

Haskell/Print version

From Wikibooks, the open-content textbooks collection

Table Of Contents

Haskell Basics

- Getting set up
- Variables and functions
- Lists and tuples
- Next steps
- Type basics
- Simple input and output
- Type declarations

Elementary Haskell

- Recursion
- Pattern matching
- More about lists
- Control structures
- List processing
- More on functions
- Higher order functions ▣

Intermediate Haskell

- Modules
- Indentation
- More on datatypes
- Class declarations
- Classes and types
- Keeping track of State ▣

Monads

- Understanding monads ▣
- Advanced monads
- Additive monads (MonadPlus)
- Monadic parser combinators
- Monad transformers
- Value recursion (MonadFix)

Practical monads

Advanced Haskell

Arrows ▣

Understanding arrows

Continuation passing style (CPS) ▣

Mutable objects ▣

Zippers ▣

Applicative Functors ▣

Concurrency ▣

Fun with Types

Existentially quantified types

Polymorphism ▣

Advanced type classes

Phantom types ▣

Generalised algebraic data-types (GADT)

Datatype algebra ▣

Wider Theory

Denotational semantics ▣

Equational reasoning

Program derivation

Category theory ▣

The Curry-Howard isomorphism

Haskell Performance

Graph reduction ▣

Laziness ▣

Strictness ▣

Algorithm complexity

Parallelism

Choosing data structures

Libraries Reference

The Hierarchical Libraries

Lists:Arrays:Maybe:Maps

IO:Random Numbers

General Practices

Building a standalone application

Debugging
Testing
Packaging your software (Cabal)
Using the Foreign Function Interface (FFI)

Specialised Tasks

Graphical user interfaces (GUI) ▣
Databases ▣
Web programming ▣
Working with XML ▣
Using Regular Expressions ▣

Haskell Basics

Getting set up

This chapter will explore how to install the programs you'll need to start coding in Haskell.

Installing Haskell

First of all, you need a Haskell compiler. A compiler is a program that takes your code and spits out an executable which you can run on your machine.

There are several Haskell compilers available freely, the most popular and fully featured of them all being the Glasgow Haskell Compiler or GHC for short. The GHC was originally written at the University of Glasgow. GHC is available for most platforms:

- For MS Windows, see the GHC download page (<http://haskell.org/ghc/download.html>) for details
- For MacOS X, Linux or other platforms, you are most likely better off using one of the pre-packaged versions (http://haskell.org/ghc/distribution_packages.html) for your distribution or operating system.

Note

A quick note to those people who prefer to compile from source: This might be a bad idea with GHC, especially if it's the first time you install it. GHC is itself mostly written in Haskell, so trying to bootstrap it by hand from source is very tricky. Besides, the build takes a very long time and consumes a lot of disk space. If you are sure that you want to build GHC from the source, see [Building and Porting GHC at the GHC homepage](http://hackage.haskell.org/trac/ghc/wiki/Building) (<http://hackage.haskell.org/trac/ghc/wiki/Building>) .

Getting interactive


```
Prelude> pi
3.141592653589793
```

Don't worry about all those missing digits; they're just skipped when displaying the value. All the digits will be used in any future calculations.

Having variables takes some of the tedium out of things. What is the area of a circle having a radius of 5 cm? How about a radius of 25cm?

```
Prelude> pi * 5^2
78.53981633974483
Prelude> pi * 25^2
1963.4954084936207
```

Note

What we call "variables" in this book are often referred to as "symbols" in other introductions to functional programming. This is because other languages, namely the more popular imperative languages have a very different use for variables: keeping track of state. Variables in Haskell do no such thing; they store a value and an immutable one at that.

Types

Following the previous example, you might be tempted to try storing a value for that radius. Let's see what happens:

```
Prelude> let r = 25
Prelude> 2 * pi * r
<interactive>:1:9:
  Couldn't match `Double' against `Integer'
    Expected type: Double
    Inferred type: Integer
  In the second argument of `(*)', namely `r'
  In the definition of `it': it = (2 * pi) * r
```

Whoops! You've just run into a programming concept known as **types**. Types are a feature of many programming languages which are designed to catch some of your programming errors early on so that you find out about them before it's too late. We'll discuss types in more detail later on in the Type basics chapter, but for now it's useful to think in terms of plugs and connectors. For example, many of the plugs on the back of your computer are designed to have different shapes and sizes for a purpose. This is partly so that you don't inadvertently plug the wrong bits of your computer together and blow something up. Types serve a similar purpose, but in this particular example, well, types aren't so helpful.

The tricky bit here is that numbers like 25 can either be interpreted as being `Double` or `Integer` (among other types)... but for lack of other information, Haskell has "guessed" that its type must be `Integer` (which cannot be multiplied with a `Double`). To work around this, we simply insist that it is to be treated as a `Double`

```
Prelude> let r = 25 :: Double
Prelude> 2 * pi * r
157.07963267948966
```

Note that Haskell only has this "guessing" behaviour in contexts where it does not have enough information to infer the type of something. As we will see below, most of the time, the surrounding context gives us all of the information that is needed to determine, say, if a number is to be treated as an `Integer` or not.

Note

There is actually a little bit more subtlety behind this problem. It involves a language feature known as the **monomorphism restriction**. You don't actually need to know about this for now, so you can skip over this note if you just want to keep a brisk pace. Instead of specifying the type `Double`, you also could have given it a **polymorphic** type, like `Num a => a`, which means "any type `a` which belongs in the class `Num`". The corresponding code looks like this and works just as seamlessly as before:

```
Prelude> let r = 25 :: Num a => a
Prelude> 2 * pi * r
157.07963267948966
```

Haskell could *in theory* assign such polymorphic types systematically, instead of defaulting to some potentially incorrect guess, like `Integer`. But in the real world, this could lead to values being needlessly duplicated or recomputed. To avoid this potential trap, the designers of the Haskell language opted for a more prudent "monomorphism restriction". It means that values may only have a polymorphic type if it can be inferred from the context, or if you explicitly give it one. Otherwise, the compiler is forced to choose a default monomorphic (i.e. non-polymorphic) type. This feature is somewhat controversial. It can even be disabled with the GHC flag `(-fno-monomorphism-restriction)`, but it comes with some risk for inefficiency. Besides, in most cases, it is just as easy to specify the type explicitly.

Variables within variables

Variables can contain much more than just simple values such as 3.14. Indeed, they can contain any Haskell expression whatsoever. So, if we wanted to keep around, say the area of a circle with radius of 5, we could write something like this:

```
Prelude> let area = pi * 5^2
```

What's interesting about this is that we've stored a complicated chunk of Haskell (an arithmetic expression containing a variable) into yet another variable.

We can use variables to store any arbitrary Haskell code, so let's use this to get our acts together.

```

Prelude> let r = 25.0
Prelude> let area2 = pi * r ^ 2
Prelude> area2
1963.4954084936207

```

So far so good.

```

Prelude> let r = 2.0
Prelude> area2
1963.4954084936207

```

Wait a second, why didn't this work? That is, why is it that we get the same value for area as we did back when r was 25? The reason this is the case is that *variables in Haskell do not change*. What actually happens when you defined r the second time is that you are talking about a *different* r . This is something that happens in real life as well. How many people do you know that have the name John? What's interesting about people named John is that most of the time, you can talk about "John" to your friends, and depending on the context, your friends will know which John your are refering to. Programming has something similar to context, called **scope**. We won't explain scope (at least not now), but Haskell's lexical scope is the magic that lets us define two different r and always get the right one back. Scope, however, does not solve the current problem. What we want to do is define a generic `area` function that always gives you the area of a circle. What we could do is just define it a second time:

Variables do not vary

```

Prelude> let area3 = pi * r ^ 2
Prelude> area3
12.566370614359172

```

But we are programmers, and programmers *loathe* repetition. Is there a better way?

Functions

What we are really trying to accomplish with our generic `area` is to define a **function**. Defining functions in Haskell is dead-simple. It is exactly like defining a variable, except with a little extra stuff on the left hand side. For instance, below is our definition of `pi`, followed by our definition of `area`:

```

Prelude> let pi = 3.14159265358979323846264338327950
Prelude> let area r = pi * r ^ 2

```

To calculate the area of our two circles, we simply pass it a different value:

```

Prelude> area 5
78.53981633974483
Prelude> area 25
1963.4954084936207

```

Functions allow us to make a great leap forward in the reusability of our code. But let's slow down for a moment, or rather, back up to dissect things. See the r in our definition `area r = ...`? This is what we call a **parameter**. A parameter is what we use to provide input to the function. When Haskell is interpreting the function, the value of its parameter must come from the outside. In the case of `area`, the value of r is 5 when you say `area 5`, but it is 25 if you say `area 25`.

Exercises

Say I type something in like this (don't type it in yet):

```
Prelude> let r = 0
Prelude> let area r = pi * r ^ 2
Prelude> area 5
```

1. What do you think should happen? Are we in for an unpleasant surprise?
2. What actually happens? Why? (Hint: remember what was said before about "scope")

Scope and parameters

Warning: this section contains spoilers to the previous exercise

We hope you have completed the very short exercise (I would say thought experiment) above. Fortunately, the following fragment of code does not contain any unpleasant surprises:

```
Prelude> let r = 0
Prelude> let area r = pi * r ^ 2
Prelude> area 5
78.53981633974483
```

An unpleasant surprise here would have been getting the value 0. This is just a consequence of what we wrote above, namely the value of a parameter is strictly what you pass in when you call the function. And *that* is directly a consequence of our old friend scope. Informally, the `r` in `let r = 0` is not the same `r` as the one inside our defined function `area` - the `r` inside `area` overrides the other `r`; you can think of it as Haskell picking the most specific version of `r` there is. If you have many friends all named John, you go with the one which just makes more sense and is specific to the context; similarly, what value of `r` we get depends on the scope.

Multiple parameters

Another thing you might want to know about functions is that they can accept more than one parameter. Say for instance, you want to calculate the area of a rectangle. This is quite simple to express:

```
Prelude> let areaRect l w = l * w
Prelude> areaRect 5 10
50
```

Or say you want to calculate the area of a right angle triangle $\left(A = \frac{bh}{2}\right)$:

```
Prelude> let areaTriangle b h = (b * h) / 2
Prelude> areaTriangle 3 9
13.5
```

Passing parameters in is pretty straightforward: you just give them in the same order that they are defined. So, whereas `areaTriangle 3 9` gives us the area of a triangle with base 3 and height 9, `areaTriangle 9 3` gives us the area with the base 9 and height 3.

Exercises

Write a function to calculate the volume of a box. A box has width, height and depth. You have to multiply them all to get the volume.

Functions within functions

To further cut down the amount of repetition it is possible to call functions from within other functions. A simple example showing how this can be used is to create a function to compute the area of a Square. We can think of a square as a special case of a rectangle (the area is still the width multiplied by the length); however, we also know that the width and length are the same, so why should we need to type it in twice?

```
Prelude> let areaRect l w = l * w
Prelude> let areaSquare s = areaRect s s
Prelude> areaSquare 5
25
```

Exercises

Write a function to calculate the volume of a cylinder. The volume of a cylinder is the area of the base, which is a circle (you already programmed this function in this chapter, so reuse it) multiplied by the height.

Summary

1. Variables store values. In fact, they store any arbitrary Haskell expression.
2. Variables do not change.
3. Functions help you write reusable code.
4. Functions can accept more than one parameter.

Notes

1. ^ For readers with prior programming experience: Variables don't change? I only get constants? Shock! Horror! No... trust us, as we hope to show you in the rest of this book, you can go a *very* long way without changing a single variable! In fact, this non-changing of variables makes life easier because it makes programs so much more predictable.

Lists and tuples

Lists and tuples are two ways of crushing several values down into a single value.

Lists

The functional programmer's next best friend

In the last section we introduced the concept of variables and functions in Haskell. Functions are one of the two major building blocks of any Haskell program. The other is the versatile list. So, without further ado, let's switch over to the interpreter and build some lists:

Example - Building Lists in the Interpreter

```
Prelude> let numbers = [1,2,3,4]
Prelude> let truths  = [True, False, False]
Prelude> let strings = ["here", "are", "some", "strings"]
```

The square brackets denote the beginning and the end of the list. List elements are separated by the comma ",", operator. Further, list elements must be all of the same type. Therefore, `[42, "life, universe and everything else"]` is not a legal list because it contains two elements of different types, namely, integer and string respectively. However, `[12, 80]` or `["beer", "sandwiches"]` are valid lists because they are both type-homogeneous.

Here is what happens if you try to define a list with mixed-type elements:

```
Prelude> let mixed = [True, "bonjour"]
<interactive>:1:19:
  Couldn't match `Bool' against `[Char]'
    Expected type: Bool
    Inferred type: [Char]
  In the list element: "bonjour"
  In the definition of `mixed': mixed = [True, "bonjour"]
```

If you're confused about this business of lists and types, don't worry about it. We haven't talked very much about types yet and we are confident that this will clear up as the book progresses.

Building lists

Square brackets and commas aren't the only way to build up a list. Another thing you can do with them is to build them up piece by piece, by **consing** things on to them, via the `(:)` operator.

Example: Consing something on to a list

```
Prelude> let numbers = [1,2,3,4]
Prelude> numbers
[1,2,3,4]
Prelude> 0:numbers
[0,1,2,3,4]
```



When you cons something on to a list (`something:someList`), what you get back is another list. So, unsurprisingly, you could keep on consing your way up.

Example: Consing lots of things to a list

```

Prelude> 1:0:numbers
[1,0,1,2,3,4]
Prelude> 2:1:0:numbers
[2,1,0,1,2,3,4]
Prelude> 5:4:3:2:1:0:numbers
[5,4,3,2,1,0,1,2,3,4]

```

In fact, this is just about how all lists are built, by consing them up from the empty list (`[]`). The commas and brackets notation is actually a pleasant form of **syntactic sugar**. In other words, a list like `[1,2,3,4,5]` is exactly equivalent to `1:2:3:4:5:[]`

You will, however, want to watch out for a potential pitfall in list construction. Whereas `1:2:[]` is perfectly good Haskell, `1:2` is *not*. In fact, if you try it out in the interpreter, you get a nasty error message.

Example: Whoops!

```

Prelude> 1:2
<interactive>:1:2:
  No instance for (Num [a])
    arising from the literal `2' at <interactive>:1:2
  Probable fix: add an instance declaration for (Num [a])
  In the second argument of `(:)', namely `2'
  In the definition of `it': it = 1 : 2

```

Well, to be fair, the error message is nastier than usual because numbers are slightly funny beasts in Haskell. Let's try this again with something simpler, but still wrong, `True:False`

Example: Simpler but still wrong

```

Prelude> True:False
<interactive>:1:5:
  Couldn't match `[Bool]' against `Bool'
    Expected type: [Bool]
    Inferred type: Bool
  In the second argument of `(:)', namely `False'
  In the definition of `it': it = True : False

```

The basic intuition for this is that the cons operator, `(:)` works with this pattern `something:someList`; however, what we gave it is more `something:somethingElse`. Cons only knows how to stick things onto lists. We're starting to run into a bit of reasoning about **types**. Let's summarize so far:

- The elements of the list must have the same type.
- You can only cons `(:)` something onto a list.

Well, sheesh, aren't types annoying? They are indeed, but as we will see in Type basics, they can also be a life

saver. In either case, when you are programming in Haskell and something blows up, you'll probably want to get used to thinking "probably a type error".

Exercises

1. Would the following piece of Haskell work: `3:[True,False]`? Why or why not?
2. Write a function `cons8` that takes a list and conses 8 on to it. Test it out on the following lists by doing:
 1. `cons8 []`
 2. `cons8 [1,2,3]`
 3. `cons8 [True,False]`
 4. `let foo = cons8 [1,2,3]`
 5. `cons8 foo`
3. Write a function that takes two arguments, a list and a thing, and conses the thing onto the list. You should start out with `let myCons list thing =`

Lists within lists

Lists can contain *anything*, just as long as they are all of the same type. Well, then, chew on this: lists are things too, therefore, lists can contain... yes indeed, other lists! Try the following in the interpreter:

Example: Lists can contain lists

```
Prelude> let listOfLists = [[1,2],[3,4],[5,6]]
Prelude> listOfLists
[[1,2],[3,4],[5,6]]
```



Lists of lists can be pretty tricky sometimes, because a list of things does not have the same type as a thing all by itself. Let's sort through these implications with a few exercises:

Exercises

1. Which of these are valid Haskell and which are not? Rewrite in cons notation.
 1. `[1,2,3,[]]`
 2. `[1,[2,3],4]`
 3. `[[1,2,3],[[]]]`
2. Which of these are valid Haskell, and which are not? Rewrite in comma and bracket notation.
 1. `[]:[[1,2,3],[4,5,6]]`
 2. `[]:[]`
 3. `[]:[]:[]`
 4. `[1]:[]:[]`

3. Can Haskell have lists of lists of lists? Why or why not?
4. Why is the following list invalid in Haskell? Don't worry too much if you don't get this one yet.
 1. `[[1,2],3,[4,5]]`

Lists of lists are extremely useful, because they allow you to express some very complicated, structured data (two-dimensional matrices, for example). They are also one of the places where the Haskell type system truly shines. Human programmers, or at least this wikibook author, get confused *all* the time when working with lists of lists, and having restrictions of types often helps in wading through the potential mess.

Tuples

A different notion of many

Tuples are another way of storing multiple values in a single value, but they are subtly different in a number of ways. They are useful when you *know*, in advance, how many values you want to store, and they lift the restriction that all the values have to be of the same type. For example, we might want a type for storing pairs of co-ordinates. We know how many elements there are going to be (two: an x and y co-ordinate), so tuples are applicable. Or, if we were writing a phonebook application, we might want to crunch three values into one: the name, phone number and address of someone. Again, we know how many elements there are going to be. Also, those three values aren't likely to have the same type, but that doesn't matter here, because we're using tuples.

Let's look at some sample tuples.

Example: Some tuples

```
!(True, 1)
!("Hello world", False)
!(4, 5, "Six", True, 'b')
```



The first example is a tuple containing two elements. The first one is `True` and the second is `1`. The next example again has two elements, the first is `"Hello world"` and the second, `False`. The third example is a bit more complex. It's a tuple consisting of *five* elements, the first is the number `4`, the second the number `5`, the third `"Six"`, the fourth `True`, and the last one the character `'b'`. So the syntax for tuples is: separate the different elements with a comma, and surround the whole thing in parentheses.

A quick note on nomenclature: in general you write n -tuple for a tuple of size n . 2-tuples (that is, tuples with 2 elements) are normally called 'pairs' and 3-tuples triples. Tuples of greater sizes aren't actually all that common, although, if you were to logically extend the naming system, you'd have 'quadruples', 'quintuples' and so on, hence the general term 'tuple'.

So tuples are a bit like lists, in that they can store multiple values. However, there is a very key difference: pairs don't have the same type as triples, and triples don't have the same type as quadruples, and in general, two tuples of different sizes have different types. You might be getting a little disconcerted because we keep mentioning this word 'type', but for now, it's just important to grasp how lists and tuples differ in their approach to sizes.

You can have, say, a list of numbers, and add a new number on the front, and it remains a list of numbers. If you

have a pair and wish to add a new element, it becomes a triple, and this is a *fundamentally different object* ^[1].

Exercises

1. Write down the 3-tuple whose first element is 4, second element is "hello" and third element is True.
2. Which of the following are valid tuples ?
 1. (4, 4)
 2. (4, "hello")
 3. (True, "Blah", "foo")
3. Lists can be built by consing new elements on to them: you cons a number onto a list of numbers, and get back a list of numbers. It turns out that there is no such way to build up tuples.
 1. Why do you think that is?
 2. Say for the sake of argument, that there was such a function. What would you get if you "consed" something on a tuple?

What are tuples for?

Tuples are handy when you want to return more than one value from a function. In most languages trying to return two or more things at once means wrapping them up in a special data structure, maybe one that only gets used in that function. In Haskell, just return them as a tuple.

You can also use tuples as a primitive kind of data structure. But that needs an understanding of types, which we haven't covered yet.

Getting data out of tuples

In this section, we concentrate solely on pairs. This is mostly for simplicity's sake, but pairs are by far and away the most commonly used size of tuple.

Okay, so we've seen how we can put values in to tuples, simply by using the (x, y, z) syntax. How can we get them out again? For example, a typical use of tuples is to store the (x, y) co-ordinate pair of a point: imagine you have a chess board, and want to specify a specific square. You could do this by labeling all the rows from 1 to 8, and similarly with the columns, then letting, say, $(2, 5)$ represent the square in row 2 and column 5. Say we want to define a function for finding all the pieces in a given row. One way of doing this would be to find the co-ordinates of all the pieces, then look at the row part and see if it's equal to whatever row we're being asked to examine. This function would need, once it had the co-ordinate pair (x, y) of a piece, to extract the x (the row part). To do this there are two functions, `fst` and `snd`, which *project* the first and second elements out of a pair, respectively (in math-speak a function that gets some data out of a structure is called a "Projection"). Let's see some examples:

Example: Using `fst` and `snd`



```

Prelude> fst (2, 5)
2
Prelude> fst (True, "boo")
True
Prelude> snd (5, "Hello")
"Hello"

```

It should be fairly obvious what these functions do. Note that you can *only* use these functions on pairs. Why? It all harks back to the fact that tuples of different sizes are different beasts entirely. `fst` and `snd` are specialized to pairs, and so you can't use them on anything else [2].

Exercises

1. Use a combination of `fst` and `snd` to extract the 4 from the tuple `(("Hello", 4), True)`.
2. Normal chess notation is somewhat different to ours: it numbers the rows from 1-8 but then labels the columns A-H. Could we label a specific point with a number and a character, like `(4, 'a')`? What important difference with lists does this illustrate?

Tuples within tuples (and other combinations)

We can apply the same reasoning to tuples about storing lists within lists. Tuples are things too, so you can store tuples with tuples (within tuples up to any arbitrary level of complexity). Likewise, you could also have lists of tuples, tuples of lists, all sorts of combinations along the same lines.

Example: Nesting tuples and lists

```

i((2,3), True)
i((2,3), [2,3])
i[(1,2), (3,4), (5,6)]

```



Some discussion about this - what you get out of this, maybe, what's the big idea behind grouping things together

There is one bit of trickiness to watch out for, however. The type of a tuple is defined not only by its size, but by the types of objects it contains. For example, the tuples like `("Hello", 32)` and `(47, "World")` are fundamentally different. One is of type `(String, Int)` tuples, whereas the other is `(Int, String)`. This has implications for building up lists of tuples. We could very well have lists like `[("a", 1), ("b", 9), ("c", 9)]`, but having a list like `[("a", 1), (2, "b"), (9, "c")]` is right out. Can you spot the difference?

Exercises

1. Which of these are valid Haskell, and why?

- `fst [1,2]`
- `1:(2,3)`
- `(2,4):(2,3)`
- `(2,4):[]`
- `[(2,4),(5,5),('a','b')]`
- `[(2,4],[2,2])`

2. *FIXME: to be added*

Summary

We have introduced two new notions in this chapter, lists and tuples. To sum up:

1. Lists are defined by square brackets and commas : `[1,2,3]`.
 - They can contain *anything* as long as all the elements of the list are of the same type
 - They can also be built by the cons operator, `(:)`, but you can only cons things onto lists
2. Tuples are defined by parentheses and commas : `("Bob",32)`
 - They can contain *anything*, even things of different types
 - They have a fixed length, or at least their length is encoded in their type. That is, two tuples with different lengths will have different types.
3. Lists and tuples can be combined in any number of ways: lists within lists, tuples with lists, etc

We hope that at this point, you're somewhat comfortable enough manipulating them as part of the fundamental Haskell building blocks (variables, functions and lists), because we're now going to move to some potentially heady topics, types and recursion. Types, we have alluded to thrice in this chapter without really saying what they are, so these shall be the next major topic that we cover. But before we get to that, we're going to make a short detour to help you make better use of the GHC interpreter.

Notes

1. ↑ At least as far as types are concerned, but we're trying to avoid that word :)
2. ↑ More technically, `fst` and `snd` have types which limit them to pairs. It would be impossible to define projection functions on tuples in general, because they'd have to be able to accept tuples of different sizes, so the type of the function would vary.

Next steps

Haskell files

Up to now, we've made heavy use of the GHC interpreter. The interpreter is indeed a useful tool for trying things out quickly and for debugging your code. But we're getting to the point where typing everything directly into the

interpreter isn't very practical. So now, we'll be writing our first Haskell source files.

Open up a file `Varfun.hs` in your favourite text editor (the `hs` stands for Haskell) and paste the following definition in. Haskell uses indentations and spaces to decide where functions (and other things) begin and end, so make sure there are no leading spaces and that indentations are correct, otherwise GHC will report parse errors.

```
area r = pi * r^2
```

(In case you're wondering, `pi` is actually predefined in Haskell, no need to include it here). Now change into the directory where you saved your file, open up `ghci`, and use `:load` (or `:l` for short):

```
Prelude> :load Varfun.hs
Compiling Main          ( Varfun.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

If `ghci` gives an error, "Could not find module 'Varfun.hs'", then use `:cd` to change the current directory to the directory containing `Varfun.hs`:

```
Prelude> :cd c:\myDirectory
Prelude> :load Varfun.hs
Compiling Main          ( Varfun.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Now you can execute the bindings found in that file:

```
*Main> area 5
78.53981633974483
```

If you make changes to the file, just use `:reload` (`:r` for short) to reload the file.

Note

GHC can also be used as a compiler. That is, you could use GHC to convert your Haskell files into a program that can then be run without running the interpreter. See the documentation for details.

You'll note that there are a couple of differences between how we do things when we type them directly into `ghci`, and how we do them when we load them from files. The differences may seem awfully arbitrary for now, but they're actually quite sensible consequences of the scope, which, rest assured, we will explain later.

No `let`

For starters, you no longer say something like

```
let x = 3
let y = 2
let area r = pi * r ^ 2
```

Instead, you say things like

```
x = 3
y = 2
area r = pi * r ^ 2
```

The keyword `let` is actually something you use a lot in Haskell, but not exactly in this context. We'll see further on in this chapter when we discuss the use of `let` **bindings**.

You can't define the same thing twice

Previously, the interpreter cheerfully allowed us to write something like this

```
Prelude> let r = 5
Prelude> r
5
Prelude> let r = 2
Prelude> r
2
```

On the other hand, writing something like this in a source file does not work

```
-- this does not work
r = 5
r = 2
```

As we mentioned above, variables do not change, and this is even more the case when you're working in a source file. This has one very nice implication. It means that:

Order does not matter

The order in which you declare things does not matter. For example, the following fragments of code do exactly the same thing:

<pre>y = x * 2 x = 3</pre>	<pre>x = 3 y = x * 2</pre>
----------------------------	----------------------------

This is a unique feature of Haskell and other functional programming languages. The fact that variables never change means that we can opt to write things in any order that we want (but this is also why you can't declare something more than once... it would be ambiguous otherwise).

Exercises

Save the functions you had written in the previous module's exercises into a Haskell file. Load the file in GHCi and test the functions on a few parameters

More about functions

Working with actual source code files instead of typing things into the interpreter makes things convenient to define much more substantial functions than those we've seen up to now. Let's flex some Haskell muscle here and examine the kinds of things we can do with our functions.

Conditional expressions

if / then / else

Haskell supports standard conditional expressions. For instance, we could define a function that returns - 1 if its argument is less than 0; 0 if its argument *is* 0; and 1 if its argument is greater than 0 (this is called the signum function). Actually, such a function already exists, but let's define one of our own, what we'll call `mySignum`.

```
mySignum x =
  if x < 0
  then -1
  else if x > 0
  then 1
  else 0
```

You can experiment with this as:

Example:

```
*Main> mySignum 5
1
*Main> mySignum 0
0
*Main> mySignum (5-10)
-1
*Main> mySignum (-1)
-1
```



Note that the parenthesis around "-1" in the last example are required; if missing, the system will think you are trying to subtract the value "1" from the value "mySignum," which is ill-typed.

The if/then/else construct in Haskell is very similar to that of most other programming languages; however, you must have both a **then** and an **else** clause. It evaluates the condition (in this case $x < 0$ and, if this evaluates to `True`, it evaluates the **then** condition; if the condition evaluated to `False`, it evaluates the **else** condition).

You can test this program by editing the file and loading it back into your interpreter. Instead of typing `:l varfun.hs` again, you can simply type `:reload` or just `:r` to reload the current file. This is usually much faster.

case

Haskell, like many other languages, also supports **case** constructions. These are used when there are multiple values that you want to check against (case expressions are actually quite a bit more powerful than this -- see the Pattern matching chapter for all of the details).

Suppose we wanted to define a function that had a value of 1 if its argument were 0; a value of 5 if its argument were 1; a value of 2 if its argument were 2; and a value of - 1 in all other instances. Writing this function using `if` statements would be long and very unreadable; so we write it using a `case` statement as follows (we call this function `f`):

```
if x =
  case x of
    0 -> 1
    1 -> 5
    2 -> 2
    _ -> -1
```

In this program, we're defining `f` to take an argument `x` and then inspecting the value of `x`. If it matches 0, the value of `f` is 1. If it matches 1, the value of `f` is 5. If it matches 2, then the value of `f` is 2; and if it hasn't matched anything by that point, the value of `f` is - 1 (the underscore can be thought of as a "wildcard" -- it will match anything).

The indentation here is important. Haskell uses a system called "layout" to structure its code (the programming language Python uses a similar system). The layout system allows you to write code without the explicit semicolons and braces that other languages like C and Java require.

Indentation

The general rule for layout is that an open-brace is inserted after the keywords `where`, `let`, `do` and `of`, and the column position at which the next command appears is remembered. From then on, a semicolon is inserted before every new line that is indented the same amount. If a following line is indented less, a close-brace is inserted. This may sound complicated, but if you follow the general rule of indenting after each of those keywords, you'll never have to remember it (see the Indentation chapter for a more complete discussion of layout).

Some people prefer not to use layout and write the braces and semicolons explicitly. This is perfectly acceptable. In this style, the above function might look like:

```
if x = case x of
      { 0 -> 1 ; 1 -> 5 ; 2 -> 2 ; _ -> -1 }
```

Of course, if you write the braces and semicolons explicitly, you're free to structure the code as you wish. The following is also equally valid:

```
if x =
  case x of { 0 -> 1 ;
            1 -> 5 ; 2 -> 2
            ; _ -> -1 }
```

However, structuring your code like this only serves to make it unreadable (in this case).

Defining one function for different parameters

Functions can also be defined piece-wise, meaning that you can write one version of your function for certain parameters and then another version for other parameters. For instance, the above function `f` could also be written as:

```

f 0 = 1
f 1 = 5
f 2 = 2
f _ = -1

```

Here, the order is important. If we had put the last line first, it would have matched every argument, and `f` would return `-1`, regardless of its argument (most compilers will warn you about this, though, saying something about overlapping patterns). If we had not included this last line, `f` would produce an error if anything other than `0`, `1` or `2` were applied to it (most compilers will warn you about this, too, saying something about incomplete patterns). This style of piece-wise definition is very popular and will be used quite frequently throughout this tutorial. These two definitions of `f` are actually equivalent -- this piece-wise version is translated into the case expression.

Function composition

More complicated functions can be built from simpler functions using *function composition*. Function composition is simply taking the result of the application of one function and using that as an argument for another. We've already seen this back in the Getting set up chapter, when we wrote `5*4+3`. In this, we were evaluating `5 * 4` and then applying `+ 3` to the result. We can do the same thing with our `square` and `f` functions:

```

square x = x^2

```

Example:

```

*Main> square (f 1)
25
*Main> square (f 2)
4
*Main> f (square 1)
5
*Main> f (square 2)
-1

```



The result of each of these function applications is fairly straightforward. The parentheses around the inner function are necessary; otherwise, in the first line, the interpreter would think that you were trying to get the value of `square f`, which has no meaning. Function application like this is fairly standard in most programming languages. There is another, more mathematical way to express function composition: the `(.)` enclosed period function. This `(.)` function is modeled after the \circ operator in mathematics.

Note

In mathematics we write $f \circ g$ to mean "f following g." In Haskell, we write `f . g` also to mean "f following g."

The meaning of $f \circ g$ is simply that $(f \circ g)(x) = f(g(x))$. That is, applying the function $f \circ g$ to the value x is the same as applying g to x , taking the result, and then applying f to that.

The `(.)` function (called the function composition function), takes two functions and makes them into one. For instance, if we write `(square . f)`, this means that it creates a new function that takes an argument, applies `f` to that argument and then applies `square` to the result. Conversely, `(f . square)` means that a new function is created that takes an argument, applies `square` to that argument and then applies `f` to the result. We can see this by testing it as before:

Example:

```

-----
!*Main> (square . f) 1
!25
!*Main> (square . f) 2
!4
!*Main> (f . square) 1
!5
!*Main> (f . square) 2
!1
-----

```



Here, we must enclose the function composition in parentheses; otherwise, the Haskell compiler will think we're trying to compose `square` with the value `f 1` in the first line, which makes no sense since `f 1` isn't even a function.

It would probably be wise to take a little time-out to look at some of the functions that are defined in the Prelude. Undoubtedly, at some point, you will accidentally rewrite some already-existing function (I've done it more times than I can count), but if we can keep this to a minimum, that would save a lot of time.

Let Bindings

Often we wish to provide local declarations for use in our functions. For instance, if you remember back to your grade school mathematics courses, the following equation is used to find the roots (zeros) of a polynomial of the form $ax^2 + bx + c = 0$: $x = (-b \pm \sqrt{b^2 - 4ac})/2a$. We could write the following function to compute the two values of x :

```

-----
roots a b c =
  ((-b + sqrt(b*b - 4*a*c)) / (2*a),
   (-b - sqrt(b*b - 4*a*c)) / (2*a))
-----

```

Notice that our definition here has a bit of redundancy. It is not quite as nice as the mathematical definition because we have needlessly repeated the code for `sqrt(b*b - 4*a*c)`. To remedy this problem, Haskell allows for local bindings. That is, we can create values inside of a function that only that function can see. For instance, we could create a local binding for `sqrt(b*b-4*a*c)` and call it, say, `disc` and then use that in both places where `sqrt(b*b - 4*a*c)` occurred. We can do this using a `let/in` declaration:

```

-----
roots a b c =
  let disc = sqrt (b*b - 4*a*c)
  in  ((-b + disc) / (2*a),
      (-b - disc) / (2*a))
-----

```

In fact, you can provide multiple declarations inside a `let`. Just make sure they're indented the same amount, or you will have layout problems:

```

roots a b c =
  let disc = sqrt (b*b - 4*a*c)
      twice_a = 2*a
  in  ((-b + disc) / twice_a,
      (-b - disc) / twice_a)

```

Type basics

Types in programming are a way of grouping similar values. In Haskell, the type system is a powerful way of ensuring there are fewer mistakes in your code.

Introduction

Programming deals with different sorts of entities. For example, consider adding two numbers together:

2 + 3

What are 2 and 3? They are numbers, clearly. But how about the plus sign in the middle? That's certainly not a number. So what is it?

Similarly, consider a program that asks you for your name, then says "Hello". Neither your name nor the word Hello is a number. What are they then? We might refer to all words and sentences and so forth as Text. In fact, it's more normal in programming to use a slightly more esoteric word, that is, *String*.

In Haskell, the rule is that all type names have to begin with a capital letter. We shall adhere to this convention henceforth.

If you've ever set up a database before, you'll likely have come across types. For example, say we had a table in a database to store details about a person's contacts; a kind of personal telephone book. The contents might look like this:

First Name	Last Name	Telephone number	Address
Sherlock	Holmes	743756	221B Baker Street London
Bob	Jones	655523	99 Long Road Street Villestown

The fields contain values. Sherlock is a value as is 99 Long Road Street Villestown as well as 655523. As we've said, types are a way of grouping different sorts of data. What do we have in the above table? Two of the columns, First name and Last name contain text, so we say that the values are of type String. The type of the third column is a dead giveaway by its name, Telephone number. Values in that column have the type of Number!

At first glance one may be tempted to class address as a string. However, the semantics behind an innocent address are quite complex. There's a whole lot of human conventions that dictate. For example, if the first line contains a number, then that's the number of the house, if not, then it's probably the name of the house, except if the line begins with PO Box then it's just a postal box address and doesn't indicate where the person lives at all... Clearly, there's more going on here than just Text. We could say that addresses are Text; there'd be nothing

wrong with that. However, claiming they're of some different type, say, Address, is more powerful. If we know some piece of data has the type of Text, that's not very helpful. However, if we know it has the type of Address, we instantly know much more about the piece of data.

We might also want to apply this line of reasoning to our telephone number column. Indeed, it would be a good idea to come up with a TelephoneNumber type. Then if we were to come across some arbitrary sequence of digits, knowing that sequence of digits was of type TelephoneNumber, we would have access to a lot more information than if it were just a Number.

Another reason to not consider the TelephoneNumber as a Number is that numbers are arithmetic entities allowing them to be used for computing other numbers. What would be then the meaning and expected effect of adding 1 to a TelephoneNumber? It would not allow calling anyone by phone. That's a good enough reason why you would like a stronger type than just a mere Number. Also, each digit making a telephone number is important, it's not acceptable to lose some of them, by rounding it, or even by omitting some leading zeroes. Other reasons would be that telephone numbers can't be used the same way from different locations, and you may also need to specify within a TelephoneNumber value some other information like a area number or a country prefix. One good way to specify that is to provide some abstraction for telephone numbers and to design your database with a separate type instead of just Number.

Why types are useful

So far, what we've done just seems like categorizing things -- hardly a feature which would cause every modern programming language designer to incorporate into their language! In the next section we explore how Haskell uses types to the programmer's benefit.

Using the interactive `:type` command

Characters and strings

The best way to explore how types work in Haskell is to fire up GHCi. Let's do it! Once we're up and running, let us get to know the `:type` command.

Example: Using the `:t` command in GHCi on a literal character

```
Prelude> :type 'H'
'H' :: Char
```



(The `:type` can be also shortened to `:t`, which we shall use from now on.)

And there we have it. You give GHCi an expression and it returns its type. In this case we gave it the literal value `'H'` - the letter H enclosed in single quotation marks (a.k.a. apostrophe, ANSI 39) and GHCi printed it followed by the `::` symbol which reads "is of type" followed by `Char`. The whole thing reads: `'H'` is of type `Char`.

If we try to give it a string of characters, we need to enclose them in quotation marks:

Example: Using the `:t` command in GHCi on a literal string



```
Prelude> :t "Hello World"
"Hello World" :: [Char]
```

In this case we gave it some text enclosed in double quotation marks and GHCi printed `"Hello World" :: [Char]`. `[Char]` means *a list of characters*. Notice the difference between `Char` and `[Char]` - the square brackets are used to construct literal lists, and they are also used to describe the list type.

Exercises

1. Try using the `:type` command on the literal value `"H"` (notice the double quotes). What happens? Why?
2. Try using the `:type` command on the literal value `'Hello World'` (notice the single quotes). What happens? Why?

This is essentially what strings are in Haskell - lists of characters. A string in Haskell can be initialized in several ways: It may be entered as a sequence of characters enclosed in double quotation marks (ANSI 34); it may be constructed similar to any other list as individual elements of type `Char` joined together with the `:"` function and terminated by an empty list or, built with individual `Char` values enclosed in brackets and separated by commas.

So, for the final time, what precisely is this concept of text that we're throwing around? One way of interpreting it is to say it's basically a sequence of characters. Think about it: the word "Hey" is just the character 'H' followed by the character 'e' followed by the character 'y'. Haskell uses a list to hold this sequence of characters. Square brackets indicate a list of things, for example here `[Char]` means 'a list of Chars'.

Haskell has a concept of type synonyms. Just as in the English language, two words that mean the same thing, for example 'fast' and 'quick', are called synonyms, in Haskell two types which are exactly the same are called 'type synonyms'. Everywhere you can use `[Char]`, you can use `String`. So to say:

```
"Hello World" :: String
```

Is also perfectly valid. From here on we'll mostly refer to text as `String`, rather than `[Char]`.

Boolean values

One of the other types found in most languages is called a Boolean, or `Bool` for short. This has two values: `true` or `false`. This turns out to be very useful. For example consider a program that would ask the user for a name then look that name up in a spreadsheet. It might be useful to have a function, `nameExists`, which indicates whether or not the name of the user exists in the spreadsheet. If it *does exist*, you could say that it is *true* that the name exists, and if not, you could say that it is *false* that the name exists. So we've come across Booleans. The two *values* of booleans are, as we've mentioned, `true` and `false`. In Haskell boolean values are capitalized (for reasons that will later become clear):

Example: Exploring the types of True and False in GHCi

```
Prelude> :t True
True :: Bool
Prelude> :t False
False :: Bool
```

This shouldn't need too much explaining at this point. The values True and False are categorized as Booleans, that is to say, they have type Bool.

Numeric types

If you've been playing around with typing `:t` on all the familiar values you've come across, perhaps you've run into the following complication:

```
Prelude> :t 5
5 :: (Num t) => t
```

We'll defer the explanation of this until later. The short version of the story is that there are many different types of numbers (fractions, whole numbers, etc) and 5 can be any one of them. This weird-looking type relates to a Haskell feature called type classes, which we will be playing with later in this book.

Functional types

So far, the types we have talked about apply to values (strings, booleans, characters, etc), and we have explained how types not only help to categorize them, but also describe them. The next thing we'll look at is what makes the type system truly powerful: We can assign types not only to values, but to functions as well^[3]. Let's look at some examples.

Example: not**Example:** Negating booleans

```
not True = False
not False = True
```

`not` is a standard Prelude function that simply negates Booleans, in the sense that truth turns into falsity and vice versa. For example, given the above example we gave using Booleans, `nameExists`, we could define a similar function that would test whether a name doesn't exist in the spreadsheet. It would likely look something like this:

Example: `nameDoesntExist`: using `not`



```
nameDoesntExist name = not (nameExists name)
```

To assign a type to `not` we look at two things: the type of values it takes as its input, and the type of values it returns. In our example, things are easy. `not` takes a `Bool` (the `Bool` to be negated), and returns a `Bool` (the negated `Bool`). Therefore, we write that:

Example: Type signature for `not`



```
not :: Bool -> Bool
```

You can read this as '`not` is a function from things of type `Bool` to things of type `Bool`'.

Example: `unlines` and `unwords`

A common programming task is to take a list of `Strings`, then join them all up into a single string, but insert a newline character between each one, so they all end up on different lines. For example, say you had the list `["Bacon", "Sausages", "Egg"]`, and wanted to convert it to something resembling a shopping list, the natural thing to do would be to join the list together into a single string, placing each item from the list onto a new line. This is precisely what `unlines` does. `unwords` is similar, but it uses a space instead of a newline as a separator. (mnemonic: `un` = unite)

Example: `unlines` and `unwords`



```
Prelude> unlines ["Bacon", "Sausages", "Egg"]
"Bacon\nSausages\nEgg\n"
Prelude> unwords ["Bacon", "Sausages", "Egg"]
"Bacon Sausages Egg"
```

Notice the weird output from `unlines`. This isn't particularly related to types, but it's worth noting anyway, so we're going to digress a little and explore why this is. Basically, any output from `GHCi` is first run through the `show` function, which converts it into a `String`. This makes sense, because `GHCi` shows you the result of your commands as text, so it has to be a `String`. However, what does `show` do if you give it something which is already a `String`? Although the obvious answer would be 'do nothing', the behaviour is actually slightly different: any 'special characters', like tabs, newlines and so on in the `String` are converted to their 'escaped forms', which means that rather than a newline actually making the stuff following it appear on the next line, it is shown as `"\n"`. To avoid this, we can use the `putStrLn` function, which `GHCi` sees and doesn't run your output through `show`.

Example: Using `putStrLn` in GHCi

```

Prelude> putStrLn (unlines ["Bacon", "Sausages", "Egg"])
Bacon
Sausages
Egg
Prelude> putStrLn (unwords ["Bacon", "Sausages", "Egg"])
Bacon Sausages Egg

```

The second result may look identical, but notice the lack of quotes. `putStrLn` outputs exactly what you give it (actually `putStrLn` appends a newline character to its input before printing it; the function `putStr` outputs *exactly* what you give it). Also, note that you can only pass it a `String`. Calls like `putStrLn 5` will fail. You'd need to convert the number to a `String` first, that is, use `show`: `putStrLn (show 5)` (or use the equivalent function `print`: `print 5`).

Getting back to the types. What would the types of `unlines` and `unwords` be? Well, again, let's look at both what they take as an argument, and what they return. As we've just seen, we've been feeding these functions a list, and each of the items in the list has been a `String`. Therefore, the type of the argument is `[String]`. They join all these `Strings` together into one long `String`, so the return type has to be `String`. Therefore, both of the functions have type `[String] -> String`. Note that we didn't mention the fact that the two functions use different separators. This is totally inconsequential when it comes to types — all that matters is that they return a `String`. The type of a `String` with some newlines is precisely the same as the type of a `String` with some spaces.

Example: chr and ord

Text presents a problem to computers. Once everything is reduced to its lowest level, all a computer knows how to deal with is 1's and 0's: computers work in binary. As working with binary isn't very convenient, humans have come up with ways of making computers store text. Every character is first converted to a number, then that number is converted to binary and stored. Hence, a piece of text, which is just a sequence of characters, can be encoded into binary. Normally, we're only interested in how to encode characters into their numerical representations, because the number to binary bit is very easy.

The easiest way of converting characters to numbers is simply to write all the possible characters down, then number them. For example, we might decide that 'a' corresponds to 1, then 'b' to 2, and so on. This is exactly what a thing called the ASCII standard is: 128 of the most commonly-used characters, numbered. Of course, it would be a bore to sit down and look up a character in a big lookup table every time we wanted to encode it, so we've got two functions that can do it for us, `chr` (pronounced 'char') and `ord` ^[4]:

Example: Type signatures for `chr` and `ord`

```

chr :: Int -> Char
ord :: Char -> Int

```

Remember earlier when we stated Haskell has many numeric types? The simplest is `Int`, which represents whole numbers, or integers, to give them their proper name. ^[5] So what do the above type signatures say? Recall how

the process worked for `not` above. We look at the type of the function's argument, then at the type of the function's result. In the case of `chr` (find the character corresponding to a specific numeric encoding), the type signature tells us that it takes arguments of type `Int` and has a result of type `Char`. The converse is the case with `ord` (find the specific numeric encoding for a given character): it takes things of type `Char` and returns things of type `Int`.

To make things more concrete, here are a few examples of function calls to `chr` and `ord`, so you can see how the types work out. Notice that the two functions aren't in the standard prelude, but instead in the `Data.Char` module, so you have to load that module with the `:m` (or `:module`) command.

Example: Function calls to `chr` and `ord`

```

Prelude> :m Data.Char
Prelude Data.Char> chr 97
'a'
Prelude Data.Char> chr 98
'b'
Prelude Data.Char> ord 'c'
'99

```



Functions in more than one argument

So far, all we've seen is functions that take a single argument. This isn't very interesting! For example, the following is a perfectly valid Haskell function, but what would its type be?

Example: A function in more than one argument

```

f x y = x + 5 + 2 * y

```



As we've said a few times, there's more than one type for numbers, but we're going to cheat here and pretend that `x` and `y` have to be `Ints`.

The general technique for forming the type of a function in more than one argument, then, is to just write down all the types of the arguments in a row, in order (so in this case `x` first then `y`), then write `->` in between all of them. Finally, add the type of the result to the end of the row and stick a final `->` in just before it. So in this case, we have:

FIXME: use images here.

1. Write down the types of the arguments. We've already said that `x` and `y` have to be `Ints`, so it becomes:

```

Int          Int
^^ x is an Int ^^ y is an Int as well

```

2. Fill in the gaps with `->`:

There are very deep reasons for this, which we'll cover in the chapter on Currying.

```
Int -> Int
```

3. Add in the result type and a final `->`. In our case, we're just doing some basic arithmetic so the result remains an `Int`.

```
Int -> Int -> Int
      ^^ We're returning an Int
      ^^ There's the extra -> that got added in
```

Real-World Example: `openWindow`

As you'll learn in the Practical Haskell section of the course, one popular group of Haskell libraries are the GUI ones. These provide functions for dealing with all the parts of Windows or Linux you're familiar with: opening and closing application windows, moving the mouse around etc. One of the functions from one of these libraries is called `openWindow`, and you can use it to open a new window in your application. For example, say you're writing a word processor like Microsoft Word, and the user has clicked on the 'Options' button. You need to open a new window which contains all the options that they can change. Let's look at the type signature for this function ^[6].

A library is a collection of common code used by many programs.

Example: `openWindow`

```
openWindow :: WindowTitle -> WindowSize -> Window
```



Don't panic! Here are a few more types you haven't come across yet. But don't worry, they're quite simple. All three of the types there, `WindowTitle`, `WindowSize` and `Window` are defined by the GUI library that provides `openWindow`. As we saw when constructing the types above, because there are two arrows, the first two types are the types of the parameters, and the last is the type of the result. `WindowTitle` holds the title of the window (what appears in the blue bar - you didn't change the color, did you? - at the top), `WindowSize` how big the window should be. The function then returns a value of type `Window` which you can use to get information on and manipulate the window.

Exercises

Finding types for functions is a basic Haskell skill that you should become very familiar with. What are the types of the following functions?

1. The `negate` function, which takes an `Int` and returns that `Int` with its sign swapped. For example, `negate 4 = -4`, and `negate (-2) = 2`
2. The `&&` function, pronounced 'and', that takes two `Bools` and returns a third `Bool` which is `True` if both the arguments were, and `False` otherwise.
3. The `||` function, pronounced 'or', that takes two `Bools` and returns a third `Bool` which is `True` if either of the arguments were, and `False` otherwise.

For any functions hereafter involving numbers, you can just assume the numbers are Ints.

1. `f x y = not x && y`
2. `g x = (2*x - 1)^2`
3. `h x y z = chr (x - 2)`

Polymorphic types

So far all we've looked at are functions and values with a single type. However, if you start playing around with `:t` in GHCi you'll quickly run into things that don't have types beginning with the familiar capital letter. For example, there's a function that finds the length of a list, called (rather predictably) `length`. Remember that `[Foo]` is a list of things of type `Foo`. However, we'd like `length` to work on lists of any type. I.e. we'd rather not have a `lengthInts :: [Int] -> Int`, as well as a `lengthBools :: [Bool] -> Int`, as well as a `lengthStrings :: [String] -> Int`, as well as a...

That's too complicated. We want one single function that will find the length of any type of list. The way Haskell does this is using type variables. For example, the actual type of `length` is as follows:

Example: Our first polymorphic type

```
length :: [a] -> Int
```



The "a" you see there in the square brackets is called a **type variable**. Type variables begin with a lowercase letter. Indeed, this is why types have to begin with an uppercase letter — so they can be distinguished from type variables. When Haskell sees a type variable, it allows any type to take its place. This is exactly what we want. In type theory (a branch of mathematics), this is called polymorphism: functions or values with only a single type (like all the ones we've looked at so far except `length`) are called monomorphic, and things that use type variables to admit more than one type are therefore polymorphic.

We'll look at the theory behind polymorphism in much more detail later in the course.

Example: `fst` and `snd`

As we saw, you can use the `fst` and `snd` functions to extract parts of pairs. By this time you should be in the habit of thinking "What type is that function?" about every function you come across. Let's examine `fst` and `snd`. First, a few sample calls to the functions:

Example: Example calls to `fst` and `snd`




```

Prelude> fst (1, 2)
1
Prelude> fst ("Hello", False)
"Hello"
Prelude> snd (("Hello", False), 4)
4

```

To begin with, let's point out the obvious: these two functions take a pair as their parameter and return one part of this pair. The important thing about pairs, and indeed tuples in general, is that they don't have to be homogeneous with respect to types; their different parts can be different types. Indeed, that is the case in the second and third examples above. If we were to say:

```
fst :: (a, a) -> a
```

That would force the first and second part of input pair to be the same type. That illustrates an important aspect to type variables: although they can be replaced with any type, they have to be replaced with the same type everywhere. So what's the correct type? Simply:

Example: The types of `fst` and `snd`

```
fst :: (a, b) -> a
snd :: (a, b) -> b
```



Note that if you were just given the type signatures, you might guess that they return the first and second parts of a pair, respectively. In fact this is not necessarily true, they just have to return something with the same type of the first and second parts of the pair.

Type signatures in code

Now we've explored the basic theory behind types and types in Haskell, let's look at how they appear in code. Most Haskell programmers will *annotate* every function they write with its associated type. That is, you might be writing a module that looks something like this:

Example: Module without type signatures

```

module StringManip where
import Data.Char
uppercase = map toUpper
lowercase = map toLower
capitalise x =
  let capWord []      = []
      capWord (x:xs) = toUpper x : xs
  in unwords (map capWord (words x))

```



This is a small library that provides some frequently used string manipulation functions. `uppercase` converts a

string to uppercase, lowercase to lowercase, and `capitalize` capitalizes the first letter of every word. Providing a type for these functions makes it more obvious what they do. For example, most Haskellers would write the above module something like the following:

Example: Module with type signatures

```
module StringManip where
import Data.Char

uppercase, lowercase :: String -> String
uppercase = map toUpper
lowercase = map toLower

capitalise :: String -> String
capitalise x =
  let capWord []      = []
      capWord (x:xs) = toUpper x : xs
  in unwords (map capWord (words x))
```



Note that you can group type signatures together into a single type signature (like ours for `uppercase` and `lowercase` above) if the two functions share the same type.

Type inference

So far, we've explored types by using the `:t` command in GHCi. However, before you came across this chapter, you were still managing to write perfectly good Haskell code, and it has been accepted by the compiler. In other words, it's not necessary to add type signatures. However, if you don't add type signatures, that doesn't mean Haskell simply forgets about typing altogether! Indeed, when you didn't tell Haskell the types of your functions and variables, it *worked them out*. This is a process called *type inference*, whereby the compiler starts with the types of things it knows, then works out the types of the rest of the things. Type inference for Haskell is *decidable*, which means that the compiler can *always* work out the types, even if you never write them in [7]. Lets look at some examples to see how the compiler works out types.

Example: Simple type inference

```
-- We're deliberately not providing a type signature for this function
isL c = c == 'l'
```



This function takes a character and sees if it is an 'l' character. The compiler *derives* the type for `isL` something like the following:

Example: A typing derivation



```

(==)  :: a -> a -> Bool
'l'   :: Char
Replacing the second 'a' in the signature for (==) with the type of 'l':
(==)  :: Char -> Char -> Bool
isL   :: Char -> Bool

```

The first line indicates that the type of the function `(==)`, which tests for equality, is `a -> a -> Bool` [8]. (We include the function name in parentheses because it's an *operator*: its name consists only of non-alphanumeric characters. More on this later.) The compiler also knows that something in 'single quotes' has type `Char`, so clearly the literal `'l'` has type `Char`. Next, the compiler starts replacing the type variables in the signature for `(==)` with the types it knows. Note that in one step, we went from `a -> a -> Bool` to `Char -> Char -> Bool`, because the type variable `a` was used in both the first and second argument, so they need to be the same. And so we arrive at a function that takes a single argument (whose type we don't know yet, but hold on!) and applies it as the first argument to `(==)`. We have a particular *instance* of the polymorphic type of `(==)`, that is, here, we're talking about `(==) :: Char -> Char -> Bool` because we know that we're comparing `Chars`. Therefore, as `(==) :: Char -> Char -> Bool` and we're feeding the parameter into the first argument to `(==)`, we know that the parameter has the type of `Char`. Phew!

But wait, we're not even finished yet! What's the return type of the function? Thankfully, this bit is a bit easier. We've fed two `Chars` into a function which (in this case) has type `Char -> Char -> Bool`, so we must have a `Bool`. Note that the return value from the call to `(==)` becomes the return value of our `isL` function.

So, let's put it all together. `isL` is a function which takes a single argument. We discovered that this argument must be of type `Char`. Finally, we derived that we return a `Bool`. So, we can confidently say that `isL` has the type:

Example: `isL` with a type

```

isL :: Char -> Bool
isL c = c == 'l'

```



And, indeed, if you miss out the type signature, the Haskell compiler will discover this on its own, using exactly the same method we've just run through.

Reasons to use type signatures

So if type signatures are optional, why bother with them at all? Here are a few reasons:

- **Documentation:** the most prominent reason is that it makes your code easier to read. With most functions, the name of the function along with the type of the function are sufficient to guess at what the function does. (Of course, you should always comment your code anyway.)
- **Debugging:** if you annotate a function with a type, then make a typo in the body of the function, the compiler will tell you *at compile-time* that your function is wrong. Leaving off the type signature could have the effect of allowing your function to compile, and the compiler would assign it an erroneous type. You wouldn't know until you ran your program that it was wrong. In fact, this is so important, let's explore it some more.

Types prevent errors

Imagine you have a few functions set up like the following:

Example: Type inference at work

```
fiveOrSix :: Bool -> Int
fiveOrSix True  = 5
fiveOrSix False = 6

pairToInt :: (Bool, String) -> Int
pairToInt x = fiveOrSix (fst x)
```



Our function `fiveOrSix` takes a `Bool`. When `pairToInt` receives its arguments, it knows, because of the type signature we've annotated it with, that the first element of the pair is a `Bool`. So, we could extract this using `fst` and pass that into `fiveOrSix`, and this would work, because the type of the first element of the pair and the type of the argument to `fiveOrSix` are the same.

This is really central to typed languages. When passing expressions around you have to make sure the types match up like they did here. If they don't, you'll get *type errors* when you try to compile; your program won't *typecheck*. This is really how types help you to keep your programs bug-free. To take a very trivial example:

Example: A non-typechecking program

```
"hello" + " world"
```



Having that line as part of your program will make it fail to compile, because you can't add two strings together! More likely, you wanted to use the string concatenation operator, which joins two strings together into a single one:

Example: Our erroneous program, fixed

```
"hello" ++ " world"
```



An easy typo to make, but because you use Haskell, it was caught when you tried to compile. You didn't have to wait until you ran the program for the bug to become apparent.

This was only a simple example. However, the idea of types being a system to catch mistakes works on a much larger scale too. In general, when you make a change to your program, you'll change the type of one of the elements. If this change isn't something that you intended, then it will show up immediately. A lot of Haskell programmers remark that once they have fixed all the type errors in their programs, and their programs compile, that they tend to 'just work': function flawlessly first time, with only minor problems. *Run-time errors*, where your program goes wrong when you run it rather than when you compile it, are much rarer in Haskell than in other languages. This is a huge advantage of a strong type system like Haskell's.

Exercises

Infer the types of following functions:

1. `f x y = uppercase (x ++ y)`
2. `g (x,y) = fiveOrSix (isL x) - ord y`
3. `h x y = pairToInt (fst x,y) + snd x + length y`

FIXME more to come...

Notes

1. ↑ At least as far as types are concerned, but we're trying to avoid that word :)
2. ↑ More technically, `fst` and `snd` have types which limit them to pairs. It would be impossible to define projection functions on tuples in general, because they'd have to be able to accept tuples of different sizes, so the type of the function would vary.
3. ↑ In fact, these are one and the same concept in Haskell.
4. ↑ This isn't quite what `chr` and `ord` do, but that description fits our purposes well, and it's close enough.
5. ↑ To make things even more confusing, there's actually even more than one type for integers! Don't worry, we'll come on to this in due course.
6. ↑ This has been somewhat simplified to fit our purposes. Don't worry, the essence of the function is there.
7. ↑ Some of the newer type system extensions to GHC *do* break this, however, so you're better off just always putting down types anyway.
8. ↑ This is a slight lie. That type signature would mean that you can compare two values of any type whatsoever, but this clearly isn't true: how can you see if two functions are equal? Haskell includes a kind of 'restricted polymorphism' that allows type variables to range over some, but not all types. Haskell implements this using *type classes*, which we'll learn about later. In this case, the correct type of `(==)` is `Eq a => a -> a -> Bool`.

Simple input and output

So far this tutorial has discussed functions that return values, which is well and good. But how do we write "Hello world"? To give you a first taste of it, here is a small variant of the "Hello world" program:

Example: Hello! What is your name?

```
main = do
  putStrLn "Please enter your name: "
  name <- getLine
  putStrLn ("Hello, " ++ name ++ ", how are you?")
```



At the very least, what should be clear is that dealing with input and output (IO) in Haskell is not a lost cause! Functional languages have always had a problem with input and output because they require side effects.

Functions always have to return the same results for the same arguments. But how can a function "getLine" return the same value every time it is called? Before we give the solution, let's take a step back and think about the difficulties inherent in such a task.

Any IO library should provide a host of functions, containing (at a minimum) operations like:

- print a string to the screen
- read a string from a keyboard
- write data to a file
- read data from a file

There are two issues here. Let's first consider the initial two examples and think about what their types should be. Certainly the first operation (I hesitate to call it a "function") should take a `String` argument and produce something, but what should it produce? It could produce a unit `()`, since there is essentially no return value from printing a string. The second operation, similarly, should return a `String`, but it doesn't seem to require an argument.

We want both of these operations to be functions, but they are by definition not functions. The item that reads a string from the keyboard cannot be a function, as it will not return the same `String` every time. And if the first function simply returns `()` every time, then referential transparency tells us we should have no problem with replacing it with a function `ε _ = ()`. But clearly this does not have the desired effect.

