listable (Tree Int) where
  toList Empty        = []
  toList (Node x l r) = toList l ++ [x] ++ toList r
```

If we implement other functions in terms of `toList`, they also get a `Listable` constraint. For example:

```
-- to compute sumL, first convert to a list of Ints, then sum
sumL x = sum (toList x)
```

`ghci` informs us that type type of `sumL` is

```
sumL :: Listable a => a -> Int
```

which makes sense: `sumL` will work only for types which are instances of `Listable`, since it uses `toList`. What about this one?

```
foo x y = sum (toList x) == sum (toList y) || x < y
```

`ghci` informs us that the type of `foo` is

```
foo :: (Listable a, Ord a) => a -> a -> Bool
```

That is, `foo` works over types which are instances of *both* `Listable` and `Ord`, since it uses both `toList` and comparison on the arguments.

As a final, and more complex, example, consider this instance:

```
instance (Listable a, Listable b) => Listable (a,b) where
  toList (x,y) = toList x ++ toList y
```

Notice how we can put type class constraints on an instance as well as on a function type. This says that a pair type `(a,b)` is an instance of `Listable` as long as `a` and `b` both are. Then we get to use `toList` on values of types `a` and `b` in our definition of `toList` for a pair. Note that this definition is *not* recursive! The version of `toList` that we are defining is calling *other* versions of `toList`, not itself.

Lazy evaluation

CIS 194 Week 6
18 February 2012

Suggested reading:

- foldr foldl foldl' from the Haskell wiki

On the first day of class I mentioned that Haskell is *lazy*, and promised to eventually explain in more detail what this means. The time has come!

Strict evaluation

Before we talk about *lazy evaluation* it will be useful to look at some examples of its opposite, *strict evaluation*.

Under a strict evaluation strategy, function arguments are completely evaluated *before* passing them to the function. For example, suppose we have defined

```
f x y = x + 2
```

In a strict language, evaluating `f 5 (29^35792)` will first completely evaluate `5` (already done) and `29^35792` (which is a lot of work) before passing the results to `f`.

Of course, in this *particular* example, this is silly, since `f` ignores its second argument, so all the work to compute `29^35792` was wasted. So why would we want this?

The benefit of strict evaluation is that it is easy to predict *when* and *in what order* things will happen. Usually languages with strict evaluation will even specify the order in which function arguments should be evaluated (*e.g.* from left to right).

For example, in Java if we write

```
f (release_monkeys(), increment_counter())
```

we know that the monkeys will be released, and then the counter will be incremented, and then the results of doing those things will be passed to `f`, and it does not matter whether `f` actually ends up using those results.

If the releasing of monkeys and incrementing of the counter could independently happen, or not, in either order, depending on whether `f` happens to use their results, it would be extremely confusing. When such "side effects" are allowed, strict evaluation is really what you want.

Side effects and purity

So, what's really at issue here is the presence or absence of *side effects*. By "side effect" we mean *anything that causes evaluation of an expression to interact with something outside itself*. The root issue is that such outside interactions are time-sensitive. For example:

- Modifying a global variable — it matters when this happens since it may affect the evaluation of other expressions
- Printing to the screen — it matters when this happens since it may need to be in a certain order with respect to other writes to the screen

- Reading from a file or the network — it matters when this happens since the contents of the file can affect the outcome of the expression

As we have seen, lazy evaluation makes it hard to reason about when things will be evaluated; hence including side effects in a lazy language would be extremely unintuitive. Historically, this is the reason Haskell is pure: initially, the designers of Haskell wanted to make a *lazy* functional language, and quickly realized it would be impossible unless it also disallowed side effects.

But… a language with *no* side effects would not be very useful. The only thing you could do with such a language would be to load up your programs in an interpreter and evaluate expressions. (Hmm… that sounds familiar…) You would not be able to get any input from the user, or print anything to the screen, or read from a file. The challenge facing the Haskell designers was to come up with a way to allow such effects in a principled, restricted way that did not interfere with the essential purity of the language. They finally did come up with something (namely, the `IO` monad) which we'll talk about in a few weeks.

Lazy evaluation

So now that we understand strict evaluation, let's see what lazy evaluation actually looks like. Under a lazy evaluation strategy, evaluation of function arguments is *delayed as long as possible*: they are not evaluated until it actually becomes necessary to do so. When some expression is given as an argument to a function, it is simply packaged up as an *unevaluated expression* (called a "thunk", don't ask me why) without doing any actual work.

For example, when evaluating `f 5 (29^35792)`, the second argument will simply be packaged up into a thunk without doing any actual computation, and `f` will be called immediately. Since `f` never uses its second argument the thunk will just be thrown away by the garbage collector.

Pattern matching drives evaluation

So, when is it "necessary" to evaluate an expression? The examples above concentrated on whether a function *used* its arguments, but this is actually not the most important distinction. Consider the following examples:

```
f1 :: Maybe a -> [Maybe a]
f1 m = [m,m]

f2 :: Maybe a -> [a]
f2 Nothing  = []
f2 (Just x) = [x]
```

`f1` and `f2` both *use* their argument. But there is still a big difference between them. Although `f1` uses its argument `m`, it does not need to know anything about it. `m` can remain completely unevaluated, and the unevaluated expression is simply put in a list. Put another way, the result of `f1 e` does not depend on the shape of `e`.

`f2`, on the other hand, needs to know something about its argument in order to proceed: was it constructed with `Nothing` or `Just`? That is, in order to evaluate `f2 e`, we must first evaluate `e`, because the result of `f2` depends on the shape of `e`.

The other important thing to note is that thunks are evaluated *only enough* to allow a pattern match to proceed, and no further! For example, suppose we wanted to evaluate `f2 (safeHead [3^500, 49])`. `f2` would force evaluation of the call to `safeHead [3^500, 49]`, which would evaluate to `Just (3^500)`—note that the `3^500` is *not* evaluated, since `safeHead` does not need to look at it, and neither does `f2`. Whether the `3^500` gets evaluated later depends on how the result of `f2` is used.

The slogan to remember is "*pattern matching drives evaluation*". To reiterate the important points:

- Expressions are only evaluated when pattern-matched
- …only as far as necessary for the match to proceed, and no farther!

Let's do a slightly more interesting example: we'll evaluate `take 3 (repeat 7)`. For reference, here are the definitions of `repeat` and `take`:

```
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _       | n <= 0 =  []
take _ []               =  []
take n (x:xs)           =  x : take (n-1) xs
```

Carrying out the evaluation step-by-step looks something like this:

```
  take 3 (repeat 7)
      { 3 <= 0 is False, so we proceed to the second clause, which
    needs to match on the second argument. So we must expand
    repeat 7 one step. }
= take 3 (7 : repeat 7)
      { the second clause does not match but the third clause
        does. Note that (3-1) does not get evaluated yet! }
= 7 : take (3-1) (repeat 7)
      { In order to decide on the first clause, we must test (3-1)
        <= 0 which requires evaluating (3-1). }
= 7 : take 2 (repeat 7)
      { 2 <= 0 is False, so we must expand repeat 7 again. }
= 7 : take 2 (7 : repeat 7)
      { The rest is similar. }
= 7 : 7 : take (2-1) (repeat 7)
= 7 : 7 : take 1 (repeat 7)
= 7 : 7 : take 1 (7 : repeat 7)
= 7 : 7 : 7 : take (1-1) (repeat 7)
= 7 : 7 : 7 : take 0 (repeat 7)
= 7 : 7 : 7 : []
```

(Note that although evaluation *could* be implemented exactly like the above, most Haskell compilers will do something a bit more sophisticated. In particular, GHC uses a technique called *graph reduction*, where the expression being evaluated is actually represented as a *graph*, so that different parts of the expression can share pointers to the same subexpression. This ensures that work is not duplicated unnecessarily. For example, if `f x = [x,x]`, evaluating `f (1+1)` will only do *one* addition, because the subexpression `1+1` will be shared between the two occurrences of `x`.)

Consequences

Laziness has some very interesting, pervasive, and nonobvious consequences. Let's explore a few of them.

Purity

As we've already seen, choosing a lazy evaluation strategy essentially *forces* you to also choose purity (assuming you don't want programmers to go insane).

Understanding space usage

Laziness is not all roses. One of the downsides is that it sometimes becomes tricky to reason about the space usage of your programs. Consider the following (innocuous-seeming) example:

```
-- Standard library function foldl, provided for reference
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Let's consider how evaluation proceeds when we evaluate `foldl (+) 0 [1,2,3]` (which sums the numbers in a list):

```
  foldl (+) 0 [1,2,3]
= foldl (+) (0+1) [2,3]
= foldl (+) ((0+1)+2) [3]
= foldl (+) (((0+1)+2)+3) []
= (((0+1)+2)+3)
= ((1+2)+3)
= (3+3)
= 6
```

Since the value of the accumulator is not demanded until recursing through the entire list, the accumulator simply builds up a big unevaluated expression `(((0+1)+2)+3)`, which finally gets reduced to a value at the end. There are at least two problems with this. One is that it's simply inefficient: there's no point in transferring all the numbers from the list into a different list-like thing (the accumulator thunk) before actually adding them up. The second problem is more subtle, and more insidious: evaluating the expression `(((0+1)+2)+3)` actually requires pushing the `3` and `2` onto a stack before being able to compute `0+1` and then unwinding the stack, adding along the way. This is not a problem for this small example, but for very long lists it's a big problem: there is usually not as much space available for the stack, so this can lead to a stack overflow.

The solution in this case is to use the `foldl'` function instead of `foldl`, which adds a bit of strictness: in particular, `foldl'` requires its second argument (the accumulator) to be evaluated before it proceeds, so a large thunk never builds up:

```
  foldl' (+) 0 [1,2,3]
= foldl' (+) (0+1) [2,3]
= foldl' (+) 1 [2,3]
= foldl' (+) (1+2) [3]
= foldl' (+) 3 [3]
= foldl' (+) (3+3) []
= foldl' (+) 6 []
= 6
```

As you can see, `foldl'` does the additions along the way, which is what we really want. But the point is that in this case laziness got in the way and we had to make our program *less* lazy.

(If you're interested in learning about *how* `foldl'` achieves this, you can read about `seq` on the Haskell wiki.)

Short-circuiting operators

In some languages (Java, C++) the boolean operators `&&` and `||` (logical AND and OR) are *short-circuiting*: for example, if the first argument to `&&` evaluates to false, the whole expression will immediately evaluate to

false without touching the second argument. However, this behavior has to be wired into the Java and C++ language standards as a special case. Normally, in a strict langauge, both arguments of a two-argument function are be evaluated before calling the function. So the short-circuiting behavior of `&&` and `||` is a special exception to the usual strict semantics of the language.

In Haskell, however, we can define short-circuiting operators without any special cases. In fact, `(&&)` and `(||)` are just plan old library functions! For example, here's how `(&&)` is defined:

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && _ = False
```

Notice how this definition of `(&&)` does not pattern-match on its second argument. Moreover, if the first argument is `False`, the second argument is entirely ignored. Since `(&&)` does not pattern-match on its second argument at all, it is short-circuiting in exactly the same way as the `&&` operator in Java or C++.

Notice that `(&&)` also could have been defined like this:

```
(&&!) :: Bool -> Bool -> Bool
True  &&! True  = True
True  &&! False = False
False &&! True  = False
False &&! False = False
```

While this version takes on the same values as `(&&)`, it has different behavior. For example, consider the following:

```
False &&  (34^9784346 > 34987345)
False &&! (34^9784346 > 34987345)
```

These will both evaluate to `False`, but the second one will take a whole lot longer! Or how about this:

```
False &&  (head [] == 'x')
False &&! (head [] == 'x')
```

The first one is again `False`, whereas the second one will crash. Try it!

All of this points out that there are some interesting issues surrounding laziness to be considered when defining a function.

User-defined control structures

Taking the idea of short-circuiting operators one step further, in Haskell we can define our own *control structures*.

Most languages have some sort of special built-in `if` construct. Some thought reveals why: in a way similar to short-circuiting Boolean operators, `if` has special behavior. Based on the value of the test, it executes/evaluates only *one* of the two branches. It would defeat the whole purpose if both branches were evaluated every time!

In Haskell, however, we can define `if` as a library function!

```
if' :: Bool -> a -> a -> a
if' True  x _ = x
if' False _ y = y
```

Of course, Haskell *does* have special built-in `if`-expressions, but I have never quite understood why. Perhaps it is simply because the language designers thought people would expect it. "What do you mean, this language doesn't have `if`!?" In any case, `if` doesn't get used that much in Haskell anyway; in most situations we prefer pattern-matching or guards.

We can also define other control structures—we'll see other examples when we discuss monads.

Infinite data structures

Lazy evaluation also means that we can work with *infinite data structures*. In fact, we've already seen a few examples, such as `repeat 7`, which represents an infinite list containing nothing but `7`. Defining an infinite data structure actually only creates a thunk, which we can think of as a "seed" out of which the entire data structure can *potentially* grow, depending on what parts actually are used/needed.

Another practical application area is "effectively infinite" data structures, such as the trees that might arise as the state space of a game (such as go or chess). Although the tree is finite in theory, it is so large as to be effectively infinite—it certainly would not fit in memory. Using Haskell, we can define the tree of all possible moves, and then write a separate algorithm to explore the tree in whatever way we want. Only the parts of the tree which are actually explored will be computed.

Pipelining/wholemeal programming

As I have mentioned before, doing "pipelined" incremental transformations of a large data structure can actually be memory-efficient. Now we can see why: due to laziness, each stage of the pipeline can operate in lockstep, only generating each bit of the result as it is demanded by the next stage in the pipeline.

Dynamic programming

As a more specific example of the cool things lazy evaluation buys us, consider the technique of *dynamic programming*. Usually, one must take great care to fill in entries of a dynamic programming table in the proper order, so that every time we compute the value of a cell, its dependencies have already been computed. If we get the order wrong, we get bogus results.

However, using lazy evaluation we can get the Haskell runtime to work out the proper order of evaluation for us! For example, here is some Haskell code to solve the 0-1 knapsack problem. Note how we simply define the array `m` in terms of itself, using the standard recurrence, and let lazy evaluation work out the proper order in which to compute its cells.

```
import Data.Array

knapsack01 :: [Double]    -- values
           -> [Integer]   -- nonnegative weights
           -> Integer     -- knapsack size
           -> Double      -- max possible value
knapsack01 vs ws maxW = m!(numItems-1, maxW)
  where numItems = length vs
        m = array ((-1,0), (numItems-1, maxW)) $
              [((-1,w), 0) | w <- [0 .. maxW]] ++
              [((i,0), 0) | i <- [0 .. numItems-1]] ++
              [((i,w), best)
                  | i <- [0 .. numItems-1]
                  , w <- [1 .. maxW]
                  , let best
                          | ws!!i > w  = m!(i-1, w)
                          | otherwise = max (m!(i-1, w))
                                            (m!(i-1, w - ws!!i) + vs!!i)
              ]

example = knapsack01 [3,4,5,8,10] [2,3,4,5,9] 20
```

Folds and monoids

CIS 194 Week 7
25 February 2013

Suggested reading:

- Learn You a Haskell, Only folds and horses
- Learn You a Haskell, Monoids
- Fold from the Haskell wiki
- Heinrich Apfelmus, Monoids and Finger Trees
- Dan Piponi, Haskell Monoids and their Uses
- Data.Monoid documentation
- Data.Foldable documentation

Folds, again

We've already seen how to define a folding function for lists… but we can generalize the idea to other data types as well!

Consider the following data type of binary trees with data stored at internal nodes:

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)
  deriving (Show, Eq)

leaf :: a -> Tree a
leaf x = Node Empty x Empty
```

Let's write a function to compute the size of a tree (*i.e.* the number of `Node`s):

```
treeSize :: Tree a -> Integer
treeSize Empty        = 0
treeSize (Node l _ r) = 1 + treeSize l + treeSize r
```

How about the sum of the data in a tree of `Integer`s?

```
treeSum :: Tree Integer -> Integer
treeSum Empty     = 0
treeSum (Node l x r)  = x + treeSum l + treeSum r
```

Or the depth of a tree?

```
treeDepth :: Tree a -> Integer
treeDepth Empty        = 0
treeDepth (Node l _ r) = 1 + max (treeDepth l) (treeDepth r)
```

Or flattening the elements of the tree into a list?

```
flatten :: Tree a -> [a]
flatten Empty        = []
flatten (Node l x r) = flatten l ++ [x] ++ flatten r
```

Are you starting to see any patterns? Each of the above functions:

1. takes a `Tree` as input
2. pattern-matches on the input `Tree`
3. in the `Empty` case, gives a simple answer
4. in the `Node` case:
    1. calls itself recursively on both subtrees
    2. somehow combines the results from the recursive calls with the data `x` to produce the final result

As good programmers, we always strive to abstract out repeating patterns, right? So let's generalize. We'll need to pass as parameters the parts of the above examples which change from example to example:

1. The return type
2. The answer in the `Empty` case
3. How to combine the recursive calls

We'll call the type of data contained in the tree `a`, and the type of the result `b`.

```
treeFold :: b -> (b -> a -> b -> b) -> Tree a -> b
treeFold e _ Empty        = e
treeFold e f (Node l x r) = f (treeFold e f l) x (treeFold e f r)
```

Now we should be able to define `treeSize`, `treeSum` and the other examples much more simply. Let's try:

```
treeSize' :: Tree a -> Integer
treeSize' = treeFold 0 (\l _ r -> 1 + l + r)

treeSum' :: Tree Integer -> Integer
treeSum' = treeFold 0 (\l x r -> l + x + r)

treeDepth' :: Tree a -> Integer
treeDepth' = treeFold 0 (\l _ r -> 1 + max l r)

flatten' :: Tree a -> [a]
flatten' = treeFold [] (\l x r -> l ++ [x] ++ r)
```

We can write new tree-folding functions easily as well:

```
treeMax :: (Ord a, Bounded a) => Tree a -> a
treeMax = treeFold minBound (\l x r -> l `max` x `max` r)
```

Much better!

Folding expressions

Where else have we seen folds?

Recall the `ExprT` type and corresponding `eval` function from Homework 5:

```
data ExprT = Lit Integer
           | Add ExprT ExprT
           | Mul ExprT ExprT

eval :: ExprT -> Integer
eval (Lit i)     = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

Hmm… this looks familiar! What would a fold for `ExprT` look like?