Actions

The breakthrough for solving this problem came when Phil Wadler realized that monads would be a good way to think about IO computations. In fact, monads are able to express much more than just the simple operations described above; we can use them to express a variety of constructions like concurrence, exceptions, IO, non-determinism and much more. Moreover, there is nothing special about them; they can be defined *within* Haskell with no special handling from the compiler (though compilers often choose to optimize monadic operations). Monads also have a somewhat undeserved reputation of being difficult to understand. So we're going to leave things at that -- knowing simply that IO somehow makes use of monads without necessarily understanding the gory details behind them (they really aren't so gory). So for now, we can forget that monads even exist.

As pointed out before, we cannot think of things like "print a string to the screen" or "read data from a file" as functions, since they are not (in the pure mathematical sense). Therefore, we give them another name: *actions*. Not only do we give them a special name, we give them a special type. One particularly useful action is `putStrLn`, which prints a string to the screen. This action has type:

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

As expected, `putStrLn` takes a string argument. What it returns is of type `IO ()`. This means that this function is actually an action (that is what the `IO` means). Furthermore, when this action is *evaluated* (or "run"), the result will have type `()`.

Note

Actually, this type means that `putStrLn` is an action "within the IO monad", but we will

gloss over this for now.

You can probably already guess the type of `getLine`:

```
-----
getLine :: IO String
-----
```

This means that `getLine` is an IO action that, when run, will have type `String`.

The question immediately arises: "how do you 'run' an action?". This is something that is left up to the compiler. You cannot actually run an action yourself; instead, a program is, itself, a single action that is run when the compiled program is executed. Thus, the compiler requires that the `main` function have type `IO ()`, which means that it is an IO action that returns nothing. The compiled code then executes this action.

However, while you are not allowed to run actions yourself, you *are* allowed to combine actions. There are two ways to go about this. The one we will focus on in this chapter is the **do** notation, which provides a convenient means of putting actions together, and allows us to get useful things done in Haskell without having to understand what *really* happens. Lurking behind the do notation is the more explicit approach using the `(>>=)` operator, but we will not be ready to cover this until the chapter Understanding monads.

Note

Do notation is just syntactic sugar for `(>>=)`. If you have experience with higher order functions, it might be worth starting with the latter approach and coming back here to see how do notation gets used.

Let's consider the following name program:

Example: What is your name?

```
-----
main = do
  putStrLn "Please enter your name: "
  name <- getLine
  putStrLn ("Hello, " ++ name ++ ", how are you?")
-----
```



We can consider the **do** notation as a way to combine a sequence of actions. Moreover, the `<-` notation is a way to get the value out of an action. So, in this program, we're sequencing three actions: a `putStrLn`, a `getLine` and another `putStrLn`. The `putStrLn` action has type `String -> IO ()`, so we provide it a `String`, so the fully applied action has type `IO ()`. This is something that we are allowed to run as a program.

Exercises

Write a program which asks the user for the base and height of a triangle,

calculates its area and prints it to the screen. The interaction should look something like:

```

The base?
3.3
The height?
5.4
The area of that triangle is 8.91

```

Hint: you can use the function `read` to convert user strings like "3.3" into numbers like 3.3 and function `show` to convert a number into string.

Left arrow clarifications

The `<-` is optional

While we are allowed to get a value out of certain actions like `getLine`, we certainly are not obliged to do so. For example, we could very well have written something like this:

Example: executing `getLine` directly

```

main = do
  putStrLn "Please enter your name: "
  getLine
  putStrLn ("Hello, how are you?")

```



Clearly, that isn't very useful: the whole point of prompting the user for his or her name was so that we could do something with the result. That being said, it is conceivable that one might wish to read a line and completely ignore the result. Omitting the `<-` will allow for that; the action will happen, but the data won't be stored anywhere.

In order to get the value out of the action, we write `name <- getLine`, which basically means "run `getLine`, and put the results in the variable called `name`."

The `<-` can be used with any action (except the last)

On the flip side, there are also very few restrictions which actions can have values gotten out of them. Consider the following example, where we put the results of each action into a variable (except the last... more on that later):

Example: putting all results into a variable

```

main = do
  x <- putStrLn "Please enter your name: "
  name <- getLine
  putStrLn ("Hello, " ++ name ++ ", how are you?")

```



The variable `x` gets the value out of its action, but that isn't very interesting because the action returns the unit value `()`. So while we could technically get the value out of any action, it isn't always worth it. But wait, what about that last action? Why can't we get a value out of that? Let's see what happens when we try:

Example: getting the value out of the last action



```
main = do
  x <- putStrLn "Please enter your name: "
  name <- getLine
  y <- putStrLn ("Hello, " ++ name ++ ", how are you?")
```

Whoops!

```
YourName.hs:5:2:
  The last statement in a 'do' construct must be an expression
```

This is a much more interesting example, but it requires a somewhat deeper understanding of Haskell than we currently have. Suffice it to say, whenever you use `<-` to get the value of an action, Haskell is always expecting another action to follow it. So the very last action better not have any `<-`s.

Controlling actions

Normal Haskell constructions like **if/then/else** and **case/of** can be used within the **do** notation, but you need to be somewhat careful. For instance, in a simple "guess the number" program, we have:

```
doGuessing num = do
  putStrLn "Enter your guess:"
  guess <- getLine
  if (read guess) < num
    then do putStrLn "Too low!"
           doGuessing num
    else if (read guess) > num
          then do putStrLn "Too high!"
                 doGuessing num
          else do putStrLn "You Win!"
```

If we think about how the **if/then/else** construction works, it essentially takes three arguments: the condition, the "then" branch, and the "else" branch. The condition needs to have type `Bool`, and the two branches can have any type, provided that they have the *same* type. The type of the entire **if/then/else** construction is then the type of the two branches.

In the outermost comparison, we have `(read guess) < num` as the condition. This clearly has the correct type. Let's just consider the "then" branch. The code here is:

```
do putStrLn "Too low!"
  doGuessing num
```

Here, we are sequencing two actions: `putStrLn` and `doGuessing`. The first has type `IO ()`, which is fine. The second also has type `IO ()`, which is fine. The type result of the entire computation is precisely the type of the final computation. Thus, the type of the "then" branch is also `IO ()`. A similar argument shows that the type of

the "else" branch is also `IO ()`. This means the type of the entire `if/then/else` construction is `IO ()`, which is just what we want.

Note

In this code, the last line is `else do putStrLn "You Win!"`. This is somewhat overly verbose. In fact, `else putStrLn "You Win!"` would have been sufficient, since `do` is only necessary to sequence actions. Since we have only one action here, it is superfluous.

It is *incorrect* to think to yourself "Well, I already started a `do` block; I don't need another one," and hence write something like:

```
do if (read guess) < num
    then putStrLn "Too low!"
      doGuessing num
    else ...
```

Here, since we didn't repeat the `do`, the compiler doesn't know that the `putStrLn` and `doGuessing` calls are supposed to be sequenced, and the compiler will think you're trying to call `putStrLn` with three arguments: the string, the function `doGuessing` and the integer `num`. It will certainly complain (though the error may be somewhat difficult to comprehend at this point).

We can write the same `doGuessing` function using a `case` statement. To do this, we first introduce the Prelude function `compare`, which takes two values of the same type (in the `Ord` class) and returns one of `GT`, `LT`, `EQ`, depending on whether the first is greater than, less than or equal to the second.

```
doGuessing num = do
  putStrLn "Enter your guess:"
  guess <- getLine
  case compare (read guess) num of
    LT -> do putStrLn "Too low!"
           doGuessing num
    GT -> do putStrLn "Too high!"
           doGuessing num
    EQ -> putStrLn "You Win!"
```

Here, again, the `dos` after the `->`s are necessary on the first two options, because we are sequencing actions.

If you're used to programming in an imperative language like C or Java, you might think that `return` will exit you from the current function. This is not so in Haskell. In Haskell, `return` simply takes a normal value (for instance, one of type `Int`) and makes it into an action that returns the given value (for the same example, the action would be of type `IO Int`). In particular, in an imperative language, you might write this function as:

```

void doGuessing(int num) {
    print "Enter your guess:";
    int guess = atoi(readLine());
    if (guess == num) {
        print "You win!";
        return ();
    }

    // we won't get here if guess == num
    if (guess < num) {
        print "Too low!";
        doGuessing(num);
    } else {
        print "Too high!";
        doGuessing(num);
    }
}

```

Here, because we have the `return ()` in the first `if` match, we expect the code to exit there (and in most imperative languages, it does). However, the equivalent code in Haskell, which might look something like:

```

doGuessing num = do
    putStrLn "Enter your guess:"
    guess <- getLine
    case compare (read guess) num of
        EQ -> do putStrLn "You win!"
              return ()

    -- we don't expect to get here unless guess == num
    if (read guess < num)
        then do print "Too low!";
              doGuessing
        else do print "Too high!";
              doGuessing

```

First of all, if you guess correctly, it will first print "You win!," but it won't exit, and it will check whether `guess` is less than `num`. Of course it is not, so the `else` branch is taken, and it will print "Too high!" and then ask you to guess again.

On the other hand, if you guess incorrectly, it will try to evaluate the case statement and get either `LT` or `GT` as the result of the `compare`. In either case, it won't have a pattern that matches, and the program will fail immediately with an exception.

Exercises

What does the following program print out?

```

main =
do x <- getX
  putStrLn x

getX =
do return "hello"
  return "aren't"
  return "these"
  return "returns"
  return "rather"
  return "pointless?"

```

Why?

Exercises

Write a program that asks the user for his or her name. If the name is one of Simon, John or Phil, tell the user that you think Haskell is a great programming language. If the name is Koen, tell them that you think debugging Haskell is fun (Koen Classen is one of the people who works on Haskell debugging); otherwise, tell the user that you don't know who he or she is.

Write two different versions of this program, one using `if` statements, the other using a `case` statement.

Actions under the microscope

Actions may look easy up to now, but they are actually a common stumbling block for new Haskellers. If you have run into trouble working with actions, you might consider looking to see if one of your problems or questions matches the cases below. It might be worth skimming this section now, and coming back to it when you actually experience trouble.

Mind your action types

One temptation might be to simplify our program for getting a name and printing it back out. Here is one unsuccessful attempt:

Example: Why doesn't this work?

```
main =
do putStrLn "What is your name? "
  putStrLn ("Hello " ++ getLine)
```

Ouch!

```
YourName.hs:3:26:
    Couldn't match expected type `[Char]'
```

```
    against inferred type `IO String'
```



Let us boil the example above to its simplest form. Would you expect this program to compile?

Example: This still does not work

```
main =
do putStrLn getLine
```



For the most part, this is the same (attempted) program, except that we've stripped off the superfluous "What is

your name" prompt as well as the polite "Hello". One trick to understanding this is to reason about it in terms of types. Let us compare:

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

We can use the same mental machinery we learned in Type basics to figure how everything went wrong. Simply put, `putStrLn` is expecting a `String` as input. We do not have a `String`, but something tantalisingly close, an `IO String`. This represents an action that will *give* us a `String` when it's run. To obtain the `String` that `putStrLn` wants, we need to run the action, and we do that with the ever-handy left arrow, `<-`.

Example: This time it works



```
main =
do name <- getLine
  putStrLn name
```

Working our way back up to the fancy example:

```
main =
do putStrLn "What is your name? "
  name <- getLine
  putStrLn ("Hello " ++ name)
```

Now the name is the `String` we are looking for and everything is rolling again.

Mind your expression types too

Fine, so we've made a big deal out of the idea that you can't use actions in situations that don't call for them. The converse of this is that you can't use non-actions in situations that **DO** expect actions. Say we want to greet the user, but this time we're so excited to meet them, we just have to **SHOUT** their name out:

Example: Exciting but incorrect. Why?



```
import Data.Char (toUpper)

main =
do name <- getLine
  loudName <- makeLoud name
  putStrLn ("Hello " ++ loudName ++ "!")
  putStrLn ("Oh boy! Am I excited to meet you, " ++ loudName)

-- Don't worry too much about this function; it just capitalises a String
makeLoud :: String -> String
makeLoud s = map toUpper s
```

This goes wrong...

```

Couldn't match expected type `IO' against inferred type `[]'
  Expected type: IO t
  Inferred type: String
In a 'do' expression: loudName <- makeLoud name

```

This is quite similar to the problem we ran into above: we've got a mismatch between something that is expecting an `IO` type, and something which is not. This time, the cause is our use of the left arrow `<-`; we're trying to left arrow a value of `makeLoud name`, which really isn't left arrow material. It's basically the same mismatch we saw in the previous section, except now we're trying to use regular old `String` (the loud name) as an `IO String`, which clearly are not the same thing. The latter is an action, something to be run, whereas the former is just an expression minding its own business. Note that we cannot simply use `loudName = makeLoud name` because a `do` sequences *actions*, and `loudName = makeLoud name` is not an action.

So how do we extricate ourselves from this mess? We have a number of options:

- We could find a way to turn `makeLoud` into an action, to make it return `IO String`. But this is not desirable, because the whole point of functional programming is to cleanly separate our side-effecting stuff (actions) from the pure and simple stuff. For example, what if we wanted to use `makeLoud` from some other, non-`IO`, function? An `IO makeLoud` is certainly possible (how?), but missing the point entirely.
- We could use `return` to promote the loud name into an action, writing something like `loudName <- return (makeLoud name)`. This is slightly better, in that we are at least leaving the `makeLoud` itself function nice and `IO`-free, whilst using it in an `IO`-compatible fashion. But it's still moderately clunky, because by virtue of left arrow, we're implying that there's action to be had -- how exciting! -- only to let our reader down with a somewhat anticlimatic `return`
- Or we could use a `let` binding...

It turns out that Haskell has a special extra-convenient syntax for `let` bindings in actions. It looks a little like this:

Example: `let` bindings in `do` blocks.

```

main =
do name <- getLine
  let loudName = makeLoud name
  putStrLn ("Hello " ++ loudName ++ "!")
  putStrLn ("Oh boy! Am I excited to meet you, " ++ loudName)

```



If you're paying attention, you might notice that the `let` binding above is missing an `in`. This is because `let` bindings in `do` blocks do not require the `in` keyword. You could very well use it, but then you'd have to make a mess of your `do` blocks. For what it's worth, the following two blocks of code are equivalent.

sweet	unsweet
<pre> do name <- getLine let loudName = makeLoud name putStrLn ("Hello " ++ loudName ++ "!") putStrLn ("Oh boy! Am I excited to meet you, " ++ loudName) </pre>	<pre> do name <- getLine let loudName = makeLoud name in do putStrLn ("Hello " ++ loudName ++ "!") putStrLn ("Oh boy! Am I excited to meet you, " ++ loudName) </pre>

Exercises

1. Why does the unsweet version of the `let` binding require an extra `do` keyword?
2. Do you always need the extra `do`?
3. (extra credit) Curiously, `let` without `in` is exactly how we wrote things when we were playing with the interpreter in the beginning of this book. Why can you omit the `in` keyword in the interpreter, when you'd have to put it in when typing up a source file?

Learn more

At this point, you should have the skills you need to do some fancier input/output. Here are some IO-related options to consider.

- You could continue the sequential track, by learning more about types and eventually monads.
- Alternately: you could start learning about building graphical user interfaces in the GUI chapter
- For more IO-related functionality, you could also consider learning more about the `System.IO` library

Type declarations

Haskell has three basic ways to declare a new type:

- The `data` declaration for structures and enumerations.
- The `type` declaration for type synonyms.
- The `newtype` declaration, which is a cross between the other two.

In this chapter, we will focus on the most essential way, `data`, and to make life easier, `type`. You'll find out about `newtype` later on, but don't worry too much about it; it's there mainly for optimisation.

`data` for making your own types

Here is a data structure for a simple list of anniversaries:

```
data Anniversary =  
  Birthday String Int Int Int      -- Name, year, month, day  
  | Wedding String String Int Int Int -- First partner's name, second partner's name, year, month, day
```

This declares a new data type `Anniversary` with two *constructor* functions called `Birthday` and `Wedding`. As usual with Haskell the case of the first letter is important: type names and constructor functions must always start with capital letters. Note also the vertical bar: this marks the point where one alternative ends and the next begins; you can think of it almost as an or - which you'll remember was `||` - except used in types.

The declaration says that an `Anniversary` can be one of two things; a `Birthday` or a `Wedding`. A `Birthday` contains one string and three integers, and a `Wedding` contains two strings and three integers. The comments (after the "--") explain what the fields actually mean.

Now we can create new anniversaries by calling the constructor functions. For example, suppose we have John Smith born on 3rd July 1968:

```
johnSmith :: Anniversary
johnSmith = Birthday "John Smith" 1968 7 3
```

He married Jane Smith on 4th March 1987:

```
smithWedding :: Anniversary
smithWedding = Wedding "John Smith" "Jane Smith" 1987 3 4
```

These two objects can now be put in a list:

```
anniversaries :: [Anniversary]
anniversaries = [johnSmith, smithWedding]
```

(Obviously a real application would not hard-code its entries: this is just to show how constructor functions work).

Constructor functions can do all of the things ordinary functions can do. Anywhere you could use an ordinary function you can use a constructor function.

Anniversaries will need to be converted into strings for printing. This needs another function:

```
showAnniversary :: Anniversary -> String
showAnniversary (Birthday name year month day) =
  name ++ " born " ++ showDate year month day
showAnniversary (Wedding name1 name2 year month day) =
  name1 ++ " married " ++ name2 ++ " " ++ showDate year month day
```

This shows the one way that constructor functions are special: they can also be used to deconstruct objects. `showAnniversary` takes an argument of type `Anniversary`. If the argument is a `Birthday` then the first version gets used, and the variables `name`, `month`, `date` and `year` are bound to its contents. If the argument is a `Wedding` then the second version is used and the arguments are bound in the same way. The brackets indicate that the whole thing is one argument split into five or six parts, rather than five or six separate arguments.

Notice the relationship between the type and the constructors. All versions of `showAnniversary` convert an anniversary to a string. One of them handles the `Birthday` case and the other handles the `Wedding` case.

It also needs an additional `showDate` routine:


```
showDate y m d = show y ++ "-" ++ show m ++ "-" ++ show d
```

Of course, it's a bit clumsy having the date passed around as three separate integers. What we really need is a new datatype:

```
data Date = Date Int Int Int -- Year, Month, Day
```

Constructor functions are allowed to be the same name as the type, and if there is only one then it is good practice to make it so.

type for making type synonyms

It would also be nice to make it clear that the strings in the `Anniversary` type are names, but still be able to manipulate them like ordinary strings. The `type` declaration does this:

```
type Name = String
```

This says that a `Name` is a synonym for a `String`. Any function that takes a `String` will now take a `Name` as well, and vice versa. The right hand side of a `type` declaration can be a more complex type as well. For example `String` itself is defined in the standard libraries as

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

So now we can rewrite the `Anniversary` type like this:

```
data Anniversary =  
  Birthday Name Date  
  | Wedding Name Name Date
```

which is a lot easier to read. We can also have a type for the list:

```
type AnniversaryBook = [Anniversary]
```

The rest of the code needs to be changed to match:

```

johnSmith :: Anniversary
johnSmith = Birthday "John Smith" (Date 1968 7 3)

smithWedding :: Anniversary
smithWedding = Wedding "John Smith" "Jane Smith" (Date 1987 3 4)

anniversaries :: AnniversaryBook
anniversaries = [johnSmith, smithWedding]

showAnniversary :: Anniversary -> String
showAnniversary (Birthday name date) =
  name ++ " born " ++ showDate date
showAnniversary (Wedding name1 name2 date) =
  name1 ++ " married " ++ name2 ++ showDate date

showDate :: Date -> String
showDate (Date y m d) = show y ++ "-" show m ++ "-" ++ show d

```

Elementary Haskell

Recursion

Recursion is a clever idea which plays a central role in Haskell (and computer science in general): namely, recursion is the idea of using a given function as part of its own definition. A function defined in this way is said to be **recursive**. It might sound like this always leads to infinite regress, but if done properly it doesn't have to.

Generally speaking, a recursive definition comes in two parts. First, there are one or more *base cases* which say what to do in simple cases where no recursion is necessary (that is, when the answer can be given straight away without recursively calling the function being defined). This ensures that the recursion can eventually stop. The *recursive case* is more general, and defines the function in terms of a 'simpler' call to itself. Let's look at a few examples.

Numeric recursion

The factorial function

In mathematics, especially combinatorics, there is a function used fairly frequently called the **factorial** function ^[9]. It takes a single number as an argument, finds all the numbers between one and this number, and multiplies them all together. For example, the factorial of 6 is $1 \times 2 \times 3 \times 4 \times 5 \times 6 = 720$. This is an interesting function for us, because it is a candidate to be written in a recursive style.

The idea is to look at the factorials of adjacent numbers:

Example: Factorials of adjacent numbers

```
Factorial of 6 = 6 × 5 × 4 × 3 × 2 × 1
Factorial of 5 =     5 × 4 × 3 × 2 × 1
```

Notice how we've lined things up. You can see here that the factorial of 6 involves the factorial of 5. In fact, the factorial of 6 is just $6 \times$ (factorial of 5). Let's look at some more examples:

Example: Factorials of adjacent numbers

```
Factorial of 3 = 3 × 2 × 1
Factorial of 2 =     2 × 1

Factorial of 8 = 8 × 7 × 6 × 5 × 4 × 3 × 2 × 1
Factorial of 7 =     7 × 6 × 5 × 4 × 3 × 2 × 1
```

Indeed, we can see that the factorial of any number is just that number multiplied by the factorial of the number one less than it. There's one exception to this: if we ask for the factorial of 0, we don't want to multiply 0 by the factorial of -1! In fact, we just say the factorial of 0 is 1 (we *define* it to be so. It just is, okay?^[10]). So, 0 is the *base case* for the recursion: when we get to 0 we can immediately say that the answer is 1, without using recursion. We can summarize the definition of the factorial function as follows:

- The factorial of 0 is 1.
- The factorial of any other number is that number multiplied by the factorial of the number one less than it.

We can translate this directly into Haskell:

Example: Factorial function

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

This defines a new function called `factorial`. The first line says that the factorial of 0 is 1, and the second one says that the factorial of any other number `n` is equal to `n` times the factorial of `n-1`. Note the parentheses around the `n-1`: without them this would have been parsed as `(factorial n) - 1`; function application (applying a function to a value) will happen before anything else does (we say that function application *binds more tightly* than anything else).

This all seems a little voodoo so far. How does it work? Well, let's look at what happens when you execute `factorial 3`:

- 3 isn't 0, so we *recur*: work out the factorial of 2
 - 2 isn't 0, so we recur.
 - 1 isn't 0, so we recur.
 - 0 is 0, so we return 1.

- We multiply the current number, 1, by the result of the recursion, 1, obtaining 1 (1×1).
- We multiply the current number, 2, by the result of the recursion, 1, obtaining 2 ($2 \times 1 \times 1$).
- We multiply the current number, 3, by the result of the recursion, obtaining 6 ($3 \times 2 \times 1 \times 1$).

We can see how the multiplication 'builds up' through the recursion.

(Note that we end up with the one appearing twice, since the base case is 0 rather than 1; but that's okay since multiplying by one has no effect. We could have designed `factorial` to stop at 1 if we had wanted to, but it's conventional, and often useful, to have the factorial of 0 defined.)

One more thing to note about the recursive definition of `factorial`: the order of the two declarations (one for `factorial 0` and one for `factorial n`) is important. Haskell decides which function definition to use by starting at the top and picking the first one that matches. In this case, if we had the general case (`factorial n`) before the 'base case' (`factorial 0`), then the general `n` would match *anything* passed into it -- including 0. So `factorial 0` would match the general `n` case, the compiler would conclude that `factorial 0` equals $0 * \text{factorial } (-1)$, and so on to negative infinity. Definitely not what we want. The lesson here is that one should always list multiple function definitions starting with the most specific and proceeding to the most general.

Exercises

1. Type the factorial function into a Haskell source file and load it into your favourite Haskell environment.
 - What is `factorial 5`?
 - What about `factorial 1000`? If you have a scientific calculator (that isn't your computer), try it there first. Does Haskell give you what you expected?
 - What about `factorial (-1)`? Why does this happen?
2. The *double factorial* of a number `n` is the product of *every other* number from 1 (or 2) up to `n`. For example, the double factorial of 8 is $8 \times 6 \times 4 \times 2 = 384$, and the double factorial of 7 is $7 \times 5 \times 3 \times 1 = 105$. Define a `doublefactorial` function in Haskell.

A quick aside

This section is aimed at people who are used to more imperative-style languages like C and Java.

Loops are the bread and butter of imperative languages. For example, the idiomatic way of writing a factorial function in an imperative language would be to use a *for* loop, like the following (in C):

Example: The factorial function in an imperative language

```
int factorial(int n) {
    int res = 1;
    for (i = 1; i <= n; i++)
        res *= i;
    return res;
}
```



This isn't directly possible in Haskell, since changing the value of the variables `res` and `i` (a destructive update) would not be allowed. However, you can always translate a loop into an equivalent recursive form. The idea is to make each loop variable in need of updating into a parameter of a recursive function. For example, here is a direct 'translation' of the above loop into Haskell:

Example: Using recursion to simulate a loop



```
factorial n = factorialWorker 1 n 1
factorialWorker i n res | i <= n    = factorialWorker (i+1) n (res * i)
                       | otherwise = res
```

The expressions after the vertical bars are called *guards*, and we'll learn more about them in the section on control structures. For now, you can probably figure out how they work by comparing them to the corresponding C code above.

Obviously this is not the shortest or most elegant way to implement `factorial` in Haskell (translating directly from an imperative paradigm into Haskell like this rarely is), but it can be nice to know that this sort of translation is always possible.

Another thing to note is that you shouldn't be worried about poor performance through recursion with Haskell. In general, functional programming compilers include a lot of optimization for recursion, including one important one called *tail-call optimisation*; remember too that Haskell is lazy -- if a calculation isn't needed, it won't be done. We'll learn about these in later chapters.

Other recursive functions

As it turns out, there is nothing particularly special about the `factorial` function; a great many numeric functions can be defined recursively in a natural way. For example, let's think about multiplication. When you were first introduced to multiplication (remember that moment? :)), it may have been through a process of 'repeated addition'. That is, 5×4 is the same as summing four copies of the number 5. Of course, summing four copies of 5 is the same as summing three copies, and then adding one more -- that is, $5 \times 4 = 5 \times 3 + 5$. This leads us to a natural recursive definition of multiplication:

Example: Multiplication defined recursively



```
mult n 0 = 0                -- anything times 0 is zero
mult n 1 = n                -- anything times 1 is itself
mult n m = (mult n (m - 1)) + n  -- recur: multiply by one less, and add an ext
```

Stepping back a bit, we can see how numeric recursion fits into the general recursive pattern. The base case for numeric recursion usually consists of one or more specific numbers (often 0 or 1) for which the answer can be immediately given. The recursive case computes the result by recursively calling the function with a smaller argument and using the result somehow to produce the final answer. The 'smaller argument' used is often one less than the current argument, leading to recursion which 'walks down the number line' (like the examples of `factorial` and `mult` above), but it doesn't have to be; the smaller argument could be produced in some other way as well.

Exercises

1. Expand out the multiplication 5×4 similarly to the expansion we used above for `factorial 3`.
2. Define a recursive function `power` such that `power x y` raises `x` to the `y` power.
3. You are given a function `plusOne x = x + 1`. Without using any other `(+)`s, define a recursive function `addition` such that `addition x y` adds `x` and `y` together.
4. (Harder) Implement the function `log2`, which computes the integer log (base 2) of its argument. That is, `log2` computes the exponent of the largest power of 2 which is less than or equal to its argument. For example, `log2 16 = 4`, `log2 11 = 3`, and `log2 1 = 0`. (Small hint: read the last phrase of the paragraph immediately preceding these exercises.)

List-based recursion

A *lot* of functions in Haskell turn out to be recursive, especially those concerning lists.^[11] Consider the `length` function that finds the length of a list:

Example: The recursive definition of `length`

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



Don't worry too much about the syntax; we'll learn more about it in the section on Pattern matching. For now, let's rephrase this code into English to get an idea of how it works. The first line gives the type of `length`: it takes any sort of list and produces an `Int`. The next line says that the length of an empty list is 0. This, of course, is the base case. The final line is the recursive case: if a list consists of a first element `x` and another list `xs` representing the rest of the list, the length of the list is one more than the length of `xs`.

How about the concatenation function `(++)`, which joins two lists together? (Some examples of usage are also given, as we haven't come across this function so far.)

Example: The recursive `(++)`



```

Prelude> [1,2,3] ++ [4,5,6]
'[1,2,3,4,5,6]
Prelude> "Hello " ++ "world" -- Strings are lists of Chars
'"Hello world"

(++ ) :: [a] -> [a] -> [a]
[] ++ ys      = ys
(x:xs) ++ ys = x : xs ++ ys

```

This is a little more complicated than `length` but not too difficult once you break it down. The type says that `(++)` takes two lists and produces another. The base case says that concatenating the empty list with a list `ys` is the same as `ys` itself. Finally, the recursive case breaks the first list into its head (`x`) and tail (`xs`) and says that to concatenate the two lists, concatenate the tail of the first list with the second list, and then tack the head `x` on the front.

There's a pattern here: with list-based functions, the base case usually involves an empty list, and the recursive case involves passing the tail of the list to our function again, so that the list becomes progressively smaller.

Exercises

Give recursive definitions for the following list-based functions. In each case, think what the base case would be, then think what the general case would look like, in terms of everything smaller than it.

1. `replicate :: Int -> a -> [a]`, which takes an element and a count and returns the list which is that element repeated that many times. E.g. `replicate 3 'a' = "aaa"`. (Hint: think about what `replicate` of anything with a count of 0 should be; a count of 0 is your 'base case'.)
2. `(!!) :: [a] -> Int -> a`, which returns the element at the given 'index'. The first element is at index 0, the second at index 1, and so on. Note that with this function, you're recurring *both* numerically and down a list.
3. (A bit harder.) `zip :: [a] -> [b] -> [(a, b)]`, which takes two lists and 'zips' them together, so that the first pair in the resulting list is the first two elements of the two lists, and so on. E.g. `zip [1,2,3] "abc" = [(1, 'a'), (2, 'b'), (3, 'c')]`. If either of the lists is shorter than the other, you can stop once either list runs out. E.g. `zip [1,2] "abc" = [(1, 'a'), (2, 'b')]`.

Recursion is used to define nearly all functions to do with lists and numbers. The next time you need a list-based algorithm, start with a case for the empty list and a case for the non-empty list and see if your algorithm is recursive.

Don't get TOO excited about recursion...

Although it's very important to have a solid understanding of recursion when programming in Haskell, one rarely has to write functions that are explicitly recursive. Instead, there are all sorts of standard library functions

which perform recursion for you in various ways, and one usually ends up using those instead. For example, a much simpler way to implement the `factorial` function is as follows:

Example: Implementing factorial with a standard library function



```
factorial n = product [1..n]
```

Almost seems like cheating, doesn't it? :) This is the version of `factorial` that most experienced Haskell programmers would write, rather than the explicitly recursive version we started out with. Of course, the `product` function is using some list recursion behind the scenes^[12], but writing `factorial` in this way means you, the programmer, don't have to worry about it.

Summary

Recursion is the practise of using a function you're defining in the body of the function itself. It nearly always comes in two parts: a base case and a recursive case. Recursion is especially useful for dealing with list- and number-based functions.

Notes

- ↑ At least as far as types are concerned, but we're trying to avoid that word :)
- ↑ More technically, `fst` and `snd` have types which limit them to pairs. It would be impossible to define projection functions on tuples in general, because they'd have to be able to accept tuples of different sizes, so the type of the function would vary.
- ↑ In fact, these are one and the same concept in Haskell.
- ↑ This isn't quite what `chr` and `ord` do, but that description fits our purposes well, and it's close enough.
- ↑ To make things even more confusing, there's actually even more than one type for integers! Don't worry, we'll come on to this in due course.
- ↑ This has been somewhat simplified to fit our purposes. Don't worry, the essence of the function is there.
- ↑ Some of the newer type system extensions to GHC *do* break this, however, so you're better off just always putting down types anyway.
- ↑ This is a slight lie. That type signature would mean that you can compare two values of any type whatsoever, but this clearly isn't true: how can you see if two functions are equal? Haskell includes a kind of 'restricted polymorphism' that allows type variables to range over some, but not all types. Haskell implements this using *type classes*, which we'll learn about later. In this case, the correct type of `(=)` is `Eq a => a -> a -> Bool`.
- ↑ In mathematics, $n!$ normally means the factorial of n , but that syntax is impossible in Haskell, so we don't use it here.
- ↑ Actually, defining the factorial of 0 to be 1 is not just arbitrary; it's because the factorial of 0 represents an empty product.
- ↑ This is no coincidence; without mutable variables, recursion is the only way to implement control structures. This might sound like a limitation until you get used to it (it isn't, really).
- ↑ Actually, it's using a function called `foldl`, which actually does the recursion.

Pattern matching

Pattern matching is a convenient way to *bind* variables to different parts of a given value.

Note

Pattern matching on what?

Some languages like Perl and Python use pattern matching in a very specific way, that is to match regular expressions against strings. The pattern matching we are referring to in this chapter is quite different. In fact, you're probably best of forgetting what you know about pattern matching for now. Here, pattern matching is used in the same way as in others ML-like languages : to deconstruct values according to their type specification.

What is pattern matching?

You've actually met pattern matching before, in the lists chapter. Recall functions like `map`:

```
map _ [] = []
map f (x:xs) = f x : map f xs
```

Here there are four different patterns going on: two per equation. Let's explore each one in turn (although not in the order they appeared in that example):

- `[]` is a pattern that matches *the empty list*. It doesn't bind any variables.
- `(x:xs)` is a pattern that matches something (which gets bound to `x`), which is cons'd, using the function `(:)`, onto something else (which gets bound to the variable `xs`).
- `f` is a pattern which matches *anything at all*, and binds `f` to that something.
- `_` is the pattern which matches anything at all, but doesn't do any binding.

So pattern matching is a way of *assigning names to things* (or *binding* those names to those things), and possibly *breaking down expressions into subexpressions* at the same time (as we did with the list in the definition of `map`).

However, you can't pattern match with anything. For example, you might want to define a function like the following to chop off the first three elements of a list:

```
dropThree ([x,y,z] ++ xs) = xs
```

However, that **won't work**, and will give you an error. The problem is that the function `(++)` *isn't allowed* in patterns. So what *is* allowed?

The one-word answer is **constructors**. Recall algebraic datatypes, which look something like:

```
data Foo = Bar | Baz Int
```

Here `Bar` and `Baz` are *constructors* for the type `Foo`. And so you can pattern match with them:

```
f :: Foo -> Int
f Bar      = 1
f (Baz x)  = x - 1
```

Remember that lists are defined thusly (note that the following isn't actually valid syntax: lists are in reality deeply grained into Haskell):

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

So the empty list, `[]`, and the `(:)` function, are in reality constructors of the list datatype, so you can pattern match with them.

Note, however, that as `[x, y, z]` is just syntactic sugar for `x:y:z:[]`, you can still pattern match using the latter form:

```
dropThree (_:_:_:xs) = xs
```

If the only relevant information is the type of the constructor (regardless of the number of its elements) the `{}` pattern can be used:

```
g :: Foo -> Bool
g Bar {} = True
g Baz {} = False
```

The function `g` does not have to be changed when the number of elements of the constructors `Bar` or `Baz` changes. Note: `Foo` does not have to be a record for this to work.

For constructors with many elements, it can help to use records:

```
data Foo2 = Bar2 | Baz2 {barNumber::Int, barName::String}
```

which then allows:

```
h :: Foo2 -> Int
h Baz2 {barName=name} = length name
h Bar2 {} = 0
```

The one exception

There is one exception to the rule that you can only pattern match with constructors. It's known as $n+k$ patterns. It is indeed valid Haskell 98 to write something like:

```
pred :: Int -> Int
pred (n+1) = n
```

However, this is generally accepted as bad form and not many Haskell programmers like this exception, and so try to avoid it.

Where you can use it

The short answer is that *wherever you can bind variables, you can pattern match*. Let's have a look at that more precisely.

Equations

The first place is in the left-hand side of function equations. For example, our above code for `map`:

```
map _ [] = []
map f (x:xs) = f x : map f xs
```

Here we're binding, and doing pattern-matching, on the left hand side of both of these equations.

Let expressions / Where clauses

You can obviously bind variables with a `let` expression or `where` clause. As such, you can also do pattern matching here. A trivial example:

```
let Just x = lookup "bar" [("foo", 1), ("bar", 2), ("baz", 3)]
```

Case expressions

One of the most obvious places you can use pattern binding is on the left hand side of case branches:

```
case someRandomList of
[] -> "The list was empty"
(x:xs) -> "The list wasn't empty: the first element was " ++ x ++ ", and " ++
        "there were " ++ show (length xs) ++ " more elements in the list."
```

Lambdas

As lambdas can be easily converted into functions, you can pattern match on the left-hand side of lambda expressions too:

```
head = (\(x:xs) -> x)
```

Note that here, along with on the left-hand side of equations as described above, you have to use parentheses around your patterns (unless they're just `_` or are just a binding, not a pattern, like `x`).

List comprehensions

After the `|` in list comprehensions, you can pattern match. This is actually extremely useful. For example, the function `catMaybes` from `Data.Maybe` takes a list of `Maybes`, filters all the `Just xs`, and gets rid of all the `Just` wrappers. It's easy to write it using list comprehensions:

```

catMaybes :: [Maybe a] -> [a]
catMaybes ms = [ x | Just x <- ms ]

```

If the pattern match fails, it just moves on to the next element in `ms`. (More formally, as list comprehensions are just the list monad, a failed pattern match invokes `fail`, which is the empty list in this case, and so gets ignored.)

A few other places

That's mostly it, but there are one or two other places you'll find as you progress through the book. Here's a list in case you're very eager already:

- In `p <- x in do-blocks`, `p` can be a pattern.
- Similarly, with `let` bindings in `do-blocks`, you can pattern match analogously to 'real' `let` bindings.

Exercises

1. If you have programmed in a language like Perl and Python before, how does pattern matching in Haskell compare to the pattern matching you know? What can you use it on, where? In what sense can we think of Perl/Python pattern matching as being "more powerful" than the Haskell one, and vice versa? Are they even comparable? You may also be interested in looking at the Haskell `Text.Regex` (<http://www.haskell.org/ghc/docs/latest/html/libraries/regex-compat/Text-Regex.html>) library wrapper.

More about lists

By now we have seen the basic tools for working with lists. We can build lists up from the cons operator `(:)` and the empty list `[]` (see Lists and tuples if you are unsure about this); and we can take them apart by using a combination of Recursion and Pattern matching. In this chapter, we will delve a little deeper into the inner-workings and the use of Haskell lists. We'll discover a little bit of new notation and some characteristically Haskell-ish features like infinite lists and list comprehensions. But before going into this, let us step back for a moment and combine the things we have already learned about lists.

Constructing Lists

We'll start by making a function to double every element of a list of integers. First, we must specify the type declaration for our function. For our purposes here, the function maps a list of integers to another list of integers:

```

doubleList :: [Integer] -> [Integer]

```

Then, we must specify the function definition itself. We'll be using a recursive definition, which consists of

1. the general case which iteratively generates a successive and simpler general case and
2. the base case, where iteration stops.

```
doubleList (n:ns) = (n * 2) : doubleList ns
doubleList [] = []
```

Since by definition, there are no more elements beyond the end of a list, intuition tells us iteration must stop at the end of the list. The easiest way to accomplish this is to return the null list: As a constant, it halts our iteration. As the empty list, it doesn't change the value of any list we append it to.

The general case requires some explanation. Remember that ":" is one of a special class of functions known as "constructors". The important thing about constructors is that they can be used to break things down as part of "pattern matching" on the left hand side of function definitions. In this case the argument passed to doubleList is broken down into the first element of the list (known as the "head") and the rest of the list (known as the "tail").

On the right hand side doubleList builds up a new list by using ":". It says that the first element of the result is twice the head of the argument, and the rest of the result is obtained by applying "doubleList" to the tail. Note the naming convention implicit in (n:ns). By appending an "s" to the element "n" we are forming its plural. The idea is that the head contains one item while the tail contains many, and so should be pluralised.

So what actually happens when we evaluate the following?

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

We can work this out longhand by substituting the argument into the function definition, just like schoolbook algebra:

```
doubleList 1:[2,3,4] = (1*2) : doubleList (2 : [3,4])
                    = (1*2) : (2*2) : doubleList (3 : [4])
                    = (1*2) : (2*2) : (3*2) : doubleList (4 : [])
                    = (1*2) : (2*2) : (3*2) : (4*2) : doubleList []
                    = (1*2) : (2*2) : (3*2) : (4*2) : []
                    = 2 : 4 : 6 : 8 : []
                    = [2, 4, 6, 8]
```

Notice how the definition for empty lists terminates the recursion. Without it, the Haskell compiler would have had no way to know what to do when it reached the end of the list.

Also notice that it would make no difference *when* we did the multiplications (unless one of them is an error or nontermination: we'll get to that later). If I had done them immediately it would have made absolutely no difference. This is an important property of Haskell: it is a "pure" functional programming language. Because evaluation order can never change the result, it is mostly left to the compiler to decide when to actually evaluate things. Haskell is a "lazy" evaluation language, so evaluation is usually deferred until the value is really needed, but the compiler is free to evaluate things sooner if this will improve efficiency. From the programmer's point of view evaluation order rarely matters (except in the case of infinite lists, of which more will be said shortly).

Of course a function to double a list has limited generality. An obvious generalization would be to allow multiplication by any number. That is, we could write a function "multiplyList" that takes a multiplicand as well as a list of integers. It would be declared like this:

```
multiplyList :: Integer -> [Integer] -> [Integer]
multiplyList _ [] = []
multiplyList m (n:ns) = (m*n) : multiplyList m ns
```

This example introduces the "`_`", which is used for a "don't care" argument; it will match anything, like `*` does in shells or `.*` in regular expressions. The multiplicand is not used for the null case, so instead of being bound to an unused argument name it is explicitly thrown away, by "setting" `_` to it. ("`_`" can be thought of as a write-only "variable".)

The type declaration needs some explanation. Hiding behind the rather odd syntax is a deep and clever idea. The "`->`" arrow is actually an operator for types, and is right associative. So if you add in the implied brackets the type definition is actually

```
multiplyList :: Integer -> ( [Integer] -> [Integer] )
```

Think about what this is saying. It means that "multiplyList" doesn't take two arguments. Instead it takes one (an Integer), and then returns *a new function*. This new function itself takes one argument (a list of Integers) and returns a new list of Integers. This process of functions taking one argument is called "currying", and is very important.

The new function can be used in a straightforward way:

```
evens = multiplyList 2 [1,2,3,4]
```

or it can do something which, in any other language, would be an error; this is partial function application and because we're using Haskell, we can write the following neat & elegant bits of code:

```
doubleList = multiplyList 2
evens = doubleList [1,2,3,4]
```

It may help you to understand if you put the implied brackets in the first definition of "evens":

```
evens = (multiplyList 2) [1,2,3,4]
```

In other words "multiplyList 2" returns a new function that is then applied to [1,2,3,4].

Dot Dot Notation

Haskell has a convenient shorthand for specifying a list containing a sequence of integers. Some examples are enough to give the flavor:

Code	Result
[1..10]	[1,2,3,4,5,6,7,8,9,10]
[2,4..10]	[2,4,6,8,10]
[5,4..1]	[5,4,3,2,1]
[1,3..10]	[1,3,5,7,9]

The same notation can be used for floating point numbers and characters as well. However, be careful with floating point numbers: rounding errors can cause unexpected things to happen. Try this:

```
[0,0.1 .. 1]
```

Similarly, there are limits to what kind of sequence can be written through dot-dot notation. You can't put in

```
[0,1,1,2,3,5,8..100]
```

and expect to get back the rest of the Fibonacci series, or put in the beginning of a geometric sequence like

```
[1,3,9,27..100]
```

Infinite Lists

One of the most mind-bending things about Haskell lists is that they are allowed to be *infinite*. For example, the following generates the infinite list of integers starting with 1:

```
[1..]
```

(If you try this in GHCi, remember you can stop an evaluation with C-c).

Or you could define the same list in a more primitive way by using a recursive function:

```
intsFrom n = n : intsFrom (n+1)
positiveInts = intsFrom 1
```

This works because Haskell uses lazy evaluation: it never actually evaluates more than it needs at any given moment. In most cases an infinite list can be treated just like an ordinary one. The program will only go into an infinite loop when evaluation would actually require all the values in the list. Examples of this include sorting or printing the entire list. However:

```
evens = doubleList [1..]
```

will define "evens" to be the infinite list [2,4,6,8....]. And you can pass "evens" into other functions, and it will all just work. See the exercise 4 below for an example of how to process an infinite list and then take the first few elements of the result.

Infinite lists are quite useful in Haskell. Often it's more convenient to define an infinite list and then take the first few items than to create a finite list. Functions that process two lists in parallel generally stop with the shortest, so making the second one infinite avoids having to find the length of the first. An infinite list is often a handy alternative to the traditional endless loop at the top level of an interactive program.

Exercises

Write the following functions and test them out. Don't forget the type declarations.

1. takeInt returns the first n items in a list. So takeInt 4 [11,21,31,41,51,61] returns [11,21,31,41]
2. dropInt drops the first n items in a list and returns the rest. so dropInt 3 [11,21,31,41,51] returns [41,51].
3. sumInt returns the sum of the items in a list.
4. scanSum adds the items in a list and returns a list of the running totals. So scanSum [2,3,4,5] returns [2,5,9,14]. Is there any difference between "scanSum (takeInt 10 [1..])" and "takeInt 10 (scanSum [1..])"?
5. diffs returns a list of the differences between adjacent items. So diffs [3,5,6,8] returns [2,1,2]. (Hint: write a second function that takes two lists and finds the difference between corresponding items).

Deconstructing lists

So now we know how to generate lists by appending to the empty list, or using infinite lists and their notation. Very useful.

But what happens if our function is not generating a list and handing it off to some other function, but is rather receiving a list? It needs to be analyzed and broken down in some way.

For this purpose, Haskell includes the same basic functionality as other programming languages, except with better names than "cdr" or "car": the "head" and "tail" functions.

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

From these two functions we can build pretty much all the functionality we want. If we want the first item in the list, a simple head will do:

```
Code          Result
-----
head [1,2,3]  1
head [5..100] 5
```

If we want the second item in a list, we have to be a bit clever: head gives the first item in a list, and tail effectively removes the first item in a list. They can be combined, though:

```
Code          Result
-----
head(tail [1,2,3,4,5])  2
head(tail (tail [1,2,3,4,5])) 3
```

Enough tails can reach to arbitrary elements; usually this is generalized into a function which is passed a list and a number, which gives the position in a list to return.

Exercises

Write a function which takes a list and a number and returns the given element; use `head` or `tail`, and not `!!`.

List comprehensions

This is one further way to deconstruct lists; it is called a List comprehension. List comprehensions are useful and concise expressions, although they are fairly rare.

List comprehensions are basically syntactic sugar for a common pattern dealing with lists: when one wants to take a list and generate a new list composed only of elements of the first list that meet a certain condition.

One could write this out manually. For example, suppose one wants to take a list `[1..10]`, and only retain the even numbers? One could handcraft a recursive function called `retainEven`, based on a test for evenness which we've already written called `isEven`:

```
isEven :: Integer -> Bool
isEven n
  | n < 0 = error "isEven needs a positive integer"
  | ((mod n 2) == 0) = True  -- Even numbers have no remainder when divided by 2
  | otherwise = False  -- If it has a remainder of anything but 0, it is not even
```

```
retainEven :: [Integer] -> [Integer]
retainEven [] = []
retainEven (e:es)
  | isEven e = e:retainEven es --If something is even, let's hang onto it
  | otherwise = retainEven es --If something isn't even, discard it and move on
```

Exercises

Write a function which will take a list and return only odd numbers greater than 1. Hint: `isOdd` can be defined as the negation of `isEven`.

This is fairly verbose, though, and we had to go through a fair bit of effort and define an entirely new function just to accomplish the relatively simple task of filtering a list. Couldn't it be generalized? What we want to do is construct a new list with only the elements of an old list for which some boolean condition is true. Well, we could generalize our function writing above like this, involving the higher-order functions `map` and `filter`. For example, the above can also be written as

```
retainEven es = filter isEven es
```

We can do this through the list comprehension form, which looks like this:

```
retainEven es = [ n | n <- es , isEven n ]
```

We can read the first half as an arbitrary expression modifying `n`, which will then be prepended to a new list. In this case, `n` isn't being modified, so we can think of this as repeatedly prepending the variable, like `n:n:n:n:[]` - but where `n` is different each time. `n` is drawn (the "`<-`") from the list `es` (a subtle point is that `es` can be the name of a list, or it can itself be a list).

Thus if `es` is equal to `[1,2,3,4]`, then we would get back the list `[2,4]`.

Suppose we wanted to subtract one from every even?

```
evensMinusOne es = [n - 1 | n<-es , isEven n ]
```

We can do more than that, and list comprehensions can be easily modifiable. Perhaps we wish to generalize factoring a list, instead of just factoring it by evenness (that is, by 2). Well, given that $((\text{mod } n \ x) == 0)$ returns true for numbers n which are factorizable by x , it's obvious how to use it, no? Write a function using a list comprehension which will take an integer, and a list of integers, and return a list of integers which are divisible by the first argument. In other words, the type signature is thus:

```
returnfact :: Int -> [Int] -> [Int]
```

We can load the function, and test it with:

```
returnFact 10 [10..1000]
```

which should give us this:

```
*Main> returnFact 10 [10..1000]
[10,20,30,40,50,60,70,80,90,100,110,120,130,140,150,160,170,180,190,200,...etc.]
```

Which is as it should be. But what if we want to write the *opposite*? What if we want to write a function which returns those integers which are not divisible? The modification is very simple, and the type signature the same. What decides whether a integer will be added to the list or not is the mod function, which currently returns true for those to be added. A simple 'not' suffices to reverse when it returns true, and so reverses the operation of the list:

```
rmFact :: Int -> [Int] -> [Int]
rmFact x ys = [n | n<-ys , (not ((mod n x) == 0))]
```

We can load it and give the equivalent test:

```
*Main> rmFact 10 [10..1000]
[11,12,13,14,15,16,17,18,19,21,22,23,24,25,26,27,28,29,.....etc.]
```

Of course this function is not perfect. We can still do silly things like

```
*Main> rmFact 0 [1..1000]
*** Exception: divide by zero
```

We can stack on more tests besides the one: maybe all our even numbers should be larger than 2:

```
evensLargerThanTwo = [ n | n <- [1..10] , isEven n, n > 2 ]
```

Fortunately, our Boolean tests are commutative, so it doesn't matter whether $(n > 2)$ or $(\text{isEven } 2)$ is evaluated first.

Pattern matching in list comprehensions

It's useful to note that the left arrow in list comprehensions can be used with pattern matching. For example, suppose we had a list of tuples $[(\text{Integer}, \text{Integer})]$. What we would like to do is return the first element of every tuple whose second element is even. We could write it with a filter and a map, or we could write it as follows:

```
firstOfEvens xys = [ x | (x,y) <- xys, isEven y ]
```

And if we wanted to *double* those first elements:

```
doubleFirstOfEvens xys = [ 2 * x | (x,y) <- xys, isEven y ]
```

Control structures

Haskell offers several ways of expressing a choice between different values. This section will describe them all and explain what they are for:

if Expressions

You have already seen these. The full syntax is:

```
if <condition> then <true-value> else <false-value>
```

If the `<condition>` is `True` then the `<true-value>` is returned, otherwise the `<false-value>` is returned. Note that in Haskell `if` is an expression (returning a value) rather than a statement (to be executed). Because of this the usual indentation is different from imperative languages. If you need to break an `if` expression across multiple lines then you should indent it like one of these:

The else is required!

```
if <condition>
  then <1>
  else <0>
```

```
if <condition>
  then
    <true-value>
  else
    <false-value>
```

Here is a simple example:

```
message42 :: Integer -> String
message42 n =
  if n == 42
    then "The Answer is forty two."
    else "The Answer is not forty two."
```

Unlike many other languages, in Haskell the `else` is required. Since `if` is an expression, it must return a result, and the `else` ensures this.

case Expressions

case expressions are a generalization of `if` expressions. As an example, let's clone `if` as a case:

```
case <condition> of
  True  -> <true-value>
  False -> <false-value>
  _     -> error "Neither True nor False? How can that be?"
```

First, this checks `<condition>` for a pattern match against `True`. If they match, the whole expression will evaluate to `<true-value>`, otherwise it will continue down the list. You can use `_` as the pattern wildcard. In fact, the left hand side of any case branch is just a pattern, so it can also be used for binding:

```
case str of
  (x:xs) -> "The first character is " ++ [x] ++ "; the rest of the string is " ++ xs
  ""     -> "This is the empty string."
```

This expression tells you whether `str` is the empty string or something else. Of course, you could just do this with an `if`-statement (with a condition of `null str`), but using a case binds variables to the head and tail of our list, which is convenient in this instance.

Equations and Case Expressions

You can use multiple equations as an alternative to case expressions. The case expression above could be named `describeString` and written like this:

```
describeString :: String -> String
describeString (x:xs) = "The first character is " ++ [x] ++ "; the rest of the string is " ++ xs
describeString ""    = "This is the empty string."
```

Named functions and case expressions at the top level are completely interchangeable. In fact the function definition form shown here is just syntactic sugar for a case expression.

The handy thing about case expressions is that they can go inside other expressions, or be used in an anonymous function. *TODO: this isn't really limited to case.* For example, this case expression returns a string which is then concatenated with two other strings to create the result:

```
data Colour = Black | White | RGB Int Int Int
describeColour c =
  "This colour is "
  ++ (case c of
      Black -> "black"
      White -> "white"
      RGB _ _ _ -> "freaky, man, sort of in between")
  ++ ", yeah?"
```

You can also put `where` clauses in a case expression, just as you can in functions:

```
describeColour c =
  "This colour is "
  ++ (case c of
      Black -> "black"
      White -> "white"
      RGB red green blue -> "freaky, man, sort of " ++ show av
        where av = (red + green + blue) `div` 3
      )
  ++ ", yeah?"
```

Guards

As shown, if we have a top-level `case` expression, we can just give multiple equations for the function instead, which is normally neater. Is there an analogue for `if` expressions? It turns out there is.

We use some additional syntax known as "guards". A guard is a boolean condition, like this:

```
describeLetter :: Char -> String
describeLetter c
  | c >= 'a' && c <= 'z' = "Lower case"
  | c >= 'A' && c <= 'Z' = "Upper case"
  | otherwise           = "Not a letter"
```

Note the lack of an `=` before the first `|`. Guards are evaluated in the order they appear. That is, if you have a set up similar to the following:

```
f (pattern1) | predicate1 = w
             | predicate2 = x
f (pattern2) | predicate3 = y
             | predicate4 = z
```

Then the input to `f` will be pattern-matched against `pattern1`. If it succeeds, then `predicate1` will be evaluated. If this is true, then `w` is returned. If not, then `predicate2` is evaluated. If *this* is true, then `x` is returned. Again, if not, then we jump out of this 'branch' of `f` and try to pattern match against `pattern2`, repeating the guards procedure with `predicate3` and `predicate4`. If no guards match, an error will be produced at runtime, so it's always a good idea to leave an 'otherwise' guard in there to handle the "But this can't happen!" case.

The `otherwise` you saw above is actually just a normal value defined in the Standard Prelude as:

```
otherwise :: Bool
otherwise = True
```

This works because of the sequential evaluation described a couple of paragraphs back: if none of the guards previous to your 'otherwise' one are true, then your otherwise will definitely be true and so whatever is on the

right-hand side gets returned. It's just nice for readability's sake.

'where' and guards

One nicety about guards is that **where** clauses are common to all guards.

```
doStuff x
  | x < 3 = report "less than three"
  | otherwise = report "normal"
  where
    report y = "the input is " ++ y
```

The difference between if and case

It's worth noting that there is a fundamental difference between `if`-expressions and `case`-expressions.

`if`-expressions, and guards, only *check to see if a boolean expression evaluated to True*. `case`-expressions, and multiple equations for the same function, *pattern match against the input*. Make sure you understand this important distinction.

List processing

Because lists are such a fundamental data type in Haskell, there is a large collection of standard functions for processing them. These are mostly to be found in a library module called the 'Standard Prelude' which is automatically imported in all Haskell programs. There are also additional list-processing functions to be found in the `Data.List` module.

Map

This module will explain one particularly important function, called `map`, and then describe some of the other list processing functions that work in similar ways.

Recall the `multiplyList` function from a couple of chapters ago.

```
multiplyList :: Integer -> [Integer] -> [Integer]
multiplyList _ [] = []
multiplyList m (n:ns) = (m*n) : multiplyList m ns
```

This works on a list of integers, multiplying each item by a constant. But Haskell allows us to pass functions around just as easily as we can pass integers. So instead of passing a multiplier `m` we could pass a function `f`, like this:

```
mapList1 :: (Integer -> Integer) -> [Integer] -> [Integer]
mapList1 _ [] = []
mapList1 f (n:ns) = (f n) : mapList1 f ns
```

Take a minute to compare the two functions. The difference is in the first parameter. Instead of being just an `Integer` it is now a function. This function parameter has the type `(Integer -> Integer)`, meaning that it is a function from one integer to another. The second line says that if this is applied to an empty list then the result is itself an empty list, and the third line says that for a non-empty list the result is `f` applied to the first item in the list, followed by a recursive call to `mapList1` for the rest of the list.

Remember that `(*)` has type `Integer -> Integer -> Integer`. So if we write `(2*)` then this returns a new function that doubles its argument and has type `Integer -> Integer`. But that is exactly the type of functions that can be passed to `mapList1`. So now we can write `doubleList` like this:

```
doubleList = mapList1 (2*)
```

We could also write it like this, making all the arguments explicit:

```
doubleList ns = mapList1 (2*) ns
```

(The two are equivalent because if we pass just one argument to `mapList1` we get back a new function. The second version is more natural for newcomers to Haskell, but experts often favour the first, known as 'point free' style.)

Obviously this idea is not limited to just integers. We could just as easily write

```
mapListString :: (String -> String) -> [String] -> [String]
mapListString _ [] = []
mapListString f (n:ns) = (f n) : mapList1 f ns
```

and have a function that does this for strings. But this is horribly wasteful: the code is exactly the same for both strings and integers. What is needed is a way to say that `mapList` works for both `Integers`, `Strings`, and any other type we might want to put in a list. In fact there is no reason why the input list should be the same type as the output list: we might very well want to convert a list of integers into a list of their string representations, or vice versa. And indeed Haskell provides a way to do this. The Standard Prelude contains the following definition of `map`:

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

Instead of constant types like `String` or `Integer` this definition uses type variables. These start with lower case letters (as opposed to type constants that start with upper case) and otherwise follow the same lexical rules as normal variables. However the convention is to start with "a" and go up the alphabet. Even the most complicated functions rarely get beyond "d".

So what this says is that `map` takes two parameters:

- A function from a thing of type `a` to a thing of type `b`.

- A list of things of type `a`.

Then it returns a new list containing things of type `b`, constructed by applying the function to each element of the input list.

Exercises

1. Use `map` to build functions that, given a list `l` of `Ints`, returns:
 - A list that is the element-wise negation of `l`.
 - A list of lists of `Ints` `ll` that, for each element of `l`, contains the factors of `l`. It will help to know that

```
factors p = [ f | f <- [1..p], p `mod` f == 0 ]
```

- The element-wise negation of `ll`.
2. Implement a Run Length Encoding (RLE) encoder and decoder.
 - The idea of RLE is simple; given some input:

```
"aaaabbaaa"
```

compress it by taking the length of each run of characters:

```
((4, 'a'), (2, 'b'), (3, 'a'))
```

- The `group` function might be helpful
- What is the type of your `encode` and `decode` functions?
- How would you convert the list of tuples (e.g. `[(4, 'a'), (6, 'b')]`) into a string (e.g. `"4a6b"`)?
- (bonus) Assuming numeric characters are forbidden in the original string, how would you parse that string back into a list of tuples?

Folds

A fold applies a function to a list in a way similar to `map`, but accumulates a single result instead of a list.

Take for example, a function like `sum`, which might be implemented as follows:

Example: sum

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



or `product`:

Example: product

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



or `concat`, which takes a list of lists and joins (concatenates) them into one:

Example: concat

```
concat :: [[a]] -> [a]
concat [] = []
concat (x:xs) = x ++ concat xs
```



There is a certain pattern of recursion common to all of these examples. This pattern is known as a *fold*, possibly from the idea that a list is being "folded up" into a single value, or that a function is being "folded between" the elements of the list.

The Standard Prelude defines four fold functions: `foldr`, `foldl`, `foldr1` and `foldl1`.

foldr

The most natural and commonly used of these in a lazy language like Haskell is the *right-associative* `foldr`:

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

The first argument is a function with two arguments, the second is a "zero" value for the accumulator, and the third is the list to be folded.

For example, in `sum`, `f` is `(+)`, and `z` is `0`, and in `concat`, `f` is `(++)` and `z` is `[]`. In many cases, like all of our examples so far, the function passed to a fold will have both its arguments be of the same type, but this is not necessarily the case in general.

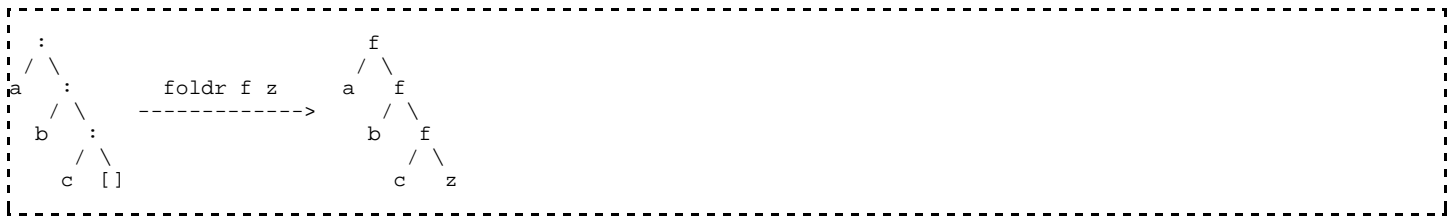
What `foldr f z xs` does is to replace each `cons` (`:`) in the list `xs` with the function `f`, and the empty list at the end with `z`. That is,

```
a : b : c : []
```

becomes

```
f a (f b (f c z))
```

This is perhaps most elegantly seen by picturing the list data structure as a tree:



It is fairly easy to see with this picture that `foldr (:) []` is just the identity function on lists.

foldl

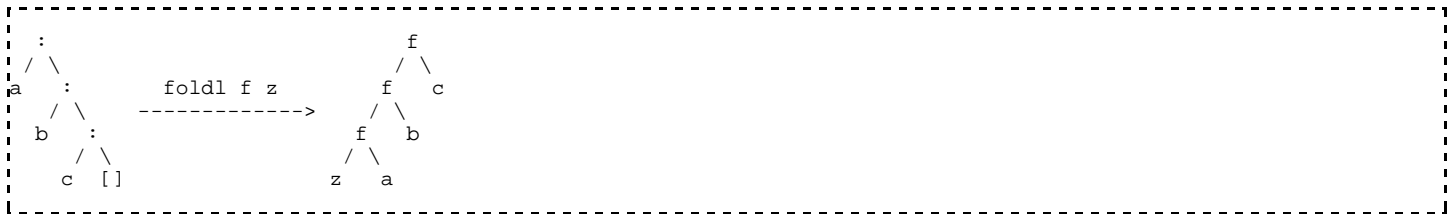
The *left-associative* `foldl` processes the list in the opposite direction:

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

So brackets in the resulting expression accumulate on the left. Our list above, after being transformed by `foldl f z` becomes:

```
f (f (f z a) b) c
```

The corresponding trees look like:



Technical Note: The left associative fold is tail-recursive, that is, it recurses immediately, calling itself. For this reason the compiler will optimise it to a simple loop, and it will then be much more efficient than `foldr`.

However, Haskell is a lazy language, and so the calls to `f` will by default be left unevaluated, building up an expression in memory whose size is linear in the length of the list, exactly what we hoped to avoid in the first place. To get back this efficiency, there is a version of `foldl` which is strict, that is, it forces the evaluation of `f` immediately, called `foldl'`. Note the single quote character: this is pronounced "fold-ell-tick". A tick is a valid character in Haskell identifiers. `foldl'` can be found in the library `Data.List`. As a rule you should use `foldr` on lists that might be infinite or where the fold is building up a data structure, and `foldl'` if the list is known to be finite and comes down to a single value. `foldl` (without the tick) should rarely be used at all.

foldr1 and foldl1

As previously noted, the type declaration for `foldr` makes it quite possible for the list elements and result to be of different types. For example, "read" is a function that takes a string and converts it into some type (the type system is smart enough to figure out which one). In this case we convert it into a float.

Example: The list elements and results can have different types



```
addStr :: String -> Float -> Float
addStr str x = read str + x

sumStr :: [String] -> Float
sumStr = foldr addStr 0.0
```

If you substitute the types `Float` and `String` for the type variables `a` and `b` in the type of `foldr` you will see that this is type correct.

There is also a variant called `foldr1` ("fold - arr - one") which dispenses with an explicit zero by taking the last element of the list instead:

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
foldr1 _ [] = error "Prelude.foldr1: empty list"
```

And `foldl1` as well:

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs
foldl1 _ [] = error "Prelude.foldl1: empty list"
```

Note: There is additionally a strict version of `foldl1` called `foldl1'` in the `Data.List` library.

Notice that in this case all the types have to be the same, and that an empty list is an error. These variants are occasionally useful, especially when there is no obvious candidate for `z`, but you need to be sure that the list is not going to be empty. If in doubt, use `foldr` or `foldl'`.

folders and laziness

One good reason that right-associative folds are more natural to use in Haskell than left-associative ones is that right folds can operate on infinite lists, which are not so uncommon in Haskell programming. If the input function `f` only needs its first parameter to produce the first part of the output, then everything works just fine. However, a left fold will continue recursing, never producing anything in terms of output until it reaches the end of the input list. Needless to say, this never happens if the input list is infinite, and the program will spin endlessly in an infinite loop.

As a toy example of how this can work, consider a function `echoes` taking a list of integers, and producing a list where if the number `n` occurs in the input list, then `n` replicated `n` times will occur in the output list. We will make use of the prelude function `replicate`: `replicate n x` is a list of length `n` with `x` the value of every element.

We can write `echoes` as a `foldr` quite handily:

```
echoes = foldr (\x xs -> (replicate x x) ++ xs) []
```

or as a `foldl`:

```
echoes = foldl (\xs x -> xs ++ (replicate x x)) []
```

but only the first definition works on an infinite list like `[1..]`. Try it!

Note the syntax in the above example: the `\xs x ->` means that `xs` is set to the first argument outside the parentheses (in this case, `[]`), and `x` is set to the second (will end up being the argument of `echoes` when it is called).

As a final example, another thing that you might notice is that `map` itself is patterned as a fold:

```
map f = foldr (\x xs -> f x : xs) []
```

Folding takes a little time to get used to, but it is a fundamental pattern in functional programming, and eventually becomes very natural. Any time you want to traverse a list and build up a result from its members you want a fold.

Exercises

Define the following functions recursively (like the definitions for `sum`, `product` and `concat` above), then turn them into a fold:

- `and :: [Bool] -> Bool`, which returns `True` if a list of `Bools` are all `True`, and `False` otherwise.
- `or :: [Bool] -> Bool`, which returns `True` if any of a list of `Bools` are `True`, and `False` otherwise.

Define the following functions using `foldl1` or `foldr1`:

- `maximum :: Ord a => [a] -> a`, which returns the maximum element of a list (hint: `max :: Ord a => a -> a -> a` returns the maximum of two values).
- `minimum :: Ord a => [a] -> a`, which returns the minimum element of a list (hint: `min :: Ord a => a -> a -> a` returns the minimum of two values).

Scans

A "scan" is much like a cross between a `map` and a fold. Folding a list accumulates a single return value, whereas mapping puts each item through a function with no accumulation. A scan does both: it accumulates a value like a fold, but instead of returning a final value it returns a list of all the intermediate values.

The Standard Prelude contains four scan functions:

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
```

This accumulates the list from the left, and the second argument becomes the first item in the resulting list. So

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

```
scanl1 :: (a -> a -> a) -> [a] -> [a]
```

This is the same as `scanl`, but uses the first item of the list as a zero parameter. It is what you would typically use if the input and output items are the same type. Notice the difference in the type signatures. `scanl1 (+) [1,2,3] = [1,3,6]`.

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr1 :: (a -> a -> a) -> [a] -> [a]
```

These two functions are the exact counterparts of `scanl` and `scanl1`. They accumulate the totals from the right. So:

```
scanr (+) 0 [1,2,3] = [6,5,3,0]
scanr1 (+) [1,2,3] = [6,5,3]
```

Exercises

Define the following functions:

- `factList :: Integer -> [Integer]`, which returns a list of factorials from 1 up to its argument. For example, `factList 4 = [1,2,6,24]`.

More to be added

More on functions

As functions are absolutely essential to functional programming, there are some nice features you can use to make using functions easier.

Private Functions

Remember the `sumStr` function from the chapter on list processing. It used another function called `addStr`:

```
addStr :: Float -> String -> Float
addStr x str = x + read str

sumStr :: [String] -> Float
sumStr = foldl addStr 0.0
```

So you could find that

```
addStr 4.3 "23.7"
```

gives 28.0, and

```
sumStr ["1.2", "4.3", "6.0"]
```

gives 11.5.

But maybe you don't want `addStr` cluttering up the top level of your program. Haskell lets you nest declarations in two subtly different ways:

```
sumStr = foldl addStr 0.0
  where addStr x str = x + read str
```

```
sumStr =
  let addStr x str = x + read str
  in foldl addStr 0.0
```

The difference between `let` and `where` lies in the fact that `let foo = 5 in foo + foo` is an expression, but `foo + foo where foo = 5` is not. (Try it: an interpreter will reject the latter *expression*.) Where clauses are part of the function declaration as a whole, which makes a difference when using guards.

Anonymous Functions

An alternative to creating a named function like `addStr` is to create an anonymous function, also known as a `lambda function`. For example, `sumStr` could have been defined like this:

```
sumStr = foldl (\x str -> x + read str) 0.0
```

The bit in the parentheses is a lambda function. The backslash is used as the nearest ASCII equivalent to the Greek letter lambda (λ). This example is a lambda function with two arguments, `x` and `str`, and the result is `"x + read str"`. So, the `sumStr` presented just above is precisely the same as the one that used `addStr` in a `let` binding.

Lambda functions are handy for one-off function parameters, especially where the function in question is simple. The example above is about as complicated as you want to get.

Infix versus Prefix

As we noted in the previous chapter, you can take an operator and turn it into a function by surrounding it in brackets:

```
2 + 4
!(+) 2 4
```

This is called making the operator *prefix*: you're using it before its arguments, so it's known as a prefix function. We can now formalise the term 'operator': it's a function which is entirely non-alphanumeric characters, and is used infix (normally). You can define your own operators just the same as functions, just don't use any

alphanumeric characters. For example, here's the set-difference definition from `Data.List`:

```
(\\) :: Eq a => [a] -> [a] -> [a]
xs \\ ys = foldl (\x y -> delete y x) xs ys
```

Note that aside from just using operators infix, you can define them infix as well. This is a point that most newcomers to Haskell miss. I.e., although one could have written:

```
(\\) xs ys = foldl (\x y -> delete y x) xs ys
```

It's more common to define operators infix. However, do note that in type declarations, you have to surround the operators by parentheses.

You can use a variant on this parentheses style for 'sections':

```
(2+) 4
(+4) 2
```

These sections are functions in their own right. `(2+)` has the type `Int -> Int`, for example, and you can pass sections to other functions, e.g. `map (+2) [1..4]`.

If you have a (prefix) function, and want to use it as an operator, simply surround it by backticks:

```
1 `elem` [1..4]
```

This is called making the function *infix*: you're using it in between its arguments. It's normally done for readability purposes: `1 `elem` [1..4]` reads better than `elem 1 [1..4]`. You can also define functions infix:

```
elem :: Eq a => a -> [a] -> Bool
x `elem` xs = any (==x) xs
```

But once again notice that in the type signature you have to use the prefix style.

Sections even work with infix functions:

```
(1 `elem`) [1..4]
(`elem` [1..4]) 1
```

You can only make binary functions (those that take two arguments) infix. Think about the functions you use, and see which ones would read better if you used them infix.

Exercises

- Lambdas are a nice way to avoid defining unnecessary separate functions. Convert the following let- or where-bindings to lambdas:
 - `map f xs` where `f x = x * 2 + 3`
 - `let f x y = read x + y in foldr f 1 xs`
- Sections are just syntactic sugar for lambda operations.

I.e. `(+2)` is equivalent to `\x -> x + 2`. What would the following sections 'desugar' to? What would be their types?

- `(4+)`
- `(1 `elem`)`
- `(`notElem` "abc")`

Higher-order functions and Currying

Higher-order functions are functions that take other functions as arguments. We have already met some of them, such as `map`, so there isn't anything really frightening or unfamiliar about them. They offer a form of abstraction that is unique to the functional programming style. In functional programming languages like Haskell, functions are just like any other value, so it doesn't get any harder to deal with higher-order functions.

Higher order functions have a separate chapter in this book, not because they are particularly difficult -- we've already worked with them, after all -- but because they are powerful enough to draw special attention to them. We will see in this chapter how much we can do if we can pass around functions as values. Generally speaking, it is a good idea to abstract over a functionality whenever we can. Besides, Haskell without higher order functions wouldn't be quite as much fun.

The Quickest Sorting Algorithm In Town

Don't get too excited, but `quickSort` is certainly *one* of the quickest. Have you heard of it? If you did, you can skip the following subsection and go straight to the next one:

The Idea Behind `quickSort`

The idea is very much simple. For a big list, we pick an element, and divide the whole list into three parts.

The first part has all elements that should go before that element, the second part consists of all of the elements that are equal to the picked element, the third has the elements that ought to go after that element. And then, of course, we are supposed to concatenate these. What we get is somewhat better, right?

The trick is to note that only the first and the third are yet to be sorted, and for the second, sorting doesn't really make sense (they are all equal!). How to go about sorting the yet-to-be-sorted sub-lists? Why... apply the same algorithm on them again! By the time the whole process is finished, you get a completely sorted list.

So Let's Get Down To It!


```

-- if the list is empty, we do nothing
-- note that this is the base case for the recursion
quickSort [] = []

-- if there's only one element, no need to sort it
-- actually, the third case takes care of this one pretty well
-- I just wanted you to take it step by step
quickSort [x] = [x]

-- this is the gist of the process
-- we pick the first element as our "pivot", the rest is to be sorted
-- don't forget to include the pivot in the middle part!
quickSort (x : xs) = (quickSort less) ++ (x : equal) ++ (quickSort more)
  where less = filter (< x) xs
        equal = filter (== x) xs
        more = filter (> x) xs

```

And we are done! I suppose if you *have* met `quickSort` before, you thought recursion is a neat trick but is hard to implement as so many things need to be kept track of.

Now, How Do We Use It?

With `quickSort` at our disposal, sorting any list is a piece of cake. Suppose we have a list of `String`, maybe from a dictionary, and we want to sort them, we just apply `quickSort` to the list. For the rest of this chapter, we will use a pseudo-dictionary of words (but a 25,000 word dictionary should do the trick as well):

```
dictionary = ["I", "have", "a", "thing", "for", "Linux"]
```

We get, for `quickSort dictionary`,

```
["I", "Linux", "a", "for", "have", "thing"]
```

But, what if we wanted to sort them in the *descending* order? Easy, just reverse the list, `reverse sortedDictionary` gives us what we want.

But wait! We didn't really *sort* in the descending order, we sorted (in the *ascending* order) and reversed it. They may have the same effect, but they are not the same thing!

Besides, you might object that the list you got isn't what you wanted. "a" should certainly be placed before "I". "Linux" should be placed between "have" and "thing". What's the problem here?

The problem is, the way `Strings` are represented in a typical programming settings is by a list of ASCII characters. ASCII (and almost all other encodings of characters) specifies that the character code for capital letters are less than the small letters. Bummer. So "Z" is less than "a". We should do something about it. Looks like we need a case insensitive `quickSort` as well. It might come handy some day.

But, there's no way you can blend that into `quickSort` as it stands. We have work to do.

Tweaking What We Already Have

What we need to do is to factor out the comparisons `quickSort` makes. We need to provide `quickSort` with a *function* that compares two elements, and gives an `Ordering`, and as you can imagine, an `Ordering` is any of `LT`, `EQ`, `GT`.

To sort in the descending order, we supply `quickSort` with a function that returns the opposite of the usual `Ordering`. For the case-insensitive sort, we may need to define the function ourselves. By all means, we want to make `quickSort` applicable to all such functions so that we don't end up writing it over and over again, each time with only minor changes.

quickSort, Take Two

So, forget the version of `quickSort` we have now, and let's think again.

Our `quickSort` will take two things this time: first, the comparison function, and second, the list to sort.

A comparison function will be a function that takes two things, say, `x` and `y`, and compares them. If `x` is less than `y` (according to the criteria we want to implement by this function), then the value will be `LT`. If they are equal (well, equal with respect to the comparison, we want "Linux" and "linux" to be equal when we are dealing with the insensitive case), we will have `EQ`. The remaining case gives us `GT` (pronounced: greater than, for obvious reasons).

```
-----  
-- no matter how we compare two things  
-- the first two equations should not change  
-- they need to accept the comparison function though  
quickSort comparison [] = []  
quickSort comparison [x] = [x]  
-----  
-- we are in a more general setting now  
-- but the changes are worth it!  
quickSort comparison (x : xs) = (quickSort comparison less) ++ (x : equal) ++ (quickSort comparison more)  
    where less = filter (\y -> comparison y x == LT) xs  
          equal = filter (\y -> comparison y x == EQ) xs  
          more = filter (\y -> comparison y x == GT) xs  
-----
```

Cool!

Note

Almost all the basic data types in Haskell are members of the `Ord` class. This class defines an ordering, the "natural" one. The functions (`or`, operators, in this case) (`<`), (`<=`) or (`>`) provide shortcuts to the `compare` function each type defines. When we *want* to use the natural ordering as defined by the types themselves, the above code can be written using those operators, as we did last time. In fact, that makes for much clearer style; however, we wrote it the long way just to make the relationship between sorting and comparing more evident.

But What Did We Gain?

Reuse. We can reuse `quickSort` to serve different purposes.

```

-- the usual ordering
-- uses the compare function from the Ord class
usual = compare

-- the descending ordering, note we flip the order of the arguments to compare
descending x y = compare y x

-- the case-insensitive version is left as an exercise!
insensitive = ...
-- can you think of anything without making a very big list of all possible cases?

```

And we are done!

```
quickSort usual dictionary
```

should, then, give

```
["I", "Linux", "a", "for", "have", "thing"]
```

The comparison is just `compare` from the `Ord` class. This was our `quickSort`, before the tweaking.

```
quickSort descending dictionary
```

now gives

```
["thing", "have", "for", "a", "Linux", "I"]
```

And finally,

```
quickSort insensitive dictionary
```

gives

```
["a", "for", "have", "I", "Linux", "thing"]
```

Exactly what we wanted!

Exercises

Write `insensitive`, such that `quickSort insensitive dictionary` gives `["a", "for", "have", "I", "Linux", "thing"]`

Higher-Order Functions and Types

Our `quickSort` has type `(a -> a -> Ordering) -> [a] -> [a]`.

Most of the time, the type of a higher-order function provides a good guideline about how to use it. A straightforward way of reading the type signature would be, "quickSort takes a function that gives an ordering of `as`, and a list of `as`, to give a list of `as`". It is then natural to guess that the function sorts the list respecting the given ordering function.

Note that the parentheses surrounding `a -> a -> Ordering` is mandatory. It says that `a -> a -> Ordering` altogether form a single argument, an argument that happens to be a function. What happens if we omit the parentheses? We would get a function of type `a -> a -> Ordering -> [a] -> [a]`, which accepts four arguments instead of the desired two (`a -> a -> Ordering` and `[a]`). Furthermore none of the four arguments, neither `a` nor `Ordering` nor `[a]` are functions, so omitting the parentheses would give us something that isn't a higher order function.

Furthermore, it's worth noting that the `->` operator is right-associative, which means that `a -> a -> Ordering -> [a] -> [a]` means the same thing as `a -> (a -> (Ordering -> ([a] -> [a])))`. We really must insist that the `a -> a -> Ordering` be clumped together by writing those parentheses... but wait... if `->` is right-associative, wouldn't that mean that the correct signature `(a -> a -> Ordering) -> [a] -> [a]` actually means... `(a -> a -> Ordering) -> ([a] -> [a])`?

Is that *really* what we want?

If you think about it, we're trying to build a function that takes two arguments, a function and a list, returning a list. Instead, what this type signature is telling us is that our function takes ONE argument (a function) and returns another function. That is profoundly odd... but if you're lucky, it might also strike you as being profoundly beautiful. Functions in multiple arguments are fundamentally the same thing as functions that take one argument and give another function back. It's OK if you're not entirely convinced. We'll go into a little bit more detail below and then show how something like this can be turned to our advantage.

Exercises

The following exercise combines what you have learned about higher order functions, recursion and IO. We are going to recreate what programmers from more popular languages call a "for loop". Implement a function

```
fFor :: a -> (a->Bool) -> (a->a) -> (a-> IO ()) -> IO ()
fFor i p f job = -- ???
```

An example of how this function would be used might be

```
fFor 1 (<10) (+1) (\x -> print x)
```

which prints the numbers 1 to 10 on the screen.

Starting from an initial value `i`, the `fFor` executes `job i`. It then modifies this value `f i` and checks to see if the modified value satisfies some condition. If it does, it stops; otherwise, the for loop continues, using the modified `f i` in place of `i`.

1. The paragraph above gives an imperative description of the for loop. What would a more functional description be?

2. Implement the for loop in Haskell.
3. Why does Haskell not have a for loop as part of the language, or in the standard library?

Some more challenging exercises you could try

1. What would be a more Haskell-like way of performing a task like 'print the list of numbers from 1 to 10'? Are there any problems with your solution?
2. Implement a function `sequenceIO :: [IO a] -> IO [a]`. Given a list of actions, this function runs each of the actions in order and returns all their results as a list.
3. Implement a function `mapIO :: (a -> IO b) -> [a] -> IO [b]` which given a function of type `a -> IO b` and a list of type `[a]`, runs that action on each item in the list, and returns the results.

This exercise was inspired from a blog post by osfameron. No peeking!

Currying

Intermediate Haskell

Modules

Modules

Haskell modules are a useful way to group a set of related functionalities into a single package and manage a set of different functions that have the same name. The module definition is the first thing that goes in your Haskell file.

Here is what a basic module definition looks like:

```
module YourModule where
```

Note that

1. Each file contains only one module
2. The name of the module begins with a capital letter

Importing

One thing your module can do is import functions from other modules. That is, in between the module declaration and the rest of your code, you may include some import declarations such as

```

-----
-- import only the functions toLower and toUpper from Data.Char
import Data.Char (toLower, toUpper)

-- import everything exported from Data.List
import Data.List

-- import everything exported from MyModule
import MyModule
-----

```

Imported datatypes are specified by their name, followed by a list of imported constructors in parenthesis. For example:

```

-----
-- import only the Tree data type, and its Node constructor from Data.Tree
import Data.Tree (Tree(Node))
-----

```

Now what to do if you import some modules, but some of them have overlapping definitions? Or if you import a module, but want to overwrite a function yourself? There are three ways to handle these cases: Qualified imports, hiding definitions and renaming imports.

Qualified imports

Say `MyModule` and `MyOtherModule` both have a definition for `remove_e`, which removes all instances of `e` from a string. However, `MyModule` only removes lower-case e's, and `MyOtherModule` removes both upper and lower case. In this case the following code is ambiguous:

```

-----
-- import everything exported from MyModule
import MyModule

-- import everything exported from MyOtherModule
import MyOtherModule

-- someFunction puts a c in front of the text, and removes all e's from the rest
someFunction :: String -> String
someFunction text = 'c' : remove_e text
-----

```

In this case, it isn't clear which `remove_e` is meant. To avoid this, use the **qualified** keyword:

```

-----
import qualified MyModule
import qualified MyOtherModule

someFunction text = 'c' : MyModule.remove_e text -- Will work, removes lower case e's
someOtherFunction text = 'c' : MyOtherModule.remove_e text -- Will work, removes all e's
someIllegalFunction text = 'c' : remove_e text -- Won't work, remove_e isn't defined.
-----

```

See the difference. In this case the function `remove_e` isn't even defined. We call the functions from the imported modules by adding the module's name. Note that `MyModule.remove_e` also works if the qualified flag isn't included. The difference lies in the fact that `remove_e` is ambiguously defined in the first case, and undefined in the second case. If we have a `remove_e` defined in the current module, then using `remove_e` without any prefix will call this function.

Note

There is an ambiguity between a qualified name like `MyModule.remove_e` and function composition (`.`). Writing `reverse.MyModule.remove_e` is bound to confuse your Haskell compiler. One solution is stylistic: to always use spaces for function composition, for example, `reverse . remove_e` or `Just . remove_e` or even `Just . MyModule.remove_e`

Hiding definitions

Now suppose we want to import both `MyModule` and `MyOtherModule`, but we know for sure we want to remove all e's, not just the lower cased ones. It will become really tedious (and disorderly) to add `MyOtherModule` before every call to `remove_e`. Can't we just *not* import `remove_e` from `MyModule`? The answer is: yes we can.

```
-----
-- Note that I didn't use qualified this time.
import MyModule hiding (remove_e)
import MyOtherModule
someFunction text = 'c' : remove_e text
-----
```

This works. Why? Because of the word **hiding** on the import line. Followed by it, is a list of functions that shouldn't be imported. Hiding more than one function works like this:

```
-----
import MyModule hiding (remove_e, remove_f)
-----
```

Note that algebraic datatypes and type synonyms cannot be hidden. These are always imported. If you have a datatype defined in more modules, you must use qualified names.

Renaming imports

This is not really a technique to allow for overwriting, but it is often used along with the qualified flag. Imagine:

```
-----
import qualified MyModuleWithAVeryLongModuleName
someFunction text = 'c' : MyModuleWithAVeryLongModuleName.remove_e $ text
-----
```

Especially when using qualified, this gets irritating. What we can do about it, is using the **as** keyword:

```
-----
import qualified MyModuleWithAVeryLongModuleName as Shorty
someFunction text = 'c' : Shorty.remove_e $ text
-----
```

This allows us to use `Shorty` instead of `MyModuleWithAVeryLongModuleName` as prefix for the imported functions. As long as there are no ambiguous definitions, the following is also possible:

```
-----
import MyModule as My
import MyCompletelyDifferentModule as My
-----
```

In this case, both the functions in `MyModule` and the functions in `MyCompletelyDifferentModule` can be prefixed with `My`.

Exporting

In the examples at the start of this article, the words "import *everything exported* from `MyModule`" were used. This raises a question. How can we decide which functions are exported and which stay "internal"? Here's how:

```
module MyModule (remove_e, add_two) where
add_one blah = blah + 1
remove_e text = filter (/= 'e') text
add_two blah = add_one . add_one $ blah
```

In this case, only `remove_e` and `add_two` are exported. While `add_two` is allowed to make use of `add_one`, functions in modules that import `MyModule` aren't allowed to try to use `add_one`, as it isn't exported.

Datatype export specifications are written quite similarly to import. You name the type, and follow with the list of constructors in parenthesis:

```
module MyModule2 (Tree(Branch, Leaf)) where
data Tree a = Branch {left, right :: Tree a}
             | Leaf a
```

In this case, the module declaration could be rewritten "`MyModule2 (Tree(..)`", declaring that all constructors are exported.

Note: maintaining an export list is good practise not only because it reduces namespace pollution, but also because it enables certain compile-time optimizations

(<http://www.haskell.org/haskellwiki/Performance/GHC#Inlining>) which are unavailable otherwise.

Notes

In Haskell98, the last standardised version of Haskell, the module system is fairly conservative. But recent common practice consists of using an hierarchical module system, using periods to section off namespaces.

A module may export functions that it imports.

See the Haskell report for more details on the module system:

- <http://www.haskell.org/onlinereport/modules.html>

Indentation

Haskell relies on indentation to reduce the verbosity of your code, but working with the indentation rules can be a bit confusing. The rules may seem many and arbitrary, but the reality of things is that there are only one or two layout rules, and all the seeming complexity and arbitrariness comes from how these rules interact with your

code. So to take the frustration out of indentation and layout, the simplest solution is to get a grip on these rules.

The golden rule of indentation

Whilst the rest of this chapter will discuss in detail Haskell's indentation system, you will do fairly well if you just remember a single rule:



Code which is part of some expression should be indented further in than the line containing the beginning of that expression

What does that mean? The easiest example is a let binding group. The equations binding the variables are part of the let expression, and so should be indented further in than the beginning of the binding group: the let keyword.

So,

```
let
  x = a
  y = b
```

Although you actually only need to indent by one extra space, it's more normal to place the first line alongside the 'let' and indent the rest to line up:

```
let x = a
    y = b
```

Here are some more examples:

```
do foo
  bar
  baz

where x = a
      y = b

case x of
  p  -> foo
  p' -> baz
```

Note that with 'case' it's less common to place the next expression on the same line as the beginning of the expression, as with 'do' and 'where'. Also note we lined up the arrows here: this is purely aesthetic and isn't counted as different layout; only *indentation*, whitespace beginning on the far-left edge, makes a difference to layout. Things get more complicated when the beginning of the expression isn't right at the left-hand edge. In this case, it's safe to just indent further than the *beginning of the line* containing the beginning of the expression. So,

```
myFunction firstArgument secondArgument = do -- the 'do' isn't right at the left-hand edge
  foo -- so indent these commands more than the beginning of the line con
  bar
  baz
```

Here are some alternative layouts to the above which would have also worked:

```

myFunction firstArgument secondArgument =
  do foo
    bar
    baz

myFunction firstArgument secondArgument = do foo
                                           bar
                                           baz

```

A mechanical translation

Did you know that layout (whitespace) is optional? It is entirely possible to treat Haskell as a one-dimensional language like C, using semicolons to separate things, and curly braces to group them back.

To understand layout, you need to understand two things: where we need semicolons/braces, and how to get there from layout. The entire layout process can be summed up in three translation rules (plus a fourth one that doesn't come up very often):

It is sometimes useful to avoid layout or to mix it with semicolons and braces.

1. If you see one of the layout keywords, (`let`, `where`, `of`, `do`), insert an open curly brace (right before the stuff that follows it)
2. If you see something indented to the SAME level, insert a semicolon
3. If you see something indented LESS, insert a closing curly brace
4. If you see something unexpected in a list, like `where`, insert a closing brace before instead of a semicolon.

Exercises

In one word, what happens if you see something indented MORE?

to be completed: work through an example

Exercises

Translate the following layout into curly braces and semicolons. Note: to underscore the mechanical nature of this process, we deliberately chose something which is probably not valid Haskell:

```

of a
  b
  c
  d
where
a
b
c
do
you
  like
the
way
i let myself
      abuse
      these
layout rules

```

Layout in action

Wrong	Right
<pre>do first thing second thing third thing</pre>	<pre>do first thing second thing third thing</pre>

do within if

What happens if we put a `do` expression with an `if`? Well, as we stated above, the keywords `if then else`, and everything besides the 4 layout keywords do *not* affect layout. So things remain exactly the same:

Wrong	Right
<pre>if foo then do first thing second thing third thing else do something else</pre>	<pre>if foo then do first thing second thing third thing else do something else</pre>

Indent to the first

Remember from the First Rule of Layout Translation (above) that although the keyword `do` tells Haskell to insert a curly brace, where the curly braces goes depends not on the `do`, but the thing that immediately follows it. For example, this weird block of code is totally acceptable:

```
do
first thing
second thing
third thing
```

As a result, you could also write combined if/do combination like this:

Wrong	Right
<pre>if foo then do first thing second thing third thing else do something else</pre>	<pre>if foo then do first thing second thing third thing else do something else</pre>

This is also the reason why you can write things like this

```
main = do
first thing
second thing
```

instead of

```
main =
do first thing
   second thing
```

Both are acceptable

if within do

This is a combination which trips up many Haskell programmers. Why does the following block of code not work?

```
-- why is this bad?
do first thing
  if condition
  then foo
  else bar
  third thing
```

Just to reiterate, the `if then else` block is not at fault for this problem. Instead, the issue is that the `do` block notices that the `then` part is indented to the same column as the `if` part, so it is not very happy, because from its point of view, it just found a new statement of the block. It is as if you had written the unsugared version on the right:

sweet (layout)	unsweet
<pre>-- why is this bad? do first thing if condition then foo else bar third thing</pre>	<pre>-- still bad, just explicitly so do { first thing ; if condition ; then foo ; else bar ; third thing }</pre>

Naturally enough, your Haskell compiler is unimpressed, because it thinks that you never finished writing your `if` expression, before charging off to write some other new statement, oh ye of little attention span. Your compiler sees that you have written something like `if condition;`, which is clearly bad, because it is unfinished. So, in order to fix this, we need to indent the bottom parts of this `if` block a little bit inwards

sweet (layout)	unsweet
<pre>-- whew, fixed it! do first thing if condition then foo else bar third thing</pre>	<pre>-- the fixed version without sugar do { first thing ; if condition then foo else bar ; third thing }</pre>

This little bit of indentation prevents the `do` block from misinterpreting your `then` as a brand new expression.

Exercises

The if-within-do problem has tripped up so many Haskellers, that one programmer has posted a proposal (<http://hackage.haskell.org/trac/haskell-prime/ticket/23>) to the Haskell prime initiative to add optional semicolons between `if then else`. How would that fix the problem?

References

- The Haskell Report (lexemes) (<http://www.haskell.org/onlinereport/lexemes.html#sect2.7>) - see 2.7 on layout

More on datatypes

Enumerations

One special case of the `data` declaration is the *enumeration*. This is simply a data type where none of the constructor functions have any arguments:

```
data Month = January | February | March | April | May | June | July
           | August | September | October | November | December
```

You can mix constructors that do and do not have arguments, but its only an enumeration if none of the constructors have arguments. The section below on "Deriving" explains why the distinction is important. For instance,

```
data Colour = Black | Red | Green | Blue | Cyan
            | Yellow | Magenta | White | RGB Int Int Int
```

The last constructor takes three arguments, so `Colour` is not an enumeration.

Incidentally, the definition of the `Bool` datatype is:

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

Named Fields (Record Syntax)

Consider a datatype whose purpose is to hold configuration settings. Usually when you extract members from this type, you really only care about one or possibly two of the many settings. Moreover, if many of the settings have the same type, you might often find yourself wondering "wait, was this the fourth or *fifth* element?" One thing you could do would be to write accessor functions. Consider the following made-up configuration type for a terminal program:

```

data Configuration =
  Configuration String      -- user name
                String      -- local host
                String      -- remote host
                Bool        -- is guest?
                Bool        -- is super user?
                String      -- current directory
                String      -- home directory
                Integer     -- time connected
  deriving (Eq, Show)

```

You could then write accessor functions, like (I've only listed a few):

```

getUserName (Configuration un _ _ _ _ _ _) = un
getLocalHost (Configuration _ lh _ _ _ _ _) = lh
getRemoteHost (Configuration _ _ rh _ _ _ _) = rh
getIsGuest (Configuration _ _ _ ig _ _ _ _) = ig
...

```

You could also write update functions to update a single element. Of course, now if you add an element to the configuration, or remove one, all of these functions now have to take a different number of arguments. This is highly annoying and is an easy place for bugs to slip in. However, there's a solution. We simply give names to the fields in the datatype declaration, as follows:

```

data Configuration =
  Configuration { username      :: String,
                 localhost     :: String,
                 remotehost    :: String,
                 isguest       :: Bool,
                 issuperuser   :: Bool,
                 currentdir    :: String,
                 homedir      :: String,
                 timeconnected :: Integer
                 }

```

This will automatically generate the following accessor functions for us:

```

username :: Configuration -> String
localhost :: Configuration -> String
...

```

Moreover, it gives us very convenient update methods. Here is a short example for a "post working directory" and "change directory" like functions that work on Configurations:

```

changeDir :: Configuration -> String -> Configuration
changeDir cfg newDir =
  -- make sure the directory exists
  if directoryExists newDir
  then -- change our current directory
       cfg{currentdir = newDir}
  else error "directory does not exist"

postWorkingDir :: Configuration -> String
-- retrieve our current directory
postWorkingDir cfg = currentdir cfg

```

So, in general, to update the field x in a datatype y to z , you write $y\{x=z\}$. You can change more than one; each should be separated by commas, for instance, $y\{x=z, a=b, c=d\}$.

It's only sugar

You can of course continue to pattern match against `Configuration`s as you did before. The named fields are simply syntactic sugar; you can still write something like:

```
getUserName (Configuration un _ _ _ _ _ _) = un
```

But there is little reason to. Finally, you can pattern match against named fields as in:

```
getHostData (Configuration {localhost=lh,remotehost=rh})
  = (lh,rh)
```

This matches the variable `lh` against the `localhost` field on the `Configuration` and the variable `rh` against the `remotehost` field on the `Configuration`. These matches of course succeed. You could also constrain the matches by putting values instead of variable names in these positions, as you would for standard datatypes.

You can create values of `Configuration` in the old way as shown in the first definition below, or in the named-field's type, as shown in the second definition below:

```
initCFG =
  Configuration "nobody" "nowhere" "nowhere"
              False False "/" "/" 0
initCFG' =
  Configuration
    { username="nobody",
      localhost="nowhere",
      remotehost="nowhere",
      isguest=False,
      issuperuser=False,
      currentdir="/",
      homedir="/",
      timeconnected=0 }
```

Though the second is probably much more understandable unless you litter your code with comments.

Parameterised Types

Parameterised types are similar to "generic" or "template" types in other languages. A parameterised type takes one or more type parameters. For example the Standard Prelude type `Maybe` is defined as follows:

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

This says that the type `Maybe` takes a type parameter `a`. You can use this to declare, for example:

```
lookupBirthday :: [Anniversary] -> String -> Maybe Anniversary
```

The `lookupBirthday` function takes a list of birthday records and a string and returns a `Maybe Anniversary`. Typically, our interpretation is that if it finds the name then it will return `Just` the corresponding record, and otherwise, it will return `Nothing`.

You can parameterise `type` and `newtype` declarations in exactly the same way. Furthermore you can combine parameterised types in arbitrary ways to construct new types.

More than one type parameter

We can also have more than one type parameter. An example of this is the `Either` type:

```
data Either a b = Left a | Right b
```

For example:

```
eitherExample :: Int -> Either Int String
eitherExample a | even a = Left (a/2)
                | a `mod` 3 == 0 = Right "three"
                | otherwise = Right "neither two or three"

otherFunction :: Int -> String
otherFunction a = case eitherExample a of
  Left c = "Even: " ++ show a ++ " = 2*" ++ show c ++ "."
  Right s = show a ++ " is divisible by " ++ s ++ "."
```

In this example, when you call `otherFunction`, it'll return a `String`. If you give it an even number as argument, it'll say so, and give half of it. If you give it anything else, `eitherExample` will determine if it's divisible by three and pass it through to `otherFunction`.

Kind Errors

The flexibility of Haskell parameterised types can lead to errors in type declarations that are somewhat like type errors, except that they occur in the type declarations rather than in the program proper. Errors in these "types of types" are known as "kind" errors. You don't program with kinds: the compiler infers them for itself. But if you get parameterised types wrong then the compiler will report a kind error.

Trees

Now let's look at one of the most important datastructures: Trees. A tree is an example of a recursive datatype. Typically, its definition will look like this:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

As you can see, it's parameterised, so we can have trees of `Ints`, trees of `Strings`, trees of `Maybe Ints`, even trees of `(Int, String)` pairs, if you really want. What makes it special is that `Tree` appears in the definition of itself. We will see how this works by using an already known example: the list.

Lists as Trees

Think about it. As we have seen in the List Processing chapter, we break lists down into two cases: An empty list (denoted by `[]`), and an element of the specified type, with another list (denoted by `(x:xs)`). This gives us valuable insight about the definition of lists:

```
data [a] = [] | (a:[a]) -- Pseudo-Haskell, will not work properly.
```

Which is sometimes written as (for Lisp-inclined people):


```
data List a = Nil | Cons a (List a)
```

As you can see this is also recursive, like the tree we had. Here, the constructor functions are `[]` and `(:)`. They represent what we have called `Leaf` and `Branch`. We can use these in pattern matching, just as we did with the empty list and the `(x:xs)`:

Maps and Folds

We already know about maps and folds for lists. With our realisation that a list is some sort of tree, we can try to write map and fold functions for our own type `Tree`. To recap:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
data [a]     = [] | (:) a [a]
-- (:) a [a] would be the same as (a:[a]) with prefix instead of infix notation.
```

I will handle map first, then folds.

Map

Let's take a look at the definition of `map` for lists:

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

First, if we were to write `treeMap`, what would its type be? Defining the function is easier if you have an idea of what its type should be.

We want it to work on a `Tree` of some type, and it should return another `Tree` of some type. What `treeMap` does is applying a function on each element of the tree, so we also need a function. In short:

```
treeMap :: (a -> b) -> Tree a -> Tree b
```

See how this is similar to the list example?

Next, we should start with the easiest case. When talking about a `Tree`, this is obviously the case of a `Leaf`. A `Leaf` only contains a single value, so all we have to do is apply the function to that value and then return a `Leaf` with the altered value:

```
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f (Leaf x) = Leaf (f x)
```

Also, this looks a lot like the empty list case with `map`. Now if we have a `Branch`, it will include two subtrees; what do we do with them? When looking at the list-`map`, you can see it uses a call to itself on the tail of the list. We also shall do that with the two subtrees. The complete definition of `treeMap` is as follows:

```
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f (Leaf x) = Leaf (f x)
treeMap f (Branch left right) = Branch (treeMap f left) (treeMap f right)
```

We can make this a bit more readable by noting that `treeMap f` is itself a function with type `Tree a -> Tree b`, and what we really need is a recursive definition of `treeMap f`. This gives us the following revised definition:

```
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f = g where
  g (Leaf x) = Leaf (f x)
  g (Branch left right) = Branch (g left) (g right)
```

If you don't understand it just now, re-read it. Especially the use of pattern matching may seem weird at first, but it is essential to the use of datatypes. The most important thing to remember is that pattern matching happens on constructor functions.

If you understand it, read on for folds.

Fold

Now we've had the `treeMap`, let's try to write a `treeFold`. Again let's take a look at the definition of `foldr` for lists, as it is easier to understand.

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

Recall that lists have two constructors:

```
(:) :: a -> [a] -> [a] -- two arguments
[] :: [a] -- zero arguments
```

Thus `foldr` takes two arguments corresponding to the two constructors:

```
f :: a -> b -> b -- a two-argument function
z :: b -- like a zero-argument function
```

We'll use the same strategy to find a definition for `treeFold` as we did for `treeMap`. First, the type. We want `treeFold` to transform a tree of some type into a value of some other type; so in place of `[a] -> b` we will have `Tree a -> b`. How do we specify the transformation? First note that `Tree a` has two constructors:

```
Branch :: Tree a -> Tree a -> Tree a
Leaf :: a -> Tree a
```

So `treeFold` will have two arguments corresponding to the two constructors:

```
fbranch :: b -> b -> b
fleaf :: a -> b
```

Putting it all together we get the following type definition:

```
treeFold :: (b -> b -> b) -> (a -> b) -> Tree a -> b
```

That is, the first argument, of type `(b -> b -> b)`, is a function specifying how to combine subtrees; the second argument, of type `a -> b`, is a function specifying what to do with leaves; and the third argument, of type `Tree a`, is the tree we want to "fold".

As with `treeMap`, we'll avoid repeating the arguments `fbranch` and `fleaf` by introducing a local function `g`:

```
treeFold :: (b -> b -> b) -> (a -> b) -> Tree a -> b
treeFold fbranch fleaf = g where
  -- definition of g goes here
```

The argument `fleaf` tells us what to do with `Leaf` subtrees:

```
g (Leaf x) = fleaf x
```

The argument `fbranch` tells us how to combine the results of "folding" two subtrees:

```
g (Branch left right) = fbranch (g left) (g right)
```

Our full definition becomes:

```
treeFold :: (b -> b -> b) -> (a -> b) -> Tree a -> b
treeFold fbranch fleaf = g where
  g (Leaf x) = fleaf x
  g (Branch left right) = fbranch (g left) (g right)
```

For examples of how these work, copy the `Tree` data definition and the `treeMap` and `treeFold` functions to a Haskell file, along with the following:

```
tree1 :: Tree Integer
tree1 =
  Branch
    (Branch
      (Branch
        (Leaf 1)
        (Branch (Leaf 2) (Leaf 3)))
      (Branch
        (Leaf 4)
        (Branch (Leaf 5) (Leaf 6))))
    (Branch
      (Branch (Leaf 7) (Leaf 8))
      (Leaf 9))

doubleTree = treeMap (*2) -- doubles each value in tree
sumTree = treeFold (+) id -- sum of the leaf values in tree
fringeTree = treeFold (++) (: []) -- list of the leaves of tree
```

Then load it into your favourite Haskell interpreter, and evaluate:

```
doubleTree tree1
sumTree tree1
fringeTree tree1
```

Other datatypes

Now, unlike mentioned in the chapter about trees, folds and maps aren't tree-only. They are very useful for any kind of data type. Let's look at the following, somewhat weird, type:

```
data Weird a b =
  First a |
  Second b |
  Third [(a,b)] |
  Fourth (Weird a b)
```

There's no way you will be using this in a program written yourself, but it demonstrates how folds and maps are really constructed.

General Map

Again, we start with `weirdMap`. Now, unlike before, this `Weird` type has *two* parameters. This means that we can't just use one function (as was the case for lists and `Tree`), but we need more. For every parameter, we need one function. The type of `weirdMap` will be:

```
weirdMap :: (a -> c) -> (b -> d) -> Weird a b -> Weird c d
```

Read it again, and it makes sense. Maps don't throw away the structure of a datatype, so if we start with a `Weird` thing, the output is also a `Weird` thing. Now we have to split it up into patterns. Remember that these patterns are the constructor functions. To avoid having to type the names of the functions again and again, I use a **where** clause:

```
weirdMap :: (a -> c) -> (b -> d) -> Weird a b -> Weird c d
weirdMap fa fb = weirdMap'
  where
    weirdMap' (First a)           = --More to follow
    weirdMap' (Second b)          = --More to follow
    weirdMap' (Third ((a,b):xs)) = --More to follow
    weirdMap' (Fourth w)          = --More to follow
```

It isn't very hard to find the definition for the `First` and `Second` constructors. The list of `(a,b)` tuples is harder. The `Fourth` is even recursive!

Remember that a map preserves structure. This is important. That means, a list of tuples stays a list of tuples. Only the types are changed in some way or another. You might have already guessed what we should do with the list of tuples. We need to make another list, of which the elements are tuples. This might sound silly to repeat, but it becomes clear that we *first* have to change individual elements into other tuples, and *then* add them to a list. Together with the `First` and `Second` constructors, we get:

```
weirdMap :: (a -> c) -> (b -> d) -> Weird a b -> Weird c d
weirdMap fa fb = weirdMap'
  where
    weirdMap' (First a)           = First (fa a)
    weirdMap' (Second b)          = Second (fb b)
    weirdMap' (Third ((a,b):xs)) = Third ( (fa a, fb b) : weirdMap' (Third xs) )
    weirdMap' (Fourth w)          = --More to follow
```

First we change `(a,b)` into `(fa a, fb b)`. Next we need the mapped version of the rest of the list to add to it. Since

we don't know a function for a list of (a,b), we must change it back to a `Weird` value, by adding `Third`. This isn't really stylish, though, as we first "unwrap" the `Weird` package, and then pack it back in. This can be changed into a more elegant solution, in which we don't even have to break list elements into tuples!

Remember we already had a function to change a list of some type into another list, of a different type? Yup, it's our good old `map` function for lists. Now what if the first type was, say (a,b), and the second type (c,d)? That seems useable. Now we must think about the function we're mapping over the list. We have already found it in the above definition: It's the function that sends (a,b) to (fa a, fb b). To write it in the Lambda Notation:

```
\(a, b) -> (fa a, fb b).
```

```
weirdMap :: (a -> c) -> (b -> d) -> Weird a b -> Weird c d
weirdMap fa fb = weirdMap'
  where
    weirdMap' (First a)   = First (fa a)
    weirdMap' (Second b) = Second (fb b)
    weirdMap' (Third list) = Third ( map \(a, b) -> (fa a, fb b) ) list)
    weirdMap' (Fourth w)  = --More to follow
```

That's it! We only have to match the list once, and call the `list-map` function on it. Now for the `Fourth` Constructor. This is actually really easy. Just `weirdMap` it again!

```
weirdMap :: (a -> c) -> (b -> d) -> Weird a b -> Weird c d
weirdMap fa fb = weirdMap'
  where
    weirdMap' (First a)   = First (fa a)
    weirdMap' (Second b) = Second (fb b)
    weirdMap' (Third list) = Third ( map \(a, b) -> (fa a, fb b) ) list)
    weirdMap' (Fourth w)  = Fourth (weirdMap w)
```

General Fold

Where we were able to define a `map`, by giving it a function for every separate type, this isn't enough for a `fold`. For a `fold`, we'll need a function for every constructor function. This is also the case with lists! Remember the constructors of a list are `[]` and `(:)`. The 'z'-argument in the `foldr` function corresponds to the `[]`-constructor. The 'f'-argument in the `foldr` function corresponds to the `(:)` constructor. The `Weird` datatype has four constructors, so we need four functions. Next, we have a parameter of the `Weird a b` type, and we want to end up with some other type of value. Even more specific: the return type of each individual function we pass to `weirdFold` will be the return type of `weirdFold` itself.

```
weirdFold :: (something1 -> c) -> (something2 -> c) -> (something3 -> c) -> (something4 -> c) -> Weird a b -> c
```

This in itself won't work. We still need the types of `something1`, `something2`, `something3` and `something4`. But since we know the constructors, this won't be much of a problem. Let's first write down a sketch for our definition. Again, I use a `where` clause, so I don't have to write the four function all the time.

```
weirdFold :: (something1 -> c) -> (something2 -> c) -> (something3 -> c) -> (something4 -> c) -> Weird a b -> c
weirdFold f1 f2 f3 f4 = weirdFold'
  where
    weirdFold' First a   = --Something of type c here
    weirdFold' Second b = --Something of type c here
    weirdFold' Third list = --Something of type c here
    weirdFold' Fourth w  = --Something of type c here
```

Again, the types and definitions of the first two functions are easy to find. The third one isn't very difficult

either, as it's just some other combination with 'a' and 'b'. The fourth one, however, is recursive, and we have to watch out. As in the case of `weirdMap`, we also need to recursively use the `weirdFold` function here. This brings us to the following, final, definition:

```
weirdFold :: (a -> c) -> (b -> c) -> [(a,b)] -> c -> (c -> c) -> Weird a b -> c
weirdFold f1 f2 f3 f4 = weirdFold'
  where
    weirdFold' First a      = f1 a
    weirdFold' Second b    = f2 b
    weirdFold' Third list  = f3 list
    weirdFold' Fourth w    = f4 (weirdFold f1 f2 f3 f4 w)
```

In which the hardest part, supplying of `f1`, `f2`, `f3` and `f4`, is left out.

Folds on recursive datatypes

Since I didn't bring enough recursiveness in the `Weird a b` datatype, here's some help for the even weirder things. Someone, please clean this up!

`weird` was a fairly nice datatype. Just one recursive constructor, which isn't even nested inside other structures. What would happen if we added a fifth constructor?

```
Fifth [Weird a b] a (Weird a a, Maybe (Weird a b))
```

A valid, and difficult, question. In general, the following rules apply:

- A function to be supplied to a fold has the same amount of arguments as the corresponding constructor.
- The type of such a function is the same as the type of the constructor.
- The only difference is that every instance of the type the constructor belongs to, should be replaced by the type of the fold.
- If a constructor is recursive, the complete fold function should be applied to the recursive part.
- If a recursive instance appears in another structure, the appropriate map function should be used

So `f5` would have the type:

```
f5 :: [c] -> a -> (Weird a a, Maybe c)
```

as the type of `Fifth` is:

```
Fifth :: [Weird a b] -> a -> (Weird a a, Maybe (Weird a b))
```

The definition of `weirdFold'` for the `Fifth` constructor will be:

```
weirdFold' Fifth list a (waa, maybe) = f5 (map (weirdFold f1 f2 f3 f4 f5) list) a (waa, maybeMap (weirdFold
  where
    maybeMap f Nothing = Nothing
    maybeMap f (Just w) = Just (f w)
```

Now note that nothing strange happens with the `weird a a` part. No `weirdFold` gets called. What's up? This is a recursion, right? Well... not really. `weird a a` has another type than `weird a b`, so it isn't a real recursion. It

isn't guaranteed that, for example, `£2` will work with something of type 'a', where it expects a type 'b'. It can be true for some cases, but not for everything.

Also look at the definition of `maybeMap`. Verify that it is indeed a map function:

- It preserves structure.
- Only types are changed.

Class declarations

Type classes are a way of ensuring you have certain operations defined on your inputs. For example, if you know a certain type *instantiates* the class `Fractional`, then you can find its reciprocal.

Please note: For programmers coming from C++, Java and other object-oriented languages: the concept of "class" in Haskell is not the same as in OO languages. There are just enough similarities to cause confusion, but not enough to let you reason by analogy with what you already know. When you work through this section try to forget everything you already know about classes and subtyping. It might help to mentally substitute the word "group" (or "interface") for "class" when reading this section. Java programmers in particular may find it useful to think of Haskell classes as being akin to Java interfaces. For C++ programmers, Haskell classes are similar to the informal notion of a "concept" used in specifying type requirements in the Standard Template Library (e.g. `InputIterator`, `EqualityComparable`, etc.)

Introduction

Haskell has several numeric types, including `Int`, `Integer` and `Float`. You can add any two numbers of the same type together, but not numbers of different types. You can also compare two numbers of the same type for equality. You can also compare two values of type `Bool` for equality, but you cannot add them together.

The Haskell type system expresses these rules using classes. A class is a template for types: it specifies the operations that the types must support. A type is said to be an "instance" of a class if it supports these operations.

For instance, here is the definition of the "Eq" class from the Standard Prelude. It defines the `==` and `/=` functions.

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

  -- Minimal complete definition:
  --   (==) or (/=)
  x /= y    = not (x == y)
  x == y    = not (x /= y)
```

This says that a type `a` is an instance of `Eq` if it supports these two functions. It also gives default definitions of the functions in terms of each other. This means that if an instance of `Eq` defines one of these functions then the other one will be defined automatically.

Here is how we declare that a type is an instance of `Eq`:

```
data Foo = Foo {x :: Integer, str :: String}
instance Eq Foo where
  (Foo x1 str1) == (Foo x2 str2) =
    (x1 == x2) && (str1 == str2)
```

There are several things to notice about this:

- The class `Eq` is defined in the standard prelude. This code sample defines the type `Foo` and then declares it to be an instance of `Eq`. The three definitions (class, data type and instance) are completely separate and there is no rule about how they are grouped. You could just as easily create a new class `Bar` and then declare the type `Integer` to be an instance of it.
- Types and classes are not the same thing. A class is a "template" for types. Again this is unlike most OO languages, where a class is also itself a type.
- The definition of `==` depends on the fact that `Integer` and `String` are also members of `Eq`. In fact almost all types in Haskell (the most notable exception being functions) are members of `Eq`.
- You can only declare types to be instances of a class if they were defined with `data` or `newtype`. Type synonyms are not allowed.

Deriving

Obviously most of the data types you create in any real program should be members of `Eq`, and for that matter a lot of them will also be members of other Standard Prelude classes such as `Ord` and `Show`. This would require large amounts of boilerplate for every new type, so Haskell has a convenient way to declare the "obvious" instance definitions using the keyword `deriving`. Using it, `Foo` would be written as:

```
data Foo = Foo {x :: Integer, str :: String}
  deriving (Eq, Ord, Show)
```

This makes `Foo` an instance of `Eq` with exactly the same definition of `==` as before, and also makes it an instance of `Ord` and `Show` for good measure. If you are only deriving from one class then you can omit the parentheses around its name, e.g.:

```
data Foo = Foo {x :: Integer, str :: String}
  deriving Eq
```

You can only use `deriving` with a limited set of built-in classes. They are:

Eq

Equality operators `==` and `/=`

Ord

Comparison operators `<` `<=` `>` `>=`. Also `min` and `max`.

Enum

For enumerations only. Allows the use of list syntax such as `[Blue .. Green]`.

Bounded

Also for enumerations, but can also be used on types that have only one constructor. Provides `minBound` and `maxBound`, the lowest and highest values that the type can take.

Show

Defines the function `show` (note the letter case of the class and function names) which converts the type to a string. Also defines some other functions that will be described later.

Read

Defines the function `read` which parses a string into a value of the type. As with `show` it also defines some other functions as well.

The precise rules for deriving the relevant functions are given in the language report. However they can generally be relied upon to be the "right thing" for most cases. The types of elements inside the data type must also be instances of the class you are deriving.

This provision of special magic for a limited set of predefined classes goes against the general Haskell philosophy that "built in things are not special". However it does save a lot of typing. Experimental work with Template Haskell is looking at how this magic, or something like it, can be extended to all classes.

Class Inheritance

Classes can inherit from other classes. For example, here is the definition of the class `Ord` from the Standard Prelude, for types that have comparison operators:

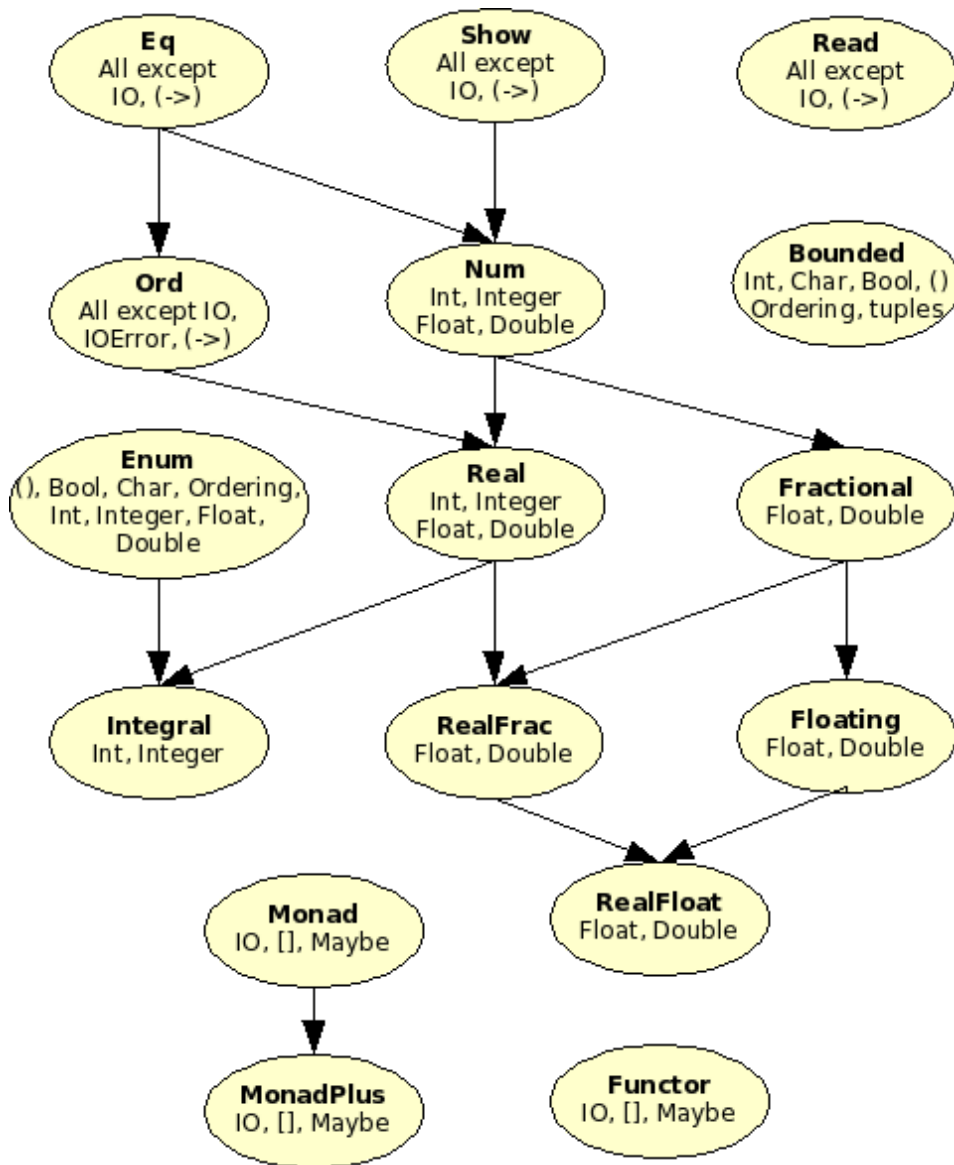
```
class (Eq a) => Ord a where
  compare      :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min     :: a -> a -> a
```

The actual definition is rather longer and includes default implementations for most of the functions. The point here is that `Ord` inherits from `Eq`. This is indicated by the `=>` symbol in the first line. It says that in order for a type to be an instance of `Ord` it must also be an instance of `Eq`, and hence must also implement the `==` and `/=` operations.

A class can inherit from several other classes: just put all the ancestor classes in the parentheses before the `=>`. Strictly speaking those parentheses can be omitted for a single ancestor, but including them acts as a visual prompt that this is not the class being defined and hence makes for easier reading.