```
exprTFold :: (Integer -> b) -> (b -> b -> b) -> (b -> b -> b) -> ExprT -> b
exprTFold f _ _ (Lit i)     = f i
exprTFold f g h (Add e1 e2) = g (exprTFold f g h e1) (exprTFold f g h e2)
exprTFold f g h (Mul e1 e2) = h (exprTFold f g h e1) (exprTFold f g h e2)

eval2 :: ExprT -> Integer
eval2 = exprTFold id (+) (*)
```

Now we can easily do other things like count the number of literals in an expression:

```
numLiterals :: ExprT -> Int
numLiterals = exprTFold (const 1) (+) (+)
```

Folds in general

The take-away message is that we can implement a fold for many (though not all) data types. The fold for `T` will take one (higher-order) argument for each of `T`'s constructors, encoding how to turn the values stored by that constructor into a value of the result type—assuming that any recursive occurrences of `T` have already been folded into a result. Many functions we might want to write on `T` will end up being expressible as simple folds.

Monoids

Here's another standard type class you should know about, found in the `Data.Monoid` module:

```
class Monoid m where
    mempty  :: m
    mappend :: m -> m -> m

    mconcat :: [m] -> m
    mconcat = foldr mappend mempty

(<>) :: Monoid m => m -> m -> m
(<>) = mappend
```

`(<>)` is defined as a synonym for `mappend` (as of GHC 7.4.1) simply because writing `mappend` is tedious.

Types which are instances of `Monoid` have a special element called `mempty`, and a binary operation `mappend` (abbreviated `(<>)`) which takes two values of the type and produces another one. The intention is that `mempty` is an identity for `<>`, and `<>` is associative; that is, for all `x`, `y`, and `z`,

1. `mempty <> x == x`

2. `x <> mempty == x`
3. `(x <> y) <> z == x <> (y <> z)`

The associativity law means that we can unambiguously write things like

`a <> b <> c <> d <> e`

because we will get the same result no matter how we parenthesize.

There is also `mconcat`, for combining a whole list of values. By default it is implemented using `foldr`, but it is included in the `Monoid` class since particular instances of `Monoid` may have more efficient ways of implementing it.

`Monoid`s show up *everywhere*, once you know to look for them. Let's write some instances (just for practice; these are all in the standard libraries).

Lists form a monoid under concatenation:

```
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

As hinted above, addition defines a perfectly good monoid on integers (or rational numbers, or real numbers...). However, so does multiplication! What to do? We can't give two different instances of the same type class to the same type. Instead, we create two `newtype`s, one for each instance:

```
newtype Sum a = Sum a
  deriving (Eq, Ord, Num, Show)

getSum :: Sum a -> a
getSum (Sum a) = a

instance Num a => Monoid (Sum a) where
  mempty  = Sum 0
  mappend = (+)

newtype Product a = Product a
  deriving (Eq, Ord, Num, Show)

getProduct :: Product a -> a
getProduct (Product a) = a

instance Num a => Monoid (Product a) where
  mempty  = Product 1
  mappend = (*)
```

Note that to find, say, the product of a list of `Integer`s using `mconcat`, we have to first turn them into values of type `Product Integer`:

```
lst :: [Integer]
lst = [1,5,8,23,423,99]

prod :: Integer
prod = getProduct . mconcat . map Product $ lst
```

(Of course, this particular example is silly, since we could use the standard `product` function instead, but this pattern does come in handy somtimes.)

Pairs form a monoid as long as the individual components do:

```
instance (Monoid a, Monoid b) => Monoid (a,b) where
  mempty = (mempty, mempty)
  (a,b) `mappend` (c,d) = (a `mappend` c, b `mappend` d)
```

Challenge: can you make an instance of `Monoid` for `Bool`? How many different instances are there?

Challenge: how would you make function types an instance of `Monoid`?

IO

CIS 194 Week 8
11 March 2013

Suggested reading:

- LYAH Chapter 9: Input and Output
- RWH Chapter 7: I/O

The problem with purity

Remember that Haskell is *lazy* and therefore *pure*. This means two primary things:

1. Functions may not have any external effects. For example, a function may not print anything on the screen. Functions may only compute their outputs.
2. Functions may not depend on external stuff. For example, they may not read from the keyboard, or filesystem, or network. Functions may depend only on their inputs—put another way, functions should give the same output for the same input every time.

But—sometimes we *do* want to be able to do stuff like this! If the only thing we could do with Haskell is write functions which we can then evaluate at the ghci prompt, it would be theoretically interesting but practically useless.

In fact, it *is* possible to do these sorts of things with Haskell, but it looks very different than in most other languages.

The `IO` type

The solution to the conundrum is a special type called `IO`. Values of type `IO a` are *descriptions of* effectful computations, which, if executed would (possibly) perform some effectful I/O operations and (eventually) produce a value of type `a`. There is a level of indirection here that's crucial to understand. A value of type `IO a`, *in and of itself*, is just an inert, perfectly safe thing with no effects. It is just a *description* of an effectful computation. One way to think of it is as a *first-class imperative program*.

As an illustration, suppose you have

```
c :: Cake
```

What do you have? Why, a delicious cake, of course. Plain and simple.

By contrast, suppose you have

```
r :: Recipe Cake
```

What do you have? A cake? No, you have some *instructions* for how to make a cake, just a sheet of paper with some writing on it.

Not only do you not actually have a cake, merely being in possession of the recipe has no effect on anything else whatsoever. Simply holding the recipe in your hand does not cause your oven to get hot or flour to be spilled all over your floor or anything of that sort. To actually produce a cake, the recipe must be *followed* (causing flour to be spilled, ingredients mixed, the oven to get hot, *etc.*).

In the same way, a value of type `IO a` is just a "recipe" for producing a value of type `a` (and possibly having some effects along the way). Like any other value, it can be passed as an argument, returned as the output of a function, stored in a data structure, or (as we will see shortly) combined with other `IO` values into more complex recipes.

So, how do values of type `IO a` actually ever get executed? There is only one way: the Haskell compiler looks for a special value

```
main :: IO ()
```

which will actually get handed to the runtime system and executed. That's it! Think of the Haskell runtime system as a master chef who is the only one allowed to do any cooking.

If you want your recipe to be followed then you had better make it part of the big recipe (`main`) that gets handed to the master chef. Of course, `main` can be arbitrarily complicated, and will usually be composed of many smaller `IO` computations.

So let's write our first actual, executable Haskell program! We can use the function

```
putStrLn :: String -> IO ()
```

which, given a `String`, returns an `IO` computation that will (when executed) print out that `String` on the screen. So we simply put this in a file called `Hello.hs`:

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

Then typing `runhaskell Hello.hs` at a command-line prompt results in our message getting printed to the screen! We can also use `ghc --make Hello.hs` to produce an executable version called `Hello` (or `Hello.exe` on Windows).

There is no `String` "inside" an `IO String`

Many new Haskell users end up at some point asking a question like "I have an `IO String`, how do I turn it into a `String`?", or, "How do I get the `String` out of an `IO String`"? Given the above intuition, it should be clear that these are nonsensical questions: a value of type `IO String` is a description of some computation, a *recipe*, for generating a `String`. There is no `String` "inside" an `IO String`, any more than there is a cake "inside" a cake recipe. To produce a `String` (or a delicious cake) requires actually *executing* the computation (or recipe). And the only way to do that is to give it (perhaps as part of some larger `IO` value) to the Haskell runtime system, via `main`.

Combining `IO`

As should be clear by now, we need a way to *combine* `IO` computations into larger ones.

The simplest way to combine two `IO` computations is with the `(>>)` operator (pronounced "and then"), which has the type

```
(>>) :: IO a -> IO b -> IO b
```

This simply creates an `IO` computation which consists of running the two input computations in sequence. Notice that the result of the first computation is discarded; we only care about it for its *effects*. For example:

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

This works fine for code of the form "do this; do this; do this" where the results don't really matter. However, in general this is insufficient. What if we don't want to throw away the result from the first computation?

A first attempt at resolving the situation might be to have something of type `IO a -> IO b -> IO (a,b)`. However, this is also insufficient. The reason is that we want the second computation to be able to *depend* on the result of the first. For example, suppose we want to read an integer from the user and then print out one more than the integer they entered. In this case the second computation (printing some number on the screen) will be different depending on the result of the first.

Instead, there is an operator `(>>=)` (pronounced "bind") with the type

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

This can be difficult to wrap one's head around at first! `(>>=)` takes a computation which will produce a value of type `a`, and a *function* which gets to *compute* a second computation based on this intermediate value of type `a`. The result of `(>>=)` is a (description of a) computation which performs the first computation, uses its result to decide what to do next, and then does that.

For example, we can write a program to read a number from the user and print out its successor. Note our use of `readLn :: Read a => IO a` which is a computation that reads input from the user and converts it into any type which is an instance of `Read`.

```
main :: IO ()
main = putStrLn "Please enter a number: " >> (readLn >>= (\n -> putStrLn (show (n+1))))
```

Of course, this looks kind of ugly, but there are better ways to write it, which we'll talk about in the future.

Record syntax

This material was not covered in lecture, but is provided as an extra resource for completing homework 8.

Suppose we have a data type such as

```
data D = C T1 T2 T3
```

We could also declare this data type with *record syntax* as follows:

```
data D = C { field1 :: T1, field2 :: T2, field3 :: T3 }
```

where we specify not just a type but also a *name* for each field stored inside the `C` constructor. This new version of `D` can be used in all the same ways as the old version (in particular we can still construct and pattern-match on values of type `D` as `C v1 v2 v3`). However, we get some additional benefits.

1. Each field name is automatically a *projection function* which gets the value of that field out of a value of type `D`. For example, `field2` is a function of type

   ```
   field2 :: D -> T2
   ```

   Before, we would have had to implement `field2` ourselves by writing

   ```
   field2 (C _ f _) = f
   ```

   This gets rid of a lot of boilerplate if we have a data type with many fields!

2. There is special syntax for *constructing*, *modifying*, and *pattern-matching* on values of type `D` (in addition to the usual syntax for such things).

   We can *construct* a value of type `D` using syntax like

   ```
   C { field3 = ..., field1 = ..., field2 = ... }
   ```

   with the `...` filled in by expressions of the right type. Note that we can specify the fields in any order.

   Suppose we have a value `d :: D`. We can *modify* `d` using syntax like

   ```
   d { field3 = ... }
   ```

   Of course, by "modify" we don't mean actually mutating `d`, but rather constructing a new value of type `D` which is the same as `d` except with the `field3` field replaced by the given value.

   Finally, we can *pattern-match* on values of type `D` like so:

   ```
   foo (C { field1 = x }) = ... x ...
   ```

   This matches only on the `field1` field from the `D` value, calling it `x` (of course, in place of `x` we could also put an arbitrary pattern), ignoring the other fields.

Functors

CIS 194 Week 9
18 March 2013

Suggested reading:

- Learn You a Haskell, The Functor typeclass
- The Typeclassopedia

Motivation

Over the past weeks we have seen a number of functions designed to "map" a function over every element of some sort of container. For example:

- `map :: (a -> b) -> [a] -> [b]`
- `treeMap :: (a -> b) -> Tree a -> Tree b`

- In Homework 5 many people ended up doing a similar thing when you had to somehow apply `eval :: ExprT -> Int` to a `Maybe ExprT` in order to get a `Maybe Int`.

  ```
  maybeEval :: (ExprT -> Int) -> Maybe ExprT -> Maybe Int
  ```

  ```
  maybeMap :: (a -> b) -> Maybe a -> Maybe b
  ```

There's a repeated pattern here, and as good Haskell programmers we want to know how to generalize it! So which parts are the same from example to example, and which parts are different?

The part that is different, of course, is the container being "mapped over":

```
thingMap :: (a -> b) -> f a -> f b
```

But what sort of things are these "containers"? Can we really assign a type variable like `f` to them?

A brief digression on kinds

Just as every expression has a type, types themselves have "types", called *kinds*. (Before you ask: no, there's not another level beyond kinds—not in Haskell at least.) In `ghci` we can ask about the kinds of types using `:kind`. For example, let's ask for the kind of `Int`:

```
Prelude> :k Int
Int :: *
```

We see that `Int` has kind `*`. In fact, every type which can actually serve as the type of some values has kind `*`.

```
Prelude> :k Bool
Bool :: *
Prelude> :k Char
Char :: *
Prelude> :k Maybe Int
Maybe Int :: *
```

If `Maybe Int` has kind `*`, then what about `Maybe`? Notice that there are no values of type `Maybe`. There are values of type `Maybe Int`, and of type `Maybe Bool`, but not of type `Maybe`. But `Maybe` is certainly a valid type-like-thing. So what is it? What kind does it have? Let's ask `ghci`:

```
Prelude> :k Maybe
Maybe :: * -> *
```

`ghci` tells us that `Maybe` has kind `* -> *`. `Maybe` is, in a sense, a *function on types* — we usually call it a *type constructor*. `Maybe` takes as input types of kind `*`, and produces another type of kind `*`. For example, it can take as input `Int :: *` and produce the new type `Maybe Int :: *`.

Are there other type constructors with kind `* -> *`? Sure. For example, `Tree`, or the list type constructor, written `[]`.

```
Prelude> :k []
[] :: * -> *
Prelude :k [] Int
[] Int :: *
```

```
Prelude> :k [Int]  -- special syntax for [] Int
[Int] :: *
Prelude> :k Tree
Tree :: * -> *
```

What about type constructors with other kinds? How about `JoinList` from Homework 7?

```
data JoinList m a = Empty
                  | Single m a
                  | Append m (JoinList m a) (JoinList m a)
```

```
Prelude> :k JoinList
JoinList :: * -> * -> *
```

This makes sense: `JoinList` expects *two* types as parameters and gives us back a new type. (Of course, it is curried, so we can also think of it as taking *one* type and giving back something of kind `* -> *`.) Here's another one:

```
Prelude> :k (->)
(->) :: * -> * -> *
```

Tthis tells us that the function type constructor takes two type arguments. Like any operator, we use it infix:

```
Prelude> :k Int -> Char
Int -> Char :: *
```

But we don't have to:

```
Prelude> :k (->) Int Char
(->) Int Char :: *
```

OK, what about this one?

```
data Funny f a = Funny a (f a)
```

```
Prelude> :k Funny
Funny :: (* -> *) -> * -> *
```

`Funny` takes two arguments, the first one a type of kind `* -> *`, and the second a type of kind `*`, and constructs a type. (How did GHCi know what the kind of `Funny` is? Well, it does *kind inference* just like it also does *type inference*.) `Funny` is a *higher-order* type constructor, in the same way that `map` is a *higher-order* function. Note that types can be partially applied, too, just like functions:

```
Prelude> :k Funny Maybe
Funny Maybe :: * -> *
Prelude> :k Funny Maybe Int
Funny Maybe Int :: *
```

Functor

The essence of the mapping pattern we saw was a higher-order function with a type like

```
thingMap :: (a -> b) -> f a -> f b
```

where `f` is a type variable standing in for some type of kind `* -> *`. So, can we write a function of this type once and for all?

```
thingMap :: (a -> b) -> f a -> f b
thingMap h fa = ???
```

Well, not really. There's not much we can do if we don't know what `f` is. `thingMap` has to work differently for each particular `f`. The solution is to make a type class, which is traditionally called `Functor`:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

(`Functor` is defined in the standard Prelude. Note that the name "functor" comes from category theory, and is *not* the same thing as functors in C++ (which are essentially first-class functions).) Now we can just implement this class in a way specific to each particular `f`. Note that the `Functor` class abstracts over types of kind `* -> *`. So it would make no sense to write

```
instance Functor Int where
  fmap = ...
```

Indeed, if we try, we get a very nice *kind mismatch error*:

```
[1 of 1] Compiling Main             ( 09-functors.lhs, interpreted )

09-functors.lhs:145:19:
    Kind mis-match
    The first argument of `Functor' should have kind `* -> *',
    but `Int' has kind `*'
    In the instance declaration for `Functor Int'
```

If we understand kinds, this error tells us exactly what is wrong.

However, it does make sense (kind-wise) to make a `Functor` instance for, say, `Maybe`. Let's do it. Following the types makes it almost trivial:

```
instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap h (Just a) = Just (h a)
```

How about lists?

```
instance Functor [] where
  fmap _ []     = []
  fmap f (x:xs) = f x : fmap f xs
  -- or just
  -- fmap = map
```

Easy peasy. What about `IO`? Could it make sense to create an instance of `Functor` for `IO`?

Sure. `fmap :: (a -> b) -> IO a -> IO b` results in the IO action which first runs the `IO a` action, then applies the function to transform the result before returning it. We can implement this without too much trouble:

```
instance Functor IO where
  fmap f ioa = ioa >>= (\a -> return (f a))
```

or even

```
instance Functor IO where
  fmap f ioa = ioa >>= (return . f)
```

Now let's try something a bit more mind-twisting:

```
instance Functor ((->) e) where
```

What!? Well, let's follow the types: if `f = (->) e` then we want

```
fmap :: (a -> b) -> (->) e a -> (->) e b
```

or, with `(->)` written infix:

```
fmap :: (a -> b) -> (e -> a) -> (e -> b)
```

Hmm, this type signature seems familiar…

```
instance Functor ((->) e) where
  fmap = (.)
```

Crazy! What does this mean? Well, one way to think of a value of type `(e -> a)` is as a "`e`-indexed container" with one value of `a` for each value of `e`. To map a function over every value in such a container corresponds exactly to function composition: to pick an element out of the transformed container, we first we apply the `(e -> a)` function to pick out an `a` from the original container, and then apply the `(a -> b)` function to transform the element we picked.

Applicative functors, Part I

CIS 194 Week 10
25 March 2012

Suggested reading:

- Applicative Functors from Learn You a Haskell
- The Typeclassopedia

Motivation

Consider the following `Employee` type:

```
type Name = String

data Employee = Employee { name    :: Name
                         , phone   :: String }
                deriving Show
```

Of course, the `Employee` constructor has type

```
Employee :: Name -> String -> Employee
```

That is, if we have a `Name` and a `String`, we can apply the `Employee` constructor to build an `Employee` object.