Standard Classes

This diagram, copied from the Haskell Report, shows the relationships between the classes and types in the Standard Prelude. The names in bold are the classes. The non-bold text are the types that are instances of each class. The `(->)` refers to functions and the `[]` refers to lists.



Classes and types

Classes and Types

Simple Type Constraints

So far we have seen how to declare classes, how to declare types, and how to declare that types are instances of classes. But there is something missing. How do we declare the type of a simple arithmetic function?

```
plus x y = x + y
```

Obviously x and y must be of the same type because you can't add different types of numbers together. So how about:

```
plus :: a -> a -> a
```

which says that `plus` takes two values and returns a new value, and all three values are of the same type. But there is a problem: the arguments to `plus` need to be of a type that supports addition. Instances of the class `Num` support addition, so we need to limit the type signature to just that class. The syntax for this is:

```
plus :: Num a => a -> a -> a
```

This says that the type of the arguments to `plus` must be an instance of `Num`, which is what we want.

You can put several limits into a type signature like this:

```
foo :: (Num a, Show a, Show b) => a -> a -> b -> String
foo x y t =
  show x ++ " plus " ++ show y ++ " is " ++ show (x+y) ++ ". " ++ show t
```

This says that the arguments `x` and `y` must be of the same type, and that type must be an instance of both `Num` and `Show`. Furthermore the final argument `t` must be of some (possibly different) type that is also an instance of `Show`.

You can omit the parentheses for a single constraint, but they are required for multiple constraints. Actually it is common practice to put even single constraints in parentheses because it makes things easier to read.

More Type Constraints

You can put a type constraint in almost any type declaration. The only exception is a `type` synonym declaration. The following is not legal:

```
type (Num a) => Foo a = a -> a -> a
```

But you can say:

```
data (Num a) => Foo a = F1 a | F2 Integer
```

This declares a type `Foo` with two constructors. `F1` takes any numeric type, while `F2` takes an integer.

You can also use type parameters in `newtype` and `instance` declarations. Class inheritance (see the previous section) also uses the same syntax.

Monads

Understanding monads



This page is undergoing a major rewrite. Meanwhile, here's the previous version (http://en.wikibooks.org/w/index.php?title=Haskell/Understanding_monads&oldid=933545)

Notes and TODOs

Loose ends:

- *Explain monadic parser combinators! But in another chapter.*
- *The basic triad "rock, scissors, paper" err, "reader, writer, state" is best introduced in another chapter, maybe entitled "A Zoo of Monads". Reader must include the famous function $[(a \rightarrow b)] \rightarrow a \rightarrow [b]$ also known as *sequence!**
- *The `Random a` is too cute to not elaborate it further to probabilistic functional programming. The Monty Hall problem can be solved with that. Key: the implementation can be changed, it effectively becomes the *Set-Monad* then. Extension: `guard` and `backtracking`.*

Introduction

What is a Monad?

We're bold and present the exact definition of "monad". This should (hopefully) prevent common confusion about the definition and remove the buzzword status. Of course, this paragraph has to be really short since it doesn't explain why I should care about monads at all or what the intuition behind `bind` and `return` is.

A **monad** is a triple $(M, \text{return}, \gg=)$ consisting of a type constructor M and two polymorphic functions

$$\begin{aligned} \text{return} &:: a \rightarrow M a \\ (\gg=) &:: M a \rightarrow (a \rightarrow M b) \rightarrow M b \end{aligned}$$

that obey the following three laws

$$\begin{aligned} \text{Right unit} & \quad m \gg= \text{return} = m \\ \text{Left unit} & \quad \text{return } x \gg= f = f x \\ \text{Associativity} & \quad (m \gg= f) \gg= g = m \gg= (\lambda x. f x \gg= g) \end{aligned}$$

The operator $\gg=$ is commonly called "**bind**". Often, one simply refers to the type constructor M as the monad.

In Haskell, we can capture this definition as a type class

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

Any instance of this class `Monad` is assumed to fulfill the three laws stated above. In other words, a monad is a

type constructor for which those two functions are implemented. This class is, slightly expanded, part of the Haskell Prelude (<http://www.haskell.org/onlinereport/standard-prelude.html>) and defined in the standard library module `Control.Monad` (<http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Monad.html>) .

What use are Monads?

After maturing in the mathematical discipline of category theory for some time, monads were introduced to programming when Eugenio Moggi showed^[13] how they can unify the description of the semantics of different programming languages. Depending on the concrete monad chosen, different semantics emerge. For instance, mutable state can be modeled by the monad $M\ a = s \rightarrow (a, s)$. Lists are a monad, too, and model nondeterminism and backtracking whereas the monad $M\ a = \text{Either } e\ a$ models exceptions.

One aim of the chapters on monads is to exemplify many of these different forms of computation. Of course, it's not our main interest to study programming language semantics, but it was Philip Wadler who noticed^[14] ^[15] that we can directly implement Moggi's monads in Haskell. This is a powerful idea since each monad is a little minilanguage specially suited for a particular task. For instance, to program a state transformer, we can use a monad to model state. To solve a logic problem, we use the list monad to transparently handle backtracking. To handle failure and exceptions easily, we have the `Either e` monad. And last but not least, there is the `IO` monad to perform input/output, something that did not seem to fit well into a purely functional language. This and subsequent chapters are to guide you to through these minilanguages and to show they can simplify and structure your daily Haskell code.

But how can the rather short definition of monads given above relate all these very different forms of computation? They all share a common use pattern, namely the ability to combine two computations f and g into a compound computation $f \gg= g$ by first "executing" f and "then" binding, i.e. feeding the result to g . This is what the operator $\gg=$ captures and that's why it's called "bind". In other words, $\gg=$ is similar to function composition. Of course, depending on the underlying monad, "executing" and "then" may have quite different meanings. Don't worry if this seems too abstract now, we will detail the genesis of $\gg=$ with our first example monad in the section `#Stateful Computations`.

Stateful Computations

Example state monads, i.e. particular state types. One of them is to be treated before/in parallel to `IO a`. The others may be useful for exercises!

- *random numbers. Drawbacks: $\gg=$ is meaningless, using an infinite list of random numbers is a better abstraction. Highlights: the state is more an implementation detail than of individual importance, this makes explanation easier!*
- *name-supply. Task: label all leaves in a tree from 1 to n.*
- *Data.Map for depth first search in a graph.*
- *SOE, chapter 19: interactive robot language. Nice example monad to program in. But not a good example for implementation.*

Of course, the problem with an example is that we need to explain the example use case before plunging into $\gg=$. But it's probably worth it. Huh, it turns out that we even have to explain the $s \rightarrow (a, s)$ pattern for threading state. Seems that random numbers are easiest to explain.

Random Number Generation

We will introduce $\gg=$ with a practical example of stateful computations: random number generation. This subsection will present the example, it's the next subsection that will show the monad behind.

Computers usually create random numbers by starting with a single random number (frequently called "**seed**") and applying some arithmetic operations to it to get a new random number. By repeating this process, we get a sequence of fairly random numbers. Of course, since each number is generated in a deterministic way from the previous one, they are not truly random, but *pseudo-random numbers*. But by choosing the arithmetic operations carefully to properly "scramble" the input number, they behave like real random numbers. To give an impression of how this "scrambling" works, here's an example function that generates a pseudo-random number from a previous one:

```
-----
type Seed = Int

randomNext :: Seed -> Seed
randomNext rand = if newRand > 0 then newRand else newRand + 2147483647
  where
    newRand = 16807 * lo - 2836 * hi
    (hi,lo) = rand `divMod` 127773
-----
```

There is much research on constructing good pseudo-random number generators, but fortunately, the Haskell standard library module `System.Random` (<http://www.haskell.org/ghc/docs/latest/html/libraries/base/System-Random.html>) already implements a ready-to-use generator for us. However, its interface is best explained with `randomNext`, so we will stick to that for now.

Let's implement a function that simulates a dice roll, i.e. that returns a random number from 1 to 6. But `randomNext` uses large numbers since they can be scrambled much better, so we need to convert a `Seed` to a number from 1 to 6. This can be done by dividing the `Seed` by 6 and taking the remainder

```
-----
toDieRoll :: Seed -> Int
toDieRoll seed = (seed `mod` 6) + 1
-----
```

So, given an initial random `Seed`, we can roll a die with it

```
-----
> toDieRoll 362354 -- hand-crafted initial random seed :-)
3
-----
```

But something is missing: what if we want to roll the die a second time? For that, we have to generate a new random `Seed` from the old one via `randomNext`. In other words, we have to change the current `Seed`, i.e. the **state** of our pseudo-random number generator. In Haskell, this can be accomplished by returning the new state in the result

```
-----
rollDie :: Seed -> (Int, Seed)
rollDie seed = ((seed `mod` 6) + 1, randomNext seed)
-----
```

This is the description of a **state transformer**: an initial state (the `Seed`) is transformed to a new one while yielding a result (the `Int` between 1 and 6). We can visualize it as ... (TODO: DRAW THE PICTURE!).

To roll two dice and sum their pips, we can now feed the `Seed` from the first roll to the second roll. Of course, we have to return the new state from the second dice roll as well for our function `sumTwoDice` to be as useful as `rollDie`:

```

sumTwoDice :: Seed -> (Int, Seed)
sumTwoDice seed0 =
  let (die1, seed1) = rollDie seed0
      (die2, seed2) = rollDie seed1
  in (die1 + die2, seed2)

```

Again, a picture shows clearly how the state is passed from one `rollDie` to the next (PICTURE!). Note that `nextRandom` does not appear in the definition of `sumTwoDice`, the state change it performs is already embedded in `rollDie`. The function `sumTwoDice` merely propagates the state updates.

This is the model that `System.Random`

(<http://www.haskell.org/ghc/docs/latest/html/libraries/base/System-Random.html>) employs, so we can now elaborate on its concrete interface. The library uses two type classes: `RandomGen` and `Random`. Any instance of the former acts similar to our `Seed`, it's just called "random number generator", not "seed". This makes sense since the seed may have more complicated internals just an `Int` and is closely linked to the function that generates new pseudo-random numbers. In any case, the module exports a convenient random number generate `StdGen` and you most likely won't have to deal with the `RandomGen`-class at all.

The interesting functions are those of the class `Random`, in particular `random` and `randomR`. They are implemented for a few types like `Bool`, `Char`, `Int` etc. so you can use them to generate different random things than numbers. The function `randomR` returns a random number in a specified range, so that we can conveniently write

```

import System.Random (http://www.haskell.org/ghc/docs/latest/html/libraries/base/System-Random.html)
rollDie :: StdGen -> (Int, StdGen)
rollDie = randomR (1,6)

```

As a final note, you may want to compare random number creation in Haskell to its counterpart in imperative languages like C. In the latter, there usually is a "function" `rand()` that returns a different and random result at each call but internally updates the random seed. Since Haskell is pure, the result of a function is determined solely by its parameters and manipulating the random seed has to manifest itself in the type.

Exercises

1. Roll two dice! With `sumTwoDice` that is :-). Use `fst` to extract the result.
2. Write a function `rollNDice :: Int -> Seed -> ([Int], Seed)` that rolls dice `n` times and returns a list of the `n` results. *Extra:* If you know about infinite lists, use `unfoldr` and `take` to get the result list (but without seed this time).
3. Reimplement `Seed` and `rollDie` with `StdGen` and `random` from `System.Random` (<http://www.haskell.org/ghc/docs/latest/html/libraries/base/System-Random.html>)
4. Now that you have random numbers, do some statistical experiments with the help of `rollNDice`. For example, do a sanity check that `rollDie` is not skewed and returns each number with equal likelihood. How is the sum of pips of a double dice roll distributed? The difference? And triple rolls?

Threading the State with *bind*

>> for the state monad is easier than >>=. But it's meaningless for random numbers :-/ PICTUREs for the plumbing! Somehow shorten the discussion, mainly introduce `return` more fluently. Expanding the definitions of the new combinators as exercises to check that the new code for `sumTwoDice` is the same as the old one.

In the last subsection, we've seen that state transformers like random number generators can be modeled by functions `s -> (a, s)` where `s` is the type of the state. Such a function takes a state and returns a result of type `a` and a transformed state. However, programming with these functions is a bit tedious since we have to explicitly pass the state from one computation to the next one like in the definition of `sumTwoDice`

```
sumTwoDice :: Seed -> (Int, Seed)
sumTwoDice seed0 =
  let (die1, seed1) = rollDie seed0
      (die2, seed2) = rollDie seed1
  in (die1 + die2, seed2)
```

Each state has to be named and we have to take care to not pass the wrong state to the next function by accident.

Of course, we are Haskell programmers: if there are common patterns or boilerplate in our code, we should search for a way to abstract and capture them in a higher order function. Thus, we want to find something that can combine state transformers `s -> (a, s)` to larger ones by passing the state from one to the next. A first attempt is an operator named "**then**"

```
(>>) :: (Seed -> (a, Seed)) -> (Seed -> (b, Seed)) -> (Seed -> (b, Seed))
```

which passes the state from the first computation to the second

```
(>>) f g seed0 =
  let (result1, seed1) = f seed0
      (result2, seed2) = g seed1
  in (result2, seed2)
```

By nesting it, we can already roll a die multiple times

```
rollDie >> (rollDie >> rollDie)
```

without seeing a single state! Unfortunately, `(>>)` doesn't allow us to use the result of the first die roll in the following ones, it's simply ignored. In other words, this combinaton changes the random seed three times but only returns the pips from the last die roll. Rather pointless for random numbers, but we're on the right track. PICTURE FOR `(>>)`!

We somehow need a way to pass the result from the first computation to the second, "then" is not yet general enough to allow the implementation of `sumTwoDice`. But first, we should introduce a type synonym to simplify the type signatures

```
type Random a = Seed -> (a, Seed)
(>>)          :: Random a -> Random b -> Random b
rollDie      :: Random Int
sumTwoDice   :: Random Int
```

Astonishingly, this gives an entirely new point of view: a value of type `Random a` can be seen as a value of type

`a` that varies randomly. So, `rollDie` can be interpreted as a number between 1 and 6 that "figdets" and is sometimes "here" and sometimes "there" when asked about its value. We will explore this idea further, but for now, let's stick to our initial goal that `Random a` is a simple shortcut for a state transformer. Just take a mental note about the observation that our aim of explicitly removing the state from our functions naturally asks for removing the state from our types, too.

Now, how to pass the result from one computation to the next? Well, we may simply give it as parameter to the next one

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

In other words, the second state transformer is now replaced by a function so that its result of type `b` may depend on the previous result `a`. The implementation is almost that of `>>`

```
(>>=) f g seed0 =
  let (result1, seed1) = f seed0
      (result2, seed2) = (g result1) seed1
  in (result2, seed2)
```

with the only difference being that `g` now takes `result1` as parameter. *PICTURE!*

This combinator named "**bind**" should finally allow us to implement `sumTwoDice`. Let's see: we roll the first die and feed the result to a function that adds a second die roll to that

```
sumTwoDice :: Random Int
sumTwoDice = rollDie >>= (\die1 -> addToDie die1)
```

Adding the second die roll uses the remaining code from our original definition of `sumTwoDice`.

```
addToDie :: Int -> Random Int
addToDie die1 seed1 =
  let (die2, seed2) = rollDie seed1
  in (die1 + die2, seed2)
```

(Remember that `Random Int = Seed -> (Int, Seed)`.) That's still unsatisfactory, since we would like to avoid all explicit state and just use `>>=` a second time to feed the second dice roll to the sum

```
addToDie die1 = rollDie >>= (\die2 -> addThem die2)
  where
    addThem die2 seed2 = (die1 + die2, seed2)
```

That's the same as

```
addToDie die1 = rollDie >>= (\die2 -> (\seed2 -> (die1 + die2, seed2)) )
```

which is almost

```
addToDie die1 = rollDie >>= (\die2 -> (die1 + die2) )
```

though not quite since the latter doesn't type check since the sum has type `Int` instead of the expected `Random Int`. But we can convert the former into the latter with a helper function called "**return**"!

```

-----
addToDie die1 = rollDie >>= (\die2 -> return (die1 + die2) )
return :: a -> Random a
return x = \seed0 -> (x, seed0)
-----

```

So, `return` doesn't change the state but simply *returns* its argument as result. For random numbers, this means that `return` creates a number that isn't random at all. Last but not least, we can drop the definition of `addToDie` and directly write

```

-----
sumTwoDice :: Random Int
sumTwoDice = rollDie >>= (\die1 -> rollDie >>= (\die2 -> return (die1 + die2)))
-----

```

Exercises

1. Implement `rollNDice :: Int -> Random [Int]` from the previous subsection with `>>=` and `return`.

To conclude, the quest of automating the passing of state from one computation to the next led us to the two operations that define a monad. Of course, this is just the beginning. The reader is probably not yet accustomed to the `>>=`-combinator, how to program with it effectively? What about the three monad laws mentioned in the introduction? But before we embark to answer these questions in the next big section, let us emphasize the need for using `>>=` as a main primitive in a slightly different example in the next subsection.

Input/Output needs *bind*

IO is the one type that requires the programmer to know what a monad is, the other monads are more or less optional. It makes abstract return and bind necessary. Explaining `World -> (a, World) = IO a` and the need to hide the `World` naturally leads to return and `>>=`. I guess we need to mention somewhere that `main :: IO ()` is the link to the real world.

Performing input/output in a purely functional language like Haskell has long been a fundamental problem. How to implement operations like `getChar` which returns the latest character that the user has typed or `putChar c` which prints the character `c` on the screen? Giving `putChar` the type `getChar :: Char` is not an option, since a pure function with no arguments must be constant. We somehow have to capture that `getChar` also performs the **side effect** of interacting with the user. Likewise, a type `putChar :: Char -> ()` is useless since the only value this function can return has to be `()`.

The breakthrough came when it was realized^[16] that monads, i.e. the operations `>>=` and `return` can be used to elegantly deal with side effects. The idea is to give our two primitive operations the types

```

-----
getChar :: IO Char
putChar :: Char -> IO ()
-----

```

and interpret a value of type `IO a` as a **computation** or **action** that performs a side effect before returning the

value `a`. This is rather abstract, so a more concrete way is to interpret `IO` as a state transformer

```
type IO a = World -> (a, World)
```

that acts on and changes the "state of the world". In other words, printing a character takes the world and returns one where the character has been printed and reading a character returns a world where the character has been read. With this model, an action `echo :: IO ()` that reads a character and immediately prints it to the screen would be written as

```
echo world0 =
  let (c , world1) = getChar world0
      ((), world2) = putChar c world1
  in ((), world2)
```

Of course, this is a case for the *bind* combinator that passes the state of the world for us:

```
echo = getChar >>= putChar
```

But for `IO a`, the use of `>>=` is not a convenience, it is *mandatory*. This is because by passing around the world explicitly, we could write (either accidentally or even consciously) something that duplicates the world:

```
havoc world0 =
  let (c , world1) = getChar world0
      ((), world2) = putChar c world0
  in ((), world2)
```

Now, where does `putChar` get the character `c` from? Did the state of world roll back similar to a time travel? This makes no sense, we have to ensure that the world is used in a single-threaded way. But this is easy to achieve: we just make `IO a` an abstract data type and export only `>>=` and `return` for combining actions, together with primitive operations like `putChar`.

There's even more: the model `World -> (a,World)` for input/output just doesn't work, one of the exercises shows why. Also, there is no hope to extend it to concurrency and exceptions. In other words, it is imperative to make `>>=` for composing effectful computations `IO a` an abstract primitive operation.

Exercises

1. Write a function `putString :: String -> IO ()` that outputs a sequence of characters with the help of `putChar`.
2. The program

```
loop :: IO ()
loop = return () >> loop
```

loops forever whereas

```
loopX :: IO ()
loopX = putChar 'X' >> loopX
```

prints an infinite sequence `xxxxxx...` of `x`-s. Clearly, a user can easily distinguish them by looking on the screen. However, show that the model `IO a = World -> (a, World)` gives the same denotation \perp for both. This means that we have to abandon this model as possible semantics for `IO a`.

Programming with *bind* and *return*

Time to write programs! More complicated stuff for `Random a`. Examples to code: St.Petersburg paradox, Lewis Carroll's pillow problem. Somewhere make explicit instances of the `Monad`-class? Hm, we really need to incorporate the monad class in the type signatures. I'm not sure whether the nuclear waste metaphor is necessary?

In the last section, we showed how the two defining operations `>>=` and `return` of a monad arise as abstraction for composing state transformers. We now want to focus on how to program effectively with these.

Nuclear Waste Containers

Random a as fuzzy a. Programming would be so much easier if we had `extract :: Random a -> a`, `bind` is sooo unwieldy. Mental prevention: think of monads as "Nuclear waste containers", the waste may not leak outside at any cost. The thing closest to `extract` we can have is `join :: m (m a) -> m a`. The text probably tells too much about "monads as containers", I'm not sure what to do.

We saw that the `bind` operation takes a computation, executes it, and feeds its result to the next, like in

```
echo      = getChar >>= \char -> putChar char
sumTwoDice = rollDie >>= \die1 -> rollDie >>= \die2 -> return (die1 + die2)
```

(Note that for parsing, lambda expressions extend as far to the right as possible, so it's not necessary to put them in parentheses.) However, it could be tempting to "execute" a monadic action like `IO a` with some hypothetical function

```
extract :: IO a -> a
```

in order to conveniently formulate

```
echo = return (putChar (extract getChar))
```

Of course, such a function does not make sense. For state transformers like `Random a = Seed -> (a, Seed)`, it would have to invent a state and discard it again, thus regressing from our goal of passing the new state to the next computation.

Here's a metaphor to strengthen your mind against *extract*:

- Think of a monadic computation $M\ a$ as a container for a value of type a that is unfortunately paired with highly dangerous **nuclear waste**. Under no circumstance should this tightly sealed container be opened to *extract* the a or the nuclear waste will leak out, resulting in a catastrophe!

So, there are some like `getChar :: IO Char` or `rollDie :: Random Int` that produce a precious value but unfortunately cannot operate without tainting it with nuclear waste. But fortunately, we have our function

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

that nonetheless allows us to operate on the value contained in $M\ a$ by entering the container and applying the given function to the value inside it. This way, everything happens inside the container and no nuclear materials leak out.

Arguably, this description of "bind" probably applies better to a function

```
fmap :: Monad m => (a -> b) -> m a -> m b
fmap f m = m >>= \x -> return (f x)
```

that takes a pure function into the container to transform the value within. You may notice that this is the defining mapping for functors, i.e. every monad is a functor. Apparently, `fmap` is less general than `>>=` since the latter expects the function to be lifted into the container to produce nuclear waste, too. The best what `fmap` can do is

```
fmap' :: Monad m => (a -> (m b)) -> m a -> m (m b)
```

to produce a nested container. Of course, it is safe to open the inner container since the outer container still shields the environment from the nuclear waste

```
join :: Monad m => m (m a) -> m a
join m = m >>= id
```

In other words, we can describe the operation of `>>=` as

```
m >>= g = join (fmap g m)
```

i.e. it lifts a waste-producing computation into the container and flattens the resulting nested containers.

We will explore this further in #Monads as Containers.

Of course, we shouldn't take the nuclear waste metaphor too literally, since there usually is some way to "run" the computation. For instance, random numbers can be observed as an infinite list of random numbers produced by an initial random seed.

```
run :: Random a -> (Seed -> [a])
```

Only the `IO` monad is a primitive in Haskell. How do we "run" it, then? The answer is that the link of a Haskell

program to outside world is the function

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

which will be run by the operating system. In other words, the Haskell program itself ultimately produces nuclear waste, so there is no need to extract `IO a -> a`.

Exercises

1. Implement `run` with `unfoldr`.

do-Notation

A common way to write the composition of multiple monadic computations is

```
sumTwoDice = do
  die1 <- rollDie
  die2 <- rollDie
  return (die1 + die2)
```

Control Structures

Needs a better title. Introduce `sequence`, `fmap`, `liftMn`, `forM`, `mapM` and `friends`.

The three Monad Laws

In the state monad, `return` doesn't touch the state. That can be formulated abstractly with the first two monad laws. Hm, what about the third? How to motivate that?

Monads as containers

Needs a better title. Introduce the second instances of monads, namely `[a]` and `Maybe a`. Shows that the operations `return` and `bind` are applicable to quite a range of problems. The more "exotic" example

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

belongs here, too, probably as exercise.

Lists

`concatMap` and `sequence..`

Maybe

Maybe `Either`, too?

References

1. ↑ *At least as far as types are concerned, but we're trying to avoid that word :)*
2. ↑ *More technically, `fst` and `snd` have types which limit them to pairs. It would be impossible to define projection functions on tuples in general, because they'd have to be able to accept tuples of different sizes, so the type of the function would vary.*
3. ↑ *In fact, these are one and the same concept in Haskell.*
4. ↑ *This isn't quite what `chr` and `ord` do, but that description fits our purposes well, and it's close enough.*
5. ↑ *To make things even more confusing, there's actually even more than one type for integers! Don't worry, we'll come on to this in due course.*
6. ↑ *This has been somewhat simplified to fit our purposes. Don't worry, the essence of the function is there.*
7. ↑ *Some of the newer type system extensions to GHC do break this, however, so you're better off just always putting down types anyway.*
8. ↑ This is a slight lie. That type signature would mean that you can compare two values of any type whatsoever, but this clearly isn't true: how can you see if two functions are equal? Haskell includes a kind of 'restricted polymorphism' that allows type variables to range over some, but not all types. Haskell implements this using *type classes*, which we'll learn about later. In this case, the correct type of `(==)` is `Eq a => a -> a -> Bool`.
9. ↑ In mathematics, $n!$ normally means the factorial of n , but that syntax is impossible in Haskell, so we don't use it here.
10. ↑ Actually, defining the factorial of 0 to be 1 is not just arbitrary; it's because the factorial of 0 represents an empty product.
11. ↑ This is no coincidence; without mutable variables, recursion is the only way to implement control structures. This might sound like a limitation until you get used to it (it isn't, really).
12. ↑ Actually, it's using a function called `fold1`, which actually does the recursion.
13. ↑ Moggi, Eugenio (1991). "Notions of Computation and Monads". *Information and Computation* **93** (1).
14. ↑ w:Philip Wadler. Comprehending Monads (<http://citeseer.ist.psu.edu/wadler92comprehending.html>) . Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice. 1990.
15. ↑ w:Philip Wadler. The Essence of Functional Programming (<http://citeseer.ist.psu.edu/wadler92essence.html>) . Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1992.
16. ↑ Simon Peyton Jones, Philip Wadler (1993). "Imperative functional programming" (<http://homepages.inf.ed.ac.uk/wadler/topics/monads.html#imperative>) . *20'th Symposium on Principles of Programming Languages*.

Advanced monads

This chapter follows on from Understanding monads, and explains a few more of the more advanced concepts.

Monads as computations

The concept

A metaphor we explored in the last chapter was that of *monads as containers*. That is, we looked at what monads are in terms of their structure. What was touched on but not fully explored is *why we use monads*. After all, monads structurally can be very simple, so why bother at all?

The secret is in the view that each monad represents a *different type of computation*. Here, and in the rest of this chapter, a 'computation' is simply a function call: we're computing the result of this function. In a minute, we'll give some examples to explain what we mean by this, but first, let's re-interpret our basic monadic operators:

```
>>=
```

The `>>=` operator is used to *sequence two monadic computations*. That means it runs the first computation, then feeds the output of the first computation into the second and runs that too.

return

`return x`, in computation-speak, is simply the computation that has result `x`, and 'does nothing'. The meaning of the latter phrase will become clear when we look at `State` below.

So how does the computations analogy work in practice? Let's look at some examples.

The Maybe monad

Computations in the Maybe monad (that is, function calls which result in a type wrapped up in a `Maybe`) represent *computations that might fail*. The easiest example is with lookup tables. A lookup table is a table which relates *keys* to *values*. You *look up* a value by knowing its key and using the lookup table. For example, you might have a lookup table of contact names as keys to their phone numbers as the values in a phonebook application. One way of implementing lookup tables in Haskell is to use a list of pairs: `[(a, b)]`. Here `a` is the type of the keys, and `b` the type of the values. Here's how the phonebook lookup table might look:

```
-----
phonebook :: [(String, String)]
phonebook = [ ("Bob",    "01788 665242"),
              ("Fred",  "01624 556442"),
              ("Alice", "01889 985333"),
              ("Jane",  "01732 187565") ]
-----
```

The most common thing you might do with a lookup table is look up values! However, this computation might fail. Everything's fine if we try to look up one of "Bob", "Fred", "Alice" or "Jane" in our phonebook, but what if we were to look up "Zoe"? Zoe isn't in our phonebook, so the lookup has failed. Hence, the Haskell function to look up a value from the table is a `Maybe` computation:

```
-----
lookup :: Eq a => a -- a key
        -> [(a, b)] -- the lookup table to use
        -> Maybe b  -- the result of the lookup
-----
```

Lets explore some of the results from `lookup`:

```
-----
Prelude> lookup "Bob" phonebook
Just "01788 665242"
Prelude> lookup "Jane" phonebook
Just "01732 187565"
Prelude> lookup "Zoe" phonebook
Nothing
-----
```

Now let's expand this into using the full power of the monadic interface. Say, we're now working for the government, and once we have a phone number from our contact, we want to look up this phone number in a big, government-sized lookup table to find out the registration number of their car. This, of course, will be

another `Maybe`-computation. But if they're not in our phonebook, we certainly won't be able to look up their registration number in the governmental database! So what we need is a function that will take the results from the first computation, and put it into the second lookup, but only if we didn't get `Nothing` the first time around. If we *did* indeed get `Nothing` from the first computation, or if we get `Nothing` from the second computation, our final result should be `Nothing`.

```
comb :: Maybe a -> (a -> Maybe b) -> Maybe b
comb Nothing _ = Nothing
comb (Just x) f = f x
```

Observant readers may have guessed where we're going with this one. That's right, `comb` is just `>>=`, but restricted to `Maybe`-computations. So we can chain our computations together:

```
getRegistrationNumber :: String -- their name
                    -> Maybe String -- their registration number
getRegistrationNumber name = lookup name phonebook >>= (\number -> lookup number governmentalDatabase)
```

If we then wanted to use the result from the governmental database lookup in a third lookup (say we want to look up their registration number to see if they owe any car tax), then we could extend our `getRegistrationNumber` function:

```
getTaxOwed :: String -- their name
           -> Maybe Double -- the amount of tax they owe
getTaxOwed name = lookup name phonebook >>= (\number -> lookup number governmentalDatabase) >>= (\registration ->
```

Or, using the `do`-block style:

```
getTaxOwed name = do
  number <- lookup name phonebook
  registration <- lookup number governmentalDatabase
  lookup registration taxDatabase
```

Let's just pause here and think about what would happen if we got a `Nothing` anywhere. Trying to use `>>=` to combine a `Nothing` from one computation with another function will result in the `Nothing` being carried on and the second function ignored (refer to our definition of `comb` above if you're not sure). That is, a `Nothing` at *any stage* in the large computation will result in a `Nothing` overall, regardless of the other functions! Thus we say that the structure of the `Maybe` monad *propagates failures*.

An important thing to note is that we're not by any means restricted to lookups! There are many, many functions whose results could fail and therefore use `Maybe`. You've probably written one or two yourself. Any computations in `Maybe` can be combined in this way.

Summary

The important features of the `Maybe` monad are that:

1. It represents computations that could fail.
2. It propagates failure.

The List monad

Computations that are in the list monad (that is, they end in a type `[a]`) represent *computations with zero or more valid answers*. For example, say we are modelling the game of noughts and crosses (known as tic-tac-toe in some parts of the world). An interesting (if somewhat contrived) problem might be to find all the possible ways the game could progress: find the possible states of the board 3 turns later, given a certain board configuration (i.e. a game in progress).

Here is the instance declaration for the list monad:

```
instance Monad [] where
  return a = [a]
  xs >>= f = concat (map f xs)
```

As monads are only really useful when we're chaining computations together, let's go into more detail on our example. The problem can be boiled down to the following steps:

1. Find the list of possible board configurations for the next turn.
2. Repeat the computation for each of these configurations: replace each configuration, call it *C*, with the list of possible configurations of the turn after *C*.
3. We will now have a list of lists (each sublist representing the turns after a previous configuration), so in order to be able to repeat this process, we need to collapse this list of lists into a single list.

This structure should look similar to the monadic instance declaration above. Here's how it might look, without using the list monad:

```
getNextConfigs :: Board -> [Board]
getNextConfigs = undefined -- details not important
```

```
tick :: [Board] -> [Board]
tick bds = concatMap getNextConfigs bds
```

```
find3rdConfig :: Board -> [Board]
find3rdConfig bd = tick $ tick $ tick [bd]
```

(`concatMap` is a handy function for when you need to concat the results of a map: `concatMap f xs = concat (map f xs)`.) Alternatively, we could define this with the list monad:

```
find3rdConfig :: Board -> [Board]
find3rdConfig bd0 = do
  bd1 <- getNextConfigs bd0
  bd2 <- getNextConfigs bd1
  bd3 <- getNextConfigs bd2
  return bd3
```

List comprehensions

An interesting thing to note is how similar list comprehensions and the list monad are. For example, the classic function to find Pythagorean triples:

```
pythags = [ (x, y, z) | z <- [1..], x <- [1..z], y <- [x..z], x^2 + y^2 == z^2 ]
```

This can be directly translated to the list monad:

```

import Control.Monad (guard)

pythags = do
  z <- [1..]
  x <- [1..z]
  y <- [x..z]
  guard (x^2 + y^2 == z^2)
  return (x, y, z)

```

The only non-trivial element here is `guard`. This is explained in the next module, Additive monads.

The State monad

The State monad actually makes a lot more sense when viewed as a computation, rather than a container. Computations in State represents computations that *depend on and modify some internal state*. For example, say you were writing a program to model the three body problem (http://en.wikipedia.org/wiki/Three_body_problem#Three_body_problem) . The internal state would be the positions, masses and velocities of all three bodies. Then a function, to, say, get the acceleration of a specific body would need to reference this state as part of its calculations.

The other important aspect of computations in State is that they can modify the internal state. Again, in the three-body problem, you could write a function that, given an acceleration for a specific body, updates its position.

The State monad is quite different from the Maybe and the list monads, in that it doesn't represent the *result* of a computation, but rather a certain property of the computation itself.

What we do is model computations that depend on some internal state as functions which take a state parameter. For example, if you had a function `f :: String -> Int -> Bool`, and we want to modify it to make it depend on some internal state of type `s`, then the function becomes `f :: String -> Int -> s -> Bool`. To allow the function to change the internal state, the function returns a pair of (new state, return value). So our function becomes `f :: String -> Int -> s -> (s, Bool)`

It should be clear that this method is a bit cumbersome. However, the types aren't the worst of it: what would happen if we wanted to run two stateful computations, call them `f` and `g`, one after another, passing the result of `f` into `g`? The second would need to be passed the new state from running the first computation, so we end up 'threading the state':

```

fThenG :: (s -> (s, a)) -> (a -> s -> (s, b)) -> s -> (s, b)
fThenG f g s =
  let (s', v) = f s      -- run f with our initial state s.
      (s'', v') = g v s' -- run g with the new state s' and the result of f, v.
  in (s'', v')          -- return the latest state and the result of g

```

All this 'plumbing' can be nicely hidden by using the State monad. The type constructor `State` takes two type parameters: the type of its environment (internal state), and the type of its output. So `State s a` indicates a stateful computation which depends on, and can modify, some internal state of type `s`, and has a result of type `a`. How is it defined? Well, simply as a function that takes some state and returns a pair of (new state, value):

```

newtype State s a = State (s -> (s, a))

```

The above example of `fThenG` is, in fact, the definition of `>>=` for the State monad, which you probably

remember from the first monads chapter.

The meaning of return

We mentioned right at the start that `return x` was the computation that 'did nothing' and just returned `x`. This idea only really starts to take on any meaning in monads with side-effects, like `State`. That is, computations in `State` have the opportunity to change the outcome of later computations by modifying the internal state. It's a similar situation with `IO` (because, of course, `IO` is just a special case of `State`).

`return x` doesn't do this. A computation produced by `return` generally won't have any side-effects. The monad law `return x >>= f == f x` basically guarantees this, for most uses of the term 'side-effect'.

Further reading

- A tour of the Haskell Monad functions (<http://members.chello.nl/hjgtuyl/tourdemonad.html>) by Henk-Jan van Tuyl
- All about monads (http://www.haskell.org/all_about_monads/html/index.html) by Jeff Newbern explains well the concept of monads as computations, using good examples. It also has a section outlining all the major monads, explains each one in terms of this computational view, and gives a full example.

MonadPlus

`MonadPlus` is a typeclass whose instances are monads which represent a number of computations.

Introduction

You may have noticed, whilst studying monads, that the `Maybe` and list monads are quite similar, in that they both represent the number of results a computation can have. That is, you use `Maybe` when you want to indicate that a computation can fail somehow (i.e. it can have 0 or 1 result), and you use the list monad when you want to indicate a computation could have many valid answers (i.e. it could have 0 results -- a failure -- or many results).

Given two computations in one of these monads, it might be interesting to amalgamate these: find *all* the valid solutions. I.e. given two lists of valid solutions, to find all of the valid solutions, you simply concatenate the lists together. It's also useful, especially when working with folds, to require a 'zero results' value (i.e. failure). For lists, the empty list represents zero results.

We combine these two features into a typeclass:

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Here are the two instance declarations for `Maybe` and the list monad:

```

instance MonadPlus [] where
  mzero = []
  mplus = (++)

instance MonadPlus Maybe where
  mzero = Nothing
  Nothing `mplus` Nothing = Nothing -- 0 solutions + 0 solutions = 0 solutions
  Just x `mplus` Nothing = Just x -- 1 solution + 0 solutions = 1 solution
  Nothing `mplus` Just x = Just x -- 0 solutions + 1 solution = 1 solution
  Just x `mplus` Just y = Just x -- 1 solution + 1 solution = 2 solutions,
  -- but as Maybe can only have up to one
  -- solution, we disregard the second one.

```

Also, if you import `Control.Monad.Error`, then `(Either e)` becomes an instance:

```

instance (Error e) => MonadPlus (Either e) where
  mzero = Left noMsg
  Left _ `mplus` n = n
  Right x `mplus` _ = Right x

```

Remember that `(Either e)` is similar to `Maybe` in that it represents computations that can fail, but it allows the failing computations to include an error message. Typically, `Left s` means a failed computation with error message `s`, and `Right x` means a successful computation with result `x`.

Example

A traditional way of parsing an input is to write functions which consume it, one character at a time. That is, they take an input string, then chop off ('consume') some characters from the front if they satisfy certain criteria (for example, you could write a function which consumes one uppercase character). However, if the characters on the front of the string don't satisfy these criteria, the parsers have *failed*, and therefore they make a valid candidate for a `Maybe`.

Here we use `mplus` to run two parsers *in parallel*. That is, we use the result of the first one if it succeeds, but if not, we use the result of the second. If that too fails, then our whole parser returns `Nothing`.

```

-- | Consume a digit in the input, and return the digit that was parsed. We use
-- a do-block so that if the pattern match fails at any point, fail of the
-- the Maybe monad (i.e. Nothing) is returned.
digit :: Int -> String -> Maybe Int
digit i s | i > 9 || i < 0 = Nothing
          | otherwise     = do
  let (c:_) = s
      if read [c] == i then Just i else Nothing

-- | Consume a binary character in the input (i.e. either a 0 or an 1)
binChar :: String -> Maybe Int
binChar s = digit 0 s `mplus` digit 1 s

```

The MonadPlus laws

Instances of `MonadPlus` are required to fulfill several rules, just as instances of `Monad` are required to fulfill the three monad laws. Unfortunately, these laws aren't set in stone anywhere and aren't fully agreed on. For example, the Haddock documentation

(<http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Monad.html#t%3AMonadPlus>) for `Control.Monad` quotes them as:

```
mzero >>= f = mzero
y >> mzero = mzero
```

All About Monads (http://www.haskell.org/all_about_monads/html/laws.html#zero) quotes the above two, but adds:

```
mzero `mplus` m = m
m `mplus` mzero = m
```

There are even more sets of laws available, and therefore you'll sometimes see monads like IO being used as a MonadPlus. The Haskell Wiki page (<http://www.haskell.org/haskellwiki/MonadPlus>) for MonadPlus has more information on this. *TODO: should that information be copied here?*

Useful functions

Beyond the basic `mplus` and `mzero` themselves, there are a few functions you should know about:

msum

A very common task when working with instances of MonadPlus is to take a list of the monad, e.g. `[Maybe a]` or `[[a]]`, and fold down the list with `mplus`. `msum` fulfills this role:

```
msum :: MonadPlus m => [m a] -> m a
msum = foldr mplus mzero
```

A nice way of thinking about this is that it generalises the list-specific `concat` operation. Indeed, for lists, the two are equivalent. For `Maybe` it finds the first `Just x` in the list, or returns `Nothing` if there aren't any.

guard

This is a very nice function which you have almost certainly used before, without knowing about it. It's used in list comprehensions, as we saw in the previous chapter. List comprehensions can be decomposed into the list monad, as we saw:

```
pythags = [ (x, y, z) | x <- [1..], y <- [x..], z <- [y..], x^2 + y^2 == z^2 ]
```

The previous can be considered syntactic sugar for:

```
pythags = do
  x <- [1..]
  y <- [x..]
  z <- [y..]
  guard (x^2 + y^2 == z^2)
  return (x, y, z)
```

`guard` looks like this:

```
guard :: MonadPlus m => Bool -> m ()
guard True  = return ()
guard False = mzero
```

Concretely, `guard` will reduce a `do`-block to `mzero` if its predicate is `False`. By the very first law stated in the 'MonadPlus laws' section above, an `mzero` on the left-hand side of an `>>=` operation will produce `mzero` again. As `do`-blocks are decomposed to lots of expressions joined up by `>>=`, an `mzero` at any point will cause the entire `do`-block to become `mzero`.

To further illustrate that, we will examine `guard` in the special case of the list monad, extending on the `pythags` function above. First, here is `guard` defined for the list monad:

```
guard :: Bool -> [()]
guard True  = [()]
guard False = []
```

`guard` *blocks off* a route. For example, in `pythags`, we want to block off all the routes (or combinations of `x`, `y` and `z`) where `x2 + y2 == z2` is `False`. Let's look at the expansion of the above `do`-block to see how it works:

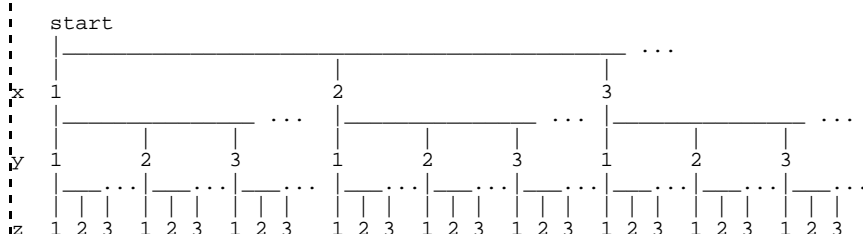
```
pythags =
  [1..] >>= \x ->
  [x..] >>= \y ->
  [y..] >>= \z ->
  guard (x2 + y2 == z2) >>= \_
  return (x, y, z)
```

Replacing `>>=` and `return` with their definitions for the list monad (and using some `let`-bindings to make things prettier), we obtain:

```
pythags =
  let ret x y z = [(x, y, z)]
      gd x y z = concatMap (\_ -> ret x y z) (guard $ x2 + y2 == z2)
      doZ x y   = concatMap (gd x y) [y..]
      doY x     = concatMap (doZ x ) [x..]
      doX       = concatMap (doY   ) [1..]
  in doX
```

Remember that `guard` returns the empty list in the case of its argument being `False`. Mapping across the empty list produces the empty list, no matter what function you pass in. So the empty list produced by the call to `guard` in the binding of `gd` will cause `gd` to be the empty list, and therefore `ret` to be the empty list.

To understand why this matters, think about list-computations as a tree. With our Pythagorean triple algorithm, we need a branch starting from the top for every choice of `x`, then a branch from each of these branches for every value of `y`, then from each of these, a branch for every value of `z`. So the tree looks like this:



Any combination of *x*, *y* and *z* represents a route through the tree. Once all the functions have been applied, each branch is concatenated together, starting from the bottom. Any route where our predicate doesn't hold evaluates to an empty list, and so has no impact on this concat operation.

Exercises

1. Prove the MonadPlus laws for Maybe and the list monad.
2. We could augment our above parser to involve a parser for any character:

```
-----
-- | Consume a given character in the input, and return the the character we
--   just consumed, paired with rest of the string. We use a do-block so that
--   if the pattern match fails at any point, fail of the Maybe monad (i.e.
--   Nothing) is returned.
char :: Char -> String -> Maybe (Char, String)
char c s = do
  let (c':s') = s
      if c == c' then Just (c, s') else Nothing
-----
```

It would then be possible to write a `hexChar` function which parses any valid hexadecimal character (0-9 or a-f). Try writing this function (hint: `map digit [0..9] :: [Maybe Int]`).

3. More to come...

Relationship with Monoids

TODO: is this at all useful? (If you don't know anything about the Monoid data structure, then don't worry about this section. It's just a bit of a muse.)

Monoids are a data structure with two operations defined: an identity (or 'zero') and a binary operation (or 'plus'), which satisfy some axioms.

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

For example, lists form a trivial monoid:

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

Note the usage of `[a]`, not `[]`, in the instance declaration. Monoids are not necessarily 'containers' of anything. For example, the integers (or indeed even the naturals) form two possible monoids:

```
-----
newtype AdditiveInt      = AI Int
newtype MultiplicativeInt = MI Int

instance Monoid AdditiveInt where
  mempty      = AI 0
  AI x `mappend` AI y = AI (x + y)

instance Monoid MultiplicativeInt where
  mempty      = MI 1
  MI x `mappend` MI y = MI (x * y)
-----
```

(A nice use of the latter is to keep track of probabilities.)

Monoids, then, look very similar to MonadPlus instances. Both feature concepts of a zero and plus, and indeed MonadPlus can be a subclass of Monoid (the following is not Haskell 98, but works with `-fglasgow-exts`):

```
instance MonadPlus m => Monoid (m a) where
  mempty = mzero
  mappend = mplus
```

However, they work at different levels. As noted, there is no requirement for monoids to be any kind of container. More formally, monoids have kind `*`, but instances of MonadPlus, as they're Monads, have kind `* -> *`.

Monad transformers

Introduction

Monad transformers are monads too!

Monad transformers are special variants of standard monads that facilitate the combining of monads. For example, `ReaderT Env IO a` is a computation which can read from some environment of type `Env`, can do some `IO` and returns a type `a`. Their type constructors are parameterized over a monad type constructor, and they produce combined monadic types. In this tutorial, we will assume that you understand the internal mechanics of the monad abstraction, what makes monads "tick". If, for instance, you are not comfortable with the bind operator (`>>=`), we

would recommend that you first read Understanding monads.

Transformers are cousins

A useful way to look at transformers is as *cousins* of some **base monad**. For example, the monad `ListT` is a cousin of its base monad `List`. Monad transformers are typically implemented almost exactly the same way that their cousins are, only more complicated because they are trying to thread some inner monad through.

The standard monads of the monad template library all have transformer versions which are defined consistently with their non-transformer versions. However, it is not the case that all monad transformers apply the same transformation. We have seen that the `ContT` transformer turns continuations of the form `(a->r)->r` into continuations of the form `(a->m r)->m r`. The `StateT` transformer is different. It turns state transformer functions of the form `s->(a,s)` into state transformer functions of the form `s->m (a,s)`. In general, there is no magic formula to create a transformer version of a monad — the form of each transformer depends on what makes sense in the context of its non-transformer type.

Standard Monad	Transformer Version	Original Type	Combined Type
Error	ErrorT	<code>Either e a</code>	<code>m (Either e a)</code>
State	StateT	<code>s -> (a,s)</code>	<code>s -> m (a,s)</code>
Reader	ReaderT	<code>r -> a</code>	<code>r -> m a</code>

Writer	WriterT	(a, w)	m (a, w)
Cont	ContT	(a -> r) -> r	(a -> m r) -> m r

In the table above, most of the transformers `FooT` differ from their base monad `Foo` by the wrapping of the result type (right-hand side of the `->` for function kinds, or the whole type for non-function types) in the threaded monad (`m`). The `Cont` monad has two "results" in its type (it maps functions to values), and so `ContT` wraps both in the threaded monad. In other words, the commonality between all these transformers is like so, with some abuse of syntax:

Original Kind	Combined Kind
*	m *
* -> *	* -> m *
(* -> *) -> *	(* -> m *) -> m *

Implementing transformers

The key to understanding how monad transformers work is understanding how they implement the `bind (>>=)` operator. You'll notice that this implementation very closely resembles that of their standard, non-transformer cousins.

Transformer type constructors

Type constructors play a fundamental role in Haskell's monad support. Recall that `Reader r a` is the type of values of type `a` within a `Reader` monad with environment of type `r`. The type constructor `Reader r` is an instance of the `Monad` class, and the `runReader :: Reader r a -> r -> a` function performs a computation in the `Reader` monad and returns the result of type `a`.

A transformer version of the `Reader` monad, called `ReaderT`, exists which adds a monad type constructor as an addition parameter. `ReaderT r m a` is the type of values of the combined monad in which `Reader` is the **base monad** and `m` is the **inner monad**.

`ReaderT r m` is an instance of the monad class, and the `runReaderT :: ReaderT r m a -> r -> m a` function performs a computation in the combined monad and returns a result of type `m a`.

The Maybe transformer

We begin by defining the data type for the `Maybe` transformer. Our `MaybeT` constructor takes a single argument. Since transformers have the same data as their non-transformer cousins, we will use the `newtype` keyword. We could very well have chosen to use `data`, but that introduces needless overhead.

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

This might seem a little off-putting at first, but it's actually simpler than it looks.

syntactic sugar

The constructor for `MaybeT` takes a single argument, of type `m (Maybe a)`. That is all. We use some syntactic sugar so that you can see `MaybeT` as a record, and access the value of this single argument by calling `runMaybeT`. One trick to understanding this is to see monad transformers as sandwiches: the bottom slice of the sandwich is the **base** monad (in this case, `Maybe`). The filling is the **inner** monad, `m`. And the top slice is the monad transformer `MaybeT`. The purpose of the `runMaybeT` function is simply to remove this top slice from the sandwich. What is the type of `runMaybeT`? It is `(MaybeT m a) -> m (Maybe a)`.

As we mentioned in the beginning of this tutorial, monad transformers are monads too. Here is a partial implementation of the `MaybeT` monad. To understand this implementation, it really helps to know how its simpler cousin `Maybe` works. For comparison's sake, we put the two monad implementations side by side

Note

Note the use of 't', 'm' and 'b' to mean 'top', 'middle', 'bottom' respectively

Maybe	MaybeT
<pre>instance Monad Maybe where b_v >>= f = -- case b_v of Nothing -> Nothing Just v -> f v</pre>	<pre>instance (Monad m) => Monad (MaybeT m) where tmb_v >>= f = MaybeT \$ runMaybeT tmb_v >>= \b_v -> case b_v of Nothing -> return Nothing Just v -> runMaybeT \$ f v</pre>

You'll notice that the `MaybeT` implementation looks a lot like the `Maybe` implementation of `bind`, with the exception that `MaybeT` is doing a lot of extra work. This extra work consists of unpacking the two extra layers of monadic sandwich (note the convention `topMidBot` to reflect the sandwich layers) and packing them up. If you really want to cut into the meat of this, read on. If you think you've understood up to here, why not try the following exercises:

Exercises

1. Implement the return function for the `MaybeT` monad
2. Rewrite the implementation of the bind operator `>>=` to be more concise.

Dissecting the bind operator

So what's going on here? You can think of this as working in three phases: first we remove the sandwich layer by layer, and then we apply a function to the data, and finally we pack the new value into a new sandwich

Unpacking the sandwich: Let us ignore the `MaybeT` constructor for now, but note that everything that's going on after the `$` is happening within the `m` monad and not the `MaybeT` monad!

1. The first step is to remove the top slice of the sandwich by calling `runMaybeT topMidBotV`
2. We use the bind operator (`>>=`) to remove the second layer of the sandwich -- remember that we are working in the confines of the `m` monad.
3. Finally, we use `case` and pattern matching to strip off the bottom layer of the sandwich, leaving behind the actual data with which we are working

Packing the sandwich back up:

- If the bottom layer was `Nothing`, we simply `return Nothing` (which gives us a 2-layer sandwich). This value then goes to the `MaybeT` constructor at the very beginning of this function, which adds the top layer and gives us back a full sandwich.
- If the bottom layer was `Just v` (note how we have pattern-matched that bottom slice of monad off): we apply the function `f` to it. But now we have a problem: applying `f` to `v` gives a full three-layer sandwich, which would be absolutely perfect except for the fact that we're now going to apply the `MaybeT` constructor to it and get a type clash! So how do we avoid this? By first running `runMaybeT` to peel the top slice off so that the `MaybeT` constructor is happy when you try to add it back on.

The List transformer

Just as with the `Maybe` transformer, we create a datatype with a constructor that takes one argument:

```
newtype ListT m a = ListT { runListT :: m [a] }
```

The implementation of the `ListT` monad is also strikingly similar to its cousin, the `List` monad. We do exactly the same things for `List`, but with a little extra support to operate within the inner monad `m`, and to pack and unpack the monadic sandwich `ListT - m - List`.

List	ListT
<pre>instance Monad [] where b_v >>= f = -- let x = map f b_v in concat x</pre>	<pre>instance (Monad m) => Monad (ListT m) where tmb_v >>= f = ListT \$ runListT tmb_v >>= \b_v -> mapM (runListT . f) b_v >>= \x -> return (concat x)</pre>

Exercises

1. Dissect the bind operator for the `(ListT m)` monad. For example, which do we now have `mapM` and `return`?
2. Now that you have seen two simple monad transformers, write a monad transformer `IdentityT`, which would be the transforming cousin of the `Identity` monad.
3. Would `IdentityT SomeMonad` be equivalent to `SomeMonadT Identity` for a given monad and its transformer cousin?

Lifting

FIXME: insert introduction

liftM

We begin with a notion which, strictly speaking, isn't about monad transformers. One small and surprisingly useful function in the standard library is `liftM`, which as the API states, is meant for lifting non-monadic functions into monadic ones. Let's take a look at that type:

```
liftM :: Monad m => (a1 -> r) -> m a1 -> m r
```

So let's see here, it takes a function `(a1 -> r)`, takes a monad with an `a1` in it, applies that function to the `a1`, and returns the result. In my opinion, the best way to understand this function is to see how it is used. The following pieces of code all mean the same thing

do notation

liftM

liftM as an operator

```
do foo <- someMonadicThing; liftM myFn someMonadicThing; myFn `liftM` someMonadicThing
return (myFn foo)
```

What made the light bulb go off for me is this third example, where we use `liftM` as an operator. `liftM` is just a monadic version of `($\$$)!`

non monadic

monadic

```
myFn $ aNonMonadicThing; myFn `liftM` someMonadicThing
```

Exercises

1. How would you write `liftM`? You can inspire yourself from the the first example

lift

When using combined monads created by the monad transformers, we avoid having to explicitly manage the inner monad types, resulting in clearer, simpler code. Instead of creating additional `do`-blocks within the computation to manipulate values in the inner monad type, we can use lifting operations to bring functions from the inner monad into the combined monad.

Recall the `liftM` family of functions which are used to lift non-monadic functions into a monad. Each monad transformer provides a `lift` function that is used to lift a monadic computation into a combined monad.

The `MonadTrans` class is defined in `Control.Monad.Trans`

(<http://www.haskell.org/ghc/docs/latest/html/base/Control.Monad.Trans.html>) and provides the single function `lift`. The `lift` function lifts a monadic computation in the inner monad into the combined monad.

```
class MonadTrans t where
  lift :: (Monad m) => m a -> t m a
```

Monads which provide optimized support for lifting IO operations are defined as members of the `MonadIO` class, which defines the `liftIO` function.

```
class (Monad m) => MonadIO m where
  liftIO :: IO a -> m a
```

Using `lift`

Implementing `lift`

Implementing `lift` is usually pretty straightforward. Consider the transformer `MaybeT`:

```
instance MonadTrans MaybeT where
  lift mon = MaybeT (mon >>= return . Just)
```

We begin with a monadic value (of the inner monad), the middle layer, if you prefer the monadic sandwich analogy. Using the bind operator and a type constructor for the base monad, we slip the bottom slice (the base monad) under the middle layer. Finally we place the top slice of our sandwich by using the constructor `MaybeT`. So using the lift function, we have transformed a lowly piece of sandwich filling into a bona-fide three-layer monadic sandwich.

As with our implementation of the `Monad` class, the bind operator is working within the confines of the inner monad.

Exercises

1. Why is it that the `lift` function has to be defined separately for each monad, where as `liftM` can be defined in a universal way?
2. Implement the `lift` function for the `ListT` transformer.
3. How would you lift a regular function into a monad transformer?
Hint: very easily.

The State monad transformer

Previously, we have pored over the implementation of two very simple monad transformers, `MaybeT` and `ListT`. We then took a short detour to talk about lifting a monad into its transformer variant. Here, we will bring the two ideas together by taking a detailed look at the implementation of one of the more interesting transformers in the standard library, `StateT`. Studying this transformer will build insight into the transformer mechanism that you can call upon when using monad transformers in your code. You might want to review the section on the `State` monad before continuing.

Just as the `State` monad was built upon the definition

```
newtype State s a = State { runState :: (s -> (a,s)) }
```

the `StateT` transformer is built upon the definition

```
newtype StateT s m a = StateT { runStateT :: (s -> m (a,s)) }
```

`StateT s` is an instance of both the `Monad` class and the `MonadState s` class, so

`StateT s m` should also be members of the `Monad` and `MonadState s` classes. Furthermore, if `m` is an instance of `MonadPlus`,

`StateT s m` should also be a member of `MonadPlus`.

To define `StateT s m` as a `Monad` instance:

State	StateT
<pre>newtype State s a = State { runState :: (s -> (a,s)) } instance Monad (State s) where return a = State \$ \s -> (a,s) (State x) >>= f = State \$ \s -> let (v,s') = x s in runState (f v) s'</pre>	<pre>newtype StateT s m a = StateT { runStateT :: (s -> m (a,s)) } instance (Monad m) => Monad (StateT s m) where return a = StateT \$ \s -> return (a,s) (StateT x) >>= f = StateT \$ \s -> do -- get new value, state (v,s') <- x s -- apply bound function to get new state transformation fn (StateT x') <- return \$ f v -- apply the state transformation fn to the new state x' s'</pre>

Our definition of `return` makes use of the `return` function of the inner monad, and the binding operator uses a `do`-block to perform a computation in the inner monad.

We also want to declare all combined monads that use the `StateT` transformer to be instances of the `MonadState` class, so we will have to give definitions for `get` and `put`:

```
instance (Monad m) => MonadState s (StateT s m) where
  get = StateT $ \s -> return (s,s)
  put s = StateT $ \_ -> return ((),s)
```

Finally, we want to declare all combined monads in which `StateT` is used with an instance of `MonadPlus` to be instances of `MonadPlus`:

```
instance (MonadPlus m) => MonadPlus (StateT s m) where
  mzero = StateT $ \s -> mzero
  (StateT x1) `mplus` (StateT x2) = StateT $ \s -> (x1 s) `mplus` (x2 s)
```

The final step to make our monad transformer fully integrated with Haskell's monad classes is to make `StateT s` an instance of the `MonadTrans` class by providing a `lift` function:

```
instance MonadTrans (StateT s) where
  lift c = StateT $ \s -> c >>= (\x -> return (x,s))
```

The `lift` function creates a `StateT` state transformation function that binds the computation in the inner monad to a function that packages the result with the input state. The result is that a function that returns a list (i.e., a computation in the `List` monad) can be lifted into `StateT s []`, where it becomes a function that returns a `StateT (s -> [(a,s)])`. That is, the lifted computation produces *multiple* (value,state) pairs from its input state. The effect of this is to "fork" the computation in `StateT`, creating a different branch of the computation for each value in the list returned by the lifted function. Of course, applying `StateT` to a different monad will produce different semantics for the `lift` function.

Acknowledgements

This module uses a large amount of text from *All About Monads* with permission from its author Jeff Newbern.

Practical monads

Parsing monads

In the beginner's track of this book, we saw how monads were used for IO. We've also started working more extensively with some of the more rudimentary monads like `Maybe`, `List` or `State`. Now let's try using monads for something quintessentially "practical". Let's try writing a very simple parser. We'll be using the `Parsec` (<http://www.cs.uu.nl/~daan/download/parsec/parsec.html>) library, which comes with GHC but may need to be downloaded separately if you're using another compiler.

Start by adding this line to the import section:

```
-----
import System
import Text.ParserCombinators.Parsec hiding (spaces)
-----
```

This makes the `Parsec` library functions and `getArgs` available to us, except the "spaces" function, whose name conflicts with a function that we'll be defining later.

Now, we'll define a parser that recognizes one of the symbols allowed in Scheme identifiers:

```
-----
symbol :: Parser Char
symbol = oneOf " ! $ % & | * + - / : < = > ? @ ^ _ ~ "
-----
```

This is another example of a monad: in this case, the "extra information" that is being hidden is all the info about position in the input stream, backtracking record, first and follow sets, etc. `Parsec` takes care of all of that for us. We need only use the `Parsec` library function `oneOf` (<http://www.cs.uu.nl/~daan/download/parsec/parsec.html#oneOf>), and it'll recognize a single one of any of the characters in the string passed to it. `Parsec` provides a number of pre-built parsers: for example, `letter` (<http://www.cs.uu.nl/~daan/download/parsec/parsec.html#letter>) and `digit` (<http://www.cs.uu.nl/~daan/download/parsec/parsec.html#digit>) are library functions. And as you're about to see, you can compose primitive parsers into more sophisticated productions.

S Let's define a function to call our parser and handle any possible errors:


```

readExpr :: String -> String
readExpr input = case parse symbol "lisp" input of
  Left err -> "No match: " ++ show err
  Right val -> "Found value"

```

As you can see from the type signature, `readExpr` is a function (`->`) from a `String` to a `String`. We name the parameter `input`, and pass it, along with the symbol action we defined above and the name of the parser ("`lisp`"), to the `Parsec` function `parse` (<http://www.cs.uu.nl/~daan/download/parsec/parsec.html#parse>) .

`Parsec` can return either the parsed value or an error, so we need to handle the error case. Following typical Haskell convention, `Parsec` returns an `Either` ([http://www.haskell.org/onlinereport/standard-prelude.html#\\$tEither](http://www.haskell.org/onlinereport/standard-prelude.html#$tEither)) data type, using the `Left` constructor to indicate an error and the `Right` one for a normal value.

We use a `case...of` construction to match the result of `parse` against these alternatives. If we get a `Left` value (error), then we bind the error itself to `err` and return "No match" with the string representation of the error. If we get a `Right` value, we bind it to `val`, ignore it, and return the string "Found value".

The `case...of` construction is an example of pattern matching, which we will see in much greater detail [[evaluator1.html#primitiveval](#) later on].

Finally, we need to change our main function to call `readExpr` and print out the result:

```

main :: IO ()
main = do args <- getArgs
        putStrLn (readExpr (args !! 0))

```

To compile and run this, you need to specify "`-package parsec`" on the command line, or else there will be link errors. For example:

```

debian:/home/jdtang/haskell_tutorial/code# ghc -package parsec -o simple_parser [../code/listing3.1.hs listing3.1.hs]
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser $
Found value
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser a
No match: "lisp" (line 1, column 1):
unexpected "a"

```

Whitespace

Next, we'll add a series of improvements to our parser that'll let it recognize progressively more complicated expressions. The current parser chokes if there's whitespace preceding our symbol:

```

debian:/home/jdtang/haskell_tutorial/code# ./simple_parser " %"
No match: "lisp" (line 1, column 1):
unexpected " %"

```

Let's fix that, so that we ignore whitespace.

First, let's define a parser that recognizes any number of whitespace characters. Incidentally, this is why we included the "hiding (spaces)" clause when we imported Parsec: there's already a function "spaces" (<http://www.cs.uu.nl/~daan/download/parsec/parsec.html#spaces>) " in that library, but it doesn't quite do what we want it to. (For that matter, there's also a parser called lexeme (<http://www.cs.uu.nl/~daan/download/parsec/parsec.html#lexeme>) that does exactly what we want, but we'll ignore that for pedagogical purposes.)

```
spaces :: Parser ()
spaces = skipMany1 space
```

Just as functions can be passed to functions, so can actions. Here we pass the Parser action space (<http://www.cs.uu.nl/~daan/download/parsec/parsec.html#space>) to the Parser action skipMany1 (<http://www.cs.uu.nl/~daan/download/parsec/parsec.html#skipMany1>), to get a Parser that will recognize one or more spaces.

Now, let's edit our parse function so that it uses this new parser. Changes are in red:

```
readExpr input = case parse (spaces >> symbol) "lisp" input of
  Left err -> "No match: " ++ show err
  Right val -> "Found value"
```

We touched briefly on the >> ("bind") operator in lesson 2, where we mentioned that it was used behind the scenes to combine the lines of a do-block. Here, we use it explicitly to combine our whitespace and symbol parsers. However, bind has completely different semantics in the Parser and IO monads. In the Parser monad, bind means "Attempt to match the first parser, then attempt to match the second with the remaining input, and fail if either fails." In general, bind will have wildly different effects in different monads; it's intended as a general way to structure computations, and so needs to be general enough to accommodate all the different types of computations. Read the documentation for the monad to figure out precisely what it does.

Compile and run this code. Note that since we defined spaces in terms of skipMany1, it will no longer recognize a plain old single character. Instead you *have to* precede a symbol with some whitespace. We'll see how this is useful shortly:

```
debian:/home/jdtang/haskell_tutorial/code# ghc -package parsec -o simple_parser [../code/listing3.2.hs listing3.
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser " %" Found value
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser %
No match: "lisp" (line 1, column 1):
unexpected "%"
expecting space
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser " abc"
No match: "lisp" (line 1, column 4):
unexpected "a"
expecting space
```

Return Values

Right now, the parser doesn't *do* much of anything - it just tells us whether a given string can be recognized or not. Generally, we want something more out of our parsers: we want them to convert the input into a data structure that we can traverse easily. In this section, we learn how to define a data type, and how to modify our parser so that it returns this data type.

First, we need to define a data type that can hold any Lisp value:

```
data LispVal = Atom String
             | List [LispVal]
             | DottedList [LispVal] LispVal
             | Number Integer
             | String String
             | Bool Bool
```

This is an example of an *algebraic data type*: it defines a set of possible values that a variable of type `LispVal` can hold. Each alternative (called a *constructor* and separated by `|`) contains a tag for the constructor along with the type of data that that constructor can hold. In this example, a `LispVal` can be:

1. An `Atom`, which stores a `String` naming the atom
2. A `List`, which stores a list of other `LispVals` (Haskell lists are denoted by brackets)
3. A `DottedList`, representing the Scheme form `(a b . c)`. This stores a list of all elements but the last, and then stores the last element as another field
4. A `Number`, containing a Haskell `Integer`
5. A `String`, containing a Haskell `String`
6. A `Bool`, containing a Haskell boolean value

Constructors and types have different namespaces, so you can have both a constructor named `String` and a type named `String`. Both types and constructor tags always begin with capital letters.

Next, let's add a few more parsing functions to create values of these types. A string is a double quote mark, followed by any number of non-quote characters, followed by a closing quote mark:

```
parseString :: Parser LispVal
parseString = do char '"'
                x <- many (noneOf "\"")
                char '"'
                return $ String x
```

We're back to using the `do`-notation instead of the `>>` operator. This is because we'll be retrieving the value of our parse (returned by `many` (<http://www.cs.uu.nl/~daan/download/parsec/parsec.html#many>) (`noneOf` (<http://www.cs.uu.nl/~daan/download/parsec/parsec.html#noneOf>) `"\""`)) and manipulating it, interleaving some other parse operations in the meantime. In general, use `>>` if the actions don't return a value, `>>=` if you'll be immediately passing that value into the next action, and `do`-notation otherwise.

Once we've finished the parse and have the Haskell `String` returned from `many`, we apply the `String` constructor (from our `LispVal` data type) to turn it into a `LispVal`. Every constructor in an algebraic data type also acts like a function that turns its arguments into a value of its type. It also serves as a pattern that can be used in the left-hand side of a pattern-matching expression; we saw an example of this in [#symbols Lesson 3.1] when we matched our parser result against the two constructors in the `Either` data type.

We then apply the built-in function `return`

([http://www.haskell.org/onlinereport/standard-prelude.html#\\$tMonad](http://www.haskell.org/onlinereport/standard-prelude.html#$tMonad)) to lift our `LispVal` into the `Parser` monad. Remember, each line of a `do`-block must have the same type, but the result of our `String` constructor is just a plain old `LispVal`. `return` lets us wrap that up in a `Parser` action that consumes no input but returns it as the inner value. Thus, the whole `parseString` action will have type `Parser LispVal`.

The `$` operator is infix function application: it's the same as if we'd written `return (String x)`, but `$` is right-associative, letting us eliminate some parentheses. Since `$` is an operator, you can do anything with it that you'd normally do to a function: pass it around, partially apply it, etc. In this respect, it functions like the `Lisp` function `apply` (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.4).

Now let's move on to `Scheme` variables. An `atom`

(http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-5.html#%_sec_2.1) is a letter or symbol, followed by any number of letters, digits, or symbols:

```
parseAtom :: Parser LispVal
parseAtom = do first <- letter <|> symbol
              rest <- many (letter <|> digit <|> symbol)
              let atom = [first] ++ rest
              return $ case atom of
                    "#t" -> Bool True
                    "#f" -> Bool False
                    otherwise -> Atom atom
```

Here, we introduce another `Parsec` combinator, the choice operator `<|>`

(<http://www.cs.uu.nl/~daan/download/parsec/parsec.html#or>). This tries the first parser, then if it fails, tries the second. If either succeeds, then it returns the value returned by that parser. The first parser must fail before it consumes any input: we'll see later how to implement backtracking.

Once we've read the first character and the rest of the atom, we need to put them together. The `"let"` statement defines a new variable `"atom"`. We use the list concatenation operator `++` for this. Recall that `first` is just a single character, so we convert it into a singleton list by putting brackets around it. If we'd wanted to create a list containing many elements, we need only separate them by commas.

Then we use a case statement to determine which `LispVal` to create and return, matching against the literal strings for true and false. The `otherwise` alternative is a readability trick: it binds a variable named `otherwise`, whose value we ignore, and then always returns the value of `atom`.

Finally, we create one more parser, for numbers. This shows one more way of dealing with monadic values:

```
parseNumber :: Parser LispVal
parseNumber = liftM (Number . read) $ many1 digit
```

It's easiest to read this backwards, since both function application (`$`) and function composition (`.`) associate to the right. The `parsec` combinator `many1` (<http://www.cs.uu.nl/~daan/download/parsec/parsec.html#many1>) matches one or more of its argument, so here we're matching one or more digits. We'd like to construct a number `LispVal` from the resulting string, but we have a few type mismatches. First, we use the built-in function `read` ([http://www.haskell.org/onlinereport/standard-prelude.html#\\$vread](http://www.haskell.org/onlinereport/standard-prelude.html#$vread)) to convert that string into a number. Then we pass the result to `Number` to get a `LispVal`. The function composition operator `"."` creates a function that

applies its right argument and then passes the result to the left argument, so we use that to combine the two function applications.

Unfortunately, the result of `many1 digit` is actually a `Parser String`, so our combined `Number . read` still can't operate on it. We need a way to tell it to just operate on the value inside the monad, giving us back a `Parser LispVal`. The standard function `liftM` does exactly that, so we apply `liftM` to our `Number . read` function, and then apply the result of that to our `Parser`.

We also have to import the `Monad` module up at the top of our program to get access to `liftM`:

```
import Monad
```

This style of programming - relying heavily on function composition, function application, and passing functions to functions - is very common in Haskell code. It often lets you express very complicated algorithms in a single line, breaking down intermediate steps into other functions that can be combined in various ways. Unfortunately, it means that you often have to read Haskell code from right-to-left and keep careful track of the types. We'll be seeing many more examples throughout the rest of the tutorial, so hopefully you'll get pretty comfortable with it.

Let's create a parser that accepts either a string, a number, or an atom:

```
parseExpr :: Parser LispVal
parseExpr = parseAtom
  <|> parseString
  <|> parseNumber
```

And edit `readExpr` so it calls our new parser:

```
readExpr :: String -> String
readExpr input = case parse parseExpr "lisp" input of
  Left err -> "No match: " ++ show err
  Right _ -> "Found value"
```

Compile and run this code, and you'll notice that it accepts any number, string, or symbol, but not other strings:

```
debian:/home/jdtang/haskell_tutorial/code# ghc -package parsec -o simple_parser [.../code/listing3.3.hs listing3
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser "\"this is a string\""
Found value
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser 25 Found value
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser symbol
Found value
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser (symbol)
bash: syntax error near unexpected token `symbol'
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser "(symbol)"
No match: "lisp" (line 1, column 1):
unexpected "("
expecting letter, "\"" or digit
```

Exercises

1. Rewrite `parseNumber` using
 1. do-notation
 2. explicit sequencing with the `>>=` (<http://www.haskell.org/onlinereport/standard-prelude.html#tMonad>) operator
2. Our strings aren't quite R5RS compliant (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.3.5), because they don't support escaping of internal quotes within the string. Change `parseString` so that `\` gives a literal quote character instead of terminating the string. You may want to replace `noneOf "\""'` with a new parser action that accepts *either* a non-quote character *or* a backslash followed by a quote mark.
3. Modify the previous exercise to support `\n`, `\r`, `\t`, `\\`, and any other desired escape characters
4. Change `parseNumber` to support the Scheme standard for different bases (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.2.4). You may find the `readOct` and `readHex` (<http://www.haskell.org/onlinereport/numeric.html#sect14>) functions useful.
5. Add a `Character` constructor to `LispVal`, and create a parser for character literals (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.3.4) as described in R5RS.
6. Add a `Float` constructor to `LispVal`, and support R5RS syntax for decimals (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.2.4). The Haskell function `readFloat` (<http://www.haskell.org/onlinereport/numeric.html#sect14>) may be useful.
7. Add data types and parsers to support the full numeric tower (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.2.1) of Scheme numeric types. Haskell has built-in types to represent many of these; check the Prelude ([http://www.haskell.org/onlinereport/standard-prelude.html#\\$tNum](http://www.haskell.org/onlinereport/standard-prelude.html#$tNum)). For the others, you can define compound types that represent eg. a `Rational` as a numerator and denominator, or a `Complex` as a real and imaginary part (each itself a `Real` number).

Recursive Parsers: Adding lists, dotted lists, and quoted datums

Next, we add a few more parser actions to our interpreter. Start with the parenthesized lists that make Lisp famous:

```
-----
parseList :: Parser LispVal
parseList = liftM List $ sepBy parseExpr spaces
-----
```

This works analogously to `parseNumber`, first parsing a series of expressions separated by whitespace (`sepBy parseExpr spaces`) and then apply the `List` constructor to it within the `Parser` monad. Note too that we can pass `parseExpr` to `sepBy` (<http://www.cs.uu.nl/~daan/download/parsec/parsec.html#sepBy>), even though it's an action we wrote ourselves.

The dotted-list parser is somewhat more complex, but still uses only concepts that we're already familiar with:

```

parseDottedList :: Parser LispVal
parseDottedList = do
  head <- endBy parseExpr spaces
  tail <- char '.' >> spaces >> parseExpr
  return $ DottedList head tail

```

Note how we can sequence together a series of Parser actions with `>>` and then use the whole sequence on the right hand side of a do-statement. The expression `char '.' >> spaces` returns a Parser `()`, then combining that with `parseExpr` gives a Parser `LispVal`, exactly the type we need for the do-block.

Next, let's add support for the single-quote syntactic sugar of Scheme:

```

parseQuoted :: Parser LispVal
parseQuoted = do
  char '\''
  x <- parseExpr
  return $ List [Atom "quote", x]

```

Most of this is fairly familiar stuff: it reads a single quote character, reads an expression and binds it to `x`, and then returns `(quote x)`, to use Scheme notation. The `Atom` constructor works like an ordinary function: you pass it the `String` you're encapsulating, and it gives you back a `LispVal`. You can do anything with this `LispVal` that you normally could, like put it in a list.

Finally, edit our definition of `parseExpr` to include our new parsers:

```

parseExpr :: Parser LispVal
parseExpr = parseAtom
  <|> parseString
  <|> parseNumber
  <|> parseQuoted
  <|> do char '('
        x <- (try parseList) <|> parseDottedList
        char ')'
        return x

```

This illustrates one last feature of `Parsec`: backtracking. `parseList` and `parseDottedList` recognize identical strings up to the dot; this breaks the requirement that a choice alternative may not consume any input before failing. The `try` (<http://www.cs.uu.nl/~daan/download/parsec/parsec.html#try>) combinator attempts to run the specified parser, but if it fails, it backs up to the previous state. This lets you use it in a choice alternative without interfering with the other alternative.

Compile and run this code:

```

debian:/home/jdtang/haskell_tutorial/code# ghc -package parsec -o simple_parser [./code/listing3.4.hs listing3.
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser "(a test)"
Found value
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser "(a (nested) test)" Found value
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser "(a (dotted . list) test)"
Found value
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser "(a '(quoted (dotted . list)) test)"
Found value
debian:/home/jdtang/haskell_tutorial/code# ./simple_parser "(a '(imbalanced parens))"
No match: "lisp" (line 1, column 24):
unexpected end of input
expecting space or ")"

```

Note that by referring to `parseExpr` within our parsers, we can nest them arbitrarily deep. Thus, we get a full Lisp reader with only a few definitions. That's the power of recursion.

Exercises

1. Add support for the backquote
(http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-7.html#%_sec_4.2.6)
syntactic sugar: the Scheme standard details what it should expand into (`quasiquote/unquote`).
2. Add support for vectors
(http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.3.6)
. The Haskell representation is up to you: GHC does have an `Array` (<http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Array.html>) data type, but it can be difficult to use. Strictly speaking, a vector should have constant-time indexing and updating, but destructive update in a purely functional language is difficult. You may have a better idea how to do this after the section on `set!`, later in this tutorial.
3. Instead of using the `try` combinator, left-factor the grammar so that the common subsequence is its own parser. You should end up with a parser that matches a string of expressions, and one that matches either nothing or a dot and a single expressions. Combining the return values of these into either a `List` or a `DottedList` is left as a (somewhat tricky) exercise for the reader: you may want to break it out into another helper function

Generic monads

Write me: The idea is that this section can show some of the benefits of not tying yourself to one single monad, but writing your code for any arbitrary monad m . Maybe run with the idea of having some elementary monad, and then deciding it's not good enough, so replacing it with a fancier one... and then deciding you need to go even further and just plug in a monad transformer

For instance: Using the Identity Monad:

```
-----
module Identity(Id(Id)) where
newtype Id a = Id a
instance Monad Id where
    (>>=) (Id x) f = f x
    return = Id
instance (Show a) => Show (Id a) where
    show (Id x) = show x
-----
```

In another File

```
-----
import Identity
type M = Id
my_fib :: Integer -> M Integer
my_fib = my_fib_acc 0 1
my_fib_acc :: Integer -> Integer -> Integer -> M Integer
my_fib_acc _ fn1 1 = return fn1
my_fib_acc fn2 _ 0 = return fn2
my_fib_acc fn2 fn1 n_rem = do
    val <- my_fib_acc fn1 (fn2+fn1) (n_rem - 1)
    return val
-----
```


Doesn't seem to accomplish much, but It allows to you add debugging facilities to a part of your program on the fly. As long as you've used return instead of explicit Id constructors, then you can drop in the following monad:

```

module PMD (Pmd(Pmd)) where --PMD = Poor Man's Debugging, Now available for haskell

import IO

newtype Pmd a = Pmd (a, IO ())

instance Monad Pmd where
  (>>=) (Pmd (x, prt)) f = let (Pmd (v, prt')) = f x
                           in Pmd (v, prt >> prt')
  return x = Pmd (x, return ())

instance (Show a) => Show (Pmd a) where
  show (Pmd (x, _) ) = show x

```

If we wanted to debug our fibonacci program above, We could modify it as follows:

```

import Identity
import PMD
import IO
type M = Pmd
...
my_fib_acc :: Integer -> Integer -> Integer -> M Integer
my_fib_acc _ fn1 1 = return fn1
my_fib_acc fn2 _ 0 = return fn2
my_fib_acc fn2 fn1 n_rem =
  val <- my_fib_acc fn1 (fn2+fn1) (n_rem - 1)
  Pmd (val, putStrLn (show fn1))

```

All we had to change is the lines where we wanted to print something for debugging, and add some code wherever you extracted the value from the Id Monad to execute the resulting IO () you've returned. Something like

```

main :: IO ()
main = do
  let (Id f25) = my_fib 25
      putStrLn ("f25 is: " ++ show f25)

```

for the Id Monad vs.

```

main :: IO ()
main = do
  let (Pmd (f25, prt)) = my_fib 25
      prt
      putStrLn ("f25 is: " ++ show f25)

```

For the Pmd Monad. Notice that we didn't have to touch any of the functions that we weren't debugging.

Advanced Haskell

```

plusOne = proc a -> returnA -< (a+1)
plusTwo = proc a -> plusOne -< (a+1)

```

One simple approach is to feed `(a+1)` as input into the `plusOne` arrow. Note the similarity between `plusOne` and `plusTwo`. You should notice that there is a basic pattern here which goes a little something like this: `proc FOO -> SOME_ARROW -< (SOMETHING_WITH_FOO)`

Exercises

1. `plusOne` is an arrow, so by the pattern above `returnA` must be an arrow too. What do you think `returnA` does?

do notation

Our current implementation of `plusTwo` is rather disappointing actually... shouldn't it just be `plusOne` twice? We can do better, but to do so, we need to introduce the `do` notation:

```

plusTwoBis =
proc a -> do b <- plusOne -< a
           plusOne -< b

```

Now try this out in GHCi:

```

Prelude> :r
Compiling Main          ( toyArrows.hs, interpreted )
Ok, modules loaded: Main.
*Main> plusTwoBis 5
7

```

You can use this `do` notation to build up sequences as long as you would like:

```

plusFive =
proc a -> do b <- plusOne -< a
           c <- plusOne -< b
           d <- plusOne -< c
           e <- plusOne -< d
           plusOne -< e

```

Monads and arrows

FIXME: I'm no longer sure, but I believe the intention here was to show what the difference is having this `proc` notation instead to just a regular chain of `dos`

Understanding arrows

We have permission to import material from the Haskell arrows page (<http://www.haskell.org/arrows>) . See the talk page for details.

The factory and conveyor belt metaphor

In this tutorial, we shall present arrows from the perspective of stream processors, using the factory metaphor from the monads module as a support. Let's get our hands dirty right away.

You are a factory owner, and as before you own a set of **processing machines**. Processing machines are just a metaphor for functions; they accept some input and produce some output. Your goal is to combine these processing machines so that they can perform richer, and more complicated tasks. Monads allow you to combine these machines in a pipeline. Arrows allow you to combine them in more interesting ways. The result of this is that you can perform certain tasks in a less complicated and more efficient manner.

In a monadic factory, we took the approach of wrapping the outputs of our machines in containers. The arrow factory takes a completely different route: rather than wrapping the outputs in containers, we wrap *the machines themselves*. More specifically, in an arrow factory, we attach a pair of conveyor belts to each machine, one for the input and one for the output.

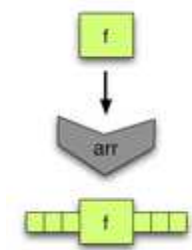
So given a function of type $b \rightarrow c$, we can construct an equivalent a arrow by attaching a b and c conveyor belt to the machine. The equivalent arrow is of type $a \rightarrow b \rightarrow c$, which we can pronounce as an arrow a from b to c .

Plethora of robots

We mentioned earlier that arrows give you more ways to combine machines together than monads did. Indeed, the arrow type class provides six distinct **robots** (compared to the two you get with monads).

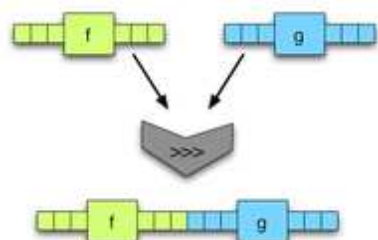
arr

The simplest robot is `arr` with the type signature `arr :: (b -> c) -> a -> b -> c`. In other words, the `arr` robot takes a processing machine of type $b \rightarrow c$, and adds conveyor belts to form an a arrow from b to c .

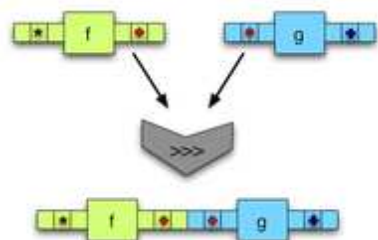


(>>>)

The next, and probably the most important, robot is `(>>>)`. This is basically the arrow equivalent to the monadic bind robot `(>>=)`. The arrow version of bind `(>>>)` puts two arrows into a sequence. That is, it connects the output conveyor belt of the first arrow to the input conveyor belt of the second one.



What we get out of this is a new arrow. One consideration to make, though is what input and output types our arrows may take. Since we're connecting output and the input conveyor belts of the first and second arrows, the second arrow must accept the same kind of input as what the first arrow outputs. If the first arrow is of type $a \rightarrow b \rightarrow c$, the second arrow must be of type $a \rightarrow c \rightarrow d$. Here is the same diagram as above, but with things on the conveyor belts to help you see the issue with types.



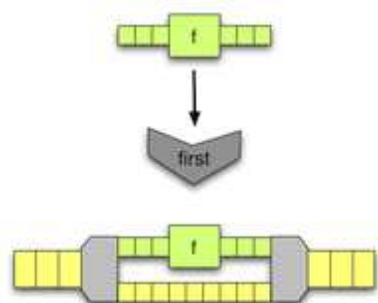
Exercises

What is the type of the combined arrow?

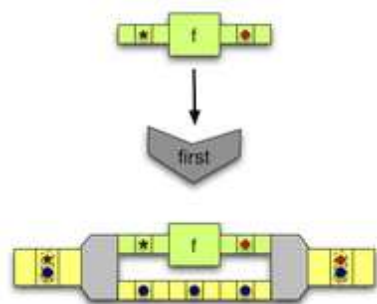
first

Up to now, our arrows can only do the same things that monads can. Here is where things get interesting! The arrows type class provides functions which allow arrows to work with *pairs* of input. As we will see later on, this leads us to be able to express parallel computation in a very succinct manner. The first of these functions, naturally enough, is `first`.

If you are skimming this tutorial, it is probably a good idea to slow down at least in this section, because the `first` robot is one of the things that makes arrows truly useful.



Given an arrow f , the `first` robot attaches some conveyor belts and extra machinery to form a new, more complicated arrow. The machines that bookend the input arrow split the input pairs into their component parts, and put them back together. The idea behind this is that the first part of every pair is fed into the f , whilst the second part is passed through on an empty conveyor belt. When everything is put back together, we have same pairs that we fed in, except that the first part of every pair has been replaced by an equivalent output from f .



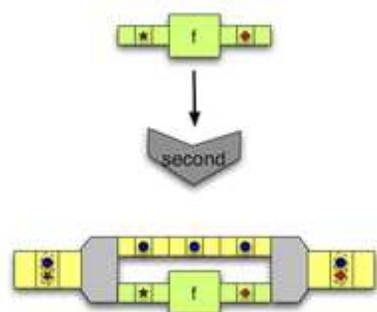
Now the question we need to ask ourselves is that of types. Say that the input tuples are of type (b, d) and the input arrow is of type $a \rightarrow b \rightarrow c$ (that is, it is an arrow from b to c). What is the type of the output? Well, the arrow converts all b s into c s, so when everything is put back together, the type of the output must be (c, d) .

Exercises

What is the type of the `first` robot?

second

If you understand the `first` robot, the `second` robot is a piece of cake. It does the same exact thing, except that it feeds the second part of every input pair into the given arrow `f` instead of the first part.

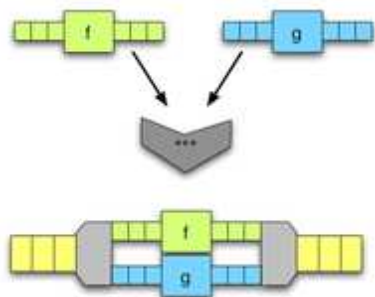


What makes the `second` robot interesting is that it can be derived from the previous robots! Strictly speaking, the only robots you need to for arrows are `arr`, `(>>>)` and `first`. The rest can be had "for free".

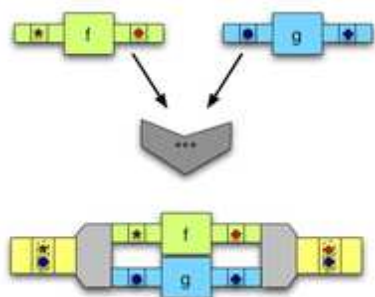
Exercises

1. Write a function to swap two components of a tuple.
2. Combine this helper function with the robots `arr`, `(>>>)` and `first` to implement the `second` robot

One of the selling points of arrows is that you can use them to express parallel computation. The `(***)` robot is just the right tool for the job. Given two arrows, `f` and `g`, the `(***)` combines them into a new arrow using the same bookend-machines we saw in the previous two robots



Conceptually, this isn't very much different from the robots `first` and `second`. As before, our new arrow accepts *pairs* of inputs. It splits them up, sends them on to separate conveyor belts, and puts them back together. The only difference here is that, rather than having one arrow and one empty conveyor belt, we have two distinct arrows. But why not?

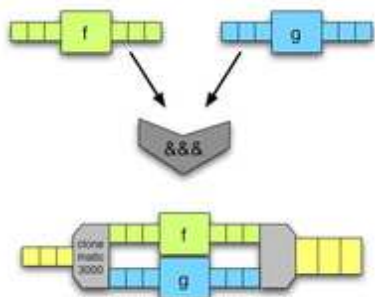


Exercises

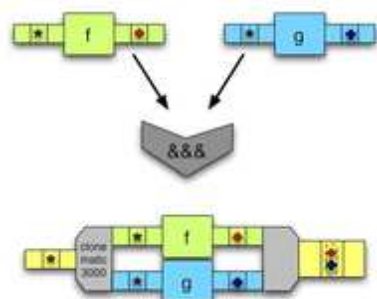
1. What is the type of the `(***)` robot?
2. Given the `(>>>)`, `first` and `second` robots, implement the `(***)` robot.

&&&

The final robot in the Arrow class is very similar to the `(***)` robot, except that the resulting arrow accepts a single input and not a pair. Yet, the rest of the machine is exactly the same. How can we work with two arrows, when we only have one input to give them?



The answer is simple: we clone the input and feed a copy into each machine!



Exercises

1. Write a simple function to clone an input into a pair.
2. Using your cloning function, as well as the robots `arr`, `(>>>)` and `***`, implement the `&&&` robot
3. Similarly, rewrite the following function without using `&&&`:

```
addA f g = f &&& g >>> arr (\ (y, z) -> y + z)
```

Functions are arrows

Now that we have presented the 6 arrow robots, we would like to make sure that you have a more solid grasp of them by walking through a simple implementations of the Arrow class. As in the monadic world, there are many different types of arrows. What is the simplest one you can think of? Functions.

Put concretely, the type constructor for functions `(->)` is an instance of `Arrow`

```
instance Arrow (->) where
  arr f = f
  f >>> g = g . f
  first f = \(x,y) -> (f x, y)
```

Now let's examine this in detail:

- `arr` - Converting a function into an arrow is trivial. In fact, the function already is an arrow.
- `(>>>)` - we want to feed the output of the first function into the input of the second function. This is nothing more than function composition.
- `first` - this is a little more elaborate. Given a function `f`, we return a function which accepts a pair of inputs `(x,y)`, and runs `f` on `x`, leaving `y` untouched.

And that, strictly speaking, is all we need to have a complete arrow, but the arrow typeclass also allows you to make up your own definition of the other three robots, so let's have a go at that:

```
first f = \(x,y) -> (f x, y) -- for comparison's sake
second f = \(x,y) -> ( x, f y) -- like first
f *** g = \(x,y) -> (f x, g y) -- takes two arrows, and not just one
f &&& g = \(x,y) -> (f x, g x) -- feed the same input into both functions
```

And that's it! Nothing could be simpler.

Note that this is not the official instance of functions as arrows. You should take a look at the haskell library (<http://darcs.haskell.org/packages/base/Control/Arrow.hs>) if you want the real deal.

The arrow notation

In the introductory Arrows chapter, we introduced the `proc` and `-<` notation. How does this tie in with all the arrow robots we just presented? Sadly, it's a little bit less straightforward than `do`-notation, but let's have a look.

Maybe functor

It turns out that any monad can be made into arrow. We'll go into that later on, but for now, *FIXME: transition*

Using arrows

At this point in the tutorial, you should have a strong enough grasp of the arrow machinery that we can start to meaningfully tackle the question of what arrows are good for.

Stream processing

Avoiding leaks

Arrows were originally motivated by an efficient parser design found by Swierstra & Duponcheel.

To describe the benefits of their design, let's examine exactly how monadic parsers work.

If you want to parse a single word, you end up with several monadic parsers stacked end to end. Taking `Parsec` as an example, the parser string "word" can also be viewed as

```
word = do char 'w' >> char 'o' >> char 'r' >> char 'd'
       return "word"
```

Each character is tried in order, if "worg" is the input, then the first three parsers will succeed, and the last one will fail, making the entire string "word" parser fail.

If you want to parse one of two options, you create a new parser for each and they are tried in order. The first one must fail and then the next will be tried with the same input.

```
ab = do char 'a' <|> char 'b' <|> char 'c'
```

To parse "c" successfully, both 'a' and 'b' must have been tried.

```

one = do char 'o' >> char 'n' >> char 'e'
      return "one"

two = do char 't' >> char 'w' >> char 'o'
      return "two"

three = do char 't' >> char 'h' >> char 'r' >> char 'e' >> char 'e'
        return "three"

nums = do one <|> two <|> three

```

With these three parsers, you can't know that the string "four" will fail the parser `nums` until the last parser has failed.

If one of the options can consume much of the input but will fail, you still must descend down the chain of parsers until the final parser fails. All of the input that can possibly be consumed by later parsers must be retained in memory in case one of them does consume it. That can lead to much more space usage than you would naively expect, this is often called a space leak.

The general pattern of monadic parsers is that each option must fail or one option must succeed.

So what's better?

Swierstra & Duponcheel (1996) noticed that a smarter parser could immediately fail upon seeing the very first character. For example, in the `nums` parser above, the choice of first letter parsers was limited to either the letter 'o' for "one" or the letter 't' for both "two" and "three". This smarter parser would also be able to garbage collect input sooner because it could look ahead to see if any other parsers might be able to consume the input, and drop input that could not be consumed. This new parser is a lot like the monadic parsers with the major difference that it exports static information. It's like a monad, but it also tells you what it can parse.

There's one major problem. This doesn't fit into the monadic interface. Monads are $(a \rightarrow m\ b)$, they're based around functions only. There's no way to attach static information. You have only one choice, throw in some input, and see if it passes or fails.

The monadic interface has been touted as a general purpose tool in the functional programming community, so finding that there was some particularly useful code that just couldn't fit into that interface was something of a setback. This is where Arrows come in. John Hughes's *Generalising monads to arrows* proposed the arrows abstraction as new, more flexible tool.

Static and dynamic parsers

Let us examine Swierstra & Duponcheel's parser in greater detail, from the perspective of arrows. The parser has two components: a fast, static parser which tells us if the input is worth trying to parse; and a slow, dynamic parser which does the actual parsing work.

```

data Parser s a b = P (StaticParser s) (DynamicParser s a b)
data StaticParser s = SP Bool [s]
newtype DynamicParser s a b = DP ((a,[s]) -> (b,[s]))

```

The static parser consists of a flag, which tells us if the parser can accept the empty input, and a list of possible **starting characters**. For example, the static parser for a single character would be as follows:

```

spCharA :: Char -> StaticParser Char
spCharA c = SP False [c]

```

It does not accept the empty string (`False`) and the list of possible starting characters consists only of `c`.

The dynamic parser needs a little more dissecting : what we see is a function that goes from $(a, [s])$ to $(b, [s])$. It is useful to think in terms of sequencing two parsers : Each parser consumes the result of the previous parser (`a`), along with the remaining bits of input stream (`[s]`), it does something with `a` to produce its own result `b`, consumes a bit of string and returns *that*. Ooof. So, as an example of this in action, consider a dynamic parser $(Int, String) \rightarrow (Int, String)$, where the `Int` represents a count of the characters parsed so far. The table belows shows what would happen if we sequence a few of them together and set them loose on the string "cake" :

	result	remaining
before	0	cake
after first parser	1	ake
after second parser	2	ke
after third parser	3	e

So the point here is that a dynamic parser has two jobs : it does something to the output of the previous parser (informally, $a \rightarrow b$), and it consumes a bit of the input string, (informally, $[s] \rightarrow [s]$), hence the type `DP ((a, [s]) -> (b, [s]))`. Now, in the case of a dynamic parser for a single character, the first job is trivial. We ignore the output of the previous parser. We return the character we have parsed. And we consume one character off the stream :

```

dpCharA :: Char -> DynamicParser Char Char Char
dpCharA c = DP \(_, x:xs) -> (c, xs)

```

This might lead you to ask a few questions. For instance, what's the point of accepting the output of the previous parser if we're just going to ignore it? The best answer we can give right now is "wait and see". If you're comfortable with monads, consider the bind operator $(>>=)$. While bind is immensely useful by itself, sometimes, when sequencing two monadic computations together, we like to ignore the output of the first computation by using the anonymous bind $(>>)$. This is the same situation here. We've got an interesting little bit of power on our hands, but we're not going to use it quite yet.

The next question, then, shouldn't the dynamic parser be making sure that the current character off the stream matches the character to be parsed? Shouldn't `x == c` be checked for? No. And in fact, this is part of the point; the work is not necessary because the check would already have been performed by the static parser.

Anyway, let us put this together. Here is our S+D style parser for a single character:

```

charA :: Char -> Parser Char Char Char
charA c = P (SP False [c]) (DP \(_, x:xs) -> (c, xs))

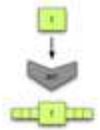
```

Arrow combinators (robots)

Up to this point, we have explored two somewhat independent trains of thought. On the one hand, we've taken a look at some arrow machinery, the combinators/robots from above, although we don't exactly know what it's for. On the other hand, we have introduced a type of parser using the `Arrow` class. We know that the goal is to avoid space leaks and that it somehow involves separating a fast static parser from its slow dynamic part, but we don't really understand how that ties in to all this arrow machinery. In this section, we will attempt to address both of these gaps in our knowledge and merge our twin trains of thought into one. We're going to implement the `Arrow` class for `Parser s`, and by doing so, give you a glimpse of what makes arrows useful. So let's get started:

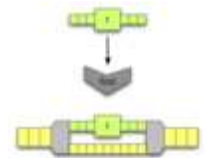
```
instance Arrow (Parser s) where
```

One of the simplest things we can do is to convert an arbitrary function into a parsing arrow. We're going to use "parse" in the loose sense of the term: our resulting arrow accepts the empty string, and *only the empty string* (its set of first characters is `[]`). Its sole job is take the output of the previous parsing arrow and do something with it. Otherwise, it does not consume any input.



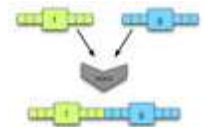
```
arr f = P (SP True []) (DP \(b,s) -> (f b,s))
```

Likewise, the `first` combinator is relatively straightforward. Recall the conveyor belts from above. Given a parser, we want to produce a new parser that accepts a pair of inputs `(b,d)`. The first part of the input `b`, is what we actually want to parse. The second part is passed through completely untouched:



```
first (P sp (DP p)) = (P sp \(b,d),s) -> let (c, s') = p (b,s) in ((c,d),s')
```

On the other hand, the implementation of `(>>>)` requires a little more thought. We want to take two parsers, and returns a combined parser incorporating the static and dynamic parsers of both arguments:



```
(P (SP empty1 start1) (DP p1)) >>>
(P (SP empty2 start2) (DP p2)) =
  P (SP (empty1 && empty2))
    (if not empty1 then start1 else start1 `union` start2)
    (DP (p2.p1))
```

Combining the dynamic parsers is easy enough; we just do function composition. Putting the static parsers together requires a little bit of thought. First of all, the combined parser can only accept the empty string if *both* parsers do. Fair enough, now how about the starting symbols? Well, the parsers are supposed to be in a sequence, so the starting symbols of the second parser shouldn't really matter. If life were simple, the starting symbols of the combined parser would only be `start1`. Alas, life is NOT simple, because parsers could very well accept the empty input. If the first parser accepts the empty input, then we have to account for this possibility by accepting the starting symbols from both the first and the second parsers.

Exercises

1. Consider the `charA` parser from above. What would `charA 'o' >>> charA 'n' >>> charA 'e'` result in?
2. Write a simplified version of that combined parser. That is: does it accept the empty string? What are its starting symbols? What is the

dynamic parser for this?

So what do arrows buy us in all this?

Monads can be arrows too

The real flexibility with arrows comes with the ones that aren't monads, otherwise it's just a clunkier syntax -- Philippa Cowderoy

It turns out that all monads can be made into arrows. Here's a central quote from the original arrows papers:

Just as we think of a monadic type $m\ a$ as representing a 'computation delivering an a '; so we think of an arrow type $a\ b\ c$, (that is, the application of the parameterised type a to the two parameters b and c) as representing 'a computation with input of type b delivering a c '; arrows make the dependence on input explicit.

One way to look at arrows is the way the English language allows you to noun a verb, for example, "I had a chat." Arrows are much like that, they turn a function from a to b into a value. This value is a first class transformation from a to b .

Arrows in practice

Arrows are a relatively new abstraction, but they already found a number of uses in the Haskell world

- Hughes' arrow-style parsers were first described in his 2000 paper, but a usable implementation wasn't available until May 2005. Einar Karttunen wrote an implementation called PArrows that approaches the features of standard Haskell parser combinator library, Parsec.
- The Fudgets library for building graphical interfaces *FIXME: complete this paragraph*
- Yampa - *FIXME: talk briefly about Yampa*
- The Haskell XML Toolbox (HXT (<http://www.fh-wedel.de/~si/HXmlToolbox/index.html>)) uses arrows for processing XML. There is a Wiki page in the Haskell Wiki with a somewhat Gentle Introduction to HXT (<http://www.haskell.org/haskellwiki/HXT>) .

Arrows Aren't The Answer To Every Question

Arrows do have some problems. Several people on the #haskell irc channel have done nifty arrows experiments, and some of those experiments ran into problems. Some notable obstacles were typified by experiments done by Jeremy Shaw, Einar Karttunen, and Peter Simons. If you would like to learn more about the limitations behind arrows, follow the references at the end of this article

See also

- Generalising Monads to Arrows - John Hughes

- <http://www.haskell.org/arrows/biblio.html>

Arrow uses

Arrow limitations

- Jeremy Shaw -
- Einar Kartunnen -
- Peter Simons -

Current research

Acknowledgements

This module uses text from *An Introduction to Arrows* by Shae Erisson, originally written for The Monad.Reader 4

Continuation passing style

Continuation passing style, or CPS, is a style of programming where functions never return values, but instead take an extra parameter which they give their result to — this extra parameter represents *what to do next*, and is called a continuation.

Starting simple

To begin with, we're going to explore two simple examples which illustrate what CPS and continuations are.

square

Let's start with a very simple module which squares a number, then outputs it:

Example: A simple module, no continuations

```
square :: Int -> Int
square x = x ^ 2

main = do
  let x = square 4
      print x
```



We're clearly doing two things here. First, we square four, then we print the result. If we were to make the `square` function take a continuation, we'd end up with something like the following:

Example: A simple module, using continuations

```
square :: Int -> (Int -> a) -> a
square x k = k (x ^ 2)

main = square 4 print
```

That is, `square` takes an extra parameter which is the function that represents *what to do next* — the continuation of the computation.

quadRoots

Consider the quadratic equation. Recall that for the equation $ax^2 + bx + c = 0$, the quadratic equation states that:

$$x = \frac{-b}{2a} \pm \frac{\sqrt{b^2 - 4ac}}{2a}$$

When considering only real numbers, we may have zero, one, or two roots. The quantity under the radical, $d \equiv b^2 - 4ac$ is known as the determinant. When $d < 0$, there are no (real) roots; when $d = 0$ we have one real root; and finally, with $d > 0$, we have two real roots. We then write a Haskell function to compute the roots of a quadratic as follows:

Example: `quadRoots`, no continuations

```
data Roots = None | One Double | Two Double Double
quadRoots :: Double -> Double -> Double -> Roots
quadRoots a b c
  | d < 0 = None
  | d == 0 = One $ -b/2/a
  | d > 0 = Two ((-b + sqrt d)/2/a) ((-b - sqrt d)/2/a)
  where d = b*b - 4*a*c
```

To use this function, we need to pattern match on the result, for instance:

Example: Using the result of `quadRoots`, still no continuations

```
printRoots :: Double -> Double -> Double -> IO ()
printRoots a b c = case quadRoots a b c of
  None -> putStrLn "There were no roots."
  One x -> putStrLn $ showString "There was one root: " $ show x
  Two x x' -> putStrLn $ showString "There were two roots found: " $
    shows x $ showString " and " $ show x'
```

To write this in continuation passing style, we will begin by modifying the `quadRoots` function. It will now take three additional parameters: functions that will be called with the resulting number of roots.

Example: quadRoots' using continuations

```

quadRoots' :: Double -> Double -> Double -- The three coefficients
           -> a                          -- What to do with no roots
           -> (Double -> a)              -- What to do with one root
           -> (Double -> Double -> a)    -- What to do with two roots
           -> a                          -- The final result
quadRoots' a b c f0 f1 f2
  | d < 0 = f0
  | d == 0 = f1 $ -b/2/a
  | d > 0 = f2 ((-b + sqrt d)/2/a) ((-b - sqrt d)/2/a)
  where d = b*b - 4*a*c

```

One may notice that the body of `quadRoots'` is identical to `quadRoots`, except that we've substituted arbitrary functions for the constructors of `Roots`. Indeed, `quadRoots` may be rewritten to use `quadRoots'`, by passing the constructors for `Roots`. Now we no longer need to pattern match on the result, we just pass in the expressions from the case matches.

This is how data is often expressed in lambda calculi: note that `quadRoots'` doesn't use `Roots` at all.

Example: Using the result of `quadRoots`, with continuations

```

printRoots :: Double -> Double -> Double -> IO ()
printRoots a b c = quadRoots' a b c f0 f1 f2
  where
    f0      = putStrLn "There were no roots."
    f1 x    = putStrLn $ "There was one root: " ++ show x
    f2 x x' = putStrLn $ "There were two roots found: "
                      ++ show x ++ " and " ++ show x'

```

**Exercises**

FIXME: write some exercises

Using the `Cont` monad

By now, you should be used to the pattern that whenever we find a pattern we like (here the pattern is using continuations), but it makes our code a little ugly, we use a monad to encapsulate the 'plumbing'. Indeed, there is a monad for modelling computations which use CPS.

Example: The `Cont` monad

```

newtype Cont r a = Cont { runCont :: (a -> r) -> r }

```

Removing the newtype and record cruft, we obtain that `Cont r a` expands to `(a -> r) -> r`. So how does this fit with our idea of continuations we presented above? Well, remember that a function in CPS basically took an extra parameter which represented 'what to do next'. So, here, the type of `Cont r a` expands to be an extra

function (the continuation), which is a function from things of type a (what the result of the function would have been, if we were returning it normally instead of throwing it into the continuation), to things of type r , which becomes the final result type of our function.

All of that was a little vague and abstract so let's crack out an example.

Example: The `square` module, using the `Cont` monad

```
square :: Int -> Cont r Int
square x = return (x ^ 2)

main = runCont (square 4) print
{- Result: 16 -}
```



If we expand the type of `square`, we obtain that `square :: Int -> (Int -> r) -> r`, which is precisely what we had before we introduced `Cont` into the picture. So we can see that type `Cont r a` expands into a type which fits our notion of a continuation, as defined above. Every function that returns a `Cont`-value actually takes an extra parameter, which is the continuation. Using `return` simply throws its argument into the continuation.

How does the `Cont` implementation of `(>>=)` work, then? It's easiest to see it at work:

Example: The `(>>=)` function for the `Cont` monad

```
square :: Int -> Cont r Int
square x = return (x ^ 2)

addThree :: Int -> Cont r Int
addThree x = return (x + 3)

main = runCont (square 4 >>= addThree) print
{- Result: 19 -}
```



The Monad instance for `(Cont r)` is given below:

```
instance Monad (Cont r) where
  return n = Cont (\k -> k n)
  m >>= f = Cont (\k -> runCont m (\a -> runCont (f a) k))
```

So `return n` is `Cont`-value that throws `n` straight away into whatever continuation it is applied to. `m >>= f` is a `Cont`-value that runs `m` with the continuation `\a -> f a k`, which receives the result of `m`, then applies it to `f` to get another `Cont`-value. This is then called with the continuation we got at the top level; in essence `m >>= f` is a `Cont`-value that takes the result from `m`, applies it to `f`, then throws that into the continuation.

Exercises

To come.

callCC

By now you should be fairly confident using the basic notions of continuations and `Cont`, so we're going to skip ahead to the next big concept in continuation-land. This is a function called `callCC`, which is short for 'call with current continuation'. We'll start with an easy example.

Example: square using `callCC`

```

-- Without callCC
square :: Int -> Cont r Int
square n = return (n ^ 2)

-- With callCC
square :: Int -> Cont r Int
square n = callCC $ \k -> k (n ^ 2)

```



We pass a *function* to `callCC` that accepts one parameter that is in turn a function. This function (`k` in our example) is our tangible continuation: we can see here we're throwing a value (in this case, n^2) into our continuation. We can see that the `callCC` version is equivalent to the `return` version stated above because we stated that `return n` is just a `Cont`-value that throws `n` into whatever continuation that it is given. Here, we use `callCC` to bring the continuation 'into scope', and immediately throw a value into it, just like using `return`.

However, these versions look remarkably similar, so why should we bother using `callCC` at all? The power lies in that we now have precise control of exactly when we call our continuation, and with what values. Let's explore some of the surprising power that gives us.

Deciding when to use `k`

We mentioned above that the point of using `callCC` in the first place was that it gave us extra power over what we threw into our continuation, and when. The following example shows how we might want to use this extra flexibility.

Example: Our first proper `callCC` function

```

foo :: Int -> Cont r String
foo n =
  callCC $ \k -> do
    let n' = n ^ 2 + 3
        when (n' > 20) $ k "over twenty"
    return (show $ n' - 4)

```



`foo` is a slightly pathological function that computes the square of its input and adds three; if the result of this computation is greater than 20, then we return from the function immediately, throwing the `String` value "over twenty" into the continuation that is passed to `foo`. If not, then we subtract four from our previous computation, show it, and throw it into the computation. If you're used to imperative languages, you can think of `k` like the 'return' statement that immediately exits the function. Of course, the advantages of an expressive language like Haskell are that `k` is just an ordinary first-class function, so you can pass it to other functions like `when`, or store it in a `Reader`, etc.

Naturally, you can embed calls to `callCC` within `do`-blocks:

Example: More developed `callCC` example involving a `do`-block

```

bar :: Char -> String -> Cont r Int
bar c s = do
  msg <- callCC $ \k -> do
    let s' = c : s
        when (s' == "hello") $ k "They say hello."
        let s'' = show s'
            return ("They appear to be saying " ++ s'')
    return (length msg)

```

When you call `k` with a value, the entire `callCC` call takes that value. In other words, `k` is a bit like a 'goto' statement in other languages: when we call `k` in our example, it pops the execution out to where you first called `callCC`, the `msg <- callCC $...` line. No more of the argument to `callCC` (the inner `do`-block) is executed. Hence the following example contains a useless line:

Example: Popping out a function, introducing a useless line

```

bar :: Cont r Int
bar = callCC $ \k -> do
  let n = 5
      k n
  return 25

```

`bar` will always return 5, and never 25, because we pop out of `bar` before getting to the `return 25` line.

A note on typing

Why do we exit using `return` rather than `k` the second time within the `foo` example? It's to do with types. Firstly, we need to think about the type of `k`. We mentioned that we can throw something into `k`, and nothing after that call will get run (unless `k` is run conditionally, like when wrapped in a `when`). So the return type of `k` doesn't matter; we can never do anything with the result of running `k`. We say, therefore, that the type of `k` is:

```

k :: a -> Cont r b

```

We universally quantify the return type of `k`. This is possible for the aforementioned reasons, and the reason it's advantageous is that we can do whatever we want with the result of `k`. In our above code, we use it as part of a `when` construct:

```

when :: Monad m => Bool -> m () -> m ()

```

As soon as the compiler sees `k` being used in this `when`, it infers that we want a `()` result type for `k`^[17]. So the final expression in that inner `do`-block has type `Cont r ()` too. This is the crux of our problem. There are two possible execution routes: either the condition for the `when` succeeds, in which case the `do`-block returns something of type `Cont r String`. (The call to `k` makes the entire `do`-block have a type of `Cont r t`, where `t` is the type of the argument given to `k`. Note that this is different from the return type of `k` itself, which is just the

return type of the expression involving the call to `k`, not the entire do-block.) If the condition fails, execution passes on and the do-block returns something of type `Cont r ()`. This is a type mismatch.

If you didn't follow any of that, just make sure you use `return` at the end of a do-block inside a call to `callCC`, not `k`.

The type of `callCC`

We've deliberately broken a trend here: normally when we've introduced a function, we've given its type straight away, but in this case we haven't. The reason is simple: the type is rather horrendously complex, and it doesn't immediately give insight into what the function does, or how it works. Nevertheless, you should be familiar with it, so now you've hopefully understood the function itself, here's its type:

```
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
```

This seems like a really weird type to begin with, so let's use a contrived example.

```
callCC $ \k -> k 5
```

You pass a *function* to `callCC`. This in turn takes a parameter, `k`, which is another function. `k`, as we remarked above, has the type:

```
k :: a -> Cont r b
```

The entire argument to `callCC`, then, is a function that takes something of the above type and returns `Cont r t`, where `t` is whatever the type of the argument to `k` was. So, `callCC`'s argument has type:

```
(a -> Cont r b) -> Cont r a
```

Finally, `callCC` is therefore a function which takes that argument and returns its result. So the type of `callCC` is:

```
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
```

The implementation of `callCC`


So far we have looked at the use of `callCC` and its type. This just leaves its implementation, which is:

```
callCC f = Cont $ \k -> runCont (f (\a -> Cont $ \_ -> k a)) k
```

This code is far from obvious. However, the amazing fact is that the implementations for `callCC f`, `return n` and `m >=> f` can all be produced automatically from their type signatures - Lennart Augustsson's Djinn [1] (<http://lambda-the-ultimate.org/node/1178>) is a program that will do this for you. See Phil Gossart's Google tech talk: [2] (<http://video.google.com/videoplay?docid=-4851250372422374791>) for background on the theory behind Djinn; and Dan Piponi's article: [3] (<http://www.haskell.org/sitewiki/images/1/14/TMR-Issue6.pdf>) which uses Djinn in deriving Continuation Passing Style.

Example: a complicated control structure

This example was originally taken from the 'The Continuation monad' section of the All about monads tutorial (http://www.haskell.org/all_about_monads/html/index.html), used with permission.

Example: Using Cont for a complicated control structure 

```

{- We use the continuation monad to perform "escapes" from code blocks.
   This function implements a complicated control structure to process
   numbers:

   Input (n)           Output           List Shown
   =====           =====
   0-9                 n                 none
   10-199              number of digits in (n/2)    digits of (n/2)
   200-19999           n                 digits of (n/2)
   20000-1999999      (n/2) backwards    none
   >= 2000000         sum of digits of (n/2)    digits of (n/2)
-}
fun :: Int -> String
fun n = (`runCont` id) $ do
  str <- callCC $ \exit1 -> do
    when (n < 10) (exit1 $ show n)
    let ns = map digitToInt (show $ n `div` 2)
        n' <- callCC $ \exit2 -> do
          when (length ns < 3) (exit2 $ length ns)
          when (length ns < 5) (exit2 n)
          when (length ns < 7) $ do
            let ns' = map intToDigit (reverse ns)
                exit1 (dropWhile (=='0') ns')
            -- escape 2 levels
            return $ sum ns
          return $ "(ns = " ++ show ns ++ ") " ++ show n'
        return $ "Answer: " ++ str

```

Because it isn't initially clear what's going on, especially regarding the usage of `callCC`, we will explore this somewhat.

Analysis of the example

Firstly, we can see that `fun` is a function that takes an integer `n`. We basically implement a control structure using `Cont` and `callCC` that does different things based on the range that `n` falls in, as explained with the comment at the top of the function. Let's dive into the analysis of how it works.

1. Firstly, the `(`runCont` id)` at the top just means that we run the `Cont` block that follows with a final continuation of `id`. This is necessary as the result type of `fun` doesn't mention `Cont`.
2. We bind `str` to the result of the following `callCC` do-block:
 1. If `n` is less than 10, we exit straight away, just showing `n`.
 2. If not, we proceed. We construct a list, `ns`, of digits of `n `div` 2`.
 3. `n'` (an `Int`) gets bound to the result of the following inner `callCC` do-block.
 1. If `length ns < 3`, i.e., if `n `div` 2` has less than 3 digits, we pop out of this inner do-block with the number of digits as the result.
 2. If `n `div` 2` has less than 5 digits, we pop out of the inner do-block returning the original `n`.
 3. If `n `div` 2` has less than 7 digits, we pop out of *both* the inner and outer do-blocks, with the result of the digits of `n `div` 2` in reverse order (a `String`).
 4. Otherwise, we end the inner do-block, returning the sum of the digits of `n `div` 2`.
4. We end this do-block, returning the `String` `"(ns = X) Y"`, where `X` is `ns`, the digits of `n `div` 2`, and `Y` is the result from the inner do-block, `n'`.

3. Finally, we return out of the entire function, with our result being the string "Answer: Z", where Z is the string we got from the `callCC` do-block.

Example: exceptions

One use of continuations is to model exceptions. To do this, we hold on to two continuations: one that takes us out to the handler in case of an exception, and one that takes us to the post-handler code in case of a success. Here's a simple function that takes two numbers and does integer division on them, failing when the denominator is zero.

Example: An exception-throwing `div`

```
divExcpT :: Int -> Int -> (String -> Cont r Int) -> Cont r Int
divExcpT x y handler =
  callCC $ \ok -> do
    err <- callCC $ \notOk -> do
      when (y == 0) $ notOk "Denominator 0"
      ok $ x `div` y
    handler err

{- For example,
runCont (divExcpT 10 2 error) id --> 5
runCont (divExcpT 10 0 error) id --> *** Exception: Denominator 0
-}
```

How does it work? We use two nested calls to `callCC`. The first labels a continuation that will be used when there's no problem. The second labels a continuation that will be used when we wish to throw an exception. If the denominator isn't 0, `x `div` y` is thrown into the `ok` continuation, so the execution pops right back out to the top level of `divExcpT`. If, however, we were passed a zero denominator, we throw an error message into the `notOk` continuation, which pops us out to the inner do-block, and that string gets assigned to `err` and given to `handler`.

A more general approach to handling exceptions can be seen with the following function. Pass a computation as the first parameter (which should be a function taking a continuation to the error handler) and an error handler as the second parameter. This example takes advantage of the generic `MonadCont` class which covers both `Cont` and `ContT` by default, plus any other continuation classes the user has defined.

Example: General `try` using continuations.

```
tryCont :: MonadCont m => ((err -> m a) -> m a) -> (err -> m a) -> m a
tryCont c h =
  callCC $ \ok -> do
    err <- callCC $ \notOk -> do x <- c notOk; ok x
    h err
```

For an example using `try`, see the following program.

Example: Using `try`

```

data SqrtException = LessThanZero deriving (Show, Eq)

sqrtIO :: (SqrtException -> ContT r IO ()) -> ContT r IO ()
sqrtIO throw = do
  ln <- lift (putStr "Enter a number to sqrt: " >> readLn)
  when (ln < 0) (throw LessThanZero)
  lift $ print (sqrt ln)

main = runContT (tryCont sqrtIO (lift . print)) return

```

Example: coroutines**Notes**

1. ↑ At least as far as types are concerned, but we're trying to avoid that word :)
2. ↑ More technically, `fst` and `snd` have types which limit them to pairs. It would be impossible to define projection functions on tuples in general, because they'd have to be able to accept tuples of different sizes, so the type of the function would vary.
3. ↑ In fact, these are one and the same concept in Haskell.
4. ↑ This isn't quite what `chr` and `ord` do, but that description fits our purposes well, and it's close enough.
5. ↑ To make things even more confusing, there's actually even more than one type for integers! Don't worry, we'll come on to this in due course.
6. ↑ This has been somewhat simplified to fit our purposes. Don't worry, the essence of the function is there.
7. ↑ Some of the newer type system extensions to GHC *do* break this, however, so you're better off just always putting down types anyway.
8. ↑ This is a slight lie. That type signature would mean that you can compare two values of any type whatsoever, but this clearly isn't true: how can you see if two functions are equal? Haskell includes a kind of 'restricted polymorphism' that allows type variables to range over some, but not all types. Haskell implements this using *type classes*, which we'll learn about later. In this case, the correct type of `(==)` is `Eq a => a -> a -> Bool`.
9. ↑ In mathematics, $n!$ normally means the factorial of n , but that syntax is impossible in Haskell, so we don't use it here.
10. ↑ Actually, defining the factorial of 0 to be 1 is not just arbitrary; it's because the factorial of 0 represents an empty product.
11. ↑ This is no coincidence; without mutable variables, recursion is the only way to implement control structures. This might sound like a limitation until you get used to it (it isn't, really).
12. ↑ Actually, it's using a function called `foldl`, which actually does the recursion.
13. ↑ Moggi, Eugenio (1991). "Notions of Computation and Monads". *Information and Computation* **93** (1).
14. ↑ w:Philip Wadler. Comprehending Monads (<http://citeseer.ist.psu.edu/wadler92comprehending.html>) . Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice. 1990.
15. ↑ w:Philip Wadler. The Essence of Functional Programming (<http://citeseer.ist.psu.edu/wadler92essence.html>) . Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1992.
16. ↑ Simon Peyton Jones, Philip Wadler (1993). "Imperative functional programming"

(<http://homepages.inf.ed.ac.uk/wadler/topics/monads.html#imperative>) . *20'th Symposium on Principles of Programming Languages*.