Suppose, however, that we don't have a `Name` and a `String`; what we actually have is a `Maybe Name` and a `Maybe String`. Perhaps they came from parsing some file full of errors, or from a form where some of the fields might have been left blank, or something of that sort. We can't necessarily make an `Employee`. But surely we can make a `Maybe Employee`. That is, we'd like to take our `(Name -> String -> Employee)` function and turn it into a `(Maybe Name -> Maybe String -> Maybe Employee)` function. Can we write something with this type?

```
(Name -> String -> Employee) ->
(Maybe Name -> Maybe String -> Maybe Employee)
```

Sure we can, and I am fully confident that you could write it in your sleep by now. We can imagine how it would work: if either the name or string is `Nothing`, we get `Nothing` out; if both are `Just`, we get out an `Employee` built using the `Employee` constructor (wrapped in `Just`). But let's keep going…

Consider this: now instead of a `Name` and a `String` we have a `[Name]` and a `[String]`. Maybe we can get an `[Employee]` out of this? Now we want

```
(Name -> String -> Employee) ->
([Name] -> [String] -> [Employee])
```

We can imagine two different ways for this to work: we could match up corresponding `Name`s and `String`s to form `Employee`s; or we could pair up the `Name`s and `String`s in all possible ways.

Or how about this: we have an `(e -> Name)` and `(e -> String)` for some type `e`. For example, perhaps `e` is some huge data structure, and we have functions telling us how to extract a `Name` and a `String` from it. Can we make it into an `(e -> Employee)`, that is, a recipe for extracting an `Employee` from the same structure?

```
(Name -> String -> Employee) ->
((e -> Name) -> (e -> String) -> (e -> Employee))
```

No problem, and this time there's really only one way to write this function.

Generalizing

Now that we've seen the usefulness of this sort of pattern, let's generalize a bit. The type of the function we want really looks something like this:

```
(a -> b -> c) -> (f a -> f b -> f c)
```

Hmm, this looks familiar… it's quite similar to the type of `fmap`!

```
fmap :: (a -> b) -> (f a -> f b)
```

The only difference is an extra argument; we might call our desired function `fmap2`, since it takes a function of two arguments. Perhaps we can write `fmap2` in terms of `fmap`, so we just need a `Functor` constraint on `f`:

```
fmap2 :: Functor f => (a -> b -> c) -> (f a -> f b -> f c)
fmap2 h fa fb = undefined
```

Try hard as we might, however, `Functor` does not quite give us enough to implement `fmap2`. What goes wrong? We have

```
h  :: a -> b -> c
fa :: f a
fb :: f b
```

Note that we can also write the type of `h` as `a -> (b -> c)`. So, we have a function that takes an `a`, and we have a value of type `f a`… the only thing we can do is use `fmap` to lift the function over the `f`, giving us a result of type:

```
h          :: a -> (b -> c)
fmap h     :: f a -> f (b -> c)
fmap h fa  :: f (b -> c)
```

OK, so now we have something of type `f (b -> c)` and something of type `f b`… and here's where we are stuck! `fmap` does not help any more. It gives us a way to apply functions to values inside a `Functor` context, but what we need now is to apply a functions *which are themselves in a* ***`Functor`*** *context* to values in a `Functor` context.

Applicative

Functors for which this sort of "contextual application" is possible are called *applicative*, and the `Applicative` class (defined in `Control.Applicative`) captures this pattern.

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The `(<*>)` operator (often pronounced "ap", short for "apply") encapsulates exactly this principle of "contextual application". Note also that the `Applicative` class requires its instances to be instances of `Functor` as well, so we can always use `fmap` with instances of `Applicative`. Finally, note that `Applicative` also has another method, `pure`, which lets us inject a value of type `a` into a container. For now, it is interesting to note that `fmap0` would be another reasonable name for `pure`:

```
pure  :: a             -> f a
fmap  :: (a -> b)      -> f a -> f b
fmap2 :: (a -> b -> c) -> f a -> f b -> f c
```

Now that we have `(<*>)`, we can implement `fmap2`, which in the standard library is actually called `liftA2`:

```
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 h fa fb = (h `fmap` fa) <*> fb
```

In fact, this pattern is so common that `Control.Applicative` defines `(<$>)` as a synonym for `fmap`,

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

so that we can write

```
liftA2 h fa fb = h <$> fa <*> fb
```

What about `liftA3`?

```
liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
liftA3 h fa fb fc = ((h <$> fa) <*> fb) <*> fc
```

(Note that the precedence and associativity of `(<$>)` and `(<*>)` are actually defined in such a way that all the parentheses above are unnecessary.)

Nifty! Unlike the jump from `fmap` to `liftA2` (which required generalizing from `Functor` to `Applicative`), going from `liftA2` to `liftA3` (and from there to `liftA4`, …) requires no extra power—`Applicative` is enough.

Actually, when we have all the arguments like this we usually don't bother calling `liftA2`, `liftA3`, and so on, but just use the `f <$> x <*> y <*> z <*> ...` pattern directly. (`liftA2` and friends do come in handly for partial application, however.)

But what about `pure`? `pure` is for situations where we want to apply some function to arguments in the context of some functor `f`, but one or more of the arguments is *not* in `f`—those arguments are "pure", so to speak. We can use `pure` to lift them up into `f` first before applying. Like so:

```
liftX :: Applicative f => (a -> b -> c -> d) -> f a -> b -> f c -> f d
liftX h fa b fc = h <$> fa <*> pure b <*> fc
```

Applicative laws

There is only one really "interesting" law for `Applicative`:

```
f `fmap` x === pure f <*> x
```

Mapping a function `f` over a container `x` ought to give the same results as first injecting the function into the container, and then applying it to `x` with `(<*>)`.

There are other laws, but they are not as instructive; you can read about them on your own if you really want.

Applicative examples

Maybe

Let's try writing some instances of `Applicative`, starting with `Maybe`. `pure` works by injecting a value into a `Just` wrapper; `(<*>)` is function application with possible failure. The result is `Nothing` if either the function or its argument are.

```
instance Applicative Maybe where
  pure              = Just
  Nothing <*> _     = Nothing
  _ <*> Nothing     = Nothing
  Just f <*> Just x = Just (f x)
```

Let's see an example:

```
m_name1, m_name2 :: Maybe Name
m_name1 = Nothing
m_name2 = Just "Brent"

m_phone1, m_phone2 :: Maybe String
m_phone1 = Nothing
m_phone2 = Just "555-1234"

exA = Employee <$> m_name1 <*> m_phone1
exB = Employee <$> m_name1 <*> m_phone2
exC = Employee <$> m_name2 <*> m_phone1
exD = Employee <$> m_name2 <*> m_phone2
```

Applicative functors, Part II

CIS 194 Week 11
1 April 2012

Suggested reading:

- Applicative Functors from Learn You a Haskell
- The Typeclassopedia

We begin with a review of the `Functor` and `Applicative` type classes:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Every `Applicative` is also a `Functor`—so can we implement `fmap` in terms of `pure` and `(<*>)`? Let's try!

```
fmap g x = pure g <*> x
```

Well, that has the right type at least! However, it's not hard to imagine making `Functor` and `Applicative` instances for some type such that this equality does not hold. Since this would be a fairly dubious situation, we stipulate as a *law* that this equality must hold—this is a formal way of stating that the `Functor` and `Applicative` instances for a given type must "play nicely together".

Now, let's see a few more examples of `Applicative` instances.

More Applicative Examples

Lists

How about an instance of `Applicative` for lists? There are actually two possible instances: one that matches up the list of functions and list of arguments elementwise (that is, it "zips" them together), and one that combines functions and arguments in all possible ways.

First, let's write the instance that does all possible combinations. (For reasons that will become clear next week, this is the default instance.) From this point of view, lists represent nondeterminism: that is, a value of type `[a]` can be thought of as a single value with multiple possibilities. Then `(<*>)` corresponds to nondeterministic function application—that is, the application of a nondeterministic function to a nondeterministic argument.

```
instance Applicative [] where
  pure a        = [a]           -- a "deterministic" value
  [] <*> _      = []
  (f:fs) <*> as = (map f as) ++ (fs <*> as)
```

Here's an example:

```
names  = ["Joe", "Sara", "Mae"]
phones = ["555-5555", "123-456-7890", "555-4321"]

employees1 = Employee <$> names <*> phones
```

Maybe this particular example doesn't make that much sense, but it's not hard to imagine situations where you want to combine things in all possible ways like this. For example, we can do nondeterministic arithmetic like so:

```
(.+) = liftA2 (+)    -- addition lifted to some Applicative context
(.*) = liftA2 (*)    -- same for multiplication

-- nondeterministic arithmetic
n = ([4,5] .* pure 2) .+ [6,1] -- (either 4 or 5) times 2, plus either 6 or 1

-- and some possibly-failing arithmetic too, just for fun
m1 = (Just 3 .+ Just 5) .* Just 8
m2 = (Just 3 .+ Nothing) .* Just 8
```

Next, let's write the instance that does elementwise combining. First, we must answer an important question: how should we handle lists of different lengths? Some thought reveals that the most sensible thing to do is to truncate the longer list to the length of the shorter, throwing away the extra elements. Of course there are other possible answers: we might, for instance, extend the shorter list by copying the last element (but then what do we do when one of the lists is empty?); or extend the shorter list with a "neutral" element (but then we would have to require an instance of `Monoid`, or an extra "default" argument for the application).

This decision in turn dictates how we must implement `pure`, since we must obey the law

```
pure f <*> xs === f <$> xs
```

Notice that the right-hand side is a list with the same length as `xs`, formed by applying `f` to every element in `xs`. The only way we can make the left-hand side turn out the same… is for `pure` to create an infinite list of copies of `f`, because we don't know in advance how long `xs` is going to be.

We implement the instance using a `newtype` wrapper to distinguish it from the other list instance. The standard Prelude function `zipWith` also comes in handy.

```
newtype ZipList a = ZipList { getZipList :: [a] }
  deriving (Eq, Show, Functor)

instance Applicative ZipList where
  pure = ZipList . repeat
  ZipList fs <*> ZipList xs = ZipList (zipWith ($) fs xs)
```

An example:

```
employees2 = getZipList $ Employee <$> ZipList names <*> ZipList phones
```

Reader/environment

Let's do one final example instance, for `(->) e`. This is known as the *reader* or *environment* applicative, since it allows "reading" from the "environment" `e`. Implementing the instance is not too hard, we just have to use our nose and follow the types:

```
instance Functor ((->) e) where
  fmap = (.)

instance Applicative ((->) e) where
  pure = const
  f <*> x = \e -> (f e) (x e)
```

An `Employee` example:

```
data BigRecord = BR { getName         :: Name
                    , getSSN          :: String
                    , getSalary       :: Integer
                    , getPhone        :: String
                    , getLicensePlate :: String
                    , getNumSickDays  :: Int
                    }

r = BR "Brent" "XXX-XX-XXX4" 600000000 "555-1234" "JGX-55T3" 2

getEmp :: BigRecord -> Employee
getEmp = Employee <$> getName <*> getPhone

exQ = getEmp r
```

Aside: Levels of Abstraction

`Functor` is a nifty tool but relatively straightforward. At first glance it seems like `Applicative` doesn't add that much beyond what `Functor` already provides, but it turns out that it's a small addition with a huge impact. `Applicative` (and as we will see next week, `Monad`) deserves to be called a "model of computation", while `Functor` doesn't.

When working with things like `Applicative` and `Monad`, it's very important to keep in mind that there are *multiple levels of abstraction* involved. Roughly speaking, an *abstraction* is something which *hides details* of a lower level, providing a "high-level" interface that can be used (ideally) without thinking about the lower level—although the details of the lower level often "leak through" in certain cases. This idea of layers of abstraction is widespread. Think about user programs—OS—kernel—integrated circuits—gates—silicon, or HTTP—TCP—IP—Ethernet, or programming languages—bytecode—assembly—machine code. As we have seen, Haskell gives us many nice tools for constructing multiple layers of abstraction *within Haskell programs themselves*, that is, we get to dynamically extend the "programming language" layer stack upwards. This is a powerful facility but can lead to confusion. One must learn to explicitly be able to think on multiple levels, and to switch between levels.

With respect to `Applicative` and `Monad` in particular, there are just two levels to be concerned with. The first is the level of implementing various `Applicative` and `Monad` instances, *i.e.* the "raw Haskell" level. You gained some experience with this level in your previous homework, when you implemented an `Applicative` instance for `Parser`.

Once we have an `Applicative` instance for a type like `Parser`, the point is that we get to "move up a layer" and program with `Parser`s *using the `Applicative` interface*, without thinking about the details of how `Parser` and its `Applicative` instance are actually implemented. You got a little bit of experience with this on last week's homework, and will get a lot more of it this week. Programming at this level has a very different feel than actually implementing the instances. Let's see some examples.

The Applicative API

One of the benefits of having a unified interface like `Applicative` is that we can write generic tools and control structures that work with *any* type which is an instance of `Applicative`. As a first example, let's try writing

```
pair :: Applicative f => f a -> f b -> f (a,b)
```

`pair` takes two values and pairs them, but all in the context of some `Applicative f`. As a first try we can take a function for pairing and "lift" it over the arguments using `(<$>)` and `(<*>)`:

```
pair fa fb = (\x y -> (x,y)) <$> fa <*> fb
```

This works, though we can simplify it a bit. First, note that Haskell allows the special syntax `(,)` to represent the pair constructor, so we can write

```
pair fa fb = (,) <$> fa <*> fb
```

But actually, we've seen this pattern before—this is the `liftA2` pattern which got us started down this whole `Applicative` road. So we can further simplify to

```
pair fa fb = liftA2 (,) fa fb
```

but now there is no need to explicitly write out the function arguments, so we reach our final simplified version:

```
pair = liftA2 (,)
```

Now, what does this function do? It depends, of course, on the particular `f` chosen. Let's consider a number of particular examples:

- `f = Maybe`: the result is `Nothing` if either of the arguments is; if both are `Just` the result is `Just` their pairing.
- `f = []`: `pair` computes the Cartesian product of two lists.
- `f = ZipList`: `pair` is the same as the standard `zip` function.
- `f = IO`: `pair` runs two `IO` actions in sequence, returning a pair of their results.
- `f = Parser`: `pair` runs two parsers in sequence (the parsers consume consecutive sections of the input), returning their results as a pair. If either parser fails, the whole thing fails.

Can you implement the following functions? Consider what each function does when `f` is replaced with each of the above types.

```
(*>)       :: Applicative f => f a -> f b -> f b
mapA       :: Applicative f => (a -> f b) -> ([a] -> f [b])
sequenceA  :: Applicative f => [f a] -> f [a]
replicateA :: Applicative f => Int -> f a -> f [a]
```

Monads

CIS 194 Week 12
8 April 2013

Suggested reading:

- The Typeclassopedia
- LYAH Chapter 12: A Fistful of Monads
- LYAH Chapter 9: Input and Output
- RWH Chapter 7: I/O
- RWH Chapter 14: Monads
- RWH Chapter 15: Programming with monads

Motivation

Over the last couple of weeks, we have seen how the `Applicative` class allows us to idiomatically handle computations which take place in some sort of "special context"—for example, taking into account possible failure with `Maybe`, multiple possible outputs with `[]`, consulting some sort of environment using `((->) e)`, or construct parsers using a "combinator" approach, as in the homework.

However, so far we have only seen computations with a fixed structure, such as applying a data constructor to a fixed set of arguments. What if we don't know the structure of the computation in advance – that is, we want to be able to decide what to do based on some intermediate results?

As an example, recall the `Parser` type from the homework, and assume that we have implemented `Functor` and `Applicative` instances for it:

```
newtype Parser a = Parser { runParser :: String -> Maybe (a, String) }
```

```
instance Functor Parser where
  ...

instance Applicative Parser where
  ...
```

Recall that a value of type `Parser a` represents a *parser* which can take a `String` as input and possibly produce a value of type `a`, along with the remaining unparsed portion of the `String`. For example, a parser for integers, given as input the string

```
"143xkkj"
```

might produce as output

```
Just (143, "xkkj")
```

As you saw in the homework, we can now write things like

```
data Foo = Bar Int Int Char

parseFoo :: Parser Foo
parseFoo = Bar <$> parseInt <*> parseInt <*> parseChar
```

assuming we have functions `parseInt :: Parser Int` and `parseChar :: Parser Char`. The `Applicative` instance automatically handles the possible failure (if parsing any of the components fail, parsing the entire `Foo` will fail) and threading through the unconsumed portion of the `String` input to each component in turn.

However, suppose we are trying to parse a file containing a sequence of numbers, like this:

```
4 78 19 3 44 3 1 7 5 2 3 2
```

The catch is that the first number in the file tells us the length of a following "group" of numbers; the next number after the group is the length of the next group, and so on. So the example above could be broken up into groups like this:

```
78 19 3 44   -- first group
1 7 5        -- second group
3 2          -- third group
```

This is a somewhat contrived example, but in fact there are many "real-world" file formats that follow a similar principle—you read some sort of header which then tells you the lengths of some following blocks, or where to find things in the file, and so on.

We would like to write a parser for this file format of type

```
parseFile :: Parser [[Int]]
```

Unfortunately, this is not possible using only the `Applicative` interface. The problem is that `Applicative` gives us no way to decide what to do next based on previous results: we must decide in advance what parsing operations we are going to run, before we see the results.

It turns out, however, that the `Parser` type *can* support this sort of pattern, which is abstracted into the `Monad` type class.

Monad

The `Monad` type class is defined as follows:

```
class Monad m where
  return :: a -> m a

  (>>=) :: m a -> (a -> m b) -> m b

  (>>)  :: m a -> m b -> m b
  m1 >> m2 = m1 >>= \_ -> m2
```

This should look familiar! We have seen these methods before in the context of `IO`, but in fact they are not specific to `IO` at all. It's just that a monadic interface to `IO` has proved useful.

`return` also looks familiar because it has the same type as `pure`. In fact, every `Monad` should also be an `Applicative`, with `pure = return`. The reason we have both is that `Applicative` was invented *after* `Monad` had already been around for a while.