- ↑ It infers a monomorphic type because κ is bound by a lambda expression, and things bound by lambdas always have monomorphic types. See Polymorphism.

Mutable objects

As one of the key strengths of Haskell is its *purity*: all side-effects are encapsulated in a monad. This makes reasoning about programs much easier, but many practical programming tasks require manipulating state and using imperative structures. This chapter will discuss advanced programming techniques for using imperative constructs, such as references and mutable arrays, without compromising (too much) purity.

The ST and IO monads

Recall the The State Monad and The IO monad from the chapter on Monads. These are two methods of structuring imperative effects. Both references and arrays can live in state monads or the IO monad, so which one is more appropriate for what, and how does one use them?

State references: STRef and IORef

Mutable arrays

Examples

Zippers

Theseus and the Zipper

The Labyrinth

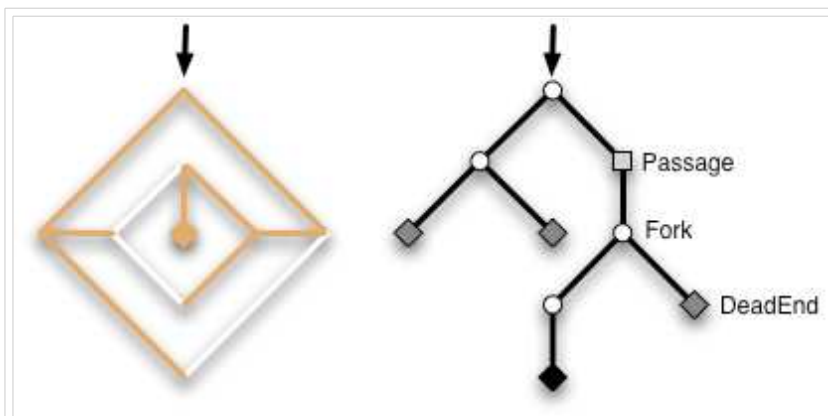
"Theseus, we have to do something" said Homer, chief marketing officer of Ancient Geeks Inc.. Theseus put the Minotaur action figure™ back onto the shelf and nods. "Today's children are no longer interested in the ancient myths, they prefer modern heroes like Spiderman or Sponge Bob." *Heroes*. Theseus knew well how much he has been a hero in the labyrinth back then on Crete^[18]. But those "modern heroes" did not even try to appear realistic. What made them so successful? Anyway, if the pending sales problems could not be resolved, the shareholders would certainly arrange a passage over the Styx for Ancient Geeks Inc.

"Heureka! Theseus, I have an idea: we implement your story with the Minotaur as a computer game! What do you say?" Homer was right. There had been several books, epic (and chart breaking) songs, a mandatory movie trilogy and uncountable Theseus & the Minotaur™ gimmicks, but a computer game was missing. "Perfect, then.

Now, Theseus, your task is to implement the game".

A true hero, Theseus chose Haskell as the language to implement the company's redeeming product in. Of course, exploring the labyrinth of the Minotaur was to become one of the game's highlights. He pondered: "We have a two-dimensional labyrinth whose corridors can point in many directions. Of course, we can abstract from the detailed lengths and angles: for the purpose of finding the way out, we only need to know how the path forks. To keep things easy, we model the labyrinth as a tree. This way, the two branches of a fork cannot join again when walking deeper and the player cannot go round in circles. But I think there is enough opportunity to get lost; and this way, if the player is patient enough, he can explore the entire labyrinth with the left-hand rule."

```
data Node a = DeadEnd a
            | Passage a (Node a)
            | Fork   a (Node a) (Node a)
```



An example labyrinth and its representation as tree.

Theseus made the nodes of the labyrinth carry an extra parameter of type `a`. Later on, it may hold game relevant information like the coordinates of the spot a node designates, the ambience around it, a list of game items that lie on the floor, or a list of monsters wandering in that section of the labyrinth. We assume that two helper functions

```
get :: Node a -> a
put :: a -> Node a -> Node a
```

retrieve and change the value of type `a` stored in the first argument of every constructor of `Node a`.

Exercises

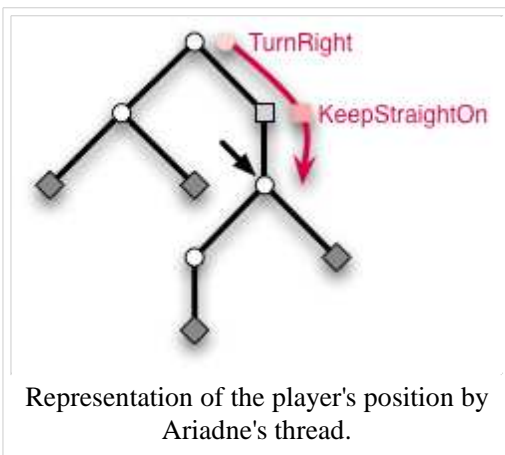
1. Implement `get` and `put`. One case for `get` is
`get (Passage x _) = x.`
2. To get a concrete example, write down the labyrinth shown in the picture as a value of type `Node (Int, Int)`. The extra parameter `(Int, Int)` holds the cartesian coordinates of a node.

"Mh, how to represent the player's current position in the labyrinth? The player can explore deeper by choosing left or right branches, like in"

```
turnRight :: Node a -> Maybe (Node a)
turnRight (Fork _ l r) = Just r
turnRight _           = Nothing
```

"But replacing the current top of the labyrinth with the corresponding sub-labyrinth this way is not an option, because he cannot go back then." He pondered. "Ah, we can apply *Ariadne's trick with the thread* for going back. We simply represent the player's position by the list of branches his thread takes, the labyrinth always remains the same."

```
data Branch = KeepStraightOn
            | TurnLeft
            | TurnRight
type Thread = [Branch]
```



"For example, a thread `[TurnRight,KeepStraightOn]` means that the player took the right branch at the entrance and then went straight down a `Passage` to reach its current position. With the thread, the player can now explore the labyrinth by extending or shortening it. For instance, the function `turnRight` extends the thread by appending the `TurnRight` to it."

```
turnRight :: Thread -> Thread
turnRight t = t ++ [TurnRight]
```

"To access the extra data, i.e. the game relevant items and such, we simply follow the thread into the labyrinth."

```
retrieve :: Thread -> Node a -> a
retrieve [] n = get n
retrieve (KeepStraightOn:bs) (Passage _ n) = retrieve bs n
retrieve (TurnLeft :bs) (Fork _ l r) = retrieve bs l
retrieve (TurnRight :bs) (Fork _ l r) = retrieve bs r
```

Exercises

Write a function `update` that applies a function of type `a -> a` to the extra data at the player's position.

Theseus' satisfaction over this solution did not last long. "Unfortunately, if we want to extend the path or go back a step, we have to change the last element of the list. We could store the list in reverse, but even then, we have to follow the thread again and again to access the data in the labyrinth at the player's position. Both actions take time proportional to the length of the thread and for large labyrinths, this will be too long. Isn't there

another way?"

Ariadne's Zipper

While Theseus was a skillful warrior, he did not train much in the art of programming and could not find a satisfying solution. After intense but fruitless cogitation, he decided to call his former love Ariadne to ask her for advice. After all, it was she who had the idea with the thread.

"Ariadne Consulting. What can I do for you?"

Our hero immediately recognized the voice.

"Hello Ariadne, it's Theseus."

An uneasy silence paused the conversation. Theseus remembered well that he had abandoned her on the island of Naxos and knew that she would not appreciate his call. But Ancient Geeks Inc. was on the road to Hades and he had no choice.

"Uhm, darling, ... how are you?"

Ariadne retorted an icy response, "Mr. Theseus, the times of *darling* are long over. What do you want?"

"Well, I uhm ... I need some help with a programming problem. I'm programming a new Theseus & the Minotaur™ computer game."

She jeered, "Yet another artifact to glorify your 'heroic being'? And you want me of all people to help you?"

"Ariadne, please, I beg of you, Ancient Geeks Inc. is on the brink of insolvency. The game is our last resort!"

After a pause, she came to a decision.

"Fine, I will help you. But only if you transfer a substantial part of Ancient Geeks Inc. to me. Let's say thirty percent."

Theseus turned pale. But what could he do? The situation was desperate enough, so he agreed but only after negotiating Ariadne's share to a tenth.

After Theseus told Ariadne of the labyrinth representation he had in mind, she could immediately give advice,

"You need a **zipper**."

"Huh? What does the problem have to do with my fly?"

"Nothing, it's a data structure first published by Gérard Huet^[19]."

"Ah."

"More precisely, it's a purely functional way to augment tree-like data structures like lists or binary trees with a single **focus** or **finger** that points to a subtree inside the data structure and allows constant time updates and lookups at the spot it points to^[20]. In our case, we want a focus on the player's position."

"I know for myself that I want fast updates, but how do I code it?"

"Don't get impatient, you cannot solve problems by coding, you can only solve them by thinking. The only place where we can get constant time updates in a purely functional data structure is the topmost node^{[21][22]}. So, the focus necessarily has to be at the top. Currently, the topmost node in your labyrinth is always the entrance, but your previous idea of replacing the labyrinth by one of its sub-labyrinths ensures that the player's position is at the topmost node."

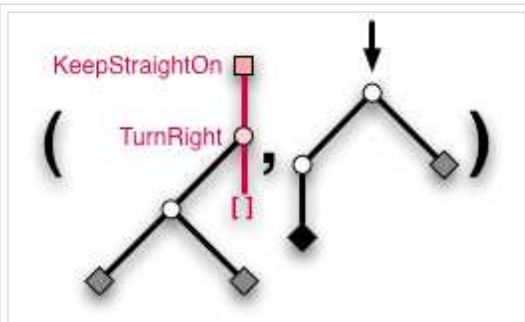
"But then, the problem is how to go back, because all those sub-labyrinths get lost that the player did not choose to branch into."

"Well, you can use my thread in order not to lose the sub-labyrinths."

Ariadne savored Theseus' puzzlement but quickly continued before he could complain that he already used Ariadne's thread,

"The key is to *glue the lost sub-labyrinths to the thread* so that they actually don't get lost at all. The intention is that the thread and the current sub-labyrinth complement one another to the whole labyrinth. With 'current' sub-labyrinth, I mean the one that the player stands on top of. The zipper simply consists of the thread and the current sub-labyrinth."

```
type Zipper a = (Thread a, Node a)
```



The zipper is a pair of Ariadne's thread and the current sub-labyrinth that the player stands on top. The main thread is colored red and has sub-labyrinths attached to it, such that the whole labyrinth can be reconstructed from the pair.

Theseus didn't say anything.

"You can also view the thread as a **context** in which the current sub-labyrinth resides. Now, let's find out how to define `Thread a`. By the way, `Thread` has to take the extra parameter `a` because it now stores sub-labyrinths. The thread is still a simple list of branches, but the branches are different from before."

```
data Branch a = KeepStraightOn a
              | TurnLeft a (Node a)
              | TurnRight a (Node a)
type Thread a = [Branch a]
```

"Most importantly, `TurnLeft` and `TurnRight` have a sub-labyrinth glued to them. When the player chooses say to turn right, we extend the thread with a `TurnRight` and now attach the untaken left branch to it, so that it doesn't get lost."

Theseus interrupts, "Wait, how would I implement this behavior as a function `turnRight`? And what about the first argument of type `a` for `TurnRight`? Ah, I see. We not only need to glue the branch that would get lost, but also the extra data of the `Fork` because it would otherwise get lost as well. So, we can generate a new branch by a preliminary"

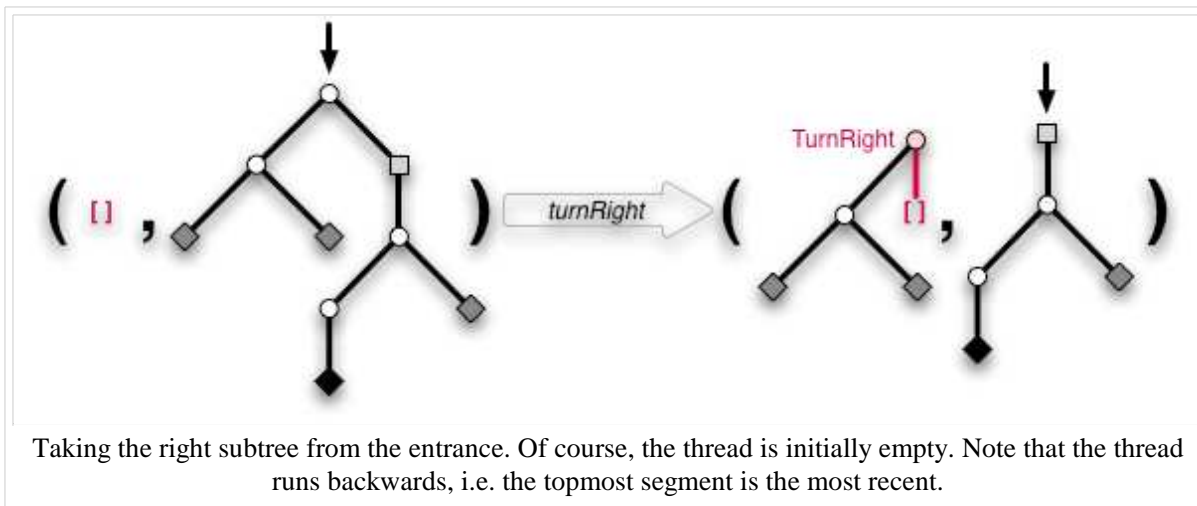
```
branchRight (Fork x l r) = TurnRight x l
```

"Now, we have to somehow extend the existing thread with it."

"Indeed. The second point about the thread is that it is stored *backwards*. To extend it, you put a new branch in front of the list. To go back, you delete the topmost element."

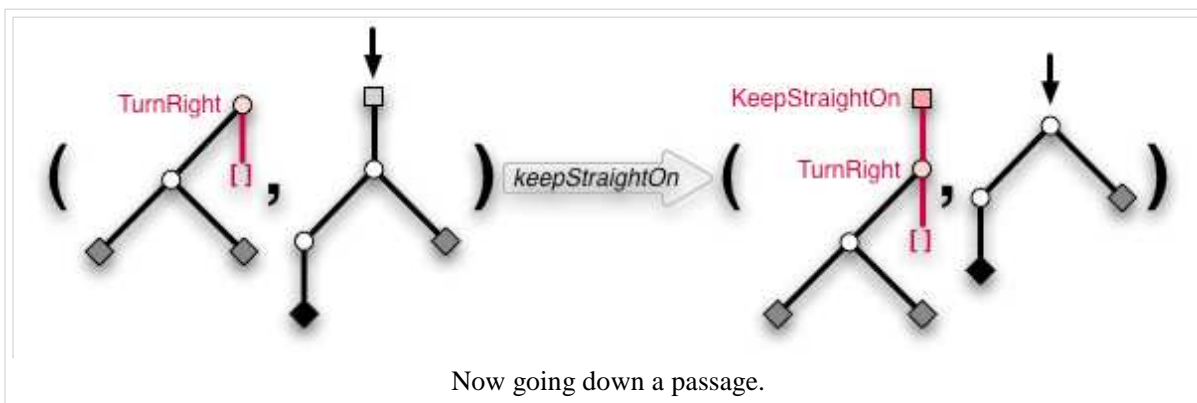
"Aha, this makes extending and going back take only constant time, not time proportional to the length as in my previous version. So the final version of `turnRight` is"

```
turnRight :: Zipper a -> Maybe (Zipper a)
turnRight (t, Fork x l r) = Just (TurnRight x l : t, r)
turnRight _               = Nothing
```



"That was not too difficult. So let's continue with `keepStraightOn` for going down a passage. This is even easier than choosing a branch as we only need to keep the extra data:"

```
keepStraightOn :: Zipper a -> Maybe (Zipper a)
keepStraightOn (t, Passage x n) = Just (KeepStraightOn x : t, n)
keepStraightOn _                = Nothing
```



Exercises

Write the function `turnLeft`.

Pleased, he continued, "But the interesting part is to go back, of course. Let's see..."

```
back :: Zipper a -> Maybe (Zipper a)
back ([], _) = Nothing
back (KeepStraightOn x : t, n) = Just (t, Passage x n)
back (TurnLeft x r : t, l) = Just (t, Fork x l r)
back (TurnRight x l : t, r) = Just (t, Fork x l r)
```

"If the thread is empty, we're already at the entrance of the labyrinth and cannot go back. In all other cases, we have to wind up the thread. And thanks to the attachments to the thread, we can actually reconstruct the sub-labyrinth we came from."

Ariadne remarked, "Note that a partial test for correctness is to check that each bound variable like `x`, `l` and `r` on the left hand side appears exactly once at the right hands side as well. So, when walking up and down a zipper, we only redistribute data between the thread and the current sub-labyrinth."

Exercises

1. Now that we can navigate the zipper, code the functions `get`, `put` and `update` that operate on the extra data at the player's position.
2. Zippers are by no means limited to the concrete example `Node a`, they can be constructed for all tree-like data types. Go on and construct a zipper for binary trees

```
data Tree a = Leaf a | Bin (Tree a) (Tree a)
```

Start by thinking about the possible branches `Branch a` that a thread can take. What do you have to glue to the thread when exploring the tree?

3. Simple lists have a zipper as well.

```
data List a = Empty | Cons a (List a)
```

What does it look like?

4. Write a complete game based on Theseus' labyrinth.

Heureka! That was the solution Theseus sought and Ancient Geeks Inc. should prevail, even if partially sold to Ariadne Consulting. But one question remained:

"Why is it called zipper?"

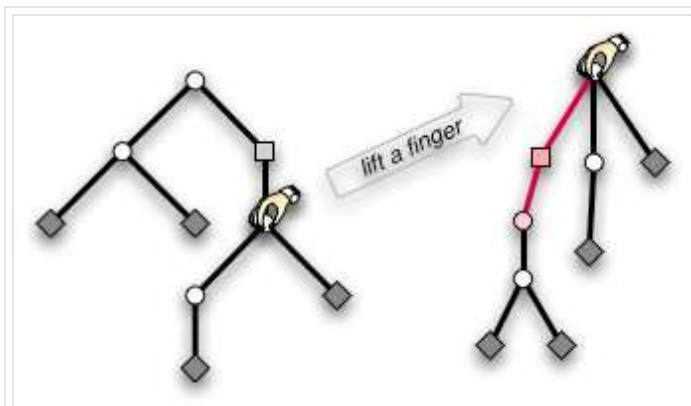
"Well, I would have called it 'Ariadne's pearl necklace'. But most likely, it's called zipper because the thread is in analogy to the open part and the sub-labyrinth is like the closed part of a zipper. Moving around in the data structure is analogous to zipping or unzipping the zipper."

"Ariadne's pearl necklace," he articulated disdainfully. "As if your thread was any help back then on Crete."

"As if the idea with the thread was yours," she replied.

"Bah, I need no thread," he defied the fact that he actually did need the thread to program the game.

Much to his surprise, she agreed, "Well, indeed you don't need a thread. Another view is to literally grab the tree at the focus with your finger and lift it up in the air. The focus will be at the top and all other branches of the tree hang down. You only have to assign the resulting tree a suitable algebraic data type, most likely that of the zipper."



Grab the focus with your finger, lift it in the air and the hanging branches will form new tree with your finger at the top, ready to be structured by an algebraic data type.

"Ah." He didn't need Ariadne's thread but he needed Ariadne to tell him? That was too much.

"Thank you, Ariadne, good bye."

She did not hide her smirk as he could not see it anyway through the phone.

Exercises

Take a list, fix one element in the middle with your finger and lift the list into the air. What type can you give to the resulting tree?

Half a year later, Theseus stopped in front of a shop window, defying the cold rain that tried to creep under his buttoned up anorak. Blinking letters announced

"Spider-Man: lost in the Web"

- find your way through the labyrinth of threads -
the great computer game by Ancient Geeks Inc.

He cursed the day when he called Ariadne and sold her a part of the company. Was it she who contrived the unfriendly takeover by WineOS Corp., led by Ariadne's husband Dionysus? Theseus watched the raindrops finding their way down the glass window. After the production line was changed, nobody would produce Theseus and the Minotaur™ merchandise anymore. He sighed. His time, the time of heroes, was over. Now came the super-heroes.

Differentiation of data types

The previous section has presented the zipper, a way to augment a tree-like data structure `Node a` with a finger that can focus on the different subtrees. While we constructed a zipper for a particular data structure `Node a`, the construction can be easily adapted to different tree data structures by hand.

Exercises

Start with a ternary tree

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

and derive the corresponding `Thread a` and `Zipper a`.

Mechanical Differentiation

But there is also an entirely mechanical way to derive the zipper of any (suitably regular) data type. Surprisingly, 'derive' is to be taken literally, for the zipper can be obtained by the **derivative** of the data type, a discovery first described by Conor McBride^[23]. The subsequent section is going to explicate this truly wonderful mathematical gem.

For a systematic construction, we need to calculate with types. The basics of structural calculations with types are outlined in a separate chapter `Generic Programming` and we will heavily rely on this material.

Let's look at some examples to see what their zippers have in common and how they hint differentiation. The type of binary tree is the fixed point of the recursive equation

$$Tree_2 = 1 + Tree_2 \times Tree_2.$$

When walking down the tree, we iteratively choose to enter the left or the right subtree and then glue the not-entered subtree to Ariadne's thread. Thus, the branches of our thread have the type

$$Branch_2 = Tree_2 + Tree_2 \cong 2 \times Tree_2.$$

Similarly, the thread for a ternary tree

$$Tree_3 = 1 + Tree_3 \times Tree_3 \times Tree_3$$

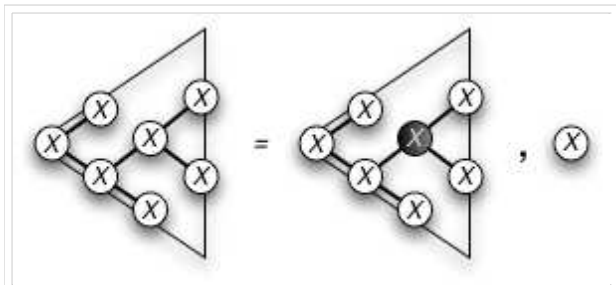
has branches of type

$$Branch_3 = 3 \times Tree_3 \times Tree_3$$

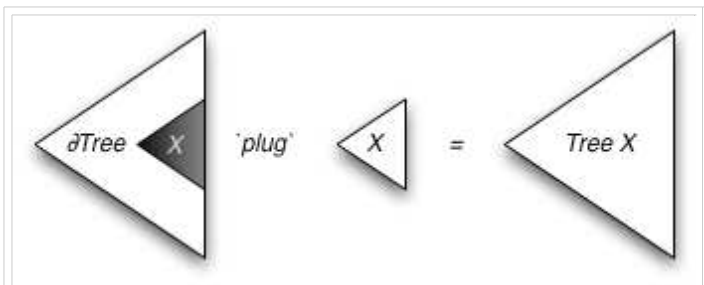
because at every step, we can choose between three subtrees and have to store the two subtrees we don't enter.

Isn't this strikingly similar to the derivatives $\frac{d}{dx}x^2 = 2 \times x$ and $\frac{d}{dx}x^3 = 3 \times x^2$?

The key to the mystery is the notion of the **one-hole context** of a data structure. Imagine a data structure parameterised over a type X , like the type of trees $Tree\ X$. If we were to remove one of the items of this type X from the structure and somehow mark the now empty position, we obtain a structure with a marked hole. The result is called "one-hole context" and inserting an item of type X into the hole gives back a completely filled $Tree\ X$. The hole acts as a distinguished position, a focus. The figures illustrate this.

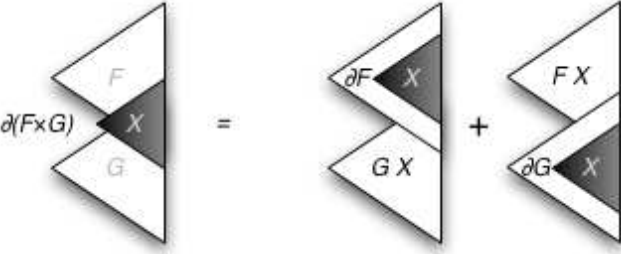


Removing a value of type X from a $Tree\ X$ leaves a hole at that position.

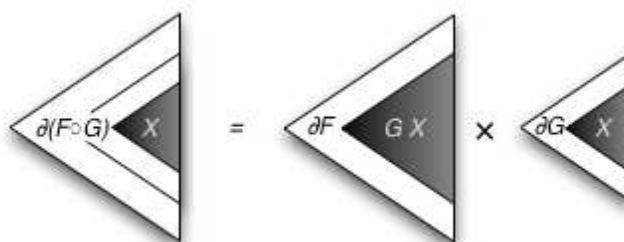


A more abstract illustration of plugging X into a one-hole context.

Of course, we are interested in the type to give to a one-hole context, i.e. how to represent it in Haskell. The problem is how to efficiently mark the focus. But as we will see, finding a representation for one-hole contexts by induction on the structure of the type we want to take the one-hole context of automatically leads to an efficient data type^[24]. So, given a data structure $F\ X$ with a functor F and an argument type X , we want to calculate the type $\partial F\ X$ of one-hole contexts from the structure of F . As our choice of notation ∂F already reveals, the rules for constructing one-hole contexts of sums, products and compositions are exactly Leibniz' rules for differentiation.

One-hole context	Illustration
$(\partial Const_A)\ X = 0$	There is no X in $A = Const_A\ X$, so the type of its one-hole contexts must be empty.
$(\partial Id)\ X = 1$	There is only one position for items X in $X = Id\ X$. Removing one X leaves no X in the result. And as there is only one position we can remove it from, there is exactly one one-hole context for $Id\ X$. Thus, the type of one-hole contexts is the singleton type.
$\partial(F + G) = \partial F + \partial G$	As an element of type $F + G$ is either of type F or of type G , a one-hole context is also either ∂F or ∂G .
$\partial(F \times G) = F \times \partial G + \partial F \times G$	 <p>The hole in a one-hole context of a pair is either in the first or in the second component.</p>

$$\partial(F \circ G) = (\partial F \circ G) \times \partial G$$



Chain rule. The hole in a composition arises by making a hole in the enclosing structure and fitting the enclosed structure in.

Of course, the function `plug` that fills a hole has the type $(\partial F X) \times X \rightarrow X$.

So far, the syntax ∂ denotes the differentiation of functors, i.e. of a kind of type functions with one argument. But there is also a handy expression oriented notation ∂_X slightly more suitable for calculation. The subscript indicates the variable with respect to which we want to differentiate. In general, we have

$$(\partial F) X = \partial_X(F X)$$

An example is

$$\partial(Id \times Id) X = \partial_X(X \times X) = 1 \times X + X \times 1 \cong 2 \times X$$

Of course, ∂_X is just point-wise whereas ∂ is point-free style.

Exercises

1. Rewrite some rules in point-wise style. For example, the left hand side of the product rule becomes $\partial_X(F X \times G X) = \dots$
2. To get familiar with one-hole contexts, differentiate the product $X^n := X \times X \times \dots \times X$ of exactly n factors formally and convince yourself that the result is indeed the corresponding one-hole context.
3. Of course, one-hole contexts are useless if we cannot plug values of type X back into them. Write the `plug` functions corresponding to the five rules.
4. Formulate the **chain rule** for **two variables** and prove that it yields one-hole contexts. You can do this by viewing a bifunctor $F X Y$ as an normal functor in the pair (X, Y) . Of course, you may need a handy notation for partial derivatives of bifunctors in point-free style.

Zippers via Differentiation

The above rules enable us to construct **zipper**s for recursive data types $\mu F := \mu X. F X$ where F is a polynomial functor. A zipper is a focus on a particular subtree, i.e. substructure of type μF inside a large tree of the same type. As in the previous chapter, it can be represented by the subtree we want to focus at and the thread, that is the context in which the subtree resides

$$\text{Zipper}_F = \mu F \times \text{Context}_F.$$

Now, the context is a series of steps each of which chooses a particular subtree μF among those in $F \mu F$. Thus, the unchosen subtrees are collected together by the one-hole context $\partial F (\mu F)$. The hole of this context comes from removing the subtree we've chosen to enter. Putting things together, we have

$$\text{Context}_F = \text{List} (\partial F (\mu F)).$$

or equivalently

$$\text{Context}_F = 1 + \partial F (\mu F) \times \text{Context}_F.$$

To illustrate how a concrete calculation proceeds, let's systematically construct the zipper for our labyrinth data type

```
data Node a = DeadEnd a
            | Passage a (Node a)
            | Fork a (Node a) (Node a)
```

This recursive type is the fixed point

$$\text{Node } A = \mu X. \text{Node} F_A X$$

of the functor

$$\text{Node} F_A X = A + A \times X + A \times X \times X.$$

In other words, we have

$$\text{Node } A \cong \text{Node} F_A (\text{Node } A) \cong A + A \times \text{Node } A + A \times \text{Node } A \times \text{Node } A.$$

The derivative reads

$$\partial_X (\text{Node} F_A X) \cong A + 2 \times A \times X$$

and we get

$$\partial \text{Node} F_A (\text{Node } A) \cong A + 2 \times A \times \text{Node } A.$$

Thus, the context reads

$$\text{Context}_{\text{Node} F} \cong \text{List} (\partial \text{Node} F_A (\text{Node } A)) \cong \text{List} (A + 2 \times A \times (\text{Node } A)).$$

Comparing with

```

data Branch a = KeepStraightOn a
              | TurnLeft a (Node a)
              | TurnRight a (Node a)
data Thread a = [Branch a]

```

we see that both are exactly the same as expected!

Exercises

1. Redo the zipper for a ternary tree, but with differentiation this time.
2. Construct the zipper for a list.
3. Rhetorical question concerning the previous exercise: what's the difference between a list and a stack?

Differentiation of Fixed Point

There is more to data types than sums and products, we also have a fixed point operator with no direct correspondence in calculus. Consequently, the table is missing a rule of differentiation, namely how to differentiate fixed points $\mu F X = \mu Y. F X Y$:

$$\partial_X(\mu F X) = ?.$$

As its formulation involves the chain rule in two variables, we delegate it to the exercises. Instead, we will calculate it for our concrete example type *Node A*:

$$\begin{aligned} \partial_A(\text{Node } A) &= \partial_A(A + A \times \text{Node } A + A \times \text{Node } A \times \text{Node } A) \\ &\cong 1 + \text{Node } A + \text{Node } A \times \text{Node } A \\ &\quad + \partial_A(\text{Node } A) \times (A + 2 \times A \times \text{Node } A). \end{aligned}$$

Of course, expanding $\partial_A(\text{Node } A)$ further is of no use, but we can see this as a fixed point equation and arrive at

$$\partial_A(\text{Node } A) = \mu X. T A + S A \times X$$

with the abbreviations

$$T A = 1 + \text{Node } A + \text{Node } A \times \text{Node } A$$

and

$$S A = A + 2 \times A \times \text{Node } A.$$

The recursive type is like a list with element types $S A$, only that the empty list is replaced by a base case of type $T A$. But given that the list is finite, we can replace the base case with 1 and pull $T A$ out of the list:

$$\partial_A(\text{Node } A) \cong T A \times (\mu X. 1 + S A \times X) = T A \times \text{List } (S A).$$

Comparing with the zipper we derived in the last paragraph, we see that the list type is our context

$$\text{List } (S A) \cong \text{Context}_{\text{Node } F}$$

and that

$$A \times T A \cong \text{Node } A.$$

In the end, we have

$$\text{Zipper}_{\text{Node}F} \cong \partial_A(\text{Node } A) \times A.$$

Thus, differentiating our concrete example *Node A* with respect to *A* yields the zipper up to an *A*!

Exercises

1. Use the chain rule in two variables to formulate a rule for the differentiation of a fixed point.
2. Maybe you know that there are inductive (μ) and coinductive fixed points (ν). What's the rule for coinductive fixed points?

Zippers vs Contexts

In general however, zippers and one-hole contexts denote different things. The zipper is a focus on arbitrary subtrees whereas a one-hole context can only focus on the argument of a type constructor. Take for example the data type

```
data Tree a = Leaf a | Bin (Tree a) (Tree a)
```

which is the fixed point

$$\text{Tree } A = \mu X. A + X \times X.$$

The zipper can focus on subtrees whose top is `Bin` or `Leaf` but the hole of one-hole context of *Tree A* may only focus a `Leaf`s because this is where the items of type *A* reside. The derivative of *Node A* only turned out to be the zipper because every top of a subtree is always decorated with an *A*.

Exercises

1. Surprisingly, $\partial_A(\text{Tree } A) \times A$ and the zipper for *Tree A* again turn out to be the same type. Doing the calculation is not difficult but can you give a reason why this has to be the case?
2. Prove that the zipper construction for μF can be obtained by introducing an auxiliary variable *Y*, differentiating $\mu X. Y \times F X$ with respect to it and re-substituting $Y = 1$. Why does this work?
3. Find a type *G A* whose zipper is different from the one-hole context.

Conclusion

We close this section by asking how it may happen that rules from calculus appear in a discrete setting. Currently, nobody knows. But at least, there is a discrete notion of **linear**, namely in the sense of "exactly once". The key feature of the function that plugs an item of type X into the hole of a one-hole context is the fact that the item is used exactly once, i.e. linearly. We may think of the plugging map as having type

$$\partial_X F X \rightarrow (X \multimap F X)$$

where $A \multimap B$ denotes a linear function, one that does not duplicate or ignore its argument like in linear logic. In a sense, the one-hole context is a representation of the function space $X \multimap F X$, which can be thought of being a linear approximation to $X \rightarrow F X$.

Notes

1. ↑ At least as far as types are concerned, but we're trying to avoid that word :)
2. ↑ More technically, `fst` and `snd` have types which limit them to pairs. It would be impossible to define projection functions on tuples in general, because they'd have to be able to accept tuples of different sizes, so the type of the function would vary.
3. ↑ In fact, these are one and the same concept in Haskell.
4. ↑ This isn't quite what `chr` and `ord` do, but that description fits our purposes well, and it's close enough.
5. ↑ To make things even more confusing, there's actually even more than one type for integers! Don't worry, we'll come on to this in due course.
6. ↑ This has been somewhat simplified to fit our purposes. Don't worry, the essence of the function is there.
7. ↑ Some of the newer type system extensions to GHC *do* break this, however, so you're better off just always putting down types anyway.
8. ↑ This is a slight lie. That type signature would mean that you can compare two values of any type whatsoever, but this clearly isn't true: how can you see if two functions are equal? Haskell includes a kind of 'restricted polymorphism' that allows type variables to range over some, but not all types. Haskell implements this using *type classes*, which we'll learn about later. In this case, the correct type of `(=)` is `Eq a => a -> a -> Bool`.
9. ↑ In mathematics, $n!$ normally means the factorial of n , but that syntax is impossible in Haskell, so we don't use it here.
10. ↑ Actually, defining the factorial of 0 to be 1 is not just arbitrary; it's because the factorial of 0 represents an empty product.
11. ↑ This is no coincidence; without mutable variables, recursion is the only way to implement control structures. This might sound like a limitation until you get used to it (it isn't, really).
12. ↑ Actually, it's using a function called `foldl`, which actually does the recursion.
13. ↑ Moggi, Eugenio (1991). "Notions of Computation and Monads". *Information and Computation* **93** (1).
14. ↑ w:Philip Wadler. Comprehending Monads (<http://citeseer.ist.psu.edu/wadler92comprehending.html>) . Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice. 1990.
15. ↑ w:Philip Wadler. The Essence of Functional Programming (<http://citeseer.ist.psu.edu/wadler92essence.html>) . Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1992.
16. ↑ Simon Peyton Jones, Philip Wadler (1993). "Imperative functional programming" (<http://homepages.inf.ed.ac.uk/wadler/topics/monads.html#imperative>) . *20'th Symposium on Principles of Programming Languages*.
17. ↑ It infers a monomorphic type because `κ` is bound by a lambda expression, and things bound by lambdas always have monomorphic types. See Polymorphism.
18. ↑ Ian Stewart. *The true story of how Theseus found his way out of the labyrinth*. Scientific American, February 1991, page 137.
19. ↑ Gérard Huet. *The Zipper*. Journal of Functional Programming, 7 (5), Sept 1997, pp. 549--554. PDF (<http://www.st.cs.uni-sb.de/edu/seminare/2005/advanced-fp/docs/huet-zipper.pdf>)

20. ↑ Note the notion of *zipper* as coined by Gérard Huet also allows to replace whole subtrees even if there is no extra data associated with them. In the case of our labyrinth, this is irrelevant. We will come back to this in the section Differentiation of data types.
21. ↑ Of course, the second topmost node or any other node at most a constant number of links away from the top will do as well.
22. ↑ Note that changing the whole data structure as opposed to updating the data at a node can be achieved in amortized constant time even if more nodes than just the top node is affected. An example is incrementing a number in binary representation. While incrementing say $111 \dots 11$ must touch all digits to yield $1000 \dots 00$, the increment function nevertheless runs in constant amortized time (but not in constant worst case time).
23. ↑ Conor Mc Bride. *The Derivative of a Regular Type is its Type of One-Hole Contexts*. Available online. PDF (<http://www.cs.nott.ac.uk/~ctm/diff.pdf>)
24. ↑ This phenomenon already shows up with generic tries.

See Also

- Zipper (<http://www.haskell.org/haskellwiki/Zipper>) on the haskell.org wiki
- Generic Zipper and its applications (<http://okmij.org/ftp/Computation/Continuations.html#zipper>)
- Zipper-based file server/OS (<http://okmij.org/ftp/Computation/Continuations.html#zipper-fs>)

Fun with Types

Existentially quantified types

Existential types, or 'existentials' for short, are a way of 'squashing' a group of types into one, single type.

Firstly, a note to those of you following along at home: existentials are part of GHC's *type system extensions*. They aren't part of Haskell98, and as such you'll have to either compile any code that contains them with an extra command-line parameter of `-fglasgow-exts`, or put `{-# LANGUAGE ExistentialQuantification #-}` at the top of your sources that use existentials.

The `forall` keyword

The `forall` keyword is used to explicitly bring type variables into scope. For example, consider something you've innocuously seen written a hundred times so far:

Example: A polymorphic function

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



But what are these a and b ? Well, they're type variables, you answer. The compiler sees that they begin with a lowercase letter and as such allows any type to fill that role. Another way of putting this is that those variables are 'universally quantified'. If you've studied formal logic, you will have undoubtedly come across the quantifiers: 'for all' (or \forall) and 'exists' (or \exists). They 'quantify' whatever comes after them: for example, $\exists x.P$ (where P is any assertion. For example, P could be $x > 5$) means that there is at least one x such that P . $\forall x.P$ means that for every x you could imagine, P .

The `forall` keyword quantifies *types* in a similar way. We would rewrite `map`'s type as follows:

Example: Explicitly quantifying the type variables



```
map :: forall a b. (a -> b) -> [a] -> [b]
```

The `forall` can be seen to be 'bringing the type variables `a` and `b` into scope'. In Haskell, any use of a lowercase type implicitly begins with a `forall` keyword, so the two type declarations for `map` are equivalent, as are the declarations below:

Example: Two equivalent type statements



```
id :: a -> a
id :: forall a . a -> a
```

What makes life really interesting is that you can override this default behaviour by explicitly telling Haskell where the `forall` keyword goes. One use of this is for building **existentially quantified types**, also known as existential types, or simply existentials.

But wait... isn't `forall` the *universal* quantifier? How do you get an existential type out of that? We look at this in a later section. However, first, let's see an example of the power of existential types in action.

Example: heterogeneous lists

Haskell's typeclass system is powerful because it allows extensible groupings of types. So if you know a type instantiates some class `C`, you know certain things about that type. For example, `Int` instantiates `Eq`, so we know that `Ints` can be compared for equality.

Suppose we have a group of values. We don't know if they are all the same type, but we do know they all instantiate some class, i.e. we know we can do a certain thing with all the values (like compare them for equality were the class `Eq`). It might be useful to throw all these values into a list. We can't do this normally because lists are homogeneous with respect to types: they can only contain a single type. However, existential types allow us to loosen this requirement by defining a 'type hider' or 'type box':

Example: Constructing a heterogeneous list

```
data ShowBox = forall s. Show s => SB s

hetroList :: [ShowBox]
hetroList = [SB (), SB 5, SB True]
```

Now we know something about all the elements of this list: they can be converted to a string via `show`. In fact, that's pretty much the only thing we know about them.

Example: Using our heterogeneous list

```
instance Show ShowBox where
  show (SB s) = show s

main :: IO ()
main = mapM_ print hetroList
```

How does this work? In the definition of `show` for `ShowBox`, we don't know the type of `s`: when we originally wrapped the value, it didn't matter what its type was (as long as it was an instance of `Show`), so its type has been forgotten. We *do* know that the type is an instance of `Show` due to the constraint on the `SB` constructor. Therefore, it's legal to use the function `show` on `s`, as seen in the right-hand side of the function definition.

As for `main`, recall the type of `print`:

Example: Types of the functions involved

```
print :: Show s => s -> IO () -- print x = putStrLn (show x)
mapM_ :: (a -> m b) -> [a] -> m ()
mapM_ print :: Show s => [s] -> IO ()
```

As we just declared `ShowBox` an instance of `Show`, we can print the values in the list.

True *existential* types

Let's get back to the question we asked ourselves a couple of sections back. Why are we calling these existential types if `forall` is the universal quantifier?

Firstly, `forall` really does mean 'for all'. One way of thinking about types is as sets of values with that type, for example, `Bool` is the set `{True, False, ⊥}` (remember that bottom (often written `⊥`) is a member of every type!), `Integer` is the set of integers (and bottom), `String` the set of all possible strings (and bottom), and so on. `forall` serves as an intersection over those sets. For example, `forall a. a` is the intersection over all types, `{⊥}`, that is, the type (i.e. set) whose only value (i.e. element) is bottom. Why? Think about it: how many of the elements of `Bool`

Since you can get existential types with `forall`, Haskell forgoes the use of an `exists` keyword, which would just

appear in `String`? Bottom is the only value common to all types.

be redundant.

A few more examples:

1. `[forall a. a]` is the type of a list whose elements all have the type `forall a. a`, i.e. a list of bottoms.
2. `[forall a. Show a => a]` is the type of a list whose elements all have the type `forall a. Show a => a`. The `Show` class constraint limits the sets you intersect over (here we're only intersect over instances of `Show`), but `⊥` is still the only value common to all these types, so this too is a list of bottoms.
3. `[forall a. Num a => a]`. Again, the list where each element is a member of all types that instantiate `Num`. This could involve numeric literals, which have the type `forall a. Num a => a`, as well as bottom.
4. `forall a. [a]` is the type of the list whose elements have some (the same) type `a`, which can be assumed to be any type at all by a callee (and therefore this too is a list of bottoms).

In the last section, we developed a heterogeneous list using a 'type hider'. Conceptually, the type of a heterogeneous list is `[exists a. a]`, i.e. the list where all elements have type `exists a. a`. This 'exists' keyword (which isn't present in Haskell) is, as you may guess, a *union* of types. Therefore the aforementioned type is that of a list where all elements could take any type at all (and the types of different elements needn't be the same).

We can't get the same behaviour using `forall`s except by using the approach we showed above: datatypes. Let's declare one.

Example: An existential datatype

```
data T = forall a. MkT a
```



This means that:

Example: The type of our existential constructor

```
MkT :: forall a. a -> T
```



So we can pass any type we want to `MkT` and it'll convert it into a `T`. So what happens when we deconstruct a `MkT` value?

Example: Pattern matching on our existential constructor

```
foo (MkT x) = ... -- what is the type of x?
```



As we've just stated, `x` could be of any type. That means it's a member of some arbitrary type, so has the type `x :: exists a. a`. In other words, our declaration for `T` is isomorphic to the following one:

Example: An equivalent version of our existential datatype (pseudo-Haskell)

```
data T = MkT (exists a. a)
```



And suddenly we have existential types. Now we can make a heterogeneous list:

Example: Constructing the heterogeneous list

```
heteroList = [MkT 5, MkT (), MkT True, MkT map]
```



Of course, when we pattern match on `heteroList` we can't do anything with its elements^[25], as all we know is that they have some arbitrary type. However, if we are to introduce class constraints:

Example: A new existential datatype, with a class constraint

```
data T' = forall a. Show a => MkT' a
```



Which is isomorphic to:

Example: The new datatype, translated into 'true' existential types

```
data T' = MkT' (exists a. Show a => a)
```



Again the class constraint serves to limit the types we're unioning over, so that now we know the values inside a `MkT'` are elements of some arbitrary type *which instantiates Show*. The implication of this is that we can apply `show` to a value of type `exists a. Show a => a`. It doesn't matter exactly which type it turns out to be.

Example: Using our new heterogeneous setup

```
heteroList' = [MkT' 5, MkT' (), MkT' True]
main = mapM_ (\(MkT' x) -> print x) heteroList'

{- prints:
5
()
True
-}
```



To summarise, the interaction of the universal quantifier with datatypes produces existential types. As most interesting applications of `forall`-involving types use this interaction, we label such types 'existential'.

Example: `runST`

One monad that you haven't come across so far is the `ST` monad. This is essentially the `State` monad on steroids: it has a much more complicated structure and involves some more advanced topics. It was originally written to provide Haskell with IO. As we mentioned in the Understanding monads chapter, IO is basically just a `State` monad with an environment of all the information about the real world. In fact, inside GHC at least, `ST` is used, and the environment is a type called `RealWorld`.

To get out of the `State` monad, you can use `runState`. The analogous function for `ST` is called `runST`, and it has a rather particular type:

Example: The `runST` function

```
runST :: forall a. (forall s. ST s a) -> a
```



This is actually an example of a more complicated language feature called rank-2 polymorphism, which we don't go into detail here. It's important to notice that there is no parameter for the initial state. Indeed, `ST` uses a different notion of state to `State`; while `State` allows you to `get` and `put` the current state, `ST` provides an interface to *references*. You create references, which have type `STRef`, with `newSTRef :: a -> ST s (STRef s a)`, providing an initial value, then you can use `readSTRef :: STRef s a -> ST s a` and `writeSTRef :: STRef s a -> a -> ST s ()` to manipulate them. As such, the internal environment of a `ST` computation is not one specific value, but a mapping from references to values. Therefore, you don't need to provide an initial state to `runST`, as the initial state is just the empty mapping containing no references.

However, things aren't quite as simple as this. What stops you creating a reference in one `ST` computation, then using it in another? We don't want to allow this because (for reasons of thread-safety) no `ST` computation should be allowed to assume that the initial internal environment contains any specific references. More concretely, we want the following code to be invalid:

Example: Bad `ST` code

```
let v = runST (newSTRef True)
in runST (readSTRef v)
```



What would prevent this? The effect of the rank-2 polymorphism in `runST`'s type is to *constrain the scope of the type variable `s`* to be within the first parameter. In other words, if the type variable `s` appears in the first parameter it cannot also appear in the second. Let's take a look at how exactly this is done. Say we have some code like the following:

Example: Briefer bad ST code

```
... runST (newSTRef True) ...
```

The compiler tries to fit the types together:

Example: The compiler's typechecking stage

```
newSTRef True :: forall s. ST s (STRef s Bool)
runST :: forall a. (forall s. ST s a) -> a
together, forall a. (forall s. ST s (STRef s Bool)) -> STRef s Bool
```

The importance of the `forall` in the first bracket is that we can change the name of the `s`. That is, we could write:

Example: A type mismatch!

```
together, forall a. (forall s'. ST s' (STRef s' Bool)) -> STRef s Bool
```

This makes sense: in mathematics, saying $\forall x.x > 5$ is precisely the same as saying $\forall y.y > 5$; you're just giving the variable a different label. However, we have a problem with our above code. Notice that as the `forall` does *not* scope over the return type of `runST`, we don't rename the `s` there as well. But suddenly, we've got a type mismatch! The result type of the ST computation in the first parameter must match the result type of `runST`, but now it doesn't!

The key feature of the existential is that it allows the compiler to generalise the type of the state in the first parameter, and so the result type cannot depend on it. This neatly sidesteps our dependence problems, and 'compartmentalises' each call to `runST` into its own little heap, with references not being able to be shared between different calls.

Further reading

- GHC's user guide contains useful information (http://haskell.org/ghc/docs/latest/html/users_guide/type-extensions.html#existential-quantification) on existentials, including the various limitations placed on them (which you should know about).
- *Lazy Functional State Threads* (<http://citeseer.ist.psu.edu/launchbury94lazy.html>), by Simon Peyton-Jones and John Launchbury, is a paper which explains more fully the ideas behind ST.

Polymorphism

Terms depending on types

Ad-hoc and parametric polymorphism

Polymorphism in Haskell

Higher-rank polymorphism

Advanced type classes

Type classes may seem innocuous, but research on the subject has resulted in several advancements and generalisations which make them a very powerful tool.

Multi-parameter type classes

Multi-parameter type classes are a generalisation of the single parameter type classes, and are supported by some Haskell implementations.

Suppose we wanted to create a 'Collection' type class that could be used with a variety of concrete data types, and supports two operations -- 'insert' for adding elements, and 'member' for testing membership. A first attempt might look like this:

```
-----  
class Collection c where  
    insert :: c -> e -> c  
    member :: c -> e -> Bool  
-- Make lists an instance of Collection:  
instance Collection [a] where  
    insert xs x = x:xs  
    member = flip elem  
-----
```

This won't compile, however. The problem is that the 'e' type variable in the Collection operations comes from nowhere -- there is nothing in the type of an instance of Collection that will tell us what the 'e' actually is, so we can never define implementations of these methods. Multi-parameter type classes solve this by allowing us to put 'e' into the type of the class. Here is an example that compiles and can be used:

```
-----  
class Eq e => Collection c e where  
    insert :: c -> e -> c  
    member :: c -> e -> Bool  
instance Eq a => Collection [a] a where  
    insert = flip (:)  
    member = flip elem  
-----
```

Functional dependencies

A problem with the above example is that, in this case, we have extra information that the compiler doesn't know, which can lead to false ambiguities and over-generalised function signatures. In this case, we can see intuitively that the type of the collection will always determine the type of the element it contains - so if `c` is `[a]`, then `e` will be `a`. If `c` is `HashMap a`, then `e` will be `a`. (The reverse is not true: many different collection types can hold the same element type, so knowing the element type was e.g. `Int`, would not tell you the collection type).

In order to tell the compiler this information, we add a **functional dependency**, changing the class declaration to

```
class Eq e => Collection c e | c -> e where ...
```

A functional dependency is a constraint that we can place on type class parameters. Here, the extra `| c -> e` should be read '`c` uniquely identifies `e`', meaning for a given `c`, there will only be one `e`. You can have more than one functional dependency in a class -- for example you could have `c -> e`, `e -> c` in the above case. And you can have more than two parameters in multi-parameter classes.

Examples

Matrices and vectors

Suppose you want to implement some code to perform simple linear algebra:

```
data Vector = Vector Int Int deriving (Eq, Show)
data Matrix = Matrix Vector Vector deriving (Eq, Show)
```

You want these to behave as much like numbers as possible. So you might start by overloading Haskell's `Num` class:

```
instance Num Vector where
  Vector a1 b1 + Vector a2 b2 = Vector (a1+a2) (b1+b2)
  Vector a1 b1 - Vector a2 b2 = Vector (a1-a2) (b1-b2)
  {- ... and so on ... -}
instance Num Matrix where
  Matrix a1 b1 + Matrix a2 b2 = Matrix (a1+a2) (b1+b2)
  Matrix a1 b1 - Matrix a2 b2 = Matrix (a1-a2) (b1-b2)
  {- ... and so on ... -}
```

The problem comes when you want to start multiplying quantities. You really need a multiplication function which overloads to different types:

```
!(*) :: Matrix -> Matrix -> Matrix
!(*) :: Matrix -> Vector -> Vector
!(*) :: Matrix -> Int -> Matrix
!(*) :: Int -> Matrix -> Matrix
!{- ... and so on ... -}
```

How do we specify a type class which allows all these possibilities?

We could try this:

```
class Mult a b c where
  (*) :: a -> b -> c

instance Mult Matrix Matrix Matrix where
  {- ... -}

instance Mult Matrix Vector Vector where
  {- ... -}
```

That, however, isn't really what we want. As it stands, even a simple expression like this has an ambiguous type unless you supply an additional type declaration on the intermediate expression:

```
m1, m2, m3 :: Matrix
(m1 * m2) * m3      -- type error; type of (m1*m2) is ambiguous
(m1 * m2) :: Matrix * m3  -- this is ok
```

After all, nothing is stopping someone from coming along later and adding another instance:

```
instance Mult Matrix Matrix (Maybe Char) where
  {- whatever -}
```

The problem is that `c` shouldn't really be a free type variable. When you know the types of the things that you're multiplying, the result type should be determined from that information alone.

You can express this by specifying a functional dependency:

```
class Mult a b c | a b -> c where
  (*) :: a -> b -> c
```

This tells Haskell that `c` is uniquely determined from `a` and `b`.



At least part of this page was imported from the Haskell wiki article [Functional dependencies](http://www.haskell.org/haskellwiki/Functional_dependencies) (http://www.haskell.org/haskellwiki/Functional_dependencies), in accordance to its Simple Permissive License. If you wish to modify this page and if your changes will also be useful on that wiki, you might consider modifying that source page instead of this one, as changes from that page may propagate here, but not the other way around. Alternately, you can explicitly dual license your contributions under the Simple Permissive License.

Phantom types

Phantom types are a way to embed a language with a stronger type system than Haskell's. *FIXME: that's about*

all I know, and it's probably wrong. :) I'm yet to be convinced of PT's usefulness, I'm not sure they should have such a prominent position. DavidHouse 17:42, 1 July 2006 (UTC)

Phantom types

An ordinary type

```
data T = TI Int | TS String
plus :: T -> T -> T
concat :: T -> T -> T
```

its phantom type version

```
data T a = TI Int | TS String
```

Nothing's changed - just a new argument `a` that we don't touch. But magic!

```
plus :: T Int -> T Int -> T Int
concat :: T String -> T String -> T String
```

Now we can enforce a little bit more!

This is useful if you want to increase the type-safety of your code, but not impose additional runtime overhead:

```
-- Peano numbers at the type level.
data Zero = Zero
data Succ a = Succ a
-- Example: 3 can be modeled as the type
-- Succ (Succ (Succ Zero))

data Vector n a = Vector [a] deriving (Eq, Show)

vector2d :: Vector (Succ (Succ Zero)) Int
vector2d = Vector [1,2]

vector3d :: Vector (Succ (Succ (Succ Zero))) Int
vector3d = Vector [1,2,3]

-- vector2d == vector3d raises a type error
-- at compile-time, while vector2d == Vector [2,3] works.
```

GADT

Introduction

Explain what a GADT (Generalised Algebraic Datatype) is, and what it's for

GADT-style syntax

Before getting into GADT-proper, let's start out by getting used to the new syntax. Here is a representation for the familiar `List` type in both normal Haskell style and the new GADT one:

normal style	GADT style
<pre>data List x = Nil Cons x (List x)</pre>	<pre>data List x where Nil :: List x Cons :: x -> List x -> List x</pre>

Up to this point, we have not introduced any new capabilities, just a little new syntax. Strictly speaking, we are not working with GADTs yet, but GADT syntax. The new syntax should be very familiar to you in that it closely resembles typeclass declarations. It should also be easy to remember if you like to think of constructors as just being functions. Each constructor is just defined like a type signature for any old function.

What GADTs give us

Given a data type `Foo a`, a constructor for `Foo` is merely a function that takes some number of arguments and gives you back a `Foo a`. So what do GADTs add for us? The ability to control exactly what kind of `Foo` you return. With GADTs, a constructor for `Foo a` is not obliged to return `Foo a`; it can return any `Foo ???` that you can think of. In the code sample below, for instance, the `GadtedFoo` constructor returns a `GadtedFoo Int` even though it is for the type `GadtedFoo x`.

Example: GADT gives you more control

```
data BoringFoo x where
  MkBoringFoo :: x -> BoringFoo x

data GadtedFoo x where
  MkGadtedFoo :: x -> GadtedFoo Int
```



But note that you can only push the idea so far... if your datatype is a `Foo`, you *must* return some kind of `Foo` or another. Returning anything else simply wouldn't work

Example: Try this out. It doesn't work

```
data Bar where
  MkBar :: Bar -- This is ok

data Foo where
  MkFoo :: Bar -- This is bad
```



Safe Lists

Prerequisite: *We assume in this section that you know how a List tends to be represented in functional languages*

We've now gotten a glimpse of the extra control given to us by the GADT syntax. The only thing new is that you can control exactly what kind of data structure you return. Now, what can we use it for? Consider the humble Haskell list. What happens when you invoke `head []`? Haskell blows up. Have you ever wished you could have a magical version of `head` that only accepts lists with at least one element, lists on which it will never blow up?

To begin with, let's define a new type, `SafeList x y`. The idea is to have something similar to normal Haskell lists `[x]`, but with a little extra information in the type. This extra information (the type variable `y`) tells us whether or not the list is empty. Empty lists are represented as `SafeList x Empty`, whereas non-empty lists are represented as `SafeList x NonEmpty`.

```

-----
-- we have to define these types
data Empty
data NonEmpty
|
-- the idea is that you can have either
--   SafeList x Empty
-- or SafeList x NonEmpty
data SafeList x y where
-- to be implemented
-----

```

Since we have this extra information, we can now define a function `safeHead` on only the non-empty lists! Calling `safeHead` on an empty list would simply refuse to type-check.

```

-----
safeHead :: SafeList x NonEmpty -> x
-----

```

So now that we know what we want, `safeHead`, how do we actually go about getting it? The answer is GADT. The key is that we take advantage of the GADT feature to return two different kinds of lists, `SafeList x Empty` for the `Nil` constructor, and `SafeList x NonEmpty` for the `Cons` constructors respectively:

```

-----
data SafeList x y where
  Nil  :: SafeList x Empty
  Cons :: x -> SafeList x y -> SafeList x NonEmpty
-----

```

This wouldn't have been possible without GADT, because all of our constructors would have been required to return the same type of list; whereas with GADT we can now return different types of lists with different constructors. Anyway, let's put this altogether, along with the actual definition of `SafeList`:

Example: safe lists via GADT

```

-----
data Empty
data NonEmpty
|
data SafeList x y where
  Nil  :: SafeList x Empty
  Cons :: x -> SafeList x y -> SafeList x NonEmpty
|
safeHead :: SafeList x NonEmpty -> x
safeHead (Cons x _) = x
-----

```



Copy this listing into a file and load in `ghci -fglasgow-exts`. You should notice the following difference, calling `safeHead` on a non-empty and an empty-list respectively:

Example: `safeHead` is... safe

```

Prelude Main> safeHead (Cons "hi" Nil)
"hi"
Prelude Main> safeHead Nil
<interactive>:1:9:
  Couldn't match `NonEmpty' against `Empty'
    Expected type: SafeList a NonEmpty
    Inferred type: SafeList a Empty
    In the first argument of `safeHead', namely `Nil'
    In the definition of `it': it = safeHead Nil

```



This complaint is a good thing: it means that we can now ensure during compile-time if we're calling `safeHead` on an appropriate list. However, this is a potential pitfall that you'll want to look out for.

Consider the following function. What do you think its type is?

Example: Trouble with GADTs

```

silly 0 = Nil
silly 1 = Cons 1 Nil

```



Now try loading the example up in GHCi. You'll notice the following complaint:

Example: Trouble with GADTs - the complaint

```

Couldn't match `Empty' against `NonEmpty'
  Expected type: SafeList a Empty
  Inferred type: SafeList a NonEmpty
  In the application `Cons 1 Nil'
  In the definition of `silly': silly 1 = Cons 1 Nil

```



FIXME: insert discussion

Exercises

1. Could you implement a `safeTail` function?

A simple expression evaluator

Insert the example used in Wobbly Types paper... I thought that was quite pedagogical

Discussion

More examples, thoughts

From FOSDEM 2006, I vaguely recall that there is some relationship between GADT and the below... what?

Phantom types

Existential types

If you like Existentially quantified types, you'd probably want to notice that they are now subsumed by GADTs. As the GHC manual says, the following two type declarations give you the same thing.

```
data TE a = forall b. MkTE b (b->a)
data TG a where { MkTG :: b -> (b->a) -> TG a }
```

Heterogeneous lists are accomplished with GADTs like this:

```
data TE2 = forall b. Show b => MkTE2 [b]
data TG2 where
  MkTG2 :: Show b => [b] -> TG2
```

Witness types

References



At least part of this page was imported from the Haskell wiki article Generalised algebraic datatype (http://www.haskell.org/haskellwiki/Generalised_algebraic_datatype), in accordance to its Simple Permissive License. If you wish to modify this page and if your changes will also be useful on that wiki, you might consider modifying that source page instead of this one, as changes from that page may propagate here, but not the other way around. Alternately, you can explicitly dual license your contributions under the Simple Permissive License.