`(>>)` is just a specialized version of `(>>=)` (it is included in the `Monad` class in case some instance wants to provide a more efficient implementation, but usually the default implementation is just fine). So to understand it we first need to understand `(>>=)`.

There is actually a fourth method called `fail`, but putting it in the `Monad` class was a mistake, and you should never use it, so I won't tell you about it (you can read about it in the Typeclassopedia if you are interested).

`(>>=)` (pronounced "bind") is where all the action is! Let's think carefully about its type:

```
(>>=) :: m a -> (a -> m b) -> m b
```

`(>>=)` takes two arguments. The first one is a value of type `m a`. (Incidentally, such values are sometimes called *monadic values*, or *computations*. It has also been proposed to call them *mobits*. The one thing you must *not* call them is "monads", since that is a kind error: the type constructor `m` is a monad.) In any case, the idea is that a mobit of type `m a` represents a computation which results in a value (or several values, or no values) of type `a`, and may also have some sort of "effect":

- `c1 :: Maybe a` is a computation which might fail but results in an `a` if it succeeds.
- `c2 :: [a]` is a computation which results in (multiple) `a`s.
- `c3 :: Parser a` is a computation which implicitly consumes part of a `String` and (possibly) produces an `a`.
- `c4 :: IO a` is a computation which potentially has some I/O effects and then produces an `a`.

And so on. Now, what about the second argument to `(>>=)`? It is a *function* of type `(a -> m b)`. That is, it is a function which will *choose* the next computation to run based on the result(s) of the first computation. This is precisely what embodies the promised power of `Monad` to encapsulate computations which can choose what to do next based on the results of previous computations.

So all `(>>=)` really does is put together two mobits to produce a larger one, which first runs one and then the other, returning the result of the second one. The all-important twist is that we get to decide which mobit to run second based on the output from the first.

The default implementation of `(>>)` should make sense now:

```
(>>)  :: m a -> m b -> m b
m1 >> m2 = m1 >>= \_ -> m2
```

`m1 >> m2` simply does `m1` and then `m2`, ignoring the result of `m1`.

Examples

Let's start by writing a `Monad` instance for `Maybe`:

```
instance Monad Maybe where
  return  = Just
  Nothing >>= _ = Nothing
  Just x  >>= k = k x
```

`return`, of course, is `Just`. If the first argument of `(>>=)` is `Nothing`, then the whole computation fails; otherwise, if it is `Just x`, we apply the second argument to `x` to decide what to do next.

Incidentally, it is common to use the letter `k` for the second argument of `(>>=)` because `k` stands for "continuation". I wish I was joking.

Some examples:

```
check :: Int -> Maybe Int
check n | n < 10    = Just n
        | otherwise = Nothing

halve :: Int -> Maybe Int
halve n | even n    = Just $ n `div` 2
        | otherwise = Nothing

exM1 = return 7 >>= check >>= halve
exM2 = return 12 >>= check >>= halve
exM3 = return 12 >>= halve >>= check
```

How about a `Monad` instance for the list constructor `[]`?

```
instance Monad [] where
  return x = [x]
  xs >>= k = concat (map k xs)
```

A simple example:

```
addOneOrTwo :: Int -> [Int]
addOneOrTwo x = [x+1, x+2]

exL1 = [10,20,30] >>= addOneOrTwo
```

Monad combinators

One nice thing about the `Monad` class is that using only `return` and `(>>=)` we can build up a lot of nice general combinators for programming with monads. Let's look at a couple.

First, `sequence` takes a list of monadic values and produces a single monadic value which collects the results. What this means depends on the particular monad. For example, in the case of `Maybe` it means that the entire computation succeeds only if all the individual ones do; in the case of `IO` it means to run all the computations in sequence; in the case of `Parser` it means to run all the parsers on sequential parts of the input (and succeed only if they all do).

```
sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (ma:mas) =
  ma >>= \a ->
  sequence mas >>= \as ->
  return (a:as)
```

Using `sequence` we can also write other combinators, such as

```
replicateM :: Monad m => Int -> m a -> m [a]
replicateM n m = sequence (replicate n m)
```

And now we are finally in a position to write the parser we wanted to write: it is simply

```
parseFile :: Parser [[Int]]
parseFile = many parseLine

parseLine :: Parser [Int]
parseLine = parseInt >>= \i -> replicateM i parseInt
```

(`many` was also known as `zeroOrMore` on the homework).

CIS 194: Homework 1

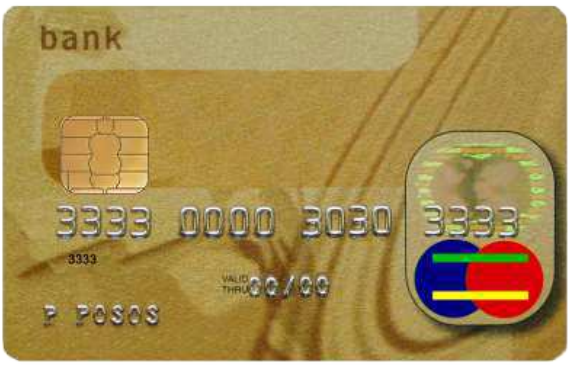
Due Monday, January 14

When solving the homework, strive to create not just code that works, but code that is stylish and concise. See the style guide on the website for some general guidelines. Try to write small functions which perform just a single task, and then combine those smaller pieces to create more complex functions. Don't repeat yourself: write one function for each logical task, and reuse functions as necessary.

Be sure to write functions with exactly the specified name and type signature for each exercise (to help us test your code). You may create additional helper functions with whatever names and type signatures you wish.

Validating Credit Card Numbers[1]

Have you ever wondered how websites validate your credit card number when you shop online? They don't check a massive database of numbers, and they don't use magic. In fact, most credit providers rely on a checksum formula for distinguishing valid numbers from random collections of digits (or typing mistakes).



In this section, you will implement the validation algorithm for credit cards. It follows these steps:

- Double the value of every second digit beginning from the right. That is, the last digit is unchanged; the second-to-last digit is doubled; the third-to-last digit is unchanged; and so on. For example, `[1,3,8,6]` becomes `[2,3,16,6]`.
- Add the digits of the doubled values and the undoubled digits from the original number. For example, `[2,3,16,6]` becomes `2+3+1+6+6 = 18`.

[1] Adapted from the first practicum assigned in the University of Utrecht functional programming course taught by Doaitse Swierstra, 2008-2009.

- Calculate the remainder when the sum is divided by 10. For the above example, the remainder would be `8`.

  If the result equals 0, then the number is valid.

Exercise 1 We need to first find the digits of a number. Define the functions

```
toDigits    :: Integer -> [Integer]
toDigitsRev :: Integer -> [Integer]
```

`toDigits` should convert positive `Integer`s to a list of digits. (For `0` or negative inputs, `toDigits` should return the empty list.) `toDigitsRev` should do the same, but with the digits reversed.

Example: `toDigits 1234 == [1,2,3,4]`

Example: `toDigitsRev 1234 == [4,3,2,1]`

Example: `toDigits 0 == []`

Example: `toDigits (-17) == []`

Exercise 2 Once we have the digits in the proper order, we need to double every other one. Define a function

```
doubleEveryOther :: [Integer] -> [Integer]
```

Remember that `doubleEveryOther` should double every other number *beginning from the right*, that is, the second-to-last, fourth-to-last, ... numbers are doubled.

Example: `doubleEveryOther [8,7,6,5] == [16,7,12,5]`

Example: `doubleEveryOther [1,2,3] == [1,4,3]`

Exercise 3 The output of `doubleEveryOther` has a mix of one-digit and two-digit numbers. Define the function

```
sumDigits :: [Integer] -> Integer
```

to calculate the sum of all digits.

Example: `sumDigits [16,7,12,5] = 1 + 6 + 7 + 1 + 2 + 5 = 22`

Exercise 4 Define the function

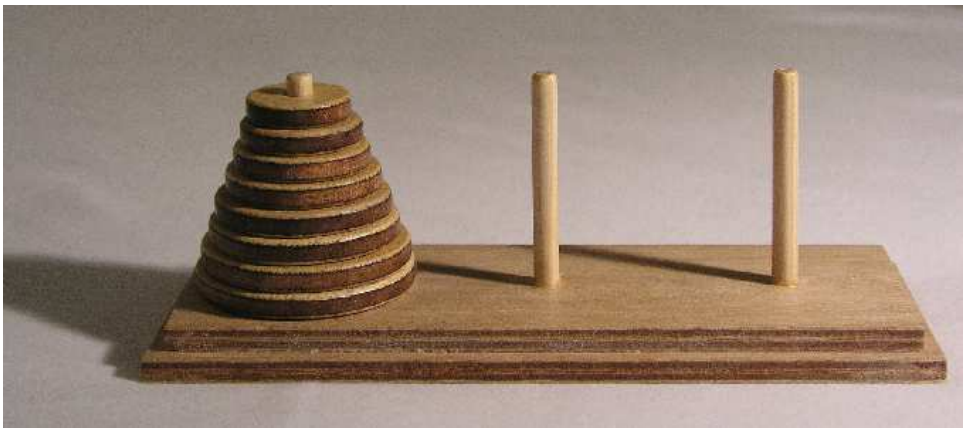
```
validate :: Integer -> Bool
```

that indicates whether an `Integer` could be a valid credit card number. This will use all functions defined in the previous exercises.

Example: `validate 4012888888881881 = True`

Example: `validate 4012888888881882 = False`

The Towers of Hanoi[2]



Exercise 5 The *Towers of Hanoi* is a classic puzzle with a solution that can be described recursively. Disks of different sizes are stacked on three pegs; the goal is to get from a starting configuration with all disks stacked on the first peg to an ending configuration with all disks stacked on the last peg, as shown in Figure 1.

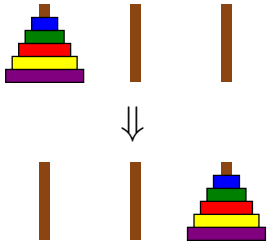


Figure 1: The Towers of Hanoi

The only rules are

- you may only move one disk at a time, and
- a larger disk may never be stacked on top of a smaller one.

For example, as the first move all you can do is move the topmost, smallest disk onto a different peg, since only one disk may be moved at a time.

Figure 2: A valid first move.

From this point, it is *illegal* to move to the configuration shown in Figure 3, because you are not allowed to put the green disk on top of the smaller blue one.

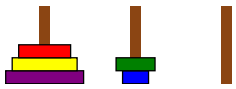


Figure 3: An illegal configuration.

To move n discs (stacked in increasing size) from peg a to peg b using peg c as temporary storage,

1. move $n-1$ discs from a to c using b as temporary storage
2. move the top disc from a to b
3. move $n-1$ discs from c to b using a as temporary storage.

For this exercise, define a function `hanoi` with the following type:

```
type Peg = String
type Move = (Peg, Peg)
hanoi :: Integer -> Peg -> Peg -> Peg -> [Move]
```

[2] Adapted from an assignment given in UPenn CIS 552, taught by Benjamin Pierce

Given the number of discs and names for the three pegs, `hanoi` should return a list of moves to be performed to move the stack of discs from the first peg to the second.

Note that a `type` declaration, like `type Peg = String` above, makes a *type synonym*. In this case `Peg` is declared as a synonym for `String`, and the two names `Peg` and `String` can now be used interchangeably. Giving more descriptive names to types in this way can be used to give shorter names to complicated types, or (as here) simply to help with documentation.

Example: `hanoi 2 "a" "b" "c" == [("a","c"), ("a","b"), ("c","b")]`

Exercise 6 (Optional) What if there are four pegs instead of three? That is, the goal is still to move a stack of discs from the first peg to the last peg, without ever placing a larger disc on top of a smaller one, but now there are two extra pegs that can be used as "temporary" storage instead of only one. Write a function similar to `hanoi` which solves this problem in as few moves as possible.

It should be possible to do it in far fewer moves than with three pegs. For example, with three pegs it takes $2^{15} - 1 = 32767$ moves to transfer 15 discs. With four pegs it can be done in 129 moves. (See Exercise 1.17 in Graham, Knuth, and Patashnik, *Concrete Mathematics*, second ed., Addison-Wesley, 1994.)

CIS 194: Homework 2

Due Monday January 28

Something has gone terribly wrong!

- Files you will need: `Log.hs`, `error.log`, `sample.log`
- Files you should submit: `LogAnalysis.hs`

Log file parsing

We're really not sure what happened, but we did manage to recover the log file `error.log`. It seems to consist of a different log message on each line. Each line begins with a character indicating the type of log message it represents:

- `'I'` for informational messages,
- `'W'` for warnings, and
- `'E'` for errors.

The error message lines then have an integer indicating the severity of the error, with 1 being the sort of error you might get around to caring about sometime next summer, and 100 being epic, catastrophic failure. All the types of log messages then have an integer time stamp followed by textual content that runs to the end of the line. Here is a snippet of the log file including an informational message followed by a level 2 error message:

```
I 147 mice in the air, I'm afraid, but you might catch a bat, and
E 2 148 #56k istereadeat lo d200ff] BOOTMEM
```

It's all quite confusing; clearly we need a program to sort through this mess. We've come up with some data types to capture the structure of this log file format:

```
data MessageType = Info
                 | Warning
                 | Error Int
                 deriving (Show, Eq)

type TimeStamp = Int

data LogMessage = LogMessage MessageType TimeStamp String
                | Unknown String
                deriving (Show, Eq)
```

Note that `LogMessage` has two constructors: one to represent normally-formatted log messages, and one to represent anything else that does not fit the proper format.

We've provided you with a module `Log.hs` containing these data type declarations, along with some other useful functions. Download `Log.hs` and put it in the same folder where you intend to put your homework assignment. Please name your homework assignment `LogAnalysis.hs` (or `.lhs` if you want to make it a literate Haskell document). The first few lines of `LogAnalysis.hs` should look like this:

```
{-# OPTIONS_GHC -Wall #-}
module LogAnalysis where

import Log
```

which sets up your file as a module named `LogAnalysis`, and imports the module from `Log.hs` so you can use the types and functions it provides.

Exercise 1 The first step is figuring out how to parse an individual message. Define a function

```
parseMessage :: String -> LogMessage
```

which parses an individual line from the log file. For example,

```
parseMessage "E 2 562 help help"
    == LogMessage (Error 2) 562 "help help"
```

```
parseMessage "I 29 la la la"
    == LogMessage Info 29 "la la la"

parseMessage "This is not in the right format"
    == Unknown "This is not in the right format"
```

Once we can parse one log message, we can parse a whole log file. Define a function

```
parse :: String -> [LogMessage]
```

which parses an entire log file at once and returns its contents as a list of `LogMessage`s.

To test your function, use the `testParse` function provided in the `Log` module, giving it as arguments your `parse` function, the number of lines to parse, and the log file to parse from (which should also be in the same folder as your assignment). For example, after loading your assignment into GHCi, type something like this at the prompt:

```
testParse parse 10 "error.log"
```

Don't reinvent the wheel! (That's so *last* week.) Use Prelude functions to make your solution as concise, high-level, and functional as possible. For example, to convert a `String` like `"562"` into an `Int`, you can use the `read` function. Other functions which may (or may not) be useful to you include `lines`, `words`, `unwords`, `take`, `drop`, and `(.)`.

Putting the logs in order

Unfortunately, due to the error messages being generated by multiple servers in multiple locations around the globe, a lightning storm, a failed disk, and a bored yet incompetent programmer, the log messages are horribly out of order. Until we do some organizing, there will be no way to make sense of what went wrong! We've designed a data structure that should help—a binary search tree of `LogMessage`s:

```
data MessageTree = Leaf
                 | Node MessageTree LogMessage MessageTree
```

Note that `MessageTree` is a recursive data type: the `Node` constructor itself takes two children as arguments, representing the left and right subtrees, as well as a `LogMessage`. Here, `Leaf` represents the empty tree.

A `MessageTree` should be sorted by timestamp: that is, the timestamp of a `LogMessage` in any `Node` should be greater than all timestamps of any `LogMessage` in the left subtree, and less than all timestamps of any `LogMessage` in the right child.

`Unknown` messages should not be stored in a `MessageTree` since they lack a timestamp.

Exercise 2 Define a function

```
insert :: LogMessage -> MessageTree -> MessageTree
```

which inserts a new `LogMessage` into an existing `MessageTree`, producing a new `MessageTree`. `insert` may assume that it is given a sorted `MessageTree`, and must produce a new sorted `MessageTree` containing the new `LogMessage` in addition to the contents of the original `MessageTree`.

However, note that if `insert` is given a `LogMessage` which is `Unknown`, it should return the `MessageTree` unchanged.

Exercise 3 Once we can insert a single `LogMessage` into a `MessageTree`, we can build a complete `MessageTree` from a list of messages. Specifically, define a function

```
build :: [LogMessage] -> MessageTree
```

which builds up a `MessageTree` containing the messages in the list, by successively inserting the messages into a `MessageTree` (beginning with a `Leaf`).

Exercise 4 Finally, define the function

```
inOrder :: MessageTree -> [LogMessage]
```

which takes a sorted `MessageTree` and produces a list of all the `LogMessage`s it contains, sorted by timestamp from smallest to biggest. (This is known as an *in-order traversal* of the `MessageTree`.)

With these functions, we can now remove `Unknown` messages and sort the well-formed messages using an expression such as:

```
inOrder (build tree)
```

[Note: there are much better ways to sort a list; this is just an exercise to get you working with recursive data structures!]

Log file postmortem

Exercise 5 Now that we can sort the log messages, the only thing left to do is extract the relevant information. We have decided that "relevant" means "errors with a severity of at least 50".

Write a function

```
whatWentWrong :: [LogMessage] -> [String]
```

which takes an *unsorted* list of `LogMessage`s, and returns a list of the messages corresponding to any errors with a severity of 50 or greater, sorted by timestamp. (Of course, you can use your functions from the previous exercises to do the sorting.)

For example, suppose our log file looked like this:

```
I 6 Completed armadillo processing
I 1 Nothing to report
E 99 10 Flange failed!
I 4 Everything normal
I 11 Initiating self-destruct sequence
E 70 3 Way too many pickles
E 65 8 Bad pickle-flange interaction detected
W 5 Flange is due for a check-up
I 7 Out for lunch, back in two time steps
E 20 2 Too many pickles
I 9 Back from lunch
```

This file is provided as `sample.log`. There are four errors, three of which have a severity of greater than 50. The output of `whatWentWrong` on `sample.log` ought to be

```
[ "Way too many pickles"
, "Bad pickle-flange interaction detected"
, "Flange failed!"
]
```

You can test your `whatWentWrong` function with `testWhatWentWrong`, which is also provided by the `Log` module. You should provide `testWhatWentWrong` with your `parse` function, your `whatWentWrong` function, and the name of the log file to parse.

Miscellaneous

- We will test your solution on log files other than the ones we have given you, so no hardcoding!
- You are free (in fact, encouraged) to discuss the assignment with any of your classmates as long as you type up your own solution.

Epilogue

Exercise 6 (Optional) For various reasons we are beginning to suspect that the recent mess was caused by a single, egotistical

hacker. Can you figure out who did it?

CIS 194: Homework 3

Due Monday, Feburary 4

Code golf!

This assignment is simple: there are three tasks listed below. For each task, you should submit a Haskell function with the required name and type signature which accomplishes the given task and is *as short as possible*.

Rules

- Along with your solution for each task you *must* include a comment explaining your solution and how it works. **Solutions without an explanatory comment will get a score of zero.** Your comment should demonstrate a complete understanding of your solution. In other words, anything is fair game but you must demonstrate that your understand how it works. If in doubt, include more detail.
- Comments do not count towards the length of your solutions.
- Type signatures do not count towards the length of your solutions.
- `import` statements do not count towards the length of your solutions. You may import any modules included in the Haskell Platform.
- **Whitespace does not count towards the length** of your solutions. So there is no need to shove all your code onto one line and take all the spaces out. Use space as appropriate, indent nicely, *etc.*, but otherwise try making your code as short as you can.

- You are welcome to include additional functions beyond the ones required. That is, you are welcome to break up your solutions into several functions if you wish (indeed, sometimes this may lead to a very short solution). Of course, such additional functions will be counted towards the length of your solution (excluding their type signatures).
- Your final submission should be named `Golf.hs`. Your file should define a module named `Golf`, that is, at the top of your file you should have

```
module Golf where
```

- The three shortest solutions (counting the total number of characters, excluding whitespace and the other exceptions listed above) for each task will receive two points of extra credit each. You can get up to a total of four extra credit points.
- Otherwise, the length does not really matter; long but correct solutions will receive full credit for correctness (although they may or may not get full credit for style, depending on their style).

Hints

- Use functions from the standard libraries as much as possible—that's part of the point of this assignment. Using (say) `map` is much shorter than implementing it yourself!
- In particular, try to use functions from the standard libraries that encapsulate recursion patterns, rather than writing explicitly recursive functions yourself.
- You may want to start by getting something that works, without worrying about the length. Once you have solved the task, try to figure out ways to make your solution shorter.
- If the specification of a task is unclear, feel free to ask for a clarification on Piazza.
- We will test your functions on other inputs besides the ones given as examples, so to be safe, so should you!

Tasks

Exercise 1 Hopscotch

Your first task is to write a function

```
skips :: [a] -> [[a]]
```

The output of `skips` is a list of lists. The first list in the output should be the same as the input list. The second list in the output should contain every second element from the input list... and the nth list in the output should contain every nth element from the input list.

For example:

```
skips "ABCD"       == ["ABCD", "BD", "C", "D"]
skips "hello!"     == ["hello!", "el!", "l!", "l", "o", "!"]
skips [1]          == [[1]]
skips [True,False] == [[True,False], [False]]
skips []           == []
```

Note that the output should be the same length as the input.

Exercise 2 Local maxima

A *local maximum* of a list is an element of the list which is strictly greater than both the elements immediately before and after it. For example, in the list `[2,3,4,1,5]`, the only local maximum is `4`, since it is greater than the elements immediately before and after it (`3` and `1`). `5` is not a local maximum since there is no element that comes after it.

Write a function

```
localMaxima :: [Integer] -> [Integer]
```

which finds all the local maxima in the input list and returns them in order. For example:

```
localMaxima [2,9,5,6,1] == [9,6]
localMaxima [2,3,4,1,5] == [4]
localMaxima [1,2,3,4,5] == []
```

Exercise 3 Histogram

For this task, write a function

```
histogram :: [Integer] -> String
```

which takes as input a list of `Integer`s between 0 and 9 (inclusive), and outputs a vertical histogram showing how many of each number were in the input list. You may assume that the input list does not contain any numbers less than zero or greater than 9 (that is, it does not matter what your function does if the input does contain such numbers). Your output must exactly match the output shown in the examples below.

```
histogram [1,1,1,5] ==

 *
 *
 *   *
==========
0123456789

histogram [1,4,5,4,6,6,3,4,2,4,9] ==

    *
    *
    * *
 ******  *
==========
0123456789
```

Important note: If you type something like `histogram [3,5]` at the `ghci` prompt, you should see something like this:

```
"   * *    \n==========\n0123456789\n"
```

This is a textual *representation* of the `String` output, including `\n` escape sequences to indicate newline characters. To actually visualize the histogram as in the examples above, use `putStr`, for example, `putStr (histogram [3,5])`.

CIS 194: Homework 4

Due Monday, February 11

What to turn in: you should turn a single `.hs` (or `.lhs`) file, which **must type check**.

Exercise 1: Wholemeal programming

Reimplement each of the following functions in a more idiomatic Haskell style. Use *wholemeal programming* practices, breaking each function into a pipeline of incremental transformations to an entire data structure. Name your functions `fun1'` and `fun2'` respectively.

1.
```
fun1 :: [Integer] -> Integer
fun1 []     = 1
fun1 (x:xs)
  | even x    = (x - 2) * fun1 xs
  | otherwise = fun1 xs
```

2.
```
fun2 :: Integer -> Integer
fun2 1 = 0
fun2 n | even n    = n + fun2 (n `div` 2)
       | otherwise = fun2 (3 * n + 1)
```

   Hint: For this problem you may wish to use the functions `iterate` and `takeWhile`. Look them up in the Prelude documentation to see what they do.

Exercise 2: Folding with trees

Recall the definition of a *binary tree* data structure. The *height* of a binary tree is the length of a path from the root to the deepest node. For example, the height of a tree with a single node is 0; the height of a tree with three nodes, whose root has two children, is 1; and so on. A binary tree is *balanced* if the height of its left and right subtrees differ by no more than 1, and its left and right subtrees are also balanced.

`http://en.wikipedia.org/wiki/Binary_tree`

You should use the following data structure to represent binary trees. Note that each node stores an extra `Integer` representing the height at that node.

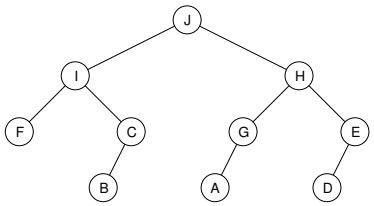
```
data Tree a = Leaf
            | Node Integer (Tree a) a (Tree a)
  deriving (Show, Eq)
```

For this exercise, write a function

```
foldTree :: [a] -> Tree a
foldTree = ...
```

which generates a balanced binary tree from a list of values using `foldr`.

For example, one sample output might be the following, also visualized at right:




```
foldTree "ABCDEFGHIJ" ==
  Node 3
    (Node 2
      (Node 0 Leaf 'F' Leaf)
      'I'
      (Node 1 (Node 0 Leaf 'B' Leaf) 'C' Leaf))
    'J'
    (Node 2
      (Node 1 (Node 0 Leaf 'A' Leaf) 'G' Leaf)
      'H'
      (Node 1 (Node 0 Leaf 'D' Leaf) 'E' Leaf))
```

Your solution might not place the nodes in the same exact order, but it should result in balanced trees, with each subtree having a correct computed height.

Exercise 3: More folds!

1. Implement a function

   ```
   xor :: [Bool] -> Bool
   ```

   which returns `True` if and only if there are an odd number of `True` values contained in the input list. It does not matter how many `False` values the input list contains. For example,

   ```
   xor [False, True, False] == True
   ```

   ```
   xor [False, True, False, False, True] == False
   ```

   Your solution must be implemented using a fold.

2. Implement `map` as a fold. That is, complete the definition

   ```
   map' :: (a -> b) -> [a] -> [b]
   map' f = foldr ...
   ```

   in such a way that `map'` behaves identically to the standard `map` function.

3. **(Optional)** Implement `foldl` using `foldr`. That is, complete the definition

```
myFoldl :: (a -> b -> a) -> a -> [b] -> a
myFoldl f base xs = foldr ...
```

   in such a way that `myFoldl` behaves identically to the standard `foldl` function.

   Hint: Study how the application of `foldr` and `foldl` work out:

```
foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...)
foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...) `f` xn
```

Exercise 4: Finding primes

Read about the *Sieve of Sundaram*. Implement the algorithm using function composition. Given an integer n, your function should generate all the odd prime numbers up to $2n + 2$.

http://en.wikipedia.org/wiki/Sieve_of_Sundaram

```
sieveSundaram :: Integer -> [Integer]
sieveSundaram = ...
```

To give you some help, below is a function to compute the *Cartesian product* of two lists. This is similar to `zip`, but it produces all possible pairs instead of matching up the list elements. For example,

```
cartProd [1,2] ['a','b'] == [(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

It's written using a *list comprehension*, which we haven't talked about in class (but feel free to research them).

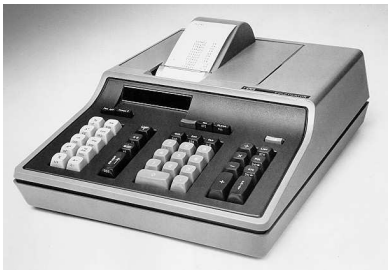
```
cartProd :: [a] -> [b] -> [(a, b)]
cartProd xs ys = [(x,y) | x <- xs, y <- ys]
```

CIS 194: Homework 5

Due Monday, 18 February

- Files you should submit: `Calc.hs`, containing a module of the same name.

As we saw in class, Haskell's *type classes* provide *ad-hoc polymorphism*, that is, the ability to decide what to do based on the type of an input. This homework explores one interesting use of type classes in constructing *domain-specific languages*.



Expressions

On day one of your new job as a software engineer, you've been asked to program the brains of the company's new blockbuster product: a calculator. But this isn't just any calculator! Extensive focus group analysis has revealed that what people really want out of their calculator is something that can add and multiply integers. Anything more just clutters the interface.

Your boss has already started by modeling the domain with the following data type of arithmetic expressions:

```
data ExprT  = Lit Integer
            | Add ExprT ExprT
            | Mul ExprT ExprT
  deriving (Show, Eq)
```

This type is capable of representing expressions involving integer constants, addition, and multiplication. For example, the expression $(2 + 3) \times 4$ would be represented by the value

```
Mul (Add (Lit 2) (Lit 3)) (Lit 4).
```

Your boss has already provided the definition of `ExprT` in `ExprT.hs`, so as usual you just need to add `import ExprT` to the top of your file. However, this is where your boss got stuck.

Exercise 1

Write Version 1 of the calculator: an evaluator for `ExprT`, with the signature

```
eval :: ExprT -> Integer
```

For example, `eval (Mul (Add (Lit 2) (Lit 3)) (Lit 4)) == 20`.

Exercise 2

The UI department has internalized the focus group data and is ready to synergize with you. They have developed the front-facing user-interface: a parser that handles the textual representation of the selected language. They have sent you the module `Parser.hs`, which exports `parseExp`, a parser for arithmetic expressions. If you pass the constructors of `ExprT` to it as arguments, it will convert `String`s representing arithmetic expressions into values of type `ExprT`. For example:

```
*Calc> parseExp Lit Add Mul "(2+3)*4"
Just (Mul (Add (Lit 2) (Lit 3)) (Lit 4))
*Calc> parseExp Lit Add Mul "2+3*4"
Just (Add (Lit 2) (Mul (Lit 3) (Lit 4)))
*Calc> parseExp Lit Add Mul "2+3*"
Nothing
```

Leverage the assets of the UI team to implement the value-added function

```
evalStr :: String -> Maybe Integer
```

which evaluates arithmetic expressions given as a `String`, producing `Nothing` for inputs which are not well-formed expressions, and `Just n` for well-formed inputs that evaluate to n.

Exercise 3

Good news! Early customer feedback indicates that people really do love the interface! Unfortunately, there seems to be some disagreement over exactly how the calculator should go about its calculating business. The problem the software department (*i.e.* you) has is that while `ExprT` is nice, it is also rather inflexible, which makes catering to diverse demographics a bit clumsy. You decide to abstract away the properties of `ExprT` with a type class.

Create a type class called `Expr` with three methods called `lit`, `add`, and `mul` which parallel the constructors of `ExprT`. Make an instance of `Expr` for the `ExprT` type, in such a way that

```
mul (add (lit 2) (lit 3)) (lit 4) :: ExprT
  == Mul (Add (Lit 2) (Lit 3)) (Lit 4)
```

Think carefully about what types `lit`, `add`, and `mul` should have. It may be helpful to consider the types of the `ExprT` constructors, which you can find out by typing (for example)

```
*Calc> :t Lit
```

at the `ghci` prompt.

Remark. Take a look at the type of the foregoing example expression:

```
*Calc> :t mul (add (lit 2) (lit 3)) (lit 4)
Expr a => a
```

What does this mean? The expression `mul (add (lit 2) (lit 3)) (lit 4)` has *any type* which is an instance of the `Expr` type class. So writing it by itself is ambiguous: GHC doesn't know what concrete type you want to use, so it doesn't know which implementations of `mul`, `add`, and `lit` to pick.

One way to resolve the ambiguity is by giving an explicit type signature, as in the above example. Another way is by using such an expression as part of some larger expression so that the context in which it is used determines the type. For example, we may write a function `reify` as follows:

```
reify :: ExprT -> ExprT
reify = id
```

To the untrained eye it may look like `reify` does no actual work! But its real purpose is to constrain the type of its argument to `ExprT`. Now we can write things like

```
reify $ mul (add (lit 2) (lit 3)) (lit 4)
```

at the `ghci` prompt.

Exercise 4

The marketing department has gotten wind of just how flexible the calculator project is and has promised custom calculators to some big clients. As you noticed after the initial roll-out, everyone loves the interface, but everyone seems to have their own opinion on what the *semantics* should be. Remember when we wrote `ExprT` and thought that addition and multiplication of integers was pretty cut and dried? Well, it turns out that some big clients want customized calculators with behaviors that they have decided are right for them.

The point of our `Expr` type class is that we can now write down arithmetic expressions *once* and have them interpreted in various ways just by using them at various types.

Make instances of `Expr` for each of the following types:

- `Integer` — works like the original calculator
- `Bool` — every literal value less than or equal to 0 is interpreted as `False`, and all positive `Integers` are interpreted as `True`; "addition" is logical or, "multiplication" is logical and
- `MinMax` — "addition" is taken to be the `max` function, while "multiplication" is the `min` function
- `Mod7` — all values should be in the ranage $0 \ldots 6$, and all arithmetic is done modulo 7; for example, $5 + 3 = 1$.

The last two variants work with `Integers` internally, but in order to provide different instances, we wrap those `Integers` in `newtype` wrappers. These are used just like the `data` constructors we've seen before.

```
newtype MinMax  = MinMax Integer deriving (Eq, Show)
newtype Mod7    = Mod7 Integer deriving (Eq, Show)
```

Once done, the following code should demonstrate our family of calculators:

```
testExp :: Expr a => Maybe a
testExp = parseExp lit add mul "(3 * -4) + 5"

testInteger  = testExp :: Maybe Integer
testBool     = testExp :: Maybe Bool
testMM       = testExp :: Maybe MinMax
testSat      = testExp :: Maybe Mod7
```

Try printing out each of those tests in `ghci` to see if things are working. It's great how easy it is for us to swap in new semantics for the same syntactic expression!

You must complete **at least one of** the following two exercises:

Exercise 5 (do this OR exercise 6)

The folks down in hardware have finished our new custom CPU, so we'd like to target that from now on. The catch is that a stack-based architecture was chosen to save money. You need to write a

version of your calculator that will emit assembly language for the new processor.

The hardware group has provided you with `StackVM.hs`, which is a software simulation of the custom CPU. The CPU supports six operations, as embodied in the `StackExp` data type:

```
data StackExp = PushI Integer
              | PushB Bool
              | Add
              | Mul
              | And
              | Or
                deriving Show

type Program = [StackExp]
```

`PushI` and `PushB` push values onto the top of the stack, which can store both `Integer` and `Bool` values. `Add`, `Mul`, `And`, and `Or` each pop the top two items off the top of the stack, perform the appropriate operation, and push the result back onto the top of the stack. For example, executing the program

```
[PushB True, PushI 3, PushI 6, Mul]
```

will result in a stack holding `True` on the bottom, and `18` on top of that.

If there are not enough operands on top of the stack, or if an operation is performed on operands of the wrong type, the processor will melt into a puddle of silicon goo. For a more precise specification of the capabilities and behavior of the custom CPU, consult the reference implementation provided in `StackVM.hs`.

Your task is to implement a compiler for arithmetic expressions. Simply create an instance of the `Expr` type class for `Program`, so that arithmetic expressions can be interpreted as compiled programs. For any arithmetic expression `exp :: Expr a => a` it should be the case that

```
stackVM exp == Right [IVal exp]
```

Note that in order to make an instance for `Program` (which is a type synonym) you will need to enable the `TypeSynonymInstances` language extension, which you can do by adding

```
{-# LANGUAGE TypeSynonymInstances #-}
```

as the *first line* in your file.

Finally, put together the pieces you have to create a function

```
compile :: String -> Maybe Program
```

which takes `String`s representing arithmetic expressions and compiles them into programs that can be run on the custom CPU.

Exercise 6 (do this OR exercise 5)

Some users of your calculator have requested the ability to give names to intermediate values and then reuse these stored values later.

To enable this, you first need to give arithmetic expressions the ability to contain variables. Create a new type class `HasVars a` which contains a single method `var :: String -> a`. Thus, types which are instances of `HasVars` have some notion of named variables.

Start out by creating a new data type `VarExprT` which is the same as `ExprT` but with an extra constructor for variables. Make `VarExprT` an instance of both `Expr` and `HasVars`. You should now be able to write things like