Wider Theory

Denotational semantics



New readers: Please report stumbling blocks! While the material on this page is intended to explain clearly, there are always mental traps that innocent readers new to the subject fall in but that the authors are not aware of. Please report any tricky passages to the Talk page or the #haskell IRC channel so that the style of exposition can be improved.

Introduction

This chapter explains how to formalize the meaning of Haskell programs, the **denotational semantics**. It may seem to be nit-picking to formally specify that the program `square x = x*x` means the same as the mathematical square function that maps each number to its square, but what about the meaning of a program like `f x = f (x+1)` that loops forever? In the following, we will exemplify the approach first taken by Scott and Strachey to this question and obtain a foundation to reason about the correctness of functional programs in general and recursive definitions in particular. Of course, we will concentrate on those topics needed to understand Haskell programs^[26].

Another aim of this chapter is to illustrate the notions **strict** and **lazy** that capture the idea that a function needs or needs not to evaluate its argument. This is a basic ingredient to predict the course of evaluation of Haskell programs and hence of primary interest to the programmer. Interestingly, these notions can be formulated consisely with denotational semantics alone, no reference to an execution model is necessary. They will be put to good use in Graph Reduction, but it is this chapter that will familiarize the reader with the denotational definition and involved notions such as \perp ("Bottom"). The reader only interested in strictness may wish to poke around in section Bottom and Partial Functions and quickly head over to Strict and Non-Strict Semantics.

What are Denotational Semantics and what are they for?

What does a Haskell program mean? This question is answered by the **denotational semantics** of Haskell. In general, the denotational semantics of a programming language map each of its programs to a mathematical object, the *meaning* of the program in question. As an example, the mathematical object for the Haskell programs `10`, `9+1`, `2*5` and `sum [1..4]` is likely to be the integer *10*. We say that all those programs **denote** the integer *10*. The collection of mathematical objects is called the **semantic domain**.

The mapping from program codes to a semantic domain is commonly written down with double square brackets (*Wikibooks doesn't seem to support \llbrackets in math formulas...*) as

$$\llbracket 2*5 \rrbracket = 10$$

It is *compositional*, i.e. the meaning of a program like `1+9` only depends on the meaning of its constituents:

$$\llbracket a+b \rrbracket = \llbracket a \rrbracket + \llbracket b \rrbracket$$

The same notation is used for types, i.e.

$$[[\text{Integer}]] = \mathbb{Z}$$

For simplicity however, we will silently identify expressions with their semantic objects in subsequent chapters and use this notation only when clarification is needed.

It is one of the key properties of *purely functional* languages like Haskell that a direct mathematical interpretation like " $1+9$ denotes 10 " carries over to functions, too: in essence, the denotation of a program of type `Integer -> Integer` is a mathematical function $\mathbb{Z} \rightarrow \mathbb{Z}$ between integers. While we will see that this needs refinement to include non-termination, the situation for *imperative languages* is clearly worse: a procedure with that type denotes something that changes the state of a machine in possibly unintended ways. Imperative languages are tied tightly to an **operational semantics** which describes how they are executed on a machine. It is possible to define a denotational semantics for imperative programs and to use it to reason about such programs, but the semantics often has an operational nature and sometimes must extend on the denotational semantics for functional languages.^[27] In contrast, the meaning of purely functional languages is *by default* completely independent from their execution. The Haskell98 standard even goes as far as to only specify Haskell's non-strict denotational semantics and leaving open how to implement them.

In the end, denotational semantics enables us to develop formal proofs that programs indeed do what we want them to do mathematically. Ironically, for proving program properties in day-to-day Haskell, one can use Equational reasoning which transform programs into equivalent ones without seeing much of the underlying mathematical objects we are concentrating on in this chapter. But the denotational semantics actually show up whenever we have to reason about non-terminating programs, for instance in Infinite Lists.

Of course, because they only state what a program is, denotational semantics cannot answer questions about how long a program takes or how much memory it eats. This is governed by the *evaluation strategy* which dictates how the computer calculates the normal form of an expression. But on the other hand, the implementation has to respect the semantics and to a certain extent, they determine how Haskell programs must be evaluated on a machine. We will elaborate this in Strict and Non-Strict Semantics.

What to choose as Semantic Domain?

We are now looking for suitable mathematical objects that we can attribute to every Haskell program. In case of the example `10`, `2*5` and `sum [1..4]`, it is clear that all expressions should denote the integer 10 . Generalizing, every value `x` of type `Integer` is likely to be an element of the set \mathbb{Z} . The same can be done with values of type `Bool`. For functions like `f :: Integer -> Integer`, we can appeal to the mathematical definition of "function" as a set of (argument,value)-pairs, its *graph*.

But interpreting functions as their graph was too quick, because it does not work well with recursive definitions. Consider the definition

```
shaves :: Integer -> Integer -> Bool
1 `shaves` 1 = True
2 `shaves` 2 = False
0 `shaves` x = not (x `shaves` x)
_ `shaves` _ = False
```

We can think of `0`, `1` and `2` as being male persons with long beards and the question is who shaves whom. Person `1` shaves himself, but `2` gets shaved by the barber `0` because evaluating the third equation yields `0 `shaves` 2 == True`. In general, the third line says that the barber `0` shaves all persons that do not shave themselves.

What about the barber himself, is `0 `shaves` 0` true or not? If it is, then the third equation says that it is not. If

it is not, then the third equation says that it is. Puzzled, we see that we just cannot attribute `True` or `False` to `0`shaves`0`, the graph we use as interpretation for the function `shaves` must have a empty spot. We realize that our semantic objects must be able to incorporate **partial functions**, functions that are undefined for some arguments.

It is well known that this famous example gave rise to serious foundational problems in set theory. It's an example of an **impredicative** definition, a definition which uses itself, a logical circle. Unfortunately for recursive definitions, the circle is not the problem but the feature.

Bottom and Partial Functions

\perp Bottom

To handle partial functions, we introduce \perp , named **bottom** and commonly written `_|_` in typewriter font. We say that \perp is the completely "**undefined**" value or function. Every data type like `Integer`, `()` or `Integer -> Integer` contains one \perp besides their usual elements. So the possible values of type `Integer` are

$$\perp, 0, \pm 1, \pm 2, \pm 3, \dots$$

Adding \perp to the set of values is also called **lifting**. This is often depicted by a subscript like in \mathbb{Z}_{\perp} . While this is the correct notation for the mathematical set "lifted integers", we prefer to talk about "values of type `Integer`". We do this because \mathbb{Z}_{\perp} suggests that there are "real" integers \mathbb{Z} , but inside Haskell, the "integers" are `Integer`.

As another example, the type `()` with only one element actually has two inhabitants:

$$\perp, ()$$

For now, we will stick to programming with `Integers`. Arbitrary algebraic data types will be treated in section [Algebraic Data Types](#) as strict and non-strict languages diverge on how these include \perp .

In Haskell, the expression `undefined` denotes \perp . With its help, one can indeed verify some semantic properties of actual Haskell programs. `undefined` has the polymorphic type `forall a . a` which of course can be specialized to `undefined :: Integer`, `undefined :: ()`, `undefined :: Integer -> Integer` and so on. In the Haskell Prelude, it is defined as

```
undefined = error "Prelude: undefined"
```

As a side note, it follows from the Curry-Howard isomorphism that any value of the polymorphic type `forall a . a` must denote \perp .

Partial Functions and the Semantic Approximation Order

Now, \perp gives us the possibility to denote partial functions:

$$f(n) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ -2 & \text{if } n \text{ is } 1 \\ \perp & \text{else} \end{cases}$$

Here, $f(n)$ yields well defined values for $n = 0$ and $n = 1$ but gives \perp for all other n . Note that the notation \perp is overloaded: the function $\perp :: \text{Integer} \rightarrow \text{Integer}$ is given by

$$\perp (n) = \perp \text{ for all } n$$

where the \perp on the right hand side denotes a value of type `Integer`.

To formalize, **partial functions** say of type `Integer -> Integer` are at least mathematical mappings from the lifted integers $\mathbb{Z}_\perp = \{\perp, 0, \pm 1, \pm 2, \pm 3, \dots\}$ to the lifted integers. But this is not enough, it does not merit the special role of \perp . For example, the definition

$$g(n) = \begin{cases} 1 & \text{if } n \text{ is } \perp \\ \perp & \text{else} \end{cases}$$

intuitively does not make sense. Why does $g(\perp)$ yield a defined value whereas $g(1)$ is undefined? The intuition is that every partial function g should yield more defined answers for more defined arguments. To formalize, we can say that every concrete number is **more defined** than \perp :

$$\perp \sqsubset 1, \perp \sqsubset 2, \dots$$

Here, $a \sqsubset b$ denotes that b is more defined than a . Likewise, $a \sqsubseteq b$ will denote that either b is more defined than a or both are equal (and so have the same definedness). \sqsubseteq is also called the **semantic approximation order** because we can approximate defined values by less defined ones thus interpreting "more defined" as "approximating better". Of course, \perp is designed to be the least element of a data type, we always have

$$\perp \sqsubseteq x \text{ for all other } x.$$

As no number is *more defined* than another, the mathematical relation \sqsubseteq does not relate different numbers:

$$\text{neither } 1 \sqsubseteq 2 \text{ nor } 2 \sqsubseteq 1 \text{ hold.}$$

This is contrasted to the ordinary order \leq between integers which can compare any two numbers. That's also why we use the different symbol \sqsubseteq . A quick way to remember this is the sentence: "1 and 2 are different in *information content* but the same in *information quantity*".

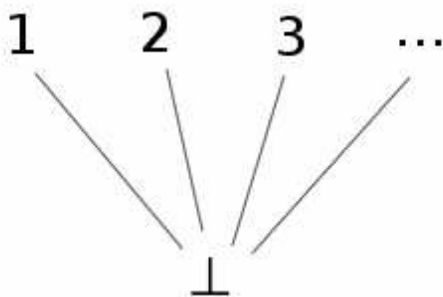
One says that \sqsubseteq specifies a **partial order** and that the values of type `Integer` form a **partially ordered set** (**poset** for short). A partial order is characterized by the following three laws

- *Reflexivity*, everything is just as defined as itself: $x \sqsubseteq x$ for all x
- *Transitivity*: if $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$
- *Antisymmetry*: if both $x \sqsubseteq y$ and $y \sqsubseteq x$ hold, then x and y must be equal: $x = y$.

Exercises

Do the integers form a poset with respect to the order \leq ?

We can depict the order \sqsubseteq on the values of type `Integer` by the following graph



where every link between two nodes specifies that the one above is more defined than the one below. Because there is only one level (excluding \perp), one says that `Integer` is a *flat domain*. The picture also explains the name of \perp : it's called *bottom* because it always sits at the bottom.

Monotonicity

Our intuition about partial functions now can be formulated as following: every partial function f is a **monotone** mapping between partially ordered sets. More defined arguments will yield more defined values:

$$x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$$

In particular, a function h with $h(\perp) = \perp$ must be constant: $h(n) = \perp$ for all n . Note that here it is crucial that $\perp \sqsubseteq 2$ etc. don't hold.

Translated to Haskell, monotonicity means that we cannot pattern match on \perp or its equivalent `undefined`. Otherwise, the example g from above could be expressed as a Haskell program. As we shall see later, \perp also denotes non-terminating programs, so that the inability to observe \perp inside Haskell is related to the halting problem.

Of course, the notion of *more defined than* can be extended to partial functions by saying that a function is more defined than another if it is so at every possible argument:

$$f \sqsubseteq g \text{ if } \forall x. f(x) \sqsubseteq g(x)$$

Thus, the partial functions also form a poset with the undefined function $\lambda x. \perp$ being the least element.

Recursive Definitions as Fixed Point Iterations

Approximations of the Factorial Function

Now that we have a means to describe partial functions, we can give an interpretation to recursive definitions. Lets take the prominent example of the factorial function $f(n) = n!$ whose recursive definition is

$$f(n) = \text{if } n == 0 \text{ then } 1 \text{ else } n \cdot f(n - 1)$$

Although we saw that interpreting this directly as a set description leads to problems, we intuitively know how to calculate $f(n)$ for every given n by iterating the right hand side. This iteration can be formalized as follows: we calculate a sequence of functions f_k with the property that each one arises from the right hand side applied to the previous one, that is

$$f_{k+1}(n) = \text{if } n == 0 \text{ then } 1 \text{ else } n \cdot f_k(n - 1)$$

Starting with the undefined function $f_0(n) = \perp$, the resulting sequence of partial functions reads

$$f_1(n) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ \perp & \text{else} \end{cases}, f_2(n) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ 1 & \text{if } n \text{ is } 1 \\ \perp & \text{else} \end{cases}, f_3(n) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ 1 & \text{if } n \text{ is } 1 \\ 2 & \text{if } n \text{ is } 2 \\ \perp & \text{else} \end{cases}$$

and so on. Clearly,

$$\perp = f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \dots$$

and we expect that the sequence converges to the factorial function.

The iteration follows the well known scheme of a fixed point iteration

$$x_0, g(x_0), g(g(x_0)), g(g(g(x_0))), \dots$$

In our case, x_0 is a function and g is a *functional*, a mapping between functions. We have

$$x_0 = \perp \text{ and}$$

$$g(x) = n \mapsto \text{if } n == 0 \text{ then } 1 \text{ else } n * x(n - 1)$$

Now, since g is monotone, and $x_0 = \perp$, the iteration sequence is monotone:

$$x_0 \sqsubseteq g(x_0) \sqsubseteq g(g(x_0)) \sqsubseteq g(g(g(x_0))) \sqsubseteq \dots$$

(The proof is roughly as follows: since $x_0 = \perp$, and $\perp \sqsubseteq$ anything, $x_0 \sqsubseteq g(x_0)$. Since g is monotone, we can successively apply g to both sides of this relation, yielding $g(x_0) \sqsubseteq g(g(x_0))$, $g(g(x_0)) \sqsubseteq g(g(g(x_0)))$, and so on.)

So each successive application of g , starting with x_0 , transforms a less defined function to a more defined one.

It is very illustrative to formulate this iteration scheme in Haskell. As functionals are just ordinary higher order functions, we have

```

-----
g :: (Integer -> Integer) -> (Integer -> Integer)
g x = \n -> if n == 0 then 1 else n * x (n-1)
-----
x0 :: Integer -> Integer
x0 = undefined
-----
(f0:f1:f2:f3:f4:fs) = iterate g x0
-----

```

We can now evaluate the functions f_0, f_1, \dots at sample arguments and see whether they yield `undefined` or not:

```

> f3 0
1
> f3 1
1
> f3 2
2
> f3 5
*** Exception: Prelude.undefined
> map f3 [0..]
[1,1,2,*** Exception: Prelude.undefined
> map f4 [0..]
[1,1,2,6,*** Exception: Prelude.undefined
> map f1 [0..]
[1,*** Exception: Prelude.undefined

```

Of course, we cannot use this to check whether `f4` is really undefined for all arguments.

Convergence

To the mathematician, the question whether this sequence of approximations converges is still to be answered. For that, we say that a poset is a **directed complete partial order (dcpo)** iff every monotone sequence $x_0 \sqsubseteq x_1 \sqsubseteq \dots$ (also called *chain*) has a least upper bound (supremum)

$$\sup_{\sqsubseteq} \{x_0 \sqsubseteq x_1 \sqsubseteq \dots\} = x.$$

If that's the case for the semantic approximation order, we clearly can be sure that monotone sequence of functions approximating the factorial function indeed has a limit. For our denotational semantics, we will only meet dcpo's which have a least element \perp which are called **complete partial orders (cpo)**.

The `Integers` clearly form a (d)cpo, because the monotone sequences consisting of more than one element must be of the form

$$\perp \sqsubseteq \dots \sqsubseteq \perp \sqsubseteq n \sqsubseteq n \sqsubseteq \dots \sqsubseteq n$$

where n is an ordinary number. Thus, n is already the least upper bound.

For functions `Integer -> Integer`, this argument fails because monotone sequences may be of infinite length. But because `Integer` is a (d)cpo, we know that for every point n , there is a least upper bound

$$\sup_{\sqsubseteq} \{\perp = f_0(n) \sqsubseteq f_1(n) \sqsubseteq f_2(n) \sqsubseteq \dots\} =: f(n).$$

As the semantic approximation order is defined point-wise, the function f is the supremum we looked for.

These have been the last touches for our aim to transform the impredicative definition of the factorial function into a well defined construction. Of course, it remains to be shown that $f(n)$ actually yields a defined value for every n , but this is not hard and far more reasonable than a completely ill-formed definition.

Bottom includes Non-Termination

It is instructive to try our newly gained insight into recursive definitions on an example that does not terminate:

$$f(n) = f(n + 1)$$

The approximating sequence reads

$$f_0 = \perp, f_1 = \perp, \dots$$

and consists only of \perp . Clearly, the resulting limit is \perp again. From an operational point of view, a machine executing this program will loop indefinitely. We thus see that \perp may also denote a **non-terminating** function or value. Hence, given the halting problem, pattern matching on \perp in Haskell is impossible.

Interpretation as Least Fixed Point

Earlier, we called the approximating sequence an example of the well known "fixed point iteration" scheme. And of course, the definition of the factorial function f can also be thought as the specification of a fixed point of the functional g :

$$f = g(f) = n \mapsto \text{if } n == 0 \text{ then } 1 \text{ else } n \cdot f(n - 1)$$

However, there might be multiple fixed points. For instance, there are several f which fulfill the specification

$$f = n \mapsto \text{if } n == 0 \text{ then } 1 \text{ else } f(n + 1),$$

Of course, when executing such a program, the machine will loop forever on $f(1)$ or $f(2)$ and thus not produce any valuable information about the value of $f(1)$. This corresponds to choosing the *least defined* fixed point as semantic object f and this is indeed a canonical choice. Thus, we say that

$$f = g(f),$$

defines the **least fixed point** f of g . Clearly, *least* is with respect to our semantic approximation order \sqsubseteq .

The existence of a least fixed point is guaranteed by our iterative construction if we add the condition that g must be **continuous** (sometimes also called "chain continuous"). That simply means that g respects suprema of monotone sequences:

$$\sup_{\sqsubseteq} \{g(x_0) \sqsubseteq g(x_1) \sqsubseteq \dots\} = g \left(\sup_{\sqsubseteq} \{x_0 \sqsubseteq x_1 \sqsubseteq \dots\} \right)$$

We can then argue that with

$$f = \sup_{\sqsubseteq} \{x_0 \sqsubseteq g(x_0) \sqsubseteq g(g(x_0)) \sqsubseteq \dots\}$$

, we have

$$\begin{aligned} g(f) &= g \left(\sup_{\sqsubseteq} \{x_0 \sqsubseteq g(x_0) \sqsubseteq g(g(x_0)) \sqsubseteq \dots\} \right) \\ &= \sup_{\sqsubseteq} \{g(x_0) \sqsubseteq g(g(x_0)) \sqsubseteq \dots\} \\ &= \sup_{\sqsubseteq} \{x_0 \sqsubseteq g(x_0) \sqsubseteq g(g(x_0)) \sqsubseteq \dots\} \\ &= f \end{aligned}$$

and the iteration limit is indeed a fixed point of g . You may also want to convince yourself that the fixed point iteration yields the *least* fixed point possible.

Exercises

Prove that the fixed point obtained by fixed point iteration starting with $x_0 = \perp$ is also the least one, that it is smaller than any other fixed point. (Hint: \perp is the least element of our cpo and g is monotone)

By the way, how do we know that each Haskell function we write down indeed is continuous? Just as with monotonicity, this has to be enforced by the programming language. Admittedly, these properties can somewhat be enforced or broken at will, so the question feels a bit void. But intuitively, monotonicity is guaranteed by not allowing pattern matches on \perp . For continuity, we note that for an arbitrary type a , every simple function $a \rightarrow \text{Integer}$ is automatically continuous because the monotone sequences of `Integer`'s are of finite length. Any infinite chain of values of type a gets mapped to a finite chain of `Integer`s and respect for suprema becomes a consequence of monotonicity. Thus, all functions of the special case $\text{Integer} \rightarrow \text{Integer}$ must be continuous. For functionals like $g :: (\text{Integer} \rightarrow \text{Integer}) \rightarrow (\text{Integer} \rightarrow \text{Integer})$, the continuity then materializes due to currying, as the type is isomorphic to $:: ((\text{Integer} \rightarrow \text{Integer}), \text{Integer}) \rightarrow \text{Integer}$ and we can take $a = ((\text{Integer} \rightarrow \text{Integer}), \text{Integer})$.

In Haskell, the fixed interpretation of the factorial function can be coded as

```
factorial = fix g
```

with the help of the fixed point combinator

```
fix :: (a -> a) -> a.
```

We can define it by

```
fix f = let x = f x in x
```

which leaves us somewhat puzzled because when expanding *factorial*, the result is not anything different from how we would have defined the factorial function in Haskell in the first place. But of course, the construction this whole section was about is not at all present when running a real Haskell program. It's just a means to put the mathematical interpretation a Haskell programs to a firm ground. Yet it is very nice that we can explore these semantics in Haskell itself with the help of `undefined`.

Strict and Non-Strict Semantics

After having elaborated on the denotational semantics of Haskell programs, we will drop the mathematical function notation $f(n)$ for semantic objects in favor of their now equivalent Haskell notation $f\ n$.

Strict Functions

A function f with one argument is called **strict**, if and only if

$$f\ \perp = \perp.$$

Here are some examples of strict functions

```

id     x = x
succ   x = x + 1
power2 0 = 1
power2 n = 2 * power2 (n-1)

```

and there is nothing unexpected about them. But why are they strict? It is instructive to prove that these functions are indeed strict. For `id`, this follows from the definition. For `succ`, we have to ponder whether $\perp + 1$ is \perp or not. If it was not, then we should for example have $\perp + 1 = 2$ or more general $\perp + 1 = k$ for some concrete number k . We remember that every function is *monotone*, so we should have for example

$$2 = \perp + 1 \sqsubseteq 4 + 1 = 5$$

as $\perp \sqsubseteq 4$. But neither of $2 \sqsubseteq 5$, $2 = 5$ nor $2 \sqsupseteq 5$ is valid so that k cannot be 2. In general, we obtain the contradiction

$$k = \perp + 1 \sqsubseteq k + 1 = k + 1.$$

and thus the only possible choice is

$$\text{succ } \perp = \perp + 1 = \perp$$

and `succ` is strict.

Exercises

Prove that `power2` is strict. While one can base the proof on the "obvious" fact that `power2 n` is 2^n , the latter is preferably proven using fixed point iteration.

Non-Strict and Strict Languages

Searching for **non-strict** functions, it happens that there is only one prototype of a non-strict function of type `Integer -> Integer`:

```

one x = 1

```

Its variants are `constk x = k` for every concrete number k . Why are these the only ones possible? Remember that `one n` has to be more defined than `one ⊥`. As `Integer` is a flat domain, both must be equal.

Why is `one` non-strict? To see that it is, we use a Haskell interpreter and try

```

> one (undefined :: Integer)
!

```

which is not \perp . This is reasonable as `one` completely ignores its argument. When interpreting \perp in an operational sense as "non-termination", one may say that the non-strictness of `one` means that it does not force its argument to be evaluated and therefore avoids the infinite loop when evaluating the argument \perp . But one might as well say that every function must evaluate its arguments before computing the result which means that `one ⊥` should be \perp , too. That is, if the program computing the argument does not halt, `one` should not halt as well.^[28] It shows

up that one can *choose freely* this or the other design for a functional programming language. One says that the language is *strict* or *non-strict* depending on whether functions are strict or non-strict by default. The choice for Haskell is non-strict. In contrast, the functional languages ML and LISP choose strict semantics.

Functions with several Arguments

The notion of strictness extends to functions with several variables. For example, a function f of two arguments is *strict in the second argument* if and only of

$$f\ x\ \perp = \perp$$

for every x . But for multiple arguments, mixed forms where the strictness depends on the given value of the other arguments, are much more common. An example is the conditional

```
cond b x y = if b then x else y
```

We see that it is strict in y depending on whether the test b is `True` or `False`:

```
cond True  \ y = \
cond False \ y = y
```

and likewise for x . Apparently, `cond` is certainly \perp if both x and y are, but not necessarily when at least one of them is defined. This behavior is called **joint strictness**.

Clearly, `cond` behaves like the if-then-else statement where it is crucial not to evaluate both the `then` and the `else` branches:

```
if null xs then 'a' else head xs
if n == 0 then 1 else 5 / n
```

Here, the `else` part is \perp when the condition is met. Thus, in a non-strict language, we have the possibility to wrap primitive control statements such as if-then-else into functions like `cond`. This way, we can define our own control operators. In a strict language, this is not possible as both branches will be evaluated when calling `cond` which makes it rather useless. This is a glimpse of the general observation that non-strictness offers more flexibility for code reuse than strictness. See the chapter Laziness^[29] for more on this subject.

Not all Functions in Strict Languages are Strict

It is important to note that even in a strict language, not all functions are strict. The choice whether to have strictness and non-strictness by default only applies to certain argument data types. Argument types that solely contain data like `Integer`, `(Bool, Integer)` or `Either String [Integer]` impose strictness, but functions are not necessarily strict in *function types* like `Integer -> Bool`. Thus, in a hypothetical strict language with Haskell-like syntax, we would have the interpreter session


```

!> let const1 _ = 1
!> const1 (undefined :: Integer)
!!! Exception: Prelude.undefined
!> const1 (undefined :: Integer -> Bool)
!

```

Why are strict languages not strict in arguments of function type? If they were, fixed point iteration would crumble to dust! Remember the fixed point iteration

$$\perp \sqsubseteq g(\perp) \sqsubseteq g(g(\perp)) \dots$$

for a functional $g :: (\text{Integer} \rightarrow \text{Integer}) \rightarrow (\text{Integer} \rightarrow \text{Integer})$. If g would be strict, the sequence would read

$$\perp \sqsubseteq \perp \sqsubseteq \perp \sqsubseteq \dots$$

which obviously converges to a useless \perp . It is crucial that g makes the argument function more defined. This means that g must not be strict in its argument to yield a useful fixed point.

As a side note, the fact that things must be non-strict in function types can be used to recover some non-strict behavior in strict languages. One simply replaces a data type like `Integer` with `() -> Integer` where `()` denotes the well known singleton type. It is clear that every such function has the only possible argument `()` (besides \perp) and therefore corresponds to a single integer. But operations may be non-strict in arguments of type `() -> Integer`.

Exercises

It's tedious to lift every `Integer` to a `() -> Integer` for using non-strict behavior in strict languages. Can you write a function

```
lift :: Integer -> (() -> Integer)
```

that does this for us? Where is the trap?

Algebraic Data Types

After treating the motivation case of partial functions between `Integers`, we now want to extend the scope of denotational semantics to arbitrary algebraic data types in Haskell.

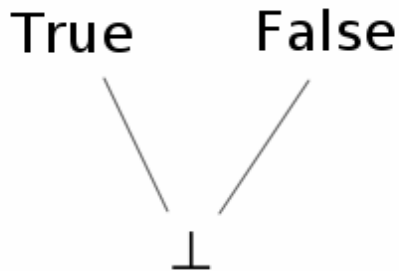
A word about nomenclature: the collection of semantic objects for a particular type is usually called a **domain**. This term is more a generic name than a particular definition and we decide that our domains are cpos (complete partial orders), that is sets of values together with a relation *more defined* that obeys some conditions to allow fixed point iteration. Usually, one adds additional conditions to the cpos that ensure that the values of our domains can be represented in some finite way on a computer and thereby avoiding to ponder the twisted ways of uncountable infinite sets. But as we are not going to prove general domain theoretic theorems, the conditions will just happen to hold by construction.

Constructors

Let's take the example types

```
data Bool    = True | False
data Maybe a = Just a | Nothing
```

Here, `True`, `False` and `Nothing` are nullary constructors whereas `Just` is a unary constructor. The inhabitants of `Bool` form the following domain:

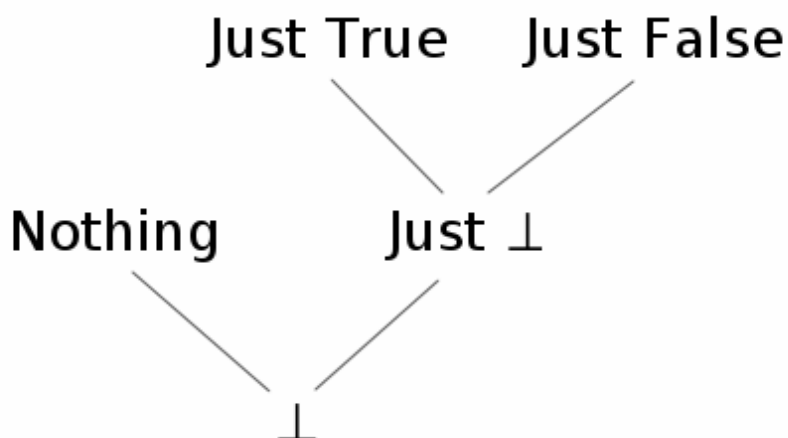


Remember that \perp is added as least element to the set of values `True` and `False`, we say that the type is **lifted**^[30]. A domain whose poset diagram consists of only one level is called a **flat domain**. We already know that *Integer* is a flat domain as well, it's just that the level above \perp has an infinite number of elements.

What are the possible inhabitants of `Maybe Bool`? They are

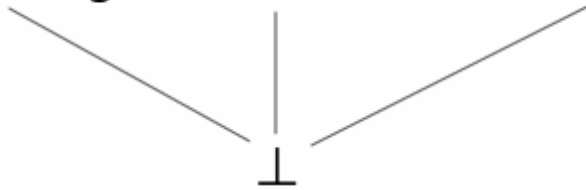
```
⊥, Nothing, Just ⊥, Just True, Just False
```

So the general rule is to insert all possible values into the unary (binary, ternary, ...) constructors as usual but without forgetting \perp . Concerning the partial order, we remember the condition that the constructors should be monotone just as any other functions. Hence, the partial order looks as follows



But there is something to ponder: why isn't `Just ⊥ = ⊥`? I mean "Just undefined" is as undefined as "undefined"! The answer is that this depends on whether the language is strict or non-strict. In a strict language, all constructors are strict by default, i.e. `Just ⊥ = ⊥` and the diagram would reduce to

Nothing Just True Just False



As a consequence, all domains of a strict language are flat.

But in a non-strict language like Haskell, constructors are non-strict by default and `Just ⊥` is a new element different from `⊥`, because we can write a function that reacts differently to them:

```
f (Just _) = 4
f Nothing  = 7
```

As `f` ignores the contents of the `Just` constructor, `f (Just ⊥)` is 4 but `f ⊥` is `⊥` (intuitively, if `f` is passed `⊥`, it will not be possible to tell whether to take the `Just` branch or the `Nothing` branch, and so `⊥` will be returned).

This gives rise to **non-flat domains** as depicted in the former graph. What should these be of use for? In the context of Graph Reduction, we may also think of `⊥` as an unevaluated expression. Thus, a value `x = Just ⊥` may tell us that a computation (say a lookup) succeeded and is not `Nothing`, but that the true value has not been evaluated yet. If we are only interested in whether `x` succeeded or not, this actually saves us from the unnecessary work to calculate whether `x` is `Just True` or `Just False` as would be the case in a flat domain. The full impact of non-flat domains will be explored in the chapter Laziness, but one prominent example are infinite lists treated in section Recursive Data Types and Infinite Lists.

Pattern Matching

In the section Strict Functions, we proved that some functions are strict by inspecting their results on different inputs and insisting on monotonicity. However, in the light of algebraic data types, there can only be one source of strictness in real life Haskell: pattern matching, i.e. `case` expressions. The general rule is that pattern matching on a constructor of a `data`-type will force the function to be strict, i.e. matching `⊥` against a constructor always gives `⊥`. For illustration, consider

```
const1 _ = 1
```

```
const1' True  = 1
const1' False = 1
```

The first function `const1` is non-strict whereas the `const1'` is strict because it decides whether the argument is `True` or `False` although its result doesn't depend on that. Pattern matching in function arguments is equivalent to `case`-expressions

```
const1' x = case x of
  True  -> 1
  False -> 1
```

which similarly impose strictness on `x`: if the argument to the `case` expression denotes `⊥` the while `case` will

denote \perp , too. However, the argument for case expressions may be more involved as in

```
foo k table = case lookup ("Foo." ++ k) table of
  Nothing -> ...
  Just x   -> ...
```

and it can be difficult to track what this means for the strictness of `foo`.

An example for multiple pattern matches in the equational style is the logical `or`:

```
or True _ = True
or _ True = True
or _ _    = False
```

Note that equations are matched from top to bottom. The first equation for `or` matches the first argument against `True`, so `or` is strict in its first argument. The same equation also tells us that `or True x` is non-strict in `x`. If the first argument is `False`, then the second will be matched against `True` and `or False x` is strict in `x`. Note that while wildcards are a general sign of non-strictness, this depends on their position with respect to the pattern matches against constructors.

Exercises

1. Give an equivalent discussion for the logical `and`
2. Can the logical "excluded or" (`xOR`) be non-strict in one of its arguments if we know the other?

There is another form of pattern matching, namely **irrefutable patterns** marked with a tilde `~`. Their use is demonstrated by

```
f ~(Just x) = 1
f Nothing   = 2
```

An irrefutable pattern always succeeds (hence the name) resulting in `f \perp = 1`. But when changing the definition of `f` to

```
f ~(Just x) = x + 1
f Nothing   = 2      -- this line may as well be left away
```

we have

```
f  $\perp$            =  $\perp + 1 = \perp$ 
f (Just 1)     = 1 + 1 = 2
```

If the argument matches the pattern, `x` will be bound to the corresponding value. Otherwise, any variable like `x` will be bound to \perp .

By default, `let` and `where` bindings are non-strict, too:

```
foo key map = let Just x = lookup key map in ...
```

is equivalent to

```
foo key map = case (lookup key map) of ~(Just x) -> ...
```

Exercises

1. The Haskell language definition (<http://www.haskell.org/onlinereport/>) gives the detailed semantics of pattern matching (<http://www.haskell.org/onlinereport/exps.html#case-semantics>) and you should now be able to understand it. So go on and have a look!
2. Consider a function `or` of two `Boolean` arguments with the following properties:

```
or ⊥      ⊥      = ⊥
or True  ⊥      = True
or ⊥      True  = True

or False y      = y
or x False      = x
```

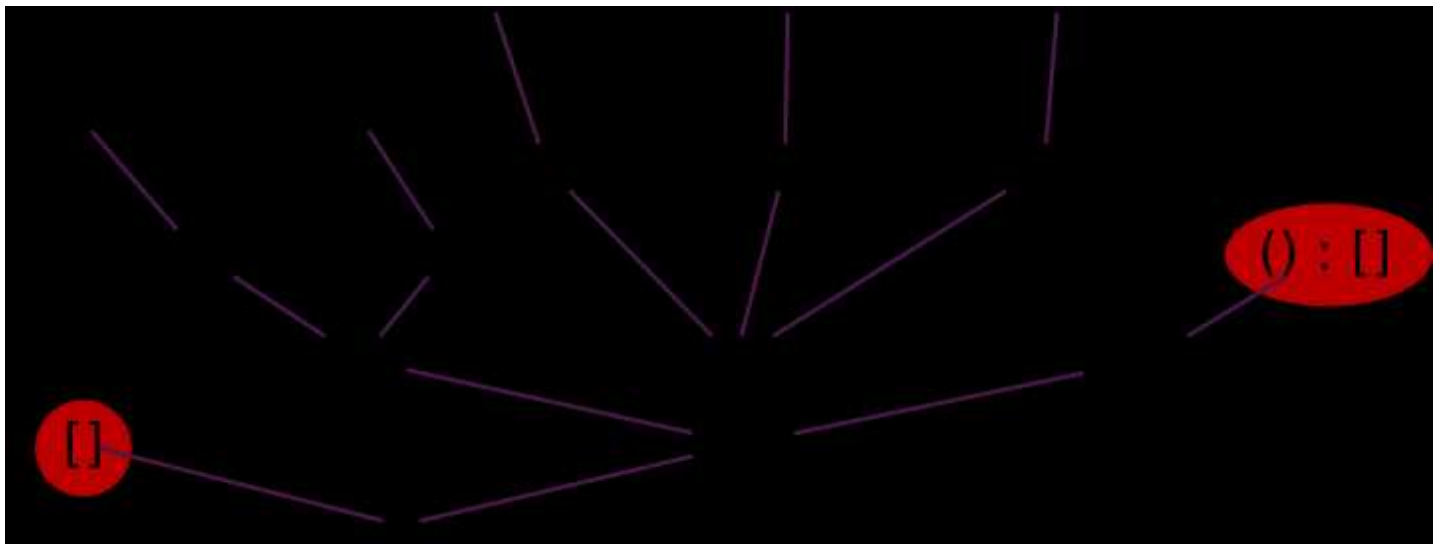
This function is another example of joint strictness, but a much sharper one: the result is only \perp if both arguments are (at least when we restrict the arguments to `True` and \perp). Can such a function be implemented in Haskell?

Recursive Data Types and Infinite Lists

The case of recursive data structures is not very different from the base case. Consider a list of unit values

```
data List = [] | () : List
```

Though this seems like a simple type, there is a surprisingly complicated number of ways you can fit \perp in here and there, and therefore the corresponding graph is complicated. The bottom of this graph is shown below. An ellipsis indicates that the graph continues along this direction. A red ellipse behind an element indicates that this is the end of a chain; the element is in normal form.



and so on. But now, there are also chains of infinite length like

$$\perp \sqsubseteq () : \perp \sqsubseteq () : () : \perp \sqsubseteq \dots$$

This causes us some trouble as we noted in section Convergence that every monotone sequence must have a least upper bound. This is only possible if we allow for **infinite lists**. Infinite lists (sometimes also called *streams*) turn out to be very useful and their manifold use cases are treated in full detail in chapter Laziness. Here, we will show what their denotational semantics should be and how to reason about them. Note that while the following discussion is restricted to lists only, it easily generalizes to arbitrary recursive data structures like trees.

In the following, we will switch back to the standard list type

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

to close the syntactic gap to practical programming with infinite lists in Haskell.

Exercises

1. Draw the non-flat domain corresponding `[Bool]`.
2. How is the graphic to be changed for `[Integer]`?

Calculating with infinite lists is best shown by example. For that, we need an infinite list

```
ones :: [Integer]
ones = 1 : ones
```

When applying the fixed point iteration to this recursive definition, we see that `ones` ought to be the supremum of

$$\perp \sqsubseteq 1 : \perp \sqsubseteq 1 : 1 : \perp \sqsubseteq 1 : 1 : 1 : \perp \sqsubseteq \dots,$$

that is an infinite list of 1. Let's try to understand what `take 2 ones` should be. With the definition of `take`

```
take 0 _      = []
take n (x:xs) = x : take (n-1) xs
take n []     = []
```

we can apply `take` to elements of the approximating sequence of `ones`:

```
take 2 ⊥      ==> ⊥
take 2 (1:⊥)  ==> 1 : take 1 ⊥      ==> 1 : ⊥
take 2 (1:1:⊥) ==> 1 : take 1 (1:⊥) ==> 1 : 1 : take 0 ⊥
              ==> 1 : 1 : []
```

We see that `take 2 (1:1:1:⊥)` and so on must be the same as `take 2 (1:1:⊥) = 1:1:[]` because `1:1:[]` is fully defined. Taking the supremum on both the sequence of input lists and the resulting sequence of output lists, we can conclude

```
take 2 ones = 1:1:[]
```

Thus, taking the first two elements of `ones` behaves exactly as expected.

Generalizing from the example, we see that reasoning about infinite lists involves considering the approximating sequence and passing to the supremum, the truly infinite list. Still, we did not give it a firm ground. The solution is to identify the infinite list with the whole chain itself and to formally add it as a new element to our domain: the infinite list *is* the sequence of its approximations. Of course, any infinite list like `ones` can compactly depicted as

```
ones = 1 : 1 : 1 : 1 : ...
```

what simply means that

```
ones = (⊥ ⊑ 1:⊥ ⊑ 1:1:⊥ ⊑ ...)
```

Exercises

- Of course, there are more interesting infinite lists than `ones`. Can you write recursive definition in Haskell for
 - the natural numbers `nats = 1:2:3:4:...`
 - a cycle like `cycle123 = 1:2:3: 1:2:3 : ...`
- Look at the Prelude functions `repeat` and `iterate` and try to solve the previous exercise with their help.
- Use the example from the text to find the value the expression `drop 3 nats` denotes.
- Assume that the work in a strict setting, i.e. that the domain of `[Integer]` is flat. What does the domain look like? What about infinite lists? What value does `ones` denote?

What about the puzzle of how a computer can calculate with infinite lists? It takes an infinite amount of time, after all? Well, this is true. But the trick is that the computer may well finish in a finite amount of time if it only

considers a finite part of the infinite list. So, infinite lists should be thought of as *potentially* infinite lists. In general, intermediate results take the form of infinite lists whereas the final value is finite. It is one of the benefits of denotational semantics that one treat the intermediate infinite data structures as truly infinite when reasoning about program correctness.

Exercises

1. To demonstrate the use of infinite lists as intermediate results, show that

```
take 2 (map (+1) nats) = take 3 nats
```

by first calculating the infinite sequence corresponding to `map (+1) nats`.

2. Of course, we should give an example where the final result indeed takes an infinite time. So, what does

```
filter (< 5) nats
```

denote?

3. Sometimes, one can replace `filter` with `takeWhile` in the previous exercise. Why only sometimes and what happens if one does?

As a last note, the construction of a recursive domain can be done by a fixed point iteration similar to recursive definition for functions. Yet, the problem of infinite chains has to be tackled explicitly. See the literature in External Links for a formal construction.

Haskell specialities: Strictness Annotations and Newtypes

Haskell offers a way to change the default non-strict behavior of data type constructors by *strictness annotations*. In a data declaration like

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

an exclamation point `!` before an argument of the constructor specifies that he should be strict in this argument. Hence we have `Just' ⊥ = ⊥` in our example. Further information may be found in chapter Strictness.

In some cases, one wants to rename a data type, like in

```
data Couldbe a = Couldbe (Maybe a)
```

However, `Couldbe a` contains both the elements `⊥` and `Couldbe ⊥`. With the help of a *newtype* definition

```
newtype Couldbe a = Couldbe (Maybe a)
```


we can arrange that `Couldbe a` is semantically equal to `Maybe a`, but different during type checking. In particular, the constructor `Couldbe` is strict. Yet, this definition is subtly different from

```
data Couldbe' a = Couldbe' !(Maybe a)
```

To explain how, consider the functions

```
f (Couldbe m) = 42
f' (Couldbe' m) = 42
```

Here, `f' ⊥` will cause the pattern match on the constructor `Couldbe'` fail with the effect that `f' ⊥ = ⊥`. But for the newtype, the match on `Couldbe` will never fail, we get `f ⊥ = 42`. In a sense, the difference can be stated as:

- for the strict case, `Couldbe' ⊥` is a synonym for `⊥`
- for the newtype, `⊥` is a synonym for `Couldbe ⊥`

with the agreement that a pattern match on `⊥` fails and that a match on `Constructor ⊥` does not.

Newtypes may also be used to define recursive types. An example is the alternate definition of the list type `[a]`

```
newtype List a = In (Maybe (a, List a))
```

Again, the point is that the constructor `In` does not introduce an additional lifting with `⊥`.

Other Selected Topics

Abstract Interpretation and Strictness Analysis

As lazy evaluation means a constant computational overhead, a Haskell compiler may want to discover where inherent non-strictness is not needed at all which allows it to drop the overhead at these particular places. To that extend, the compiler performs **strictness analysis** just like we proved in some functions to be strict section `Strict Functions`. Of course, details of strictness depending on the exact values of arguments like in our example `cond` are out of scope (this is in general undecidable). But the compiler may try to find approximate strictness information and this works in many common cases like `power2`.

Now, **abstract interpretation** is a formidable idea to reason about strictness: ...

For more about strictness analysis, see the research papers about strictness analysis on the Haskell wiki (http://haskell.org/haskellwiki/Research_papers/Compilation#Strictness).

Interpretation as Powersets

So far, we have introduced `⊥` and the semantic approximation order `⊑` abstractly by specifying their properties. However, both as well as any inhabitants of a data type like `Just ⊥` can be interpreted as ordinary sets. This is called the **powerset construction**. NOTE: *i'm not sure whether this is really true. Someone how knows, please*

correct this.

The idea is to think of \perp as the *set of all possible values* and that a computation retrieves more information this by choosing a subset. In a sense, the denotation of a value starts its life as the set of all values which will be reduced by computations until there remains a set with a single element only.

As an example, consider `Bool` where the domain looks like

```

{True}  {False}
  \    /
   \  /
     $\perp = \{True, False\}$ 

```

The values `True` and `False` are encoded as the singleton sets `{True}` and `{False}` and \perp is the set of all possible values.

Another example is `Maybe Bool`:

```

{Just True}  {Just False}
  \         /
   \       /
{Nothing}  {Just True, Just False}
  \       /
   \     /
     $\perp = \{Nothing, Just True, Just False\}$ 

```

We see that the semantic approximation order is equivalent to set inclusion, but with arguments switched:

$$x \sqsubseteq y \iff x \supseteq y$$

This approach can be used to give a semantics to exceptions in Haskell^[31].

Naïve Sets are unsuited for Recursive Data Types

In section [Naïve Sets are unsuited for Recursive Definitions](#), we argued that taking simple sets as denotation for types doesn't work well with partial functions. In the light of recursive data types, things become even worse as John C. Reynolds showed in his paper *Polymorphism is not set-theoretic*^[32].

Reynolds actually considers the recursive type

```

newtype U = In ((U -> Bool) -> Bool)

```

Interpreting `Bool` as the set `{True, False}` and the function type `A -> B` as the set of functions from `A` to `B`, the type `U` cannot denote a set. This is because `(A -> Bool)` is the set of subsets (powerset) of `A` which, due to a diagonal argument analogous to Cantor's argument that there are "more" real numbers than natural ones, always has a bigger cardinality than `A`. Thus, `(U -> Bool) -> Bool` has an even bigger cardinality than `U` and there is no way for it to be isomorphic to `U`. Hence, the set `U` must not exist, a contradiction.

In our world of partial functions, this argument fails. Here, an element of `U` is given by a sequence of approximations taken from the sequence of domains

$\perp, (\perp \rightarrow \text{Bool}) \rightarrow \text{Bool}, ((\perp \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \text{Bool}$ and so on

where \perp denotes the domain with the single inhabitant \perp . While the author of this text admittedly has no clue on what such a thing should mean, the constructor gives a perfectly well defined object for \perp . We see that the type $(\perp \rightarrow \text{Bool}) \rightarrow \text{Bool}$ merely consists of shifted approximating sequences which means that it is isomorphic to \perp .

As a last note, Reynolds actually constructs an equivalent of \perp in the second order polymorphic lambda calculus. There, it happens that all terms have a normal form, i.e. there are only total functions when we do not include a primitive recursion operator $\text{fix} :: (a \rightarrow a) \rightarrow a$. Thus, there is no true need for partial functions and \perp , yet a naïve set theoretic semantics fails. We can only speculate that this has to do with the fact that not every mathematical function is computable. In particular, the set of computable functions $A \rightarrow \text{Bool}$ should not have a bigger cardinality than A .

Footnotes

1. ↑ At least as far as types are concerned, but we're trying to avoid that word :)
2. ↑ More technically, `fst` and `snd` have types which limit them to pairs. It would be impossible to define projection functions on tuples in general, because they'd have to be able to accept tuples of different sizes, so the type of the function would vary.
3. ↑ In fact, these are one and the same concept in Haskell.
4. ↑ This isn't quite what `chr` and `ord` do, but that description fits our purposes well, and it's close enough.
5. ↑ To make things even more confusing, there's actually even more than one type for integers! Don't worry, we'll come on to this in due course.
6. ↑ This has been somewhat simplified to fit our purposes. Don't worry, the essence of the function is there.
7. ↑ Some of the newer type system extensions to GHC *do* break this, however, so you're better off just always putting down types anyway.
8. ↑ This is a slight lie. That type signature would mean that you can compare two values of any type whatsoever, but this clearly isn't true: how can you see if two functions are equal? Haskell includes a kind of 'restricted polymorphism' that allows type variables to range over some, but not all types. Haskell implements this using *type classes*, which we'll learn about later. In this case, the correct type of `(==)` is `Eq a => a -> a -> Bool`.
9. ↑ In mathematics, $n!$ normally means the factorial of n , but that syntax is impossible in Haskell, so we don't use it here.
10. ↑ Actually, defining the factorial of 0 to be 1 is not just arbitrary; it's because the factorial of 0 represents an empty product.
11. ↑ This is no coincidence; without mutable variables, recursion is the only way to implement control structures. This might sound like a limitation until you get used to it (it isn't, really).
12. ↑ Actually, it's using a function called `foldl`, which actually does the recursion.
13. ↑ Moggi, Eugenio (1991). "Notions of Computation and Monads". *Information and Computation* **93** (1).
14. ↑ w:Philip Wadler. Comprehending Monads (<http://citeseer.ist.psu.edu/wadler92comprehending.html>) . Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice. 1990.
15. ↑ w:Philip Wadler. The Essence of Functional Programming (<http://citeseer.ist.psu.edu/wadler92essence.html>) . Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1992.
16. ↑ Simon Peyton Jones, Philip Wadler (1993). "Imperative functional programming" (<http://homepages.inf.ed.ac.uk/wadler/topics/monads.html#imperative>) . *20'th Symposium on Principles of Programming Languages*.
17. ↑ It infers a monomorphic type because κ is bound by a lambda expression, and things bound by lambdas always have monomorphic types. See Polymorphism.

18. ↑ Ian Stewart. *The true story of how Theseus found his way out of the labyrinth*. Scientific American, February 1991, page 137.
19. ↑ Gérard Huet. *The Zipper*. Journal of Functional Programming, 7 (5), Sept 1997, pp. 549--554. PDF (<http://www.st.cs.uni-sb.de/edu/seminare/2005/advanced-fp/docs/huet-zipper.pdf>)
20. ↑ Note the notion of *zipper* as coined by Gérard Huet also allows to replace whole subtrees even if there is no extra data associated with them. In the case of our labyrinth, this is irrelevant. We will come back to this in the section Differentiation of data types.
21. ↑ Of course, the second topmost node or any other node at most a constant number of links away from the top will do as well.
22. ↑ Note that changing the whole data structure as opposed to updating the data at a node can be achieved in amortized constant time even if more nodes than just the top node is affected. An example is incrementing a number in binary representation. While incrementing say $111 \dots 11$ must touch all digits to yield $1000 \dots 00$, the increment function nevertheless runs in constant amortized time (but not in constant worst case time).
23. ↑ Conor Mc Bride. *The Derivative of a Regular Type is its Type of One-Hole Contexts*. Available online. PDF (<http://www.cs.nott.ac.uk/~ctm/diff.pdf>)
24. ↑ This phenomenon already shows up with generic tries.
25. ↑ Actually, we can apply them to functions whose type is $\text{forall } a. a \rightarrow R$, for some arbitrary R , as these accept values of any type as a parameter. Examples of such functions: `id`, `const k` for any `k`. So technically, we can't do anything `_useful_` with its elements.
26. ↑ In fact, there are no written down and complete denotational semantics of Haskell. This would be a tedious task void of additional insight and we happily embrace the folklore and common sense semantics.
27. ↑ Monads are one of the most successful ways to give denotational semantics to imperative programs. See also Haskell/Advanced monads.
28. ↑ Strictness as premature evaluation of function arguments is elaborated in the chapter Graph Reduction.
29. ↑ The term *Laziness* comes from the fact that the prevalent implementation technique for non-strict languages is called *lazy evaluation*
30. ↑ The term *lifted* is somewhat overloaded, see also Unboxed Types.
31. ↑ S. Peyton Jones, A. Reid, T. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. (<http://research.microsoft.com/~simonpj/Papers/imprecise-exn.htm>) In Programming Languages Design and Implementation. ACM press, May 1999.
32. ↑ John C. Reynolds. *Polymorphism is not set-theoretic*. INRIA Rapports de Recherche No. 296. May 1984.

External Links

Online books about Denotational Semantics

- Schmidt, David A. (1986). *Denotational Semantics. A Methodology for Language Development* (<http://www.cis.ksu.edu/~schmidt/text/densem.html>) . Allyn and Bacon.

Equational reasoning

Haskell/Equational reasoning

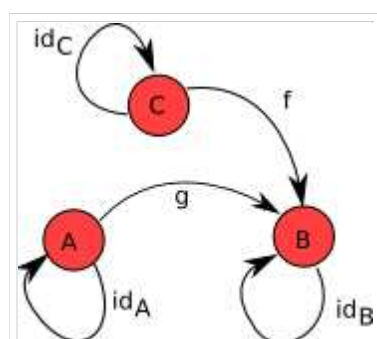
Program derivation

Haskell/Program derivation

Category theory

This article attempts to give an overview of category theory, insofar as it applies to Haskell. To this end, Haskell code will be given alongside the mathematical definitions. Absolute rigour is not followed; in its place, we seek to give the reader an intuitive feel for what the concepts of category theory are and how they relate to Haskell.

Introduction to categories



A simple category, with three objects A , B and C , three identity morphisms id_A , id_B and id_C , and two other morphisms $f : C \rightarrow B$ and $g : A \rightarrow B$. The third element (the specification of how to compose the morphisms) is not shown.

A category is, in essence, a simple collection. It has three components:

- ▮ A collection of **objects**.
- ▮ A collection of **morphisms**, each of which ties two objects (a *source object* and a *target object*) together. (These are sometimes called **arrows**, but we avoid that term here as it has other denotations in Haskell.) If f is a morphism with source object A and target object B , we write $f : A \rightarrow B$.
- ▮ A notion of **composition** of these morphisms. If h is the composition of morphisms f and g , we write $h = f \circ g$.

Lots of things form categories. For example, **Set** is the category of all sets with morphisms as standard functions and composition being standard function composition. (Category names are often typeset in bold face.) **Grp** is the category of all groups with morphisms as functions that preserve group operations (the group homomorphisms), i.e. for any two groups G with operation $*$ and H with operation \cdot , a function $f : G \rightarrow H$ is a morphism in **Grp** iff:

$$f(u * v) = f(u) \cdot f(v)$$

It may seem that morphisms are always functions, but this needn't be the case. For example, any partial order (P, \leq) defines a category where the objects are the elements of P , and there is a morphism between any two objects A and B iff $A \leq B$. Moreover, there are allowed to be multiple morphisms with the same source and target objects; using the **Set** example, \sin and \cos are both functions with source object \mathbb{R} and target object $[-1, 1]$, but they're most certainly not the same morphism!

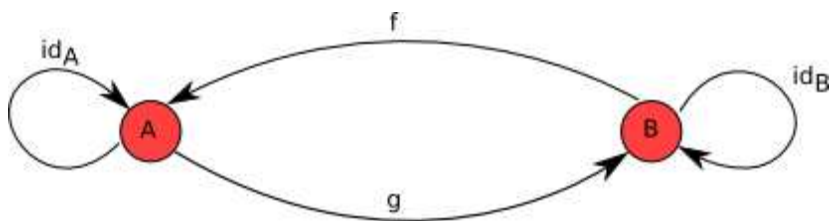
Category laws

There are three laws that categories need to follow. Firstly, and most simply, the composition of morphisms needs to be associative. Symbolically,

$$f \circ (g \circ h) = (f \circ g) \circ h$$

Secondly, the category needs to be closed under the composition operation. So if $f : A \rightarrow B$ and $g : B \rightarrow C$, then there must be some morphism $h : A \rightarrow C$ in the category such that $h = f \circ g$. We can

see how this works using the following category:



f and g are both morphisms so we must be able to compose them and get another morphism in the category. So which is the morphism $f \circ g$? The only option is id_A . Similarly, we see that $g \circ f = id_B$.

Lastly, given a category C there needs to be for every object A an *identity morphism*, $id_A : A \rightarrow A$ that is an identity of composition with other morphisms. Put precisely, for every morphism $f : A \rightarrow B$:

$$f \circ id_A = id_B \circ f = f$$

Hask, the Haskell category

The main category we'll be concerning ourselves with in this article is **Hask**, the category of Haskell types and Haskell functions as morphisms, using $(.)$ for composition: a function $f :: A \rightarrow B$ for types A and B is a morphism in **Hask**. We can check the first and last laws easily: we know $(.)$ is an associative function, and clearly, for any f and g , $f . g$ is another function. In **Hask**, the identity morphism is id , and we have trivially:

$$id . f = f . id = f$$

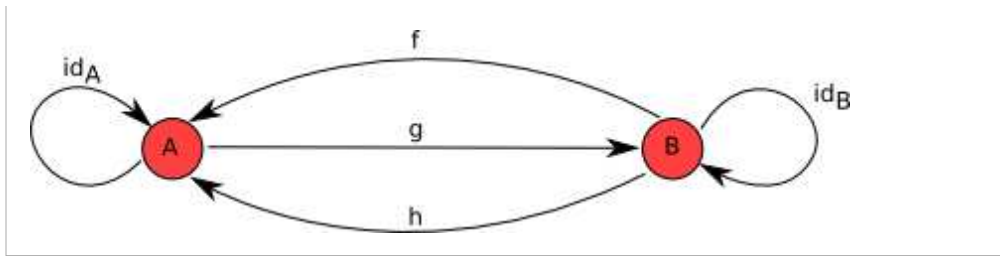
[33] This isn't an exact translation of the law above, though; we're missing subscripts. The function id in Haskell is *polymorphic* - it can take many different types for its domain and range, or, in category-speak, can have many different source and target objects. But morphisms in category theory are by definition *monomorphic* - each morphism has one specific source object and one specific target object. A polymorphic Haskell function can be made monomorphic by specifying its type (*instantiating* with a monomorphic type), so it would be more precise if we said that the identity morphism from **Hask** on a type A is $(id :: A \rightarrow A)$. With this in mind, the above law would be rewritten as:

$$(id :: B \rightarrow B) . f = f . (id :: A \rightarrow A) = f$$

However, for simplicity, we will ignore this distinction when the meaning is clear.

Exercises

- As was mentioned, any partial order (P, \leq) is a category with objects as the elements of P and a morphism between elements a and b iff $a \leq b$. Which of the above laws guarantees the transitivity of \leq ?
- (Harder.) If we add another morphism to the above example, it fails to be a category. Why? Hint: think about associativity of the composition operation.



Functors

So we have some categories which have objects and morphisms that relate our objects together. The next Big Topic in category theory is the **functor**, which relates categories together. A functor is essentially a transformation between categories, so given categories C and D , a functor $F : C \rightarrow D$:

- Maps any object A in C to $F(A)$, in D .
- Maps morphisms $f : A \rightarrow B$ in C to $F(f) : F(A) \rightarrow F(B)$ in D .

One of the canonical examples of a functor is the forgetful functor

Grp \rightarrow **Set** which maps groups to their underlying sets and group morphisms to the functions which behave the same but are defined on sets

instead of groups. Another example is

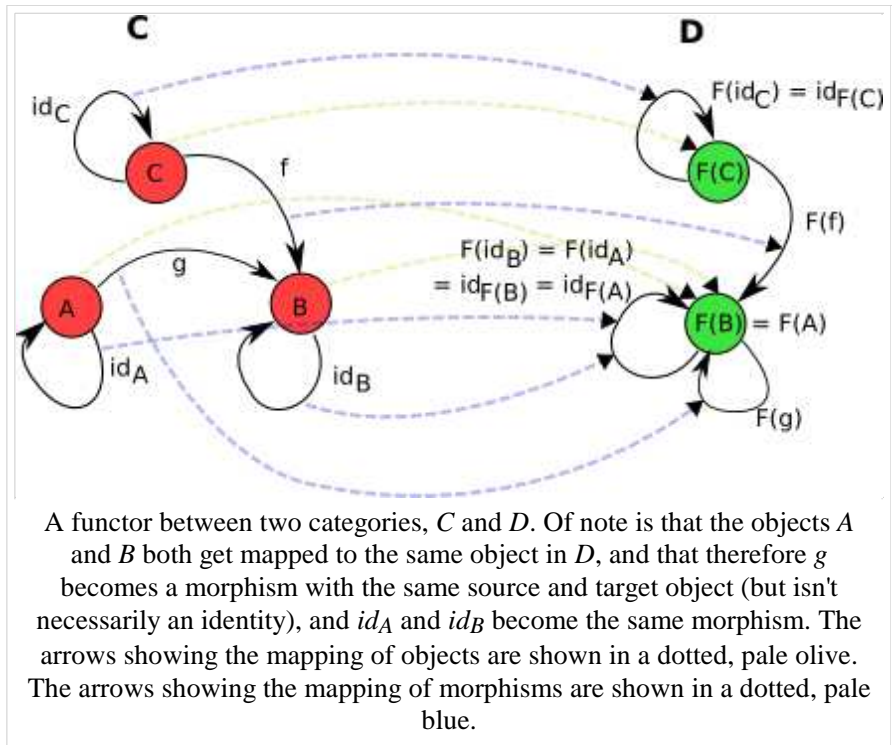
the power set functor **Set** \rightarrow **Set** which maps sets to their power sets and maps functions $f : X \rightarrow Y$ to functions $\mathcal{P}(X) \rightarrow \mathcal{P}(Y)$ which take inputs $U \subset X$ and return $f(U)$, the image of U under f , defined by $f(U) = \{f(u) : u \in U\}$. For any category C , we can define a functor known as the identity functor on C , or $1_C : C \rightarrow C$, that just maps objects to themselves and morphisms to themselves. This will turn out to be useful in the monad laws section later on.