```
*Calc> add (lit 3) (var "x") :: VarExprT
```

But we can't stop there: we want to be able to interpret expressions containing variables, given a suitable mapping from variables to values. For storing mappings from variables to values, you should use the `Data.Map` module. Add

```
import qualified Data.Map as M
```

at the top of your file. The `qualified` import means that you must prefix `M.` whenever you refer to things from `Data.Map`. This is standard practice, since `Data.Map` exports quite a few functions with names that overlap with names from the `Prelude`. Consult the `Data.Map` documentation to read about the operations that are supported on `Map`s.

`http://hackage.haskell.org/packages/archive/containers/latest/doc/html/Data-Map.html`

Implement the following instances:

```
instance HasVars (M.Map String Integer -> Maybe Integer)


instance Expr (M.Map String Integer -> Maybe Integer)
```

The first instance says that variables can be interpreted as functions from a mapping of variables to `Integer` values to (possibly) `Integer` values. It should work by looking up the variable in the mapping.

The second instance says that these same functions can be interpreted as expressions (by passing along the mapping to subexpressions and combining results appropriately).

Note: to write these instances you will need to enable the `FlexibleInstances` language extension by putting

```
{-# LANGUAGE FlexibleInstances #-}
```

as the first line in your file.

Once you have created these instances, you should be able to test them as follows:

```
withVars :: [(String, Integer)]
         -> (M.Map String Integer -> Maybe Integer)
         -> Maybe Integer
withVars vs exp = exp $ M.fromList vs
```

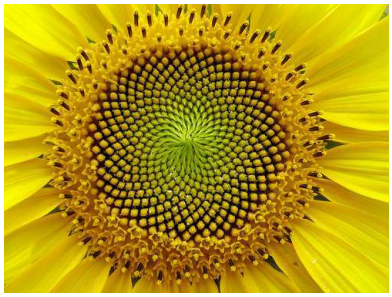
```
*Calc> :t add (lit 3) (var "x")
add (lit 3) (var "x") :: (Expr a, HasVars a) => a
*Calc> withVars [("x", 6)] $ add (lit 3) (var "x")
Just 9
*Expr> withVars [("x", 6)] $ add (lit 3) (var "y")
Nothing
*Calc> withVars [("x", 6), ("y", 3)]
         $ mul (var "x") (add (var "y") (var "x"))
Just 54
```

CIS 194: Homework 6

Due Monday, February 25

- Files you should submit: `Fibonacci.hs`

This week we learned about Haskell's *lazy evaluation*. This homework assignment will focus on one particular consequence of lazy evaluation, namely, the ability to work with infinite data structures.



Fibonacci numbers

The *Fibonacci numbers* F_n are defined as the sequence of integers, beginning with 0 and 1, where every integer in the sequence is the sum of the previous two. That is,

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad (n \geq 2) \end{aligned}$$

For example, the first fifteen Fibonacci numbers are

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$$

It's quite likely that you've heard of the Fibonacci numbers before. The reason they're so famous probably has something to do with the simplicity of their definition combined with the astounding variety of ways that they show up in various areas of mathematics as well as art and nature.[1]

[1] Note that you may have seen a definition where $F_0 = F_1 = 1$. This definition is wrong. There are several reasons; here are two of the most compelling:

- If we extend the Fibonacci sequence *backwards* (using the appropriate subtraction), we find

  $$\ldots, -8, 5, -3, 2, -1, 1, 0, 1, 1, 2, 3, 5, 8 \ldots$$

  0 is the obvious center of this pattern, so if we let $F_0 = 0$ then F_n and F_{-n} are either equal or of opposite signs, depending on the parity of n. If $F_0 = 1$ then everything is off by two.
- If $F_0 = 0$ then we can prove the lovely theorem "If m evenly divides n if and only if F_m evenly divides F_n." If $F_0 = 1$ then we have to state this as "If m evenly divides n if and only if F_{m-1} evenly divides F_{n-1}." Ugh.

Exercise 1

Translate the above definition of Fibonacci numbers directly into a recursive function definition of type

```
fib :: Integer -> Integer
```

so that `fib n` computes the nth Fibonacci number F_n.

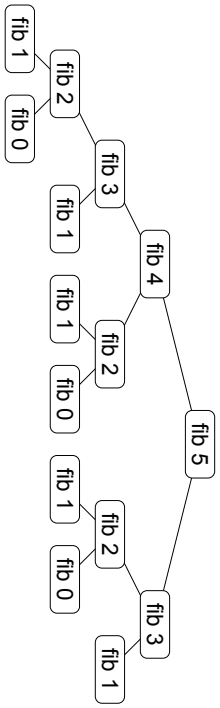
Now use `fib` to define the *infinite list* of all Fibonacci numbers,

```
fibs1 :: [Integer]
```

(*Hint*: You can write the list of all positive integers as `[0..]`.)

Try evaluating `fibs1` at the `ghci` prompt. You will probably get bored watching it after the first 30 or so Fibonacci numbers, because `fib` is ridiculously slow. Although it is a good way to *define* the Fibonacci numbers, it is not a very good way to *compute* them—in order

to compute F_n it essentially ends up adding 1 to itself F_n times! For example, shown at right is the tree of recursive calls made by evaluating `fib 5`.



As you can see, it does a lot of repeated work. In the end, `fib` has running time $O(F_n)$, which (it turns out) is equivalent to $O(\varphi^n)$, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the "golden ratio". That's right, the running time is *exponential* in n. What's more, all this work is also repeated from each element of the list `fibs1` to the next. Surely we can do better.

Exercise 2

When I said "we" in the previous sentence I actually meant "you". Your task for this exercise is to come up with more efficient implementation. Specifically, define the infinite list

```
fibs2 :: [Integer]
```

so that it has the same elements as `fibs1`, but computing the first n elements of `fibs2` requires only $O(n)$ addition operations. Be sure to use standard recursion pattern(s) from the `Prelude` as appropriate.

Of course there are several billion Haskell implementations of the Fibonacci numbers on the web, and I have no way to prevent you from looking at them; but you'll probably learn a lot more if you try to come up with something yourself first.

Streams

We can be more explicit about infinite lists by defining a type `Stream` representing lists that *must be* infinite. (The usual list type represents lists that *may* be infinite but may also have some finite length.)

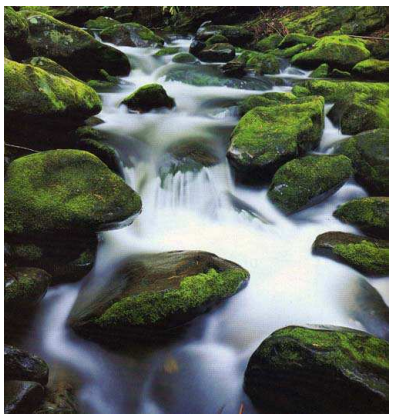
In particular, streams are like lists but with *only* a "cons" constructor—whereas the list type has two constructors, `[]` (the empty list) and `(:)` (cons), there is no such thing as an *empty stream*. So a stream is simply defined as an element followed by a stream.

Exercise 3

- Define a data type of polymorphic streams, `Stream`.
- Write a function to convert a `Stream` to an infinite list,

  ```
  streamToList :: Stream a -> [a]
  ```

- To test your `Stream` functions in the succeeding exercises, it will be useful to have an instance of `Show` for `Streams`. However, if you put `deriving Show` after your definition of `Stream`, as one usually does, the resulting instance will try to print an *entire* `Stream`—which, of course, will never finish. Instead, you should make your own instance of `Show` for `Stream`,



```
instance Show a => Show (Stream a) where
  show ...
```

which works by showing only some prefix of a stream (say, the first 20 elements).

Hint: you may find your `streamToList` function useful.

Exercise 4

Let's create some simple tools for working with `Stream`s.

- Write a function

  ```
  streamRepeat :: a -> Stream a
  ```

  which generates a stream containing infinitely many copies of the given element.

- Write a function

  ```
  streamMap :: (a -> b) -> Stream a -> Stream b
  ```

  which applies a function to every element of a `Stream`.

- Write a function

  ```
  streamFromSeed :: (a -> a) -> a -> Stream a
  ```

  which generates a `Stream` from a "seed" of type `a`, which is the first element of the stream, and an "unfolding rule" of type `a -> a` which specifies how to transform the seed into a new seed, to be used for generating the rest of the stream.

Exercise 5

Now that we have some tools for working with streams, let's create a few:

- Define the stream

  ```
  nats :: Stream Integer
  ```

  which contains the infinite list of natural numbers 0, 1, 2, ...

- Define the stream

  ```
  ruler :: Stream Integer
  ```

  which corresponds to the *ruler function*

  $$0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, \ldots$$

  where the nth element in the stream (assuming the first element corresponds to $n = 1$) is the largest power of 2 which evenly divides n.

Hint: define a function `interleaveStreams` which alternates the elements from two streams. Can you use this function to implement `ruler` in a clever way that does not have to do any divisibility testing?

Fibonacci numbers via generating functions (extra credit)

This section is optional but *very cool*, so if you have time I hope you will try it. We will use streams of `Integer`s to compute the Fibonacci numbers in an astounding way.

The essential idea is to work with *generating functions* of the form

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n + \ldots$$

where x is just a "formal parameter" (that is, we will never actually substitute any values for x; we just use it as a placeholder) and all the coefficients a_i are integers. We will store the coefficients $a_0, a_1, a_2, \ldots$ in a `Stream Integer`.

Exercise 6 (Optional)

- First, define

  ```
  x :: Stream Integer
  ```

  by noting that $x = 0 + 1x + 0x^2 + 0x^3 + \ldots$.

- Define an instance of the `Num` type class for `Stream Integer`. Here's what should go in your `Num` instance:
  - You should implement the `fromInteger` function. Note that $n = n + 0x + 0x^2 + 0x^3 + \ldots$.
  - You should implement `negate`: to negate a generating function, negate all its coefficients.
  - You should implement `(+)`, which works like you would expect: $(a_0 + a_1 x + a_2 x^2 + \ldots) + (b_0 + b_1 x + b_2 x^2 + \ldots) = (a_0 + b_0) + (a_1 + b_1)x + (a_2 + b_2)x^2 + \ldots$
  - Multiplication is a bit trickier. Suppose $A = a_0 + xA'$ and $B = b_0 + xB'$ are two generating functions we wish to multiply. We reason as follows:

    $$\begin{aligned} AB &= (a_0 + xA')B \\ &= a_0 B + xA'B \\ &= a_0(b_0 + xB') + xA'B \\ &= a_0 b_0 + x(a_0 B' + A'B) \end{aligned}$$

    That is, the first element of the product AB is the product of the first elements, $a_0 b_0$; the remainder of the coefficient stream (the part after the x) is formed by multiplying every element in B' (that is, the tail of B) by a_0, and to this adding the result of multiplying A' (the tail of A) by B.

Note that you will have to add `{-# LANGUAGE FlexibleInstances #-}` to the top of your `.hs` file in order for this instance to be allowed.

Note that there are a few methods of the `Num` class I have not told you to implement, such as `abs` and `signum`. `ghc` will complain that you haven't defined them, but don't worry about it. We won't need those methods. (To turn off these warnings you can add

```
{-# OPTIONS_GHC -fno-warn-missing-methods #-}
```

to the top of your file.)

If you have implemented the above correctly, you should be able to evaluate things at the `ghci` prompt such as

```
*Main> x^4
*Main> (1 + x)^5
*Main> (x^2 + x + 3) * (x - 5)
```

- The penultimate step is to implement an instance of the `Fractional` class for `Stream Integer`. Here the important method to define is division, `(/)`. I won't bother deriving it (though it isn't hard), but it turns out that if $A = a_0 + xA'$ and $B = b_0 + xB'$, then $A/B = Q$, where Q is defined as

$$Q = (a_0/b_0) + x((1/b_0)(A' - QB')).$$

That is, the first element of the result is a_0/b_0; the remainder is formed by computing $A' - QB'$ and dividing each of its elements by b_0.

Of course, in general, this operation might not result in a stream of `Integers`. However, we will only be using this instance in cases where it does, so just use the `div` operation where appropriate.

- Consider representing the Fibonacci numbers using a generating function,

$$F(x) = F_0 + F_1x + F_2x^2 + F_3x^3 + \dots$$

Notice that $x + xF(x) + x^2F(x) = F(x)$:

$$\begin{array}{rcccccccccc}
 & & x & & & & & & & & \\
 & & F_0x & + & F_1x^2 & + & F_2x^3 & + & F_3x^4 & + & \dots \\
 & & & & F_0x^2 & + & F_1x^3 & + & F_2x^4 & + & \dots \\
\hline
0 & + & x & + & F_2x^2 & + & F_3x^3 & + & F_4x^4 & + & \dots
\end{array}$$

Thus $x = F(x) - xF(x) - x^2F(x)$, and solving for $F(x)$ we find that

$$F(x) = \frac{x}{1 - x - x^2}.$$

Translate this into an (amazing, totally sweet) definition

```
fibs3 :: Stream Integer
```

Fibonacci numbers via matrices (extra credit)

It turns out that it is possible to compute the nth Fibonacci number with only $O(\log n)$ (arbitrary-precision) arithmetic operations. This section explains one way to do it.

Consider the 2×2 matrix $\mathbf{F}$ defined by

$$\mathbf{F} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

Notice what happens when we take successive powers of $\mathbf{F}$ (see `http://en.wikipedia.org/wiki/Matrix_multiplication` if you forget how matrix multiplication works):

$$\mathbf{F}^2 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1\cdot 1 + 1 \cdot 1 & 1 \cdot 1 + 1 \cdot 0 \\ 1 \cdot 1 + 0 \cdot 1 & 1 \cdot 1 + 0 \cdot 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\mathbf{F}^3 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix}$$

$$\mathbf{F}^4 = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix}$$

$$\mathbf{F}^5 = \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 5 & 3 \end{bmatrix}$$

Curious! At this point we might well conjecture that Fibonacci numbers are involved, namely, that

$$\mathbf{F}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

for all $n \geq 1$. Indeed, this is not hard to prove by induction on n.

The point is that exponentiation can be implemented in logarithmic time using a *binary exponentiation* algorithm. The idea is that to compute x^n, instead of iteratively doing n multiplications of x, we compute

$$x^n = \begin{cases} (x^{n/2})^2 & n \text{ even} \\ x \cdot (x^{(n-1)/2})^2 & n \text{ odd} \end{cases}$$

where $x^{n/2}$ and $x^{(n-1)/2}$ are recursively computed by the same method. Since we approximately divide n in half at every iteration, this method requires only $O(\log n)$ multiplications.

The punchline is that Haskell's exponentiation operator `(^)` *already uses* this algorithm, so we don't even have to code it ourselves!

Exercise 7 (Optional)

- Create a type `Matrix` which represents 2×2 matrices of `Integer`s.
- Make an instance of the `Num` type class for `Matrix`. In fact, you only have to implement the `(*)` method, since that is the only one we will use. (If you want to play around with matrix operations a bit more, you can implement `fromInteger`, `negate`, and `(+)` as well.)

  Don't worry about the warnings telling you that you have not implemented the other methods. (If you want to disable the warnings you can add

  ```
  {-# OPTIONS_GHC -fno-warn-missing-methods #-}
  ```

  to the top of your file.)
- We now get fast (logarithmic time) matrix exponentiation for free, since `(^)` is implemented using a binary exponentiation algorithm in terms of `(*)`. Write a function

  ```
  fib4 :: Integer -> Integer
  ```

  which computes the nth Fibonacci number by raising $\mathbf{F}$ to the nth power and projecting out F_n (you will also need a special case for zero). Try computing the one millionth or even ten millionth Fibonacci number.

  On my computer the millionth Fibonacci number takes only 0.32 seconds to compute but more than four seconds to print on the screen—after all, it has just over two hundred thousand digits.

CIS 194: Homework 7

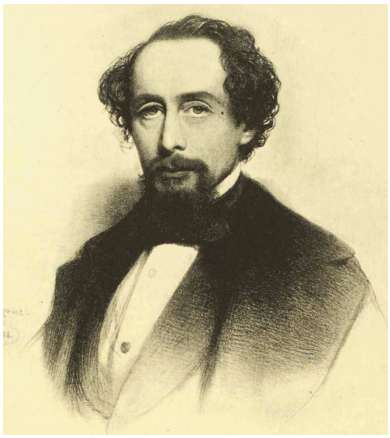
Due Monday, March 11

- Files you should submit: `JoinList.hs` and `Scrabble.hs`, containing modules of the same name.

The First Word Processor

As everyone knows, Charles Dickens was paid by the word.[1] What most people don't know, however, is the story of you, the trusty programming assistant to the great author.

[1] Actually, this is a myth.



In your capacity as Dickens's assistant, you program and operate the steam-powered Word Processing Engine which was given to him as a thoughtful birthday gift from his friend Charles Babbage.[2] To be helpful, you are developing a primitive word processor the author can use not only to facilitate his craft, but also to ease the accounting.[3] What you have done is build a word processor that keeps track of the total number of words in a document while it is being edited. This is really a great help to Mr. Dickens as the publishers use this score to determine payment, but you are not satisfied with the performance of the system and have decided to improve it.

[2] Unlike the rest of this story, the fact that Dickens and Babbage were friends is 100% true. If you don't believe it, just do a Google search for "Dickens Babbage".

[3] Of course, all of your programming is actually done by assembling steam pipes and valves and shafts and gears into machines which perform the desired computations; but as a mental shortcut you have taken to *thinking* in terms of a higher-order lazy functional programming language and compiling down to steam and gears as necessary. In this assignment we will stick to the purely mental world, but of course you should keep in mind that we could compile everything into Steam-Powered Engines if we wanted to.

Editors and Buffers

You have a working user interface for the word processor implemented in the file `Editor.hs`. The `Editor` module defines functionality for working with documents implementing the `Buffer` type class found in `Buffer.hs`. Take a look at `Buffer.hs` to see the operations that a document representation must support to work with the `Editor` module. The intention of this design is to separate the

front-end interface from the back-end representation, with the type class intermediating the two. This allows for the easy swapping of different document representation types without having to change the `Editor` module.

The editor interface is as follows:

- `v` — view the current location in the document
- `n` — move to the next line
- `p` — move to the previous line
- `l` — load a file into the editor
- `e` — edit the current line
- `q` — quit
- `?` — show this list of commands

To move to a specific line, enter the line number you wish to navigate to at the prompt. The display shows you up to two preceding and two following lines in the document surrounding the current line, which is indicated by an asterisk. The prompt itself indicates the current value of the entire document.

The first attempt at a word processor back-end was to use a single `String` to represent the entire document. You can see the `Buffer` instance for `String` in the file `StringBuffer.hs`. Performance isn't great because reporting the document score requires traversing every single character in the document every time the score is shown! Mr. Dickens demonstrates the performance issues with the following (imaginary) editor session:

```
$ runhaskell StringBufEditor.hs
33> n
 0: This buffer is for notes you don't want to save, and for
*1: evaluation of steam valve coefficients.
 2: To load a different file, type the character L followed
 3: by the name of the file.
33> l carol.txt
31559> 3640
 3638:
 3639: "An intelligent boy!" said Scrooge. "A remarkable boy!
*3640: Do you know whether they've sold the prize Turkey that
 3641: was hanging up there?--Not the little prize Turkey: the
 3642: big one?"
31559> e
Replace line 3640: Do you know whether they've sold the prize Goose that
```

```
31559> n
 3639: "An intelligent boy!" said Scrooge. "A remarkable boy!
 3640: Do you know whether they've sold the prize Goose that
*3641: was hanging up there?--Not the little prize Turkey: the
 3642: big one?"
 3643:
31559> e
Replace line 3641: was hanging up there?--Not the little one: the
31558> v
 3639: "An intelligent boy!" said Scrooge. "A remarkable boy!
 3640: Do you know whether they've sold the prize Goose that
*3641: was hanging up there?--Not the little one: the
 3642: big one?"
 3643:
31559> q
```

Sure enough, there is a small delay every time the prompt is shown.

You have chosen to address the issue by implementing a light-weight, tree-like structure, both for holding the data and caching the metadata. This data structure is referred to as a *join-list*. A data type definition for such a data structure might look like this:

```
data JoinListBasic a = Empty
                     | Single a
                     | Append (JoinListBasic a) (JoinListBasic a)
```

The intent of this data structure is to directly represent append operations as data constructors. This has the advantage of making append an `O(1)` operation: sticking two `JoinLists` together simply involves applying the `Append` data constructor. To make this more explicit, consider the function

```
jlbToList :: JoinListBasic a -> [a]
jlbToList Empty          = []
jlbToList (Single a)     = [a]
jlbToList (Append l1 l2) = jlbToList l1 ++ jlbToList l2
```

If `jl` is a `JoinList`, we can think of it as a representation of the list `jlbToList jl` where some append operations have been "deferred". For example, the join-list shown in Figure 1 corresponds to the list `['y', 'e', 'a', 'h']`.

Such a structure makes sense for text editing applications as it provides a way of breaking the document data into pieces that can be processed individually, rather than having to always traverse the entire document. This structure is also what you will be annotating with the metadata you want to track.

Monoidally Annotated Join-Lists

The `JoinList` definition to use for this assignment is

```
data JoinList m a = Empty
                  | Single m a
                  | Append m (JoinList m a) (JoinList m a)
  deriving (Eq, Show)
```

You should copy this definition into a Haskell module named `JoinList.hs`.

The `m` parameter will be used to track monoidal annotations to the structure. The idea is that the annotation at the root of a `JoinList` will always be equal to the combination of all the annotations on the `Single` nodes (according to whatever notion of "combining" is defined for the monoid in question). `Empty` nodes do not explicitly store an annotation, but we consider them to have an annotation of `mempty` (that is, the identity element for the given monoid).

For example,

```
Append (Product 210)
  (Append (Product 30)
    (Single (Product 5) 'y')
    (Append (Product 6)
      (Single (Product 2) 'e')
      (Single (Product 3) 'a')))
  (Single (Product 7) 'h')
```

is a join-list storing four values: the character `'y'` with annotation `5`, the character `'e'` with annotation `2`, `'a'` with annotation `3`, and `'h'` with annotation 7. (See Figure 1 for a graphical representation of the same structure.) Since the multiplicative monoid is being used, each `Append` node stores the product of all the annotations below it. The point of doing this is that all the subcomputations needed to compute the product of all the annotations in the join-list are cached. If we now change one of the annotations, say, the annotation on `'y'`, we need only recompute the annotations on nodes above it in the tree. In particular, in this example we don't need to descend into the subtree containing `'e'` and `'a'`, since we have cached the fact that their product is `6`. This means that for balanced join-lists, it takes only $O(\log n)$ time to rebuild the annotations after making an edit.

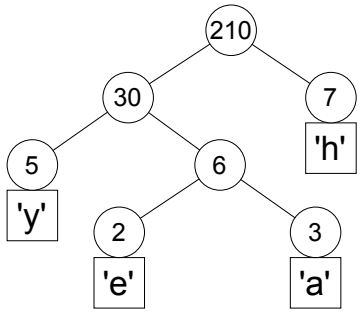


Figure 1: A sample join-list annotated with products

Exercise 1 We first consider how to write some simple operations on these `JoinLists`. Perhaps the most important operation we will consider is how to append two `JoinLists`. Previously, we said that the point of `JoinLists` is to represent append operations as data, but what about the annotations? Write an append function for `JoinLists`

that yields a new `JoinList` whose monoidal annotation is derived from those of the two arguments.

```
(+++) :: Monoid m => JoinList m a -> JoinList m a -> JoinList m a
```

You may find it helpful to implement a helper function

```
tag :: Monoid m => JoinList m a -> m
```

which gets the annotation at the root of a `JoinList`.

Exercise 2 The first annotation to try out is one for fast indexing into a `JoinList`. The idea is to cache the *size* (number of data elements) of each subtree. This can then be used at each step to determine if the desired index is in the left or the right branch.

We have provided the `Sized` module that defines the `Size` type, which is simply a `newtype` wrapper around an `Int`. In order to make `Size`s more accessible, we have also defined the `Sized` type class which provides a method for obtaining a `Size` from a value.

Use the `Sized` type class to write the following functions.

1. Implement the function

   ```
   indexJ :: (Sized b, Monoid b) =>
             Int -> JoinList b a -> Maybe a
   ```

   `indexJ` finds the `JoinList` element at the specified index. If the index is out of bounds, the function returns `Nothing`. By an *index* in a `JoinList` we mean the index in the list that it represents. That is, consider a safe list indexing function

   ```
   (!!?) :: [a] -> Int -> Maybe a
   []     !!? _         = Nothing
   _      !!? i | i < 0 = Nothing
   (x:xs) !!? 0         = Just x
   (x:xs) !!? i         = xs !!? (i-1)
   ```

   which returns `Just` the *i*th element in a list (starting at zero) if such an element exists, or `Nothing` otherwise. We also consider an updated function for converting join-lists into lists, just like `jlbToList` but ignoring the monoidal annotations:

   ```
   jlToList :: JoinList m a -> [a]
   jlToList Empty            = []
   jlToList (Single _ a)     = [a]
   jlToList (Append _ l1 l2) = jlToList l1 ++ jlToList l2
   ```

Note: you do not have to include `(!!?)` and `jlToList` in your assignment; they are just to help explain how `indexJ` ought to behave. However, you may certainly use them to help test your implementations if you wish.

We can now specify the desired behavior of `indexJ`. For any index `i` and join-list `jl`, it should be the case that

```
(indexJ i jl) == (jlToList jl !!? i)
```

That is, calling `indexJ` on a join-list is the same as first converting the join-list to a list and then indexing into the list. The point, of course, is that `indexJ` can be more efficient ($O(\log n)$ versus $O(n)$, assuming a balanced join-list), because it gets to use the size annotations to throw away whole parts of the tree at once, whereas the list indexing operation has to walk over every element.

2. Implement the function

```
dropJ :: (Sized b, Monoid b) =>
         Int -> JoinList b a -> JoinList b a
```

The `dropJ` function drops the first `n` elements from a `JoinList`. This is analogous to the standard `drop` function on lists. Formally, `dropJ` should behave in such a way that

```
jlToList (dropJ n jl) == drop n (jlToList jl).
```

3. Finally, implement the function

```
takeJ :: (Sized b, Monoid b) =>
         Int -> JoinList b a -> JoinList b a
```

The `takeJ` function returns the first `n` elements of a `JoinList`, dropping all other elements. Again, this function works similarly to the standard library `take` function; that is, it should be the case that

```
jlToList (takeJ n jl) == take n (jlToList jl).
```

Ensure that your function definitions use the `size` function from the `Sized` type class to make smart decisions about how to descend into the `JoinList` tree.

Exercise 3 Mr. Dickens's publishing company has changed their minds. Instead of paying him by the word, they have decided to pay him according to the scoring metric used by the immensely popular game of *Scrabble*™. You must therefore update your editor implementation to count Scrabble scores rather than counting words.

Scrabble™, of course, was invented in 1842, by Dr. Wilson P. Scrabble™.

Hence, the second annotation you decide to implement is one to cache the *Scrabble*™ score for every line in a buffer. Create a `Scrabble` module that defines a `Score` type, a `Monoid` instance for `Score`, and the following functions:

```
score :: Char -> Score
scoreString :: String -> Score
```

The `score` function should implement the tile scoring values as shown at `http://www.thepixiepit.co.uk/scrabble/rules.html`; any characters not mentioned (punctuation, spaces, *etc.*) should be given zero points.

To test that you have everything working, add the line `import Scrabble` to the import section of your `JoinList` module, and write the following function to test out `JoinList`s annotated with scores:

```
scoreLine :: String -> JoinList Score String
```

Example:

```
*JoinList> scoreLine "yay " +++ scoreLine "haskell!"
Append (Score 23)
       (Single (Score 9) "yay ")
       (Single (Score 14) "haskell!")
```

Exercise 4 Finally, combine these two kinds of annotations. A pair of monoids is itself a monoid:

```
instance (Monoid a, Monoid b) => Monoid (a,b) where
  mempty = (mempty, mempty)
  mappend (a1,b1) (a2,b2) = (mappend a1 a2, mappend b1 b2)
```

(This instance is defined in `Data.Monoid`.) This means that join-lists can track more than one type of annotation at once, in parallel, simply by using a pair type.

Since we want to track both the size and score of a buffer, you should provide a `Buffer` instance for the type

```
JoinList (Score, Size) String.
```

Note that you will have to enable the `FlexibleInstances` and `TypeSynonymInstances` extensions.

Due to the use of the `Sized` type class, this type will continue to work with your functions such as `indexJ`.

Finally, make a `main` function to run the editor interface using your join-list backend in place of the slow `String` backend (see `StringBufEditor.hs` for an example of how to do this). You should create an initial buffer of type `JoinList (Score, Size) String` and pass it as an argument to `runEditor editor`. Verify that the editor demonstration described in the section "Editors and Buffers" does not exhibit delays when showing the prompt.

CIS 194: Homework 8

Due Monday, March 18

- Files you should submit: `Party.hs`, containing a module of the same name (**make sure your file actually has `module Party where` at the top!**).

Planning the office party

As the most junior employee at Calculators R Us, Inc., you are tasked with organizing the office Spring Break party. As with all party organizers, your goal is, of course, to maximize the *amount of fun*[1] which is had at the party. Since some people enjoy parties more than others, you have estimated the amount of fun which will be had by each employee. So simply summing together all these values should indicate the amount of fun which will be had at the party in total, right?

[1] As measured, of course, in Standard Transnational Fun Units, or STFUs.

. . . well, there's one small problem. It is a well-known fact that anyone whose immediate boss is *also* at the party will not have any fun at all. So if *all* the company employees are at the party, only the CEO will have fun, and everyone else will stand around laughing nervously and trying to look natural while looking for their boss out of the corner of their eyes.

Your job, then, is to figure out *who to invite* to the party in order to maximize the total amount of fun.

Preliminaries

We have provided you with the file `Employee.hs`, which contains the following definitions:

```
-- Employee names are represented by Strings.
type Name = String

-- The amount of fun an employee would have at the party,
-- represented by an Integer number of STFUs
type Fun  = Integer

-- An Employee consists of a name and a fun score.
data Employee = Emp { empName :: Name, empFun :: Fun }
  deriving (Show, Read, Eq)
```

Note that the definition of `Employee` uses *record syntax*, which you can read more about in the Week 8 lecture notes (it was not covered in lecture but is provided in the lecture notes as an additional resource).

It also defines `testCompany :: Tree Employee`, a small company hierarchy which you can use for testing your code (although your actual company hierarchy is much larger).

Finally, `Employee.hs` defines a type to represent guest lists. The obvious possibility to represent a guest list would be `[Employee]`. However, we will frequently want to know the total amount of fun had by a particular guest list, and it would be inefficient to recompute it every time by adding up the fun scores for all the employees in the list. Instead, a `GuestList` contains both a list of `Employees` and a `Fun` score. Values of type `GuestList` should always satisfy the invariant that the sum of all the `Fun` scores in the list of `Employees` should be equal to the one, "cached" `Fun` score.

Exercise 1

Now define the following tools for working with `GuestLists`:

1. A function

   ```
   glCons :: Employee -> GuestList -> GuestList
   ```

   which adds an `Employee` to the `GuestList` (updating the cached `Fun` score appropriately). Of course, in general this is impossible: the updated fun score should depend on whether the `Employee` being added is already in the list, or if any of their direct subordinates are in the list, and so on. For our purposes, though, you may assume that none of these special cases will hold: that is, `glCons` should simply add the new `Employee` and add their fun score without doing any kind of checks.

2. A `Monoid` instance for `GuestList`.[2] (How is the `Monoid` instance supposed to work, you ask? You figure it out!)
3. A function `moreFun :: GuestList -> GuestList -> GuestList` which takes two `GuestLists` and returns whichever one of them is more fun, *i.e.* has the higher fun score. (If the scores are equal it does not matter which is returned.)

[2] Note that this requires creating an "orphan instance" (a type class instance `instance C T` which is defined in a module which is distinct from both the modules where `C` and `T` are defined), which GHC will warn you about. You can ignore the warning, or add `{-# OPTIONS_GHC -fno-warn-orphans #-}` to the top of your file.

Exercise 2

The `Data.Tree` module from the standard Haskell libraries defines the type of "rose trees", where each node stores a data element and has any number of children (*i.e.* a *list* of subtrees):

```
data Tree a = Node {
        rootLabel :: a,         -- label value
        subForest :: [Tree a]   -- zero or more child trees
    }
```

Strangely, `Data.Tree` does *not* define a fold for this type! Rectify the situation by implementing

```
treeFold :: ... -> Tree a -> b
```

(See if you can figure out what type(s) should replace the dots in the type of `treeFold`. If you are stuck, look back at the lecture notes from Week 7, or infer the proper type(s) from the remainder of this assignment.)

The algorithm

Now let's actually derive an algorithm to solve this problem. Clearly there must be some sort of recursion involved—in fact, it seems that we should be able to do it with a fold. This makes sense though—starting from the bottom of the tree and working our way up, we compute the best guest list for each subtree and somehow combine these to decide on the guest list for the next level up, and so on. So we need to write a combining function

I mean, why else would we have had you do Exercise 2?

```
combineGLs :: Employee -> [GuestList] -> GuestList
```

which takes an employee (the boss of some division) and the optimal guest list for each subdivision under him, and somehow combines this information to compute the best guest list for the entire division.

However, this obvious first attempt fails! The problem is that we don't get enough information from the recursive calls. If the best guest list for some subtree involves inviting that subtree's boss, then we are stuck, since we might want to consider inviting the boss of the entire tree—in which case we don't want to invite any of the subtree bosses (since they wouldn't have any fun anyway). But we might be able to do better than just taking the best possible guest list for each subtree and then excluding their bosses.

The solution is to generalize the recursion to compute *more* information, in such a way that we can actually make the recursive step. In particular, instead of just computing the best guest list for a given tree, we will compute *two* guest lists:

1. the best possible guest list we can create *if we invite the boss* (that is, the `Employee` at the root of the tree); and
2. the best possible guest list we can create if we *don't* invite the boss.

It turns out that this gives us enough information at each step to compute the optimal two guest lists for the next level up.

Exercise 3

Write a function

```
nextLevel :: Employee -> [(GuestList, GuestList)]
             -> (GuestList, GuestList)
```

which takes two arguments. The first is the "boss" of the current subtree (let's call him Bob). The second argument is a list of the results for each subtree under Bob. Each result is a pair of `GuestLists`: the first `GuestList` in the pair is the best possible guest list *with* the boss of that subtree; the second is the best possible guest list *without* the boss of that subtree. `nextLevel` should then compute the overall best guest list that includes Bob, and the overall best guest list that doesn't include Bob.

Exercise 4

Finally, put all of this together to define

```
maxFun :: Tree Employee -> GuestList
```

which takes a company hierarchy as input and outputs a fun-maximizing guest list. You can test your function on `testCompany`, provided in `Employee.hs`.

The whole company

Of course, the *actual* tree of employees in your company is much larger! We have provided you with a file, `company.txt`, containing the entire hierarchy for your company. The contents of this file were created by calling the `show` function on a `Tree Employee`,[3] so you can convert it back into a `Tree Employee` using the `read` function.

[3] We don't recommend actually looking at the contents of `company.txt`, assuming that you value your sanity.

Exercise 5

Implement `main :: IO ()` so that it reads your company's hierarchy from the file `company.txt`, and then prints out a formatted guest list, sorted by first name, which looks like

```
Total fun: 23924
Adam Debergues
Adeline Anselme
...
```

(Note: the above is just an example of the *format*; it is *not* the correct output!) You will probably find the `readFile` and `putStrLn` functions useful.

As much as possible, try to separate out the "pure" computation from the `IO` computation. In other words, your `main` function should actually be fairly short, calling out to helper functions (whose types do not involve `IO`) to do most of the work. If you find `IO` "infecting" all your function types, you are Doing It Wrong.

No homework this week. If you have some extra time, get a head start on your final project!

CIS 194: Homework 10

Due Monday, April 1

- Files you should submit: `AParser.hs`. You should take the versions that we have provided and add your solutions to them.

Introduction

A *parser* is an algorithm which takes unstructured data as input (often a `String`) and produces structured data as output. For example, when you load a Haskell file into `ghci`, the first thing it does is *parse* your file in order to turn it from a long `String` into an *abstract syntax tree* representing your code in a more structured form.

Concretely, we will represent a parser for a value of type `a` as a function which takes a `String` represnting the input to be parsed, and succeeds or fails; if it succeeds, it returns the parsed value along with whatever part of the input it did not use.

```
newtype Parser a
  = Parser { runParser :: String -> Maybe (a, String) }
```

For example, `satisfy` takes a `Char` predicate and constructs a parser which succeeds only if it sees a `Char` that satisfies the predicate (which it then returns). If it encounters a `Char` that does not satisfy the predicate (or an empty input), it fails.

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = Parser f
  where
    f [] = Nothing    -- fail on the empty input
    f (x:xs)          -- check if x satisfies the predicate
                      -- if so, return x along with the remainder
                      -- of the input (that is, xs)
        | p x       = Just (x, xs)
        | otherwise = Nothing  -- otherwise, fail
```

Using satisfy, we can also define the parser `char`, which expects to see exactly a given character and fails otherwise.

```
char :: Char -> Parser Char
char c = satisfy (== c)
```

For example:

```
*Parser> runParser (satisfy isUpper) "ABC"
Just ('A',"BC")
```

```
*Parser> runParser (satisfy isUpper) "abc"
Nothing
*Parser> runParser (char 'x') "xyz"
Just ('x',"yz")
```

For convenience, we've also provided you with a parser for positive integers:

```
posInt :: Parser Integer
posInt = Parser f
  where
    f xs
      | null ns   = Nothing
      | otherwise = Just (read ns, rest)
      where (ns, rest) = span isDigit xs
```

Tools for building parsers

However, implementing parsers explicitly like this is tedious and error-prone for anything beyond the most basic primitive parsers. The real power of this approach comes from the ability to create complex parsers by *combining* simpler ones. And this power of combining will be given to us by... you guessed it, `Applicative`.

Exercise 1

First, you'll need to implement a `Functor` instance for `Parser`. *Hint*: You may find it useful to implement a function

```
first :: (a -> b) -> (a,c) -> (b,c)
```

Exercise 2

Now implement an `Applicative` instance for `Parser`:

- `pure a` represents the parser which consumes no input and successfully returns a result of `a`.
- `p1 <*> p2` represents the parser which first runs `p1` (which will consume some input and produce a function), then passes the *remaining* input to `p2` (which consumes more input and produces some value), then returns the result of applying the function to the value. However, if either `p1` or `p2` fails then the whole thing should also fail (put another way, `p1 <*> p2` only succeeds if both `p1` and `p2` succeed).

So what is this good for? Recalling the `Employee` example from class,

```
type Name = String
data Employee = Emp { name :: Name, phone :: String }
```

we could now use the `Applicative` instance for `Parser` to make an employee parser from name and phone parsers. That is, if

```
parseName  :: Parser Name
parsePhone :: Parser String
```

then

```
Emp <$> parseName <*> parsePhone :: Parser Employee
```

is a parser which first reads a name from the input, then a phone number, and returns them combined into an `Employee` record. Of course, this assumes that the name and phone number are right next to each other in the input, with no intervening separators. We'll see later how to make parsers that can throw away extra stuff that doesn't directly correspond to information you want to parse.

Exercise 3

We can also test your `Applicative` instance using other simple applications of functions to multiple parsers. You should implement each of the following exercises using the `Applicative` interface to put together simpler parsers into more complex ones. Do *not* implement them using the low-level definition of a `Parser`! In other words, pretend that you do not have access to the `Parser` constructor or even know how the `Parser` type is defined.

- Create a parser

  ```
  abParser :: Parser (Char, Char)
  ```

  which expects to see the characters `'a'` and `'b'` and returns them as a pair. That is,

  ```
  *AParser> runParser abParser "abcdef"
  Just (('a','b'),"cdef")
  *AParser> runParser abParser "aebcdf"
  Nothing
  ```

- Now create a parser

  ```
  abParser_ :: Parser ()
  ```

  which acts in the same way as `abParser` but returns `()` instead of the characters `'a'` and `'b'`.

```
*AParser> runParser abParser_ "abcdef"
Just ((),"cdef")
*AParser> runParser abParser_ "aebcdf"
Nothing
```

- Create a parser `intPair` which reads two integer values separated by a space and returns the integer values in a list. You should use the provided `posInt` to parse the integer values.

```
*Parser> runParser intPair "12 34"
Just ([12,34],"")
```

Exercise 4

`Applicative` by itself can be used to make parsers for simple, fixed formats. But for any format involving *choice* (*e.g.* "... after the colon there can be a number **or** a word **or** parentheses...") `Applicative` is not quite enough. To handle choice we turn to the `Alternative` class, defined (essentially) as follows:

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

`(<|>)` is intended to represent *choice*: that is, `f1 <|> f2` represents a choice between `f1` and `f2`. `empty` should be the identity element for `(<|>)`, and often represents *failure*.

Write an `Alternative` instance for `Parser`:

- `empty` represents the parser which always fails.
- `p1 <|> p2` represents the parser which first tries running `p1`. If `p1` succeeds then `p2` is ignored and the result of `p1` is returned. Otherwise, if `p1` fails, then `p2` is tried instead.

Hint: there is already an `Alternative` instance for `Maybe` which you may find useful.

Exercise 5

Implement a parser

```
intOrUppercase :: Parser ()
```

which parses either an integer value or an uppercase character, and fails otherwise.

```
*Parser> runParser intOrUppercase "342abcd"
Just ((), "abcd")
*Parser> runParser intOrUppercase "XYZ"
Just ((), "YZ")
*Parser> runParser intOrUppercase "foo"
Nothing
```

Next week, we will use your parsing framework to build a more sophisticated parser for a small programming language!

CIS 194: Homework 11

Due Monday, April 8

- Files you should submit: `SExpr.hs`. You should take the version that we have provided and add your solutions. Note that we have also provided `AParser.hs`—you are welcome to use your own `AParser.hs` from last week's homework or ours, whichever you prefer.

Parsing S-expressions

In `AParser.hs` from last week's homework, we now have the following:

- the definition of a basic `Parser` type
- a few primitive parsers such as `satisfy`, `char`, and `posInt`
- `Functor`, `Applicative`, and `Alternative` instances for `Parser`

So, what can we do with this? It may not seem like we have much to go on, but it turns out we can actually do quite a lot.

Remember, for this week's homework you should only need to write code on top of the interface provided by the `Functor`, `Applicative`, and `Alternative` instances. In particular, you should not write any code that depends on the details of the `Parser` implementation. (To help with this, the version of `AParser.hs` we provided this week does not even export the `Parser` constructor, so it is literally impossible to depend on the details!)

Exercise 1

First, let's see how to take a parser for (say) widgets and turn it into a parser for *lists* of widgets. In particular, there are two functions you should implement: `zeroOrMore` takes a parser as input and runs it consecutively as many times as possible (which could be none, if it fails right away), returning a list of the results. `zeroOrMore` always succeeds. `oneOrMore` is similar, except that it requires the input parser to succeed at least once. If the input parser fails right away then `oneOrMore` also fails.

For example, below we use `zeroOrMore` and `oneOrMore` to parse a sequence of uppercase characters. The longest possible sequence of uppercase characters is returned as a list. In this case, `zeroOrMore` and `oneOrMore` behave identically:

```
*AParser> runParser (zeroOrMore (satisfy isUpper)) "ABCdEfgH"
Just ("ABC","dEfgH")
*AParser> runParser (oneOrMore (satisfy isUpper)) "ABCdEfgH"
Just ("ABC","dEfgH")
```

The difference between them can be seen when there is not an uppercase character at the beginning of the input. `zeroOrMore` succeeds and returns the empty list without consuming any input; `oneOrMore` fails.

```
*AParser> runParser (zeroOrMore (satisfy isUpper)) "abcdeFGh"
Just ("","abcdeFGh")
*AParser> runParser (oneOrMore (satisfy isUpper)) "abcdeFGh"
Nothing
```

Implement `zeroOrMore` and `oneOrMore` with the following type signatures:

```
zeroOrMore :: Parser a -> Parser [a]
oneOrMore  :: Parser a -> Parser [a]
```

Hint: To parse one or more occurrences of `p`, run `p` once and then parse zero or more occurrences of `p`. To parse zero or more occurrences of `p`, try parsing one or more; if that fails, return the empty list.

Exercise 2

There are a few more utility parsers needed before we can accomplish the final parsing task. First, `spaces` should parse a consecutive list of zero or more whitespace characters (use the `isSpace` function from the standard `Data.Char` module).

```
spaces :: Parser String
```

Next, `ident` should parse an *identifier*, which for our purposes will be an alphabetic character (use `isAlpha`) followed by zero or more alphanumeric characters (use `isAlphaNum`). In other words, an identifier can be any nonempty sequence of letters and digits, except that it may not start with a digit.

```
ident :: Parser String
```

For example:

```
*AParser> runParser ident "foobar baz"
Just ("foobar"," baz")
*AParser> runParser ident "foo33fA"
Just ("foo33fA","")
*AParser> runParser ident "2bad"
Nothing
*AParser> runParser ident ""
Nothing
```

Exercise 3

S-expressions are a simple syntactic format for tree-structured data, originally developed as a syntax for Lisp programs. We'll close out our demonstration of parser combinators by writing a simple S-expression parser.

An *identifier* is represented as just a `String`; the format for valid identifiers is represented by the `ident` parser you wrote in the previous exercise.

```
type Ident = String
```

An "atom" is either an integer value (which can be parsed with `posInt`) or an identifier.

```
data Atom = N Integer | I Ident
  deriving Show
```

Finally, an S-expression is either an atom, or a list of S-expressions.[1]

[1] Actually, this is slightly different than the usual definition of S-expressions in Lisp, which also includes binary "cons" cells; but it's good enough for our purposes.

```
data SExpr = A Atom
           | Comb [SExpr]
  deriving Show
```

Textually, S-expressions can optionally begin and end with any number of spaces; after *throwing away leading and trailing spaces* they consist of either an atom, *or* an open parenthesis followed by one or more S-expressions followed by a close parenthesis.

$$\begin{aligned} atom &::= int \\ &\mid ident \end{aligned}$$

$$\begin{aligned} S &::= atom \\ &\mid (S^*) \end{aligned}$$

For example, the following are all valid S-expressions:

```
5
foo3
(bar (foo) 3 5 874)
(((lambda x (lambda y (plus x y))) 3) 5)
(   lots  of   (  spaces   in  )  this ( one ) )
```

We have provided Haskell data types representing S-expressions in `SExpr.hs`. Write a parser for S-expressions, that is, something of type

```
parseSExpr :: Parser SExpr
```

Hints: To parse something but ignore its output, you can use the `(*>)` and `(<*)` operators, which have the types

```
(*>) :: Applicative f => f a -> f b -> f b
(<*) :: Applicative f => f a -> f b -> f a
```

`p1 *> p2` runs `p1` and `p2` in sequence, but ignores the result of `p1` and just returns the result of `p2`. `p1 <* p2` also runs `p1` and `p2` in sequence, but returns the result of `p1` (ignoring `p2`'s result) instead.

For example:

```
*AParser> runParser (spaces *> posInt) "     345"
Just (345,"")
```

CIS 194: Homework 11

Due Thursday, April 5

- Files you should submit: `Risk.hs`. You should take the version we have provided and add your solutions to it.

Risk

The game of *Risk* involves two or more players, each vying to "conquer the world" by moving armies around a board representing the world and using them to conquer territories. The central mechanic of the game is that of one army attacking another, with dice rolls used to determine the outcome of each battle.

The rules of the game make it complicated to determine the likelihood of possible outcomes. In this assignment, you will write a *simulator* which could be used by Risk players to estimate the probabilities of different outcomes before deciding on a course of action.

The `Rand StdGen` *monad*

Since battles in Risk are determined by rolling dice, your simulator will need some way to access a source of randomness. Many languages include standard functions for getting the output of a pseudorandom number generator. For example, in Java one can write

```
Random randGen = new Random();
int dieRoll = 1 + randGen.nextInt(6);
```

to get a random value between 1 and 6 into the variable `dieRoll`. It may seem like we can't do this in Haskell, because the output of `randGen.nextInt(6)` may be different each time it is called—and Haskell functions must always yield the same outputs for the same inputs.

However, if we think about what's going on a bit more carefully, we can see how to successfully model this in Haskell. The Java code first creates a `Random` object called `randGen`. This represents a *pseudorandom number generator*, which remembers a bit of state (a few numbers), and every time something like `nextInt` is called, it uses the state to (deterministically) generate an `Int` and then updates the state according to some (deterministic) algorithm. So the numbers which are generated are not truly random; they are in fact completely deterministic, but computed using an algorithm which generates random-seeming output. As long as we initialize (*seed*) the generator with some truly random data, this is often good enough for purposes such as simulations.

In Haskell we can cerainly have pseudorandom number generator objects. Instead of having methods which mutate them, however, we will have functions that take a generator and return the next pseudorandom value *along with a new generator*. That is, the type signature for `nextInt` would be something like

```
nextInt :: Generator -> (Int, Generator)
```

However, using `nextInt` would quickly get annoying: we have to manually pass around generators everywhere. For example, consider some code to generate three random `Int`s:

```
threeInts :: Generator -> ((Int, Int, Int), Generator)
threeInts g = ((i1, i2, i3), g''')
  where (i1, g')   = nextInt g
        (i2, g'')  = nextInt g'
        (i3, g''') = nextInt g''
```

Ugh! Fortunately, there is a much better way. The `MonadRandom` package[1] defines a *monad* which encapsulates this generator-passing behavior. Using it, `threeInts` can be rewritten as

[1] `http://hackage.haskell.org/package/MonadRandom`

```
threeInts :: Rand StdGen (Int, Int, Int)
threeInts =
  getRandom >>= \i1 ->
  getRandom >>= \i2 ->
  getRandom >>= \i3 ->
  return (i1,i2,i3)
```

The type signature says that `threeInts` is a computation in the `Rand StdGen` monad which returns a triple of `Int`s. `Rand StdGen` computations implicitly pass along a pseudorandom generator of type `StdGen` (which is defined in the standard Haskell library `System.Random`).

Exercise 1

Type `cabal install MonadRandom` at a command prompt (*not* the `ghci` prompt) to download and install the `MonadRandom` package from Hackage. Then visit the documentation (`http://hackage.haskell.org/package/MonadRandom`). Take a look at the `Control.Monad.Random` module, which defines various ways to "run" a `Rand` computation; in particular you will eventually (at the very end of the assignment) need to use the `evalRandIO` function. Take a look also at the `Control.Monad.Random.Class` module, which defines a `MonadRandom` class containing methods you can use to access the random generator in a `Rand` computation. For example, this is where the `getRandom` function (used above in the `threeInts` example) comes from. However, you probably won't need to use these methods directly in this assignment.

In `Risk.hs` we have provided a type

```
newtype DieValue = DV { unDV :: Int }
```

for representing the result of rolling a six-sided die. We have also provided an instance of `Random` for `DieValue` (allowing it to be used with `MonadRandom`), and a definition

```
die :: Rand StdGen DieValue
die = getRandom
```

which represents the random outcome of rolling a fair six-sided die.

The Rules

The rules of attacking in Risk are as follows.

- There is an attacking army (containing some number of units) and a defending army (containing some number of units).
- The attacking player may attack with up to three units at a time. However, they must always leave at least one unit behind. That is, if they only have three total units in their army they may only attack with two, and so on.
- The defending player may defend with up to two units (or only one if that is all they have).
- To determine the outcome of a single battle, the attacking and defending players each roll one six-sided die for every unit they have attacking or defending. So the attacking player rolls one, two, or three dice, and the defending player rolls one or two dice.
- The attacking player sorts their dice rolls in descending order. The defending player does the same.

- The dice are then matched up in pairs, starting with the highest roll of each player, then the second-highest.
- For each pair, if the attacking player's roll is higher, then one of the defending player's units die. If there is a tie, or the defending player's roll is higher, then one of the attacking player's units die.

For example, suppose player A has 3 units and player B has 5. A can attack with only 2 units, and B can defend with 2 units. So A rolls 2 dice, and B does the same. Suppose A rolls a 3 and a 5, and B rolls a 4 and a 3. After sorting and pairing up the rolls, we have

A	B
5	4
3	3

A wins the first matchup (5 *vs.* 4), so one of B's units dies. The second matchup is won by B, however (since B wins ties), so one of A's units dies. The end result is that now A has 2 units and B has 4. If A wanted to attack again they would only be able to attack with 1 unit (whereas B would still get to defend with 2—clearly this would give B an advantage because the *higher* of B's two dice rolls will get matched with A's single roll.)

Exercise 2

Given the definitions

```
type Army = Int

data Battlefield = Battlefield { attackers :: Army, defenders :: Army }
```

(which are also included in `Risk.hs`), write a function with the type

```
battle :: Battlefield -> Rand StdGen Battlefield
```

which simulates a single battle (as explained above) between two opposing armies. That is, it should simulate randomly rolling the appropriate number of dice, interpreting the results, and updating the two armies to reflect casualties. You may assume that each player will attack or defend with the maximum number of units they are allowed.

Exercise 3

Of course, usually an attacker does not stop after just a single battle, but attacks repeatedly in an attempt to destroy the entire defending army (and thus take over its territory).

Now implement a function

```
invade :: Battlefield -> Rand StdGen Battlefield
```

which simulates an entire invasion attempt, that is, repeated calls to `battle` until there are no defenders remaining, or fewer than two attackers.

Exercise 4

Finally, implement a function

```
successProb :: Battlefield -> Rand StdGen Double
```

which runs `invade` 1000 times, and uses the results to compute a `Double` between 0 and 1 representing the estimated probability that the attacking army will completely destroy the defending army. For example, if the defending army is destroyed in 300 of the 1000 simulations (but the attacking army is reduced to 1 unit in the other 700), `successProb` should return `0.3`.

Exercise 5 (Optional)

Write a function

```
exactSuccessProb :: Battlefield -> Double
```

which computes the exact probability of success based on principles of probability, without running any simulations. (This won't give you any particular practice with Haskell; it's just a potentially interesting challenge in probability theory.)