Once again there are a few axioms that functors have to obey. Firstly, given an identity morphism id_A on an object A , $F(id_A)$ must be the identity morphism on $F(A)$, i.e.:

$$F(id_A) = id_{F(A)}$$

Secondly functors must distribute over morphism composition, i.e.

$$F(f \circ g) = F(f) \circ F(g)$$



Exercises

For the diagram given on the right, check these functor laws.

Functors on Hask

The Functor typeclass you will probably have seen in Haskell does in fact tie in with the categorical notion of a functor. Remember that a functor has two parts: it maps objects in one category to objects in another and morphisms in the first category to morphisms in the second. Functors in Haskell are from **Hask** to *func*, where *func* is the subcategory of **Hask** defined on just that functor's types. E.g. the list functor goes from **Hask** to **Lst**, where **Lst** is the category containing only *list types*, that is, `[T]` for any type `T`. The morphisms in **Lst** are functions defined on list types, that is, functions `[T] -> [U]` for types `T, U`. How does this tie into the Haskell typeclass Functor? Recall its definition:

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

Let's have a sample instance, too:

```
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap _ Nothing = Nothing
```

Here's the key part: the *type constructor* `Maybe` takes any type `T` to a new type, `Maybe T`. Also, `fmap` restricted to `Maybe` types takes a function `a -> b` to a function `Maybe a -> Maybe b`. But that's it! We've defined two parts, something that takes objects in **Hask** to objects in another category (that of `Maybe` types and functions defined on `Maybe` types), and something that takes morphisms in **Hask** to morphisms in this category. So `Maybe` is a functor.

A useful intuition regarding Haskell functors is that they represent types that can be mapped over. This could be a list or a `Maybe`, but also more complicated structures like trees. A function that does some mapping could be written using `fmap`, then any functor structure could be passed into this function. E.g. you could write a generic function that covers all of `Data.List.map`, `Data.Map.map`, `Data.Array.IArray.amap`, and so on.

What about the functor axioms? The polymorphic function `id` takes the place of id_A for any A , so the first law states:

```
fmap id = id
```

With our above intuition in mind, this states that mapping over a structure doing nothing to each element is equivalent to doing nothing overall. Secondly, morphism composition is just `(.)`, so

```
fmap (f . g) = fmap f . fmap g
```

This second law is very useful in practice. Picturing the functor as a list or similar container, the right-hand side is a two-pass algorithm: we map over the structure, performing `g`, then map over it again, performing `f`. The functor axioms guarantee we can transform this into a single-pass algorithm that performs `f . g`. This is a process known as *fusion*.

Exercises

Check the laws for the `Maybe` and list functors.

Translating categorical concepts into Haskell

Functors provide a good example of how category theory gets translated into Haskell. The key points to remember are that:

- We work in the category **Hask** and its subcategories.
- Objects are types.
- Morphisms are functions.
- Things that take a type and return another type are type constructors.
- Things that take a function and return another function are higher-order functions.
- Typeclasses, along with the polymorphism they provide, make a nice way of capturing the fact that in category theory things are often defined over a number of objects at once.

Monads

Monads are obviously an extremely important concept in Haskell, and in fact they originally came from category theory. A *monad* is a special type of functor, one that supports some additional structure. Additionally, every monad is a functor from a category to that same category. So, down to definitions. A monad is a functor $M : \mathcal{C} \rightarrow \mathcal{C}$, along with two morphisms ^[34] for every object X in \mathcal{C} :

- $unit_X^M : X \rightarrow M(X)$
- $join_X^M : M(M(X)) \rightarrow M(X)$

When the monad under discussion is obvious, we'll leave out the M superscript for these functions and just talk about $unit_X$ and $join_X$ for some X .

Let's see how this translates to the Haskell typeclass `Monad`, then.

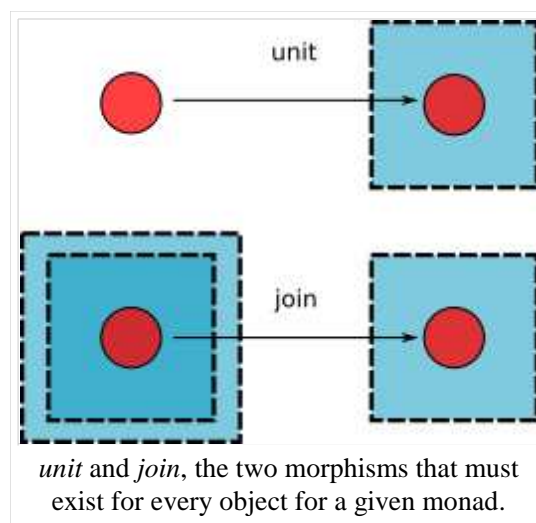
```
class Functor m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

The class constraint of `Functor m` ensures that we already have the functor structure: a mapping of objects and of morphisms. `return` is the (polymorphic) analogue to $unit_X$ for any X . But we have a problem. Although `return`'s type looks quite similar to that of $unit$, `(>>=)` bears no resemblance to $join$. The monad function `join :: Monad m => m (m a) -> m a` does however look quite similar. Indeed, we can recover `join` and `(>>=)` from each other:

```
join :: Monad m => m (m a) -> m a
join x = x >>= id

(>>=) :: Monad m => m a -> (a -> m b) -> m b
x >>= f = join (fmap f x)
```

So specifying a monad's `return` and `join` is equivalent to specifying its `return` and `(>>=)`. It just turns out that the normal way of defining a monad in category theory is to give $unit$ and $join$, whereas Haskell programmers



like to give `return` and `(>>=)` ^[35]. Often, the categorical way makes more sense. Any time you have some kind of structure M and a natural way of taking any object X into $M(X)$, as well as a way of taking $M(M(X))$ into $M(X)$, you probably have a monad. We can see this in the following example section.

Example: the powerset functor is also a monad

The power set functor $P : \mathbf{Set} \rightarrow \mathbf{Set}$ described above forms a monad. For any set S you have a $units(x) = \{x\}$, mapping elements to their singleton set. Note that each of these singleton sets are trivially a subset of S , so $units$ returns elements of the powerset of S , as is required. Also, you can define a function $joins$ as follows: we receive an input $L \in \mathcal{P}(\mathcal{P}(S))$. This is:

- A member of the powerset of the powerset of S .
- So a member of the set of all subsets of the set of all subsets of S .
- So a set of subsets of S

We then return the union of these subsets, giving another subset of S . Symbolically,

$$join_S(L) = \bigcup L.$$

Hence P is a monad ^[36].

In fact, P is almost equivalent to the list monad; with the exception that we're talking lists instead of sets, they're almost the same. Compare:

Power set functor on Set		List monad from Haskell	
Function type	Definition	Function type	Definition
Given a set S and a morphism $f : A \rightarrow B$:		Given a type \mathbb{T} and a function $f :: A \rightarrow B$	
$P(f) : \mathcal{P}(A) \rightarrow \mathcal{P}(B)$	$(P(f))(S) = \{f(a) : a \in S\}$	<code>fmap f :: [A] -> [B]</code>	<code>fmap f xs = [f b b <- xs]</code>
$units : S \rightarrow \mathcal{P}(S)$	$units(x) = \{x\}$	<code>return :: T -> [T]</code>	<code>return x = [x]</code>
$join_S : \mathcal{P}(\mathcal{P}(S)) \rightarrow \mathcal{P}(S)$	$join_S(L) = \bigcup L$	<code>join :: [[T]] -> [T]</code>	<code>join xs = concat xs</code>

The monad laws and their importance

Just as functors had to obey certain axioms in order to be called functors, monads have a few of their own. We'll first list them, then translate to Haskell, then see why they're important.

Given a monad $M : \mathcal{C} \rightarrow \mathcal{C}$ and a morphism $f : A \rightarrow B$ for $A, B \in \mathcal{C}$,

1. $join \circ M(join) = join \circ join$
2. $join \circ M(unit) = join \circ unit = id$
3. $unit \circ f = M(f) \circ unit$

$$4. \text{join} \circ M(M(f)) = M(f) \circ \text{join}$$

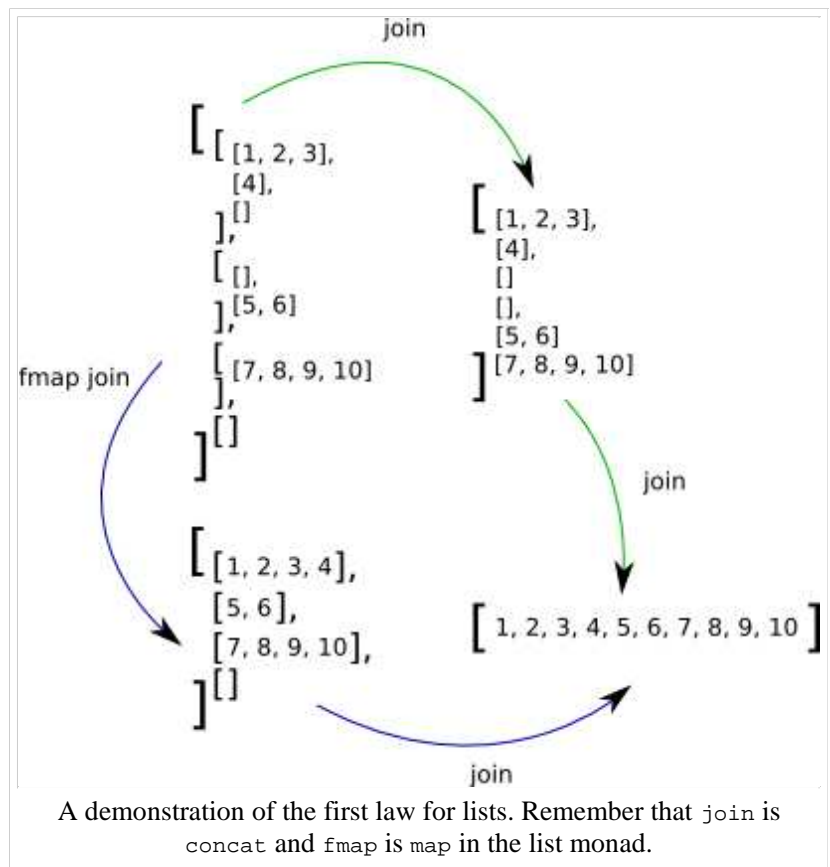
By now, the Haskell translations should be hopefully self-explanatory:

1. `join . fmap join = join . join`
2. `join . fmap return = join . return = id`
3. `return . f = fmap f . return`
4. `join . fmap (fmap f) = fmap f . join`

(Remember that `fmap` is the part of a functor that acts on morphisms.) These laws seem a bit impenetrable at first, though. What on earth do these laws mean, and why should they be true for monads? Let's explore the laws.

The first law

In order to understand this law, we'll first use the example of lists. The first law mentions two functions, `join . fmap join` (the left-hand side) and `join . join` (the right-hand side). What will the types of these functions be? Remembering that `join`'s type is `[[a]] -> [a]` (we're talking just about lists for now), the types are both `[[[a]]] -> [a]` (the fact that they're the same is handy; after all, we're trying to show they're completely the same function!). So we have a list of list of lists. The left-hand side, then, performs `fmap join` on this 3-layered list, then uses `join` on the result. `fmap` is just the familiar `map` for lists, so we first map across each of the list of lists inside the top-level list, concatenating them down into a list each. So afterward, we have a list of lists, which we then run through `join`. In summary, we 'enter' the top level, collapse the second and third levels down, then collapse this new level with the top level.



What about the right-hand side? We first run `join` on our list of list of lists. Although this is three layers, and you normally apply a two-layered list to `join`, this will still work, because a `[[[a]]]` is just `[[b]]`, where `b = [a]`, so in a sense, a three-layered list is just a two layered list, but rather than the last layer being 'flat', it is composed of another list. So if we apply our list of lists (of lists) to `join`, it will flatten those outer two layers into one. As the second layer wasn't flat but instead contained a third layer, we will still end up with a list of lists, which the other `join` flattens. Summing up, the left-hand side will flatten the inner two layers into a new layer, then flatten this with the outermost layer. The right-hand side will flatten the outer two layers, then flatten this with the innermost layer. These two operations should be equivalent. It's sort of like a law of associativity for `join`.

We can see this at work more if we recall the definition of `join` for `Maybe`:

```

!join :: Maybe (Maybe a) -> Maybe a
!join Nothing           = Nothing
!join (Just Nothing)   = Nothing
!join (Just (Just x))  = Just x

```

So if we had a *three*-layered `Maybe` (i.e., it could be `Nothing`, `Just Nothing`, `Just (Just Nothing)` or `Just (Just (Just x))`), the first law says that collapsing the inner two layers first, then that with the outer layer is exactly the same as collapsing the outer layers first, then that with the innermost layer.

Exercises

Verify that the list and `Maybe` monads do in fact obey this law with some examples to see precisely how the layer flattening works.

The second law

What about the second law, then? Again, we'll start with the example of lists. Both functions mentioned in the second law are functions `[a] -> [a]`. The left-hand side expresses a function that maps over the list, turning each element `x` into its singleton list `[x]`, so that at the end we're left with a list of singleton lists. This two-layered list is flattened down into a single-layer list again using the `join`. The right hand side, however, takes the entire list `[x, y, z, ...]`, turns it into the singleton list of lists `[[x, y, z, ...]]`, then flattens the two layers down into one again. This law is less obvious to state quickly, but it essentially says that applying `return` to a monadic value, then `joining` the result should have the same effect whether you perform the `return` from inside the top layer or from outside it.

Exercises

Prove this second law for the `Maybe` monad.

The third and fourth laws

The last two laws express more self evident fact about how we expect monads to behave. The easiest way to see how they are true is to expand them to use the expanded form:

- `\x -> return (f x) = \x -> fmap f (return x)`
- `\x -> join (fmap (fmap f) x) = \x -> fmap f (join x)`

Exercises

Convince yourself that these laws should hold true for any monad by exploring what they mean, in a similar style to how we explained the first and second laws.

Application to do-blocks

Well, we have intuitive statements about the laws that a monad must support, but why is that important? The answer becomes obvious when we consider `do`-blocks. Recall that a `do`-block is just syntactic sugar for a combination of statements involving `(>>=)` as witnessed by the usual translation:

```
do { x }           --> x
do { let { y = v }; x } --> let y = v in do { x }
do { v <- y; x }    --> y >>= \v -> do { x }
do { y; x }        --> y >>= \_ -> do { x }
```

Also notice that we can prove what are normally quoted as the monad laws using `return` and `(>>=)` from our above laws (the proofs are a little heavy in some cases, feel free to skip them if you want to):

1. `return x >>= f = f x`. Proof:

```
return x >>= f
= join (fmap f (return x)) -- By the definition of (>>=)
= join (return (f x))      -- By law 3
= (join . return) (f x)
= id (f x)                 -- By law 2
= f x
```

2. `m >>= return = m`. Proof:

```
m >>= return
= join (fmap return m)    -- By the definition of (>>=)
= (join . fmap return) m
= id m                    -- By law 2
= m
```

3. `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`. Proof (recall that `fmap f . fmap g = fmap (f . g)`):

```
(m >>= f) >>= g
= (join (fmap f m)) >>= g -- By the definition of (>>=)
= join (fmap g (join (fmap f m))) -- By the definition of (>>=)
= (join . fmap g) (join (fmap f m))
= (join . fmap g . join) (fmap f m)
= (join . join . fmap (fmap g)) (fmap f m) -- By law 4
= (join . join . fmap (fmap g) . fmap f) m
= (join . join . fmap (fmap g . f)) m -- By the distributive law of functors
= (join . join . fmap (\x -> fmap g (f x))) m
= (join . fmap join . fmap (\x -> fmap g (f x))) m -- By law 1
= (join . fmap (join . (\x -> fmap g (f x)))) m -- By the distributive law of functors
= (join . fmap (\x -> join (fmap g (f x)))) m
= (join . fmap (\x -> f x >>= g)) -- By the definition of (>>=)
= join (fmap (\x -> f x >>= g) m)
= m >>= (\x -> f x >>= g) -- By the definition of (>>=)
```

These new monad laws, using `return` and `(>>=)`, can be translated into do-block notation.

Points-free style	Do-block style
<code>return x >>= f = f x</code>	<code>do { v <- return x; f v } = do { f x }</code>
<code>m >>= return = m</code>	<code>do { v <- m; return v } = do { m }</code>
<code>(m >>= f) >>= g = m >>= (\x -> f x >>= g)</code>	<pre>do { y <- do { x <- m; f x }; g y } = do { x <- m; y <- f x; g y }</pre>

The monad laws are now common-sense statements about how do-blocks should function. If one of these laws

were invalidated, users would become confused, as you couldn't be able to manipulate things within the do-blocks as would be expected. The monad laws are, in essence, usability guidelines.

Exercises

In fact, the two versions of the laws we gave:

```

-----
-- Categorical:
!join . fmap join = join . join
!join . fmap return = join . return = id
!return . f = fmap f . return
!join . fmap (fmap f) = fmap f . join
-----
-- Functional:
m >>= return = m
!return m >>= f = f m
!(m >>= f) >>= g = m >>= (\x -> f x >>= g)
-----

```

Are entirely equivalent. We showed that we can recover the functional laws from the categorical ones. Go the other way; show that starting from the functional laws, the categorical laws hold. It may be useful to remember the following definitions:

```

-----
!join m = m >>= id
!fmap f m = m >>= return . f
-----

```

Thanks to Yitzchak Gale for suggesting this exercise.

Summary

We've come a long way in this chapter. We've looked at what categories are and how they apply to Haskell. We've introduced the basic concepts of category theory including functors, as well as some more advanced topics like monads, and seen how they're crucial to idiomatic Haskell. We haven't covered some of the basic category theory that wasn't needed for our aims, like natural transformations, but have instead provided an intuitive feel for the categorical grounding behind Haskell's structures.

Notes

- ↑ At least as far as types are concerned, but we're trying to avoid that word :)
- ↑ More technically, `fst` and `snd` have types which limit them to pairs. It would be impossible to define projection functions on tuples in general, because they'd have to be able to accept tuples of different sizes, so the type of the function would vary.
- ↑ In fact, these are one and the same concept in Haskell.
- ↑ This isn't quite what `chr` and `ord` do, but that description fits our purposes well, and it's close enough.
- ↑ To make things even more confusing, there's actually even more than one type for integers! Don't worry, we'll come on to this in due course.
- ↑ This has been somewhat simplified to fit our purposes. Don't worry, the essence of the function is there.
- ↑ Some of the newer type system extensions to GHC *do* break this, however, so you're better off just always putting down types anyway.
- ↑ This is a slight lie. That type signature would mean that you can compare two values of any type whatsoever, but this clearly isn't true: how can you see if two functions are equal? Haskell includes a kind

of 'restricted polymorphism' that allows type variables to range over some, but not all types. Haskell implements this using *type classes*, which we'll learn about later. In this case, the correct type of `(=)` is `Eq a => a -> a -> Bool`.

9. ↑ In mathematics, $n!$ normally means the factorial of n , but that syntax is impossible in Haskell, so we don't use it here.
10. ↑ Actually, defining the factorial of 0 to be 1 is not just arbitrary; it's because the factorial of 0 represents an empty product.
11. ↑ This is no coincidence; without mutable variables, recursion is the only way to implement control structures. This might sound like a limitation until you get used to it (it isn't, really).
12. ↑ Actually, it's using a function called `foldl`, which actually does the recursion.
13. ↑ Moggi, Eugenio (1991). "Notions of Computation and Monads". *Information and Computation* **93** (1).
14. ↑ w:Philip Wadler. Comprehending Monads (<http://citeseer.ist.psu.edu/wadler92comprehending.html>) . Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice. 1990.
15. ↑ w:Philip Wadler. The Essence of Functional Programming (<http://citeseer.ist.psu.edu/wadler92essence.html>) . Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1992.
16. ↑ Simon Peyton Jones, Philip Wadler (1993). "Imperative functional programming" (<http://homepages.inf.ed.ac.uk/wadler/topics/monads.html#imperative>) . *20'th Symposium on Principles of Programming Languages*.
17. ↑ It infers a monomorphic type because `κ` is bound by a lambda expression, and things bound by lambdas always have monomorphic types. See Polymorphism.
18. ↑ Ian Stewart. *The true story of how Theseus found his way out of the labyrinth*. Scientific American, February 1991, page 137.
19. ↑ Gérard Huet. *The Zipper*. Journal of Functional Programming, 7 (5), Sept 1997, pp. 549--554. PDF (<http://www.st.cs.uni-sb.de/edu/seminare/2005/advanced-fp/docs/huet-zipper.pdf>)
20. ↑ Note the notion of *zipper* as coined by Gérard Huet also allows to replace whole subtrees even if there is no extra data associated with them. In the case of our labyrinth, this is irrelevant. We will come back to this in the section Differentiation of data types.
21. ↑ Of course, the second topmost node or any other node at most a constant number of links away from the top will do as well.
22. ↑ Note that changing the whole data structure as opposed to updating the data at a node can be achieved in amortized constant time even if more nodes than just the top node is affected. An example is incrementing a number in binary representation. While incrementing say `111 . . 11` must touch all digits to yield `1000 . . 00`, the increment function nevertheless runs in constant amortized time (but not in constant worst case time).
23. ↑ Conor Mc Bride. *The Derivative of a Regular Type is its Type of One-Hole Contexts*. Available online. PDF (<http://www.cs.nott.ac.uk/~ctm/diff.pdf>)
24. ↑ This phenomenon already shows up with generic tries.
25. ↑ Actually, we can apply them to functions whose type is `forall a. a -> R`, for some arbitrary `R`, as these accept values of any type as a parameter. Examples of such functions: `id`, `const k` for any `k`. So technically, we can't do anything `_useful_` with its elements.
26. ↑ In fact, there are no written down and complete denotational semantics of Haskell. This would be a tedious task void of additional insight and we happily embrace the folklore and common sense semantics.
27. ↑ Monads are one of the most successful ways to give denotational semantics to imperative programs. See also Haskell/Advanced monads.
28. ↑ Strictness as premature evaluation of function arguments is elaborated in the chapter Graph Reduction.
29. ↑ The term *Laziness* comes from the fact that the prevalent implementation technique for non-strict languages is called *lazy evaluation*
30. ↑ The term *lifted* is somewhat overloaded, see also Unboxed Types.
31. ↑ S. Peyton Jones, A. Reid, T. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. (<http://research.microsoft.com/~simonpj/Papers/imprecise-exn.htm>) In Programming Languages Design

- and Implementation. ACM press, May 1999.
32. ↑ John C. Reynolds. *Polymorphism is not set-theoretic*. INRIA Rapports de Recherche No. 296. May 1984.
 33. ↑ Actually, there is a subtlety here: because `(.)` is a lazy function, if `f` is `undefined`, we have that `id . f = _ -> _|_`. Now, while this may seem equivalent to `_|_` for all extents and purposes, you can actually tell them apart using the strictifying function `seq`, meaning that the last category law is broken. We can define a new strict composition function, `f .! g = ((.) $! f) $! g`, that makes **Hask** a category. We proceed by using the normal `(.)`, though, and attribute any discrepancies to the fact that `seq` breaks an awful lot of the nice language properties anyway.
 34. ↑ Experienced category theorists will notice that we're simplifying things a bit here; instead of presenting *unit* and *join* as natural transformations, we treat them explicitly as morphisms, and require naturality as extra axioms alongside the the standard monad laws (laws 3 and 4). The reasoning is simplicity; we are not trying to teach category theory as a whole, simply give a categorical background to some of the structures in Haskell. You may also notice that we are giving these morphisms names suggestive of their Haskell analogues, because the names η and μ don't provide much intuition.
 35. ↑ This is perhaps due to the fact that Haskell programmers like to think of monads as a way of sequencing computations with a common feature, whereas in category theory the container aspect of the various structures is emphasised. `join` pertains naturally to containers (squashing two layers of a container down into one), but `(>>=)` is the natural sequencing operation (do something, feeding its results into something else).
 36. ↑ If you can prove that certain laws hold, which we'll explore in the next section.
-

Haskell Performance

Graph reduction

Notes and TODOs

- *TODO: Pour lazy evaluation explanation from Laziness into this mold.*
- *TODO: better section names.*
- *TODO: ponder the graphical representation of graphs.*
 - *No graphical representation, do it with `let .. in`. Pro: Reduction are easiest to perform in that way anyway. Cons: no graphic.*
 - *ASCII art / line art similar to the one in Bird&Wadler? Pro: displays only the relevant parts truly as graph, easy to perform on paper. Cons: Ugly, no large graphs with that.*
 - *Full blown graphs with @-nodes? Pro: look graphy. Cons: nobody needs to know @-nodes in order to understand graph reduction. Can be explained in the implementation section.*
 - *Graphs without @-nodes. Pro: easy to understand. Cons: what about currying?*
- *! Keep this chapter short. The sooner the reader knows how to evaluate Haskell programs by hand, the better.*
- *First sections closely follow Bird&Wadler*

Introduction

Programming is not only about writing correct programs, answered by denotational semantics, but also about writing fast ones that require little memory. For that, we need to know how they're executed on a machine, commonly given by operational semantics. This chapter explains how Haskell programs are commonly executed on a real computer and thus serves as foundation for analyzing time and space usage. Note that the Haskell standard deliberately does *not* give operational semantics, implementations are free to choose their own. But so far, every implementation of Haskell more or less closely follows the execution model of *lazy evaluation*.

In the following, we will detail lazy evaluation and subsequently use this execution model to explain and exemplify the reasoning about time and memory complexity of Haskell programs.

Evaluating Expressions by Lazy Evaluation

Reductions

Executing a functional program, i.e. evaluating an expression, means to repeatedly apply function definitions until all function applications have been expanded. Take for example the expression `pythagoras 3 4` together with the definitions

```
square x = x * x
pythagoras x y = square x + square y
```

One possible sequence of such **reductions** is

```
pythagoras 3 4
⇒ square 3 + square 4    (pythagoras)
⇒ (3*3) + square 4      (square)
⇒ 9 + square 4          (*)
⇒ 9 + (4*4)              (square)
⇒ 9 + 16                 (*)
⇒ 25
```

Every reduction replaces a subexpression, called **reducible expression** or **redex** for short, with an equivalent one, either by appealing to a function definition like for `square` or by using a built-in function like `(+)`. An expression without redexes is said to be in **normal form**. Of course, execution stops once reaching a normal form which thus is the result of the computation.

Clearly, the fewer reductions that have to be performed, the faster the program runs. We cannot expect each reduction step to take the same amount of time because its implementation on real hardware looks very different, but in terms of asymptotic complexity, this number of reductions is an accurate measure.

Reduction Strategies

There are many possible reduction sequences and the number of reductions may depend on the order in which reductions are performed. Take for example the expression `fst (square 3, square 4)`. One systematic possibility is to evaluate all function arguments before applying the function definition

```
fst (square 3, square 4)
⇒ fst (3*3, square 4)    (square)
⇒ fst (9, square 4)      (*)
⇒ fst (9, 4*4)           (square)
⇒ fst (9, 16)            (*)
⇒ 9                      (fst)
```

This is called an **innermost reduction** strategy and an **innermost redex** is a redex that has no other redex as subexpression inside.

Another systematic possibility is to apply all function definitions first and only then evaluate arguments:

```
fst (square 3, square 4)
⇒ square 3           (fst)
⇒ 3*3                (square)
⇒ 9                  (*)
```

which is named **outermost reduction** and always reduces **outermost redexes** that are not inside another redex. Here, the outermost reduction uses less reduction steps than the innermost reduction. Why? Because the function `fst` doesn't need the second component of the pair and the reduction of `square 4` was superfluous.

Termination

For some expressions like

```
loop = 1 + loop
```

no reduction sequence may terminate and program execution enters a neverending loop, those expressions do not have a normal form. But there are also expressions where some reduction sequences terminate and some do not, an example being

```
fst (42, loop)
⇒ 42                (fst)

fst (42, loop)
⇒ fst (42, 1+loop)  (loop)
⇒ fst (42, 1+(1+loop)) (loop)
⇒ ...
```

The first reduction sequence is outermost reduction and the second is innermost reduction which tries in vain to evaluate the `loop` even though it is ignored by `fst` anyway. The ability to evaluate function arguments only when needed is what makes outermost optimal when it comes to termination:

Theorem (Church Rosser II)

If there is one terminating reduction, then outermost reduction will terminate, too.

Graph Reduction

Despite the ability to discard arguments, outermost reduction doesn't always take fewer reduction steps than innermost reduction:

```
square (1+2)
⇒ (1+2)*(1+2)      (square)
⇒ (1+2)*3          (+)
⇒ 3*3              (+)
⇒ 9                (*)
```

Here, the argument `(1+2)` is duplicated and subsequently reduced twice. But because it is one and the same argument, the solution is to share the reduction `(1+2) ⇒ 3` with all other incarnations of this argument. This can be achieved by representing expressions as *graphs*. For example,



represents the expression $(1+2) * (1+2)$. Now, the **outermost graph reduction** of `square (1+2)` proceeds as follows

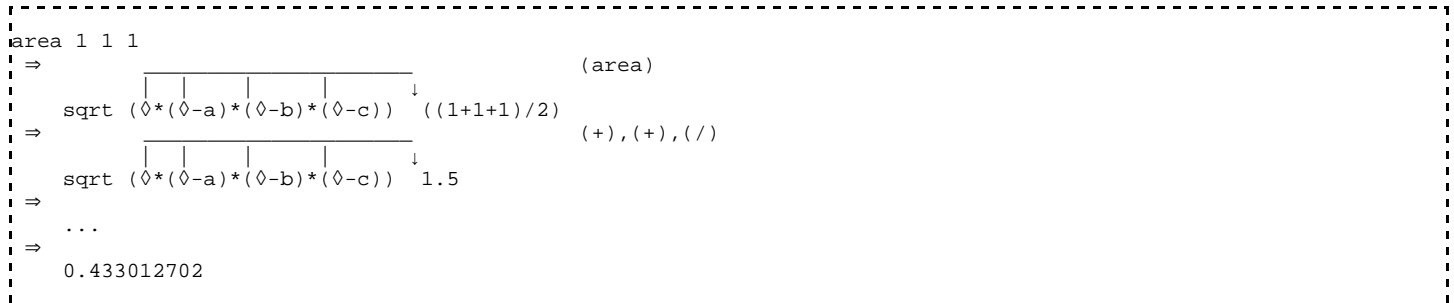


and the work has been shared. In other words, outermost graph reduction now reduces every argument at most once. For this reason, it always takes fewer reduction steps than the innermost reduction, a fact we will prove when reasoning about time.

Sharing of expressions is also introduced with `let` and `where` constructs. For instance, consider Heron's formula for the area of a triangle with sides `a`, `b` and `c`:



Instantiating this to an equilateral triangle will reduce as



which is $\sqrt{3}/4$. Put differently, `let`-bindings simply give names to nodes in the graph. In fact, one can dispense entirely with a graphical notation and solely rely on `let` to mark sharing and express a graph structure.^[37]

Any implementation of Haskell is in some form based on outermost graph reduction which thus provides a good model for reasoning about the asymptotic complexity of time and memory allocation. The number of reduction steps to reach normal form corresponds to the execution time and the size of the terms in the graph corresponds to the memory used.

Exercises

1. Reduce `square (square 3)` to normal form with innermost, outermost and outermost graph reduction.
2. Consider the fast exponentiation algorithm

```
power x 0 = 1
power x n = x' * x' * (if n `mod` 2 == 0 then 1 else x)
  where x' = power x (n `div` 2)
```

that takes x to the power of n . Reduce `power 2 5` with innermost and outermost graph reduction. How many reductions are performed? What is the asymptotic time complexity for the general `power 2 n`? What happens to the algorithm if we use "graphless" outermost reduction?

Pattern Matching

So far, our description of outermost graph reduction is still underspecified when it comes to pattern matching and data constructors. Explaining these points will enable the reader to trace most cases of the reduction strategy that is commonly the base for implementing non-strict functional languages like Haskell. It is called **call-by-need** or **lazy evaluation** in allusion to the fact that it "lazily" postpones the reduction of function arguments to the last possible moment. Of course, the remaining details are covered in subsequent chapters.

To see how pattern matching needs specification, consider for example the boolean disjunction

```
or True y = True
or False y = y
```

and the expression

```
or (1==1) loop
```

with a non-terminating `loop = not loop`. The following reduction sequence

```
or (1==1) loop
=> or (1==1) (not loop)      (loop)
=> or (1==1) (not (not loop)) (loop)
=> ...
```

only reduces outermost redexes and therefore is an outermost reduction. But

```
or (1==1) loop
=> or True loop             (or)
=> True
```

makes much more sense. Of course, we just want to apply the definition of `or` and are only reducing arguments to decide which equation to choose. This intention is captured by the following rules for pattern matching in Haskell:

- Left hand sides are matched from top to bottom
- When matching a left hand side, arguments are matched from left to right
- Evaluate arguments only as much as needed to decide whether they match or not.

Thus, for our example `or (1==1) loop`, we have to reduce the first argument to either `True` or `False`, then evaluate the second to match a variable `y` pattern and then expand the matching function definition. As the match against a variable always succeeds, the second argument will not be reduced at all. It is the second reduction section above that reproduces this behavior.

With these preparations, the reader should now be able to evaluate most Haskell expressions he encounters. Here are some random encounters to test this ability:

Exercises

Reduce the following expressions with lazy evaluation to normal form. Assume the standard function definitions from the Prelude.

- `length [42,42+1,42-1]`
- `head (map (2*) [1,2,3])`
- `head $ [1,2,3] ++ (let loop = tail loop in loop)`
- `zip [1..3] (iterate (+1) 0)`
- `head $ concatMap (\x -> [x,x+1]) [1,2,3]`
- `take (42-6*7) $ map square [2718..3146]`

Higher Order Functions

The remaining point to clarify is the reduction of higher order functions and currying. For instance, consider the definitions

```
id x = x
a = id (+1) 41
```

```
twice f = f . f
b = twice (+1) (13*3)
```

where both `id` and `twice` are only defined with one argument. The solution is to see multiple arguments as subsequent applications to one argument, this is called **currying**

```
a = (id (+1)) 41
b = (twice (+1)) (13*3)
```

To reduce an arbitrary application `expression1 expression2`, call-by-need first reduce `expression1` until this becomes a function whose definition can be unfolded with the argument `expression2`. Hence, the reduction sequences are

```

a
=> (id (+1)) 41      (a)
=> (+1) 41          (id)
=> 42               (+)

b
=> (twice (+1)) (13*3) (b)
=> ((+1).(+1)) (13*3) (twice)
=> (+1) ((+1) (13*3)) (.)
=> (+1) ((+1) 39) (*)
=> (+1) 40          (+)
=> 41               (+)

```

Admittedly, the description is a bit vague and the next section will detail a way to state it clearly.

While it may seem that pattern matching is the workhorse of time intensive computations and higher order functions are only for capturing the essence of an algorithm, functions are indeed useful as data structures. One example are difference lists (`[a] -> [a]`) that permit concatenation in $O(1)$ time, another is the representation of a stream by a fold. In fact, all data structures are represented as functions in the pure lambda calculus, the root of all functional programming languages.

Exercises! Or not? Diff-Lists Best done with `foldl (++)` but this requires knowledge of the fold example. Oh, where do we introduce the `foldl` VS. `foldr` example at all? Hm, Bird&Wadler sneak in an extra section "Meet again with fold" for the `(++)` example at the end of "Controlling reduction order and space requirements" :-/ The complexity of `(++)` is explained when arguing about `reverse`.

Weak Head Normal Form

To formulate precisely how lazy evaluation chooses its reduction sequence, it is best to abandon equational function definitions and replace them with an expression-oriented approach. In other words, our goal is to translate function definitions like `f (x:xs) = ...` into the form `f = expression`. This can be done with two primitives, namely case-expressions and lambda abstractions.

In their primitive form, case-expressions only allow the discrimination of the outermost constructor. For instance, the primitive case-expression for lists has the form

```

case expression of
  [] -> ...
  x:xs -> ...

```

Lambda abstractions are functions of one parameter, so that the following two definitions are equivalent

```

λf x = expression
λf   = \x -> expression

```

Here is a translation of the definition of `zip`

```

zip :: [a] -> [a] -> [(a,a)]
zip []   ys   = []
zip xs []   = []
zip (x:xs') (y:ys') = (x,y):zip xs' ys'

```

to case-expressions and lambda-abstractions:

```
zip = \xs -> \ys ->
  case xs of
  [] -> []
  x:xs' ->
    case ys of
    [] -> []
    y:ys' -> (x,y):zip xs' ys'
```

Assuming that all definitions have been translated to those primitives, every redex now has the form of either

- a function application ($\backslash\text{variable} \rightarrow \text{expression}_1$) expression_2
- or a case-expression `case expression of { ... }`

lazy evaluation.

Weak Head Normal Form

An expression is in weak head normal form, iff it is either

- a constructor (eventually applied to arguments) like `True`, `Just (square 42)` or `(:) 1 (42+1)`
- a built-in function applied to too few arguments (perhaps none) like `(+) 2` or `sqrt`.
- or a lambda abstraction $\backslash x \rightarrow \text{expression}$.

functions types cannot be pattern matched anyway, but the devious seq can evaluate them to WHNF nonetheless. "weak" = no reduction under lambdas. "head" = first the function application, then the arguments.

Strict and Non-strict Functions

A non-strict function doesn't need its argument. A strict function needs its argument in WHNF, as long as we do not distinguish between different forms of non-termination ($\text{f } x = \text{loop}$ doesn't need its argument, for example).

Controlling Space

NOTE: The chapter Haskell/Strictness is intended to elaborate on the stuff here.

NOTE: The notion of strict function is to be introduced before this section.

Now's the time for the space-eating fold example:

```
foldl (+) 0 [1..10]
```

Introduce seq and \$! that can force an expression to WHNF. => foldl'.

Tricky space leak example:

```
(\xs -> head xs + last xs) [1..n]
(\xs -> last xs + head xs) [1..n]
```

The first version runs on O(1) space. The second in O(n).

Sharing and CSE

NOTE: overlaps with section about time. Hm, make an extra memoization section?

How to share

```
foo x y = -- s is not shared
foo x = \y -> s + y
  where s = expensive x -- s is shared
```

"Lambda-lifting", "Full laziness". The compiler should not do full laziness.

A classic and important example for the trade between space and time:

```
sublists [] = [[]]
sublists (x:xs) = sublists xs ++ map (x:) sublists xs
sublists' (x:xs) = let ys = sublists' xs in ys ++ map (x:) ys
```

That's why the compiler should not do common subexpression elimination as optimization. (Does GHC?).

Tail recursion

NOTE: Does this belong to the space section? I think so, it's about stack space.

Tail recursion in Haskell looks different.

Reasoning about Time

Note: introducing strictness before the upper time bound saves some hassle with explanation?

Lazy eval < Eager eval

When reasoning about execution time, naively performing graph reduction by hand to get a clue on what's going is most often infeasible. In fact, the order of evaluation taken by lazy evaluation is difficult to predict by humans, it is much easier to trace the path of eager evaluation where arguments are reduced to normal form before being supplied to a function. But knowing that lazy evaluation always performs less reduction steps than eager evaluation (present the proof!), we can easily get an upper bound for the number of reductions by pretending that our function is evaluated eagerly.

Example:

```
or = foldr (||) False
isPrime n = not $ or $ map (\k -> n `mod` k == 0) [2..n-1]
```

=> eager evaluation always takes n steps, lazy won't take more than that. But it will actually take fewer.

Throwing away arguments

Time bound exact for functions that examine their argument to normal form anyway. The property that a function needs its argument can concisely be captured by denotational semantics:

```
f ⊥ = ⊥
```

Argument in WHNF only, though. Operationally: non-termination -> non-termination. (this is an approximation only, though because $f \text{ anything} = \perp$ doesn't "need" its argument). Non-strict functions don't need their argument and eager time bound is not sharp. But the information whether a function is strict or not can already be used to great benefit in the analysis.

```
isPrime n = not $ or $ (n `mod` 2 == 0) : (n `mod` 3 == 0) : ...
```

It's enough to know or `True ⊥ = True`.

Other examples:

- `foldr (:) []` vs. `foldl (flip (:)) []` with \perp .
- Can `head . mergesort` be analyzed only with \perp ? In any case, this example is too involved and belongs to Haskell/Laziness.

Persistence & Amortisation

NOTE: this section is better left to a data structures chapter because the subsections above cover most of the cases a programmer not focussing on data structures / amortization will encounter.

Persistence = no updates in place, older versions are still there. Amortisation = distribute unequal running times across a sequence of operations. Both don't go well together in a strict setting. Lazy evaluation can reconcile them. Debit invariants. Example: incrementing numbers in binary representation.

Implementation of Graph reduction

Smalltalk about G-machines and such. Main definition:

closure = thunk = code/data pair on the heap. What do they do? Consider $(\lambda x. \lambda y. x + y)2$. This is a function that returns a function, namely $\lambda y. 2 + y$ in this case. But when you want to compile code, it's prohibitive to actually perform the substitution in memory and replace all occurrences of x by 2 . So, you return a closure that consists of the function code $\lambda y. x + y$ and an environment $\{x = 2\}$ that assigns values to the free variables appearing in there.

GHC (? , most Haskell implementations?) avoid free variables completely and use supercombinators. In other words, they're supplied as extra-parameters and the observation that lambda-expressions with too few parameters don't need to be reduced since their WHNF is not very different.

Note that these terms are technical terms for implementation stuff, lazy evaluation happily lives without them. Don't use them in any of the sections above.

References

1. ↑ At least as far as types are concerned, but we're trying to avoid that word :)
2. ↑ More technically, `fst` and `snd` have types which limit them to pairs. It would be impossible to define projection functions on tuples in general, because they'd have to be able to accept tuples of different sizes, so the type of the function would vary.
3. ↑ In fact, these are one and the same concept in Haskell.
4. ↑ This isn't quite what `chr` and `ord` do, but that description fits our purposes well, and it's close enough.
5. ↑ To make things even more confusing, there's actually even more than one type for integers! Don't worry, we'll come on to this in due course.
6. ↑ This has been somewhat simplified to fit our purposes. Don't worry, the essence of the function is there.
7. ↑ Some of the newer type system extensions to GHC *do* break this, however, so you're better off just always putting down types anyway.
8. ↑ This is a slight lie. That type signature would mean that you can compare two values of any type whatsoever, but this clearly isn't true: how can you see if two functions are equal? Haskell includes a kind of 'restricted polymorphism' that allows type variables to range over some, but not all types. Haskell implements this using *type classes*, which we'll learn about later. In this case, the correct type of `(==)` is `Eq a => a -> a -> Bool`.
9. ↑ In mathematics, $n!$ normally means the factorial of n , but that syntax is impossible in Haskell, so we don't use it here.
10. ↑ Actually, defining the factorial of 0 to be 1 is not just arbitrary; it's because the factorial of 0 represents an empty product.
11. ↑ This is no coincidence; without mutable variables, recursion is the only way to implement control structures. This might sound like a limitation until you get used to it (it isn't, really).
12. ↑ Actually, it's using a function called `foldl`, which actually does the recursion.
13. ↑ Moggi, Eugenio (1991). "Notions of Computation and Monads". *Information and Computation* **93** (1).
14. ↑ w:Philip Wadler. Comprehending Monads (<http://citeseer.ist.psu.edu/wadler92comprehending.html>) . Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice. 1990.
15. ↑ w:Philip Wadler. The Essence of Functional Programming (<http://citeseer.ist.psu.edu/wadler92essence.html>) . Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1992.
16. ↑ Simon Peyton Jones, Philip Wadler (1993). "Imperative functional programming" (<http://homepages.inf.ed.ac.uk/wadler/topics/monads.html#imperative>) . *20'th Symposium on Principles of Programming Languages*.
17. ↑ It infers a monomorphic type because `κ` is bound by a lambda expression, and things bound by lambdas always have monomorphic types. See Polymorphism.
18. ↑ Ian Stewart. *The true story of how Theseus found his way out of the labyrinth*. Scientific American, February 1991, page 137.
19. ↑ Gérard Huet. *The Zipper*. Journal of Functional Programming, 7 (5), Sept 1997, pp. 549--554. PDF (<http://www.st.cs.uni-sb.de/edu/seminare/2005/advanced-fp/docs/huet-zipper.pdf>)
20. ↑ Note the notion of *zipper* as coined by Gérard Huet also allows to replace whole subtrees even if there is no extra data associated with them. In the case of our labyrinth, this is irrelevant. We will come back to this in the section Differentiation of data types.
21. ↑ Of course, the second topmost node or any other node at most a constant number of links away from the top will do as well.
22. ↑ Note that changing the whole data structure as opposed to updating the data at a node can be achieved in amortized constant time even if more nodes than just the top node is affected. An example is incrementing a number in binary representation. While incrementing say `111...11` must touch all digits to yield `1000...00`, the increment function nevertheless runs in constant amortized time (but not in constant worst case time).
23. ↑ Conor Mc Bride. *The Derivative of a Regular Type is its Type of One-Hole Contexts*. Available online. PDF (<http://www.cs.nott.ac.uk/~ctm/diff.pdf>)
24. ↑ This phenomenon already shows up with generic tries.

25. ↑ Actually, we can apply them to functions whose type is forall a. a -> R, for some arbitrary R, as these accept values of any type as a parameter. Examples of such functions: id, const k for any k. So technically, we can't do anything `_useful_` with its elements.
26. ↑ In fact, there are no written down and complete denotational semantics of Haskell. This would be a tedious task void of additional insight and we happily embrace the folklore and common sense semantics.
27. ↑ Monads are one of the most successful ways to give denotational semantics to imperative programs. See also Haskell/Advanced monads.
28. ↑ Strictness as premature evaluation of function arguments is elaborated in the chapter Graph Reduction.
29. ↑ The term *Laziness* comes from the fact that the prevalent implementation technique for non-strict languages is called *lazy evaluation*
30. ↑ The term *lifted* is somewhat overloaded, see also Unboxed Types.
31. ↑ S. Peyton Jones, A. Reid, T. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. (<http://research.microsoft.com/~simonpj/Papers/imp-precise-exn.htm>) In Programming Languages Design and Implementation. ACM press, May 1999.
32. ↑ John C. Reynolds. *Polymorphism is not set-theoretic*. INRIA Rapports de Recherche No. 296. May 1984.
33. ↑ Actually, there is a subtlety here: because `(.)` is a lazy function, if `f` is undefined, we have that `id . f = _ -> _|_`. Now, while this may seem equivalent to `_|_` for all extents and purposes, you can actually tell them apart using the strictifying function `seq`, meaning that the last category law is broken. We can define a new strict composition function, `f .! g = ((.) $! f) $! g`, that makes **Hask** a category. We proceed by using the normal `(.)`, though, and attribute any discrepancies to the fact that `seq` breaks an awful lot of the nice language properties anyway.
34. ↑ Experienced category theorists will notice that we're simplifying things a bit here; instead of presenting *unit* and *join* as natural transformations, we treat them explicitly as morphisms, and require naturality as extra axioms alongside the the standard monad laws (laws 3 and 4). The reasoning is simplicity; we are not trying to teach category theory as a whole, simply give a categorical background to some of the structures in Haskell. You may also notice that we are giving these morphisms names suggestive of their Haskell analogues, because the names η and μ don't provide much intuition.
35. ↑ This is perhaps due to the fact that Haskell programmers like to think of monads as a way of sequencing computations with a common feature, whereas in category theory the container aspect of the various structures is emphasised. `join` pertains naturally to containers (squashing two layers of a container down into one), but `(>>=)` is the natural sequencing operation (do something, feeding its results into something else).
36. ↑ If you can prove that certain laws hold, which we'll explore in the next section.
37. ↑ John Maraist, Martin Odersky, and Philip Wadler (May 1998). "The call-by-need lambda calculus" (<http://homepages.inf.ed.ac.uk/wadler/topics/call-by-need.html#need-journal>) . *Journal of Functional Programming* **8** (3): 257-317.
 - Bird, Richard (1998). *Introduction to Functional Programming using Haskell*. Prentice Hall. ISBN 0-13-484346-0.
 - Peyton-Jones, Simon (1987). *The Implementation of Functional Programming Languages* (<http://research.microsoft.com/~simonpj/papers/slpj-book-1987/>) . Prentice Hall.

Laziness



Hard work pays off later. Laziness pays off now! – Steven Wright



Introduction

By now you are aware that Haskell uses lazy evaluation in the sense that nothing is evaluated until necessary. The problem is what exactly does "until necessary" mean? In this chapter, we will see how lazy evaluation works (how little black magic there is), what exactly it means for functional programming and how to make the best use of it. But first, let's consider for having lazy evaluation. At first glance, it is tempting to think that lazy evaluation is meant to make programs more efficient. After all, what can be more efficient than not doing anything? This is only true in a superficial sense. Besides, in practice, laziness often introduces an overhead that leads programmers to hunt for places where they can make their code stricter. The real benefit of laziness is not merely that it makes things efficient, but that *it makes the right things efficient enough*. Lazy evaluation allows us to write simple, elegant code which would simply not be practical in a strict environment.

Nonstrictness versus Laziness

There is a slight difference between *laziness* and *nonstrictness*. **Nonstrict semantics** refers to a given property of Haskell programs that you can rely on: nothing will be evaluated until it is needed. **Lazy evaluation** is how you implement nonstrictness, using a device called **thunks** which we explain in the next section. However, these two concepts are so closely linked that it is beneficial to explain them both together: a knowledge of thunks is useful for understanding nonstrictness, and the semantics of nonstrictness explains why you would be using lazy evaluation in the first place. As such, we introduce the concepts simultaneously and make no particular effort to keep them from intertwining, with the exception of getting the terminology right.

Thunks and Weak head normal form

There are two principles you need to understand to get how programs execute in Haskell. Firstly, we have the property of nonstrictness: we evaluate as little as possible for as long as possible. Secondly, Haskell values are highly layered; 'evaluating' a Haskell value could mean evaluating down to any one of these layers. To see what this means, let's walk through a few examples using a pair.

```
-----  
let (x, y) = (length [1..5], reverse "olleh") in ...  
-----
```

(We'll assume that in the 'in' part, we use x and y somewhere. Otherwise, we're not forced to evaluate the let-binding at all; the right-hand side could have been `undefined` and it would still work if the 'in' part doesn't mention x or y . This assumption will remain for all the examples in this section.) What do we know about x ? Looking at it we can see it's pretty obvious x is 5 and y "hello", but remember the first principle: we don't want to evaluate the calls to `length` and `reverse` until we're forced to. So okay, we can say that x and y are both **thunks**: that is, they are *unevaluated values* with a *recipe* that explains how to evaluate them. For example, for x this recipe says 'Evaluate `length [1..5]`'. However, we are actually doing some pattern matching on the left hand side. What would happen if we removed that?

```
-----  
let z = (length [1..5], reverse "olleh") in ...  
-----
```

Although it's still pretty obvious to us that z is a pair, the compiler sees that we're not trying to deconstruct the

value on the right-hand side of the '=' sign at all, so it doesn't really care what's there. It lets z be a thunk on its own. Later on, when we try to use z , we'll probably need one or both of the components, so we'll have to evaluate z , but for now, it can be a thunk.

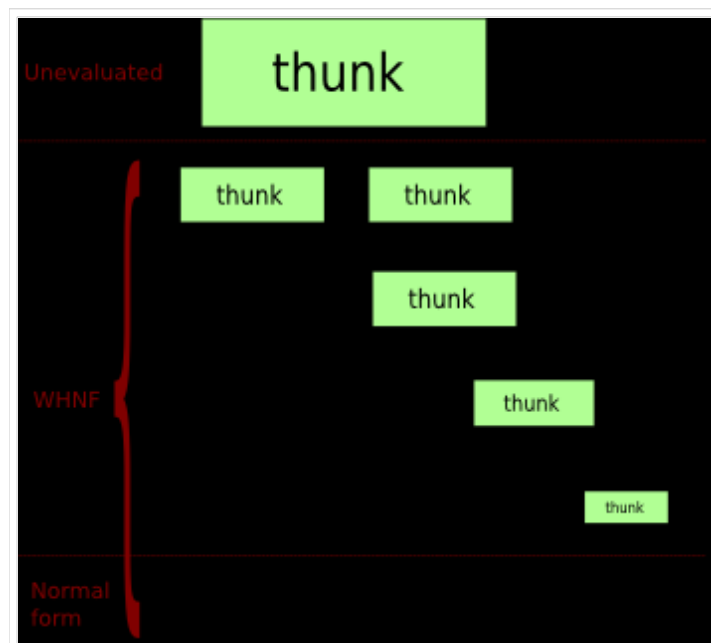
Above, we said Haskell values were layered. We can see that at work if we pattern match on z :

```
let z      = (length [1..5], reverse "olleh")
    (n, s) = z
in ...
```

After the first line has been executed, z is simply a thunk. We know nothing about the sort of value it is because we haven't been asked to find out yet. In the second line, however, we pattern match on z using a pair pattern. The compiler thinks 'I better make sure that pattern does indeed match z , and in order to do that, I need to make sure z is a pair.' Be careful, though. We're not as of yet doing anything with the component parts (the calls to `length` and `reverse`), so they can remain unevaluated. In other words, z , which was just a thunk, gets evaluated to something like $(\text{*thunk*}, \text{*thunk*})$, and n and s become thunks which, when evaluated, will be the component parts of the original z .

Let's try a slightly more complicated pattern match:

```
let z      = (length [1..5], reverse "olleh")
    (n, s) = z
    'h':ss = s
in ...
```



Evaluating the value $(4, [1, 2])$ step by step. The first stage is completely unevaluated; all subsequent forms are in WHNF, and the last one is also in normal form.

The pattern match on the second component of z causes some evaluation. The compiler wishes to check that the `'h':ss` pattern matches the second component of the pair. So, it:

- ▮ Evaluates the top level of s to ensure it's a cons cell: $s = \text{*thunk*} : \text{*thunk*}$. (If s had been an empty list we would encounter a pattern match failure error at this point.)
- ▮ Evaluates the first thunk it just revealed to make sure it's 'h': $s = \text{'h'} : \text{*thunk*}$
- ▮ The rest of the list stays unevaluated, and ss becomes a thunk which, when evaluated, will be the rest of this list.

So it seems that we can 'partially evaluate' (most) Haskell values. Also, there is some sense of the minimum amount of evaluation we can do. For example, if we have a pair thunk, then the minimum amount of evaluation takes us to the pair constructor with two unevaluated components:

$(\text{*thunk*}, \text{*thunk*})$. If we have a list, the

minimum amount of evaluation takes us either to a cons cell $\text{*thunk*} : \text{*thunk*}$ or an empty list $[]$. Note that in the second case, no more evaluation can be performed on the value; it is said to be in **normal form**. If we are at any of the intermediate steps so that we've performed at least some evaluation on a value, it is in **weak head**

normal form (WHNF). (There is also a 'head normal form', but it's not used in Haskell.) *Fully* evaluating something in WHNF reduces it to something in normal form; if at some point we needed to, say, print `z` out to the user, we'd need to fully evaluate it to, including those calls to `length` and `reverse`, to `(5, "hello")`, where it is in normal form. Performing any degree of evaluation on a value is sometimes called **forcing** that value.

Note that for some values there is only one. For example, you can't partially evaluate an integer. It's either a thunk or it's in normal form. Furthermore, if we have a constructor with strict components (annotated with an exclamation mark, as with `data MaybeS a = NothingS | JustS !a`), these components become evaluated as soon as we evaluate the level above. I.e. we can never have `JustS *thunk*`, as soon as we get to this level the strictness annotation on the component of `JustS` forces us to evaluate the component part.

So in this section we've explored the basics of laziness. We've seen that nothing gets evaluated until its needed (in fact the *only* place that Haskell values get evaluated is in pattern matches, and inside certain primitive IO functions), and that this principle even applies to evaluating values — we do the minimum amount of work on a value that we need to compute our result.

Lazy and strict functions

Functions can be lazy or strict 'in an argument'. Most functions need to do something with their arguments, and this will involve evaluating these arguments to different levels. For example, `length` needs to evaluate only the cons cells in the argument you give it, not the contents of those cons cells — `length *thunk*` might evaluate to something like `length (*thunk* : *thunk* : *thunk* : [])`, which in turn evaluates to 3. Others need to evaluate their arguments fully, like `show`. If you had `show *thunk*`, there's no way you can do anything other than evaluate that thunk to normal form.

So some functions evaluate their arguments more fully than others. Given two functions of one parameter, `f` and `g`, we say `f` is stricter than `g` if `f x` evaluates `x` to a deeper level than `g x`. Often we only care about WHNF, so a function that evaluates its argument to at least WHNF is called *strict* and one that performs no evaluation is *lazy*. What about functions of more than one parameter? Well, we can talk about functions being strict in one parameter, but lazy in another. For example, given a function like the following:

```
f x y = show x
```

Clearly we need to perform no evaluation on `y`, but we need to evaluate `x` fully to normal form, so `f` is strict in its first parameter but lazy in its second.

Exercises

1. Why must we fully evaluate `x` to normal form in `f x y = show x`?
2. Which is the stricter function?

```
f x = length [head x]
g x = length (tail x)
```

TODO: explain that it's also about how much of the input we need to consume before we can start producing output. E.g. `foldr (:) []` and `foldl (flip (:)) []` both evaluate their arguments to the same level of strictness, but

foldr can start producing values straight away, whereas *foldl* needs to evaluate cons cells all the way to the end before it starts anything.

Black-box strictness analysis

Imagine we're given some function f which takes a single parameter. We're not allowed to look at its source code, but we want to know whether f is strict or not. How might we do this? Probably the easiest way is to use the standard Prelude value `undefined`. Forcing `undefined` to any level of evaluation will halt our program and print an error, so all of these print errors:

```
-----
let (x, y) = undefined in x
!length undefined
!head undefined
!JustS undefined -- Using MaybeS as defined in the last section
-----
```

So if a function is strict, passing it `undefined` will result in an error. Were the function lazy, passing it `undefined` will print no error and we can carry on as normal. For example, none of the following produce errors:

```
-----
let (x, y) = (4, undefined) in x
!length [undefined, undefined]
!head (4 : undefined)
!Just undefined
-----
```

So we can say that f is a strict function if, and only if, f `undefined` results in an error being printed and the halting of our program.

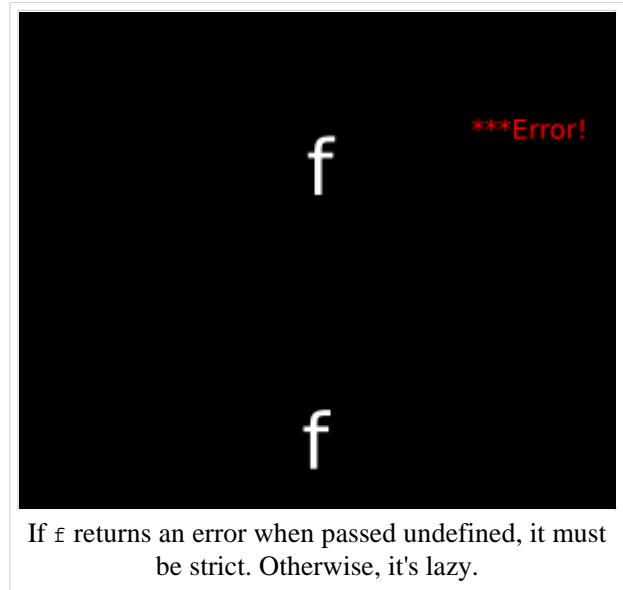
In the context of nonstrict semantics

What we've presented so far makes sense until you start to think about functions like `id`. Is `id` strict? Our gut reaction is probably to say "No! It doesn't evaluate its argument, therefore it's lazy". However, let's apply our black-box strictness analysis from the last section to `id`. Clearly, `id undefined` is going to print an error and halt our program, so shouldn't we say that `id` is strict? The reason for this mixup is that Haskell's nonstrict semantics makes the whole issue a bit murkier.

Nothing gets evaluated if it doesn't need to be, according to nonstrictness. In the following code, will `length undefined` be evaluated?

```
-----
[4, 10, length undefined, 12]
-----
```

If you type this into `GHCi`, it seems so, because you'll get an error printed. However, our question was something of a trick one; it doesn't make sense to say whether a value got evaluated, unless we're doing something to this value. Think about it: if we type in `head [1, 2, 3]` into `GHCi`, the only reason we have to do any evaluation whatsoever is because `GHCi` has to print us out the result. Typing `[4, 10, length undefined,`



12] again requires GHCi to print that list back to us, so it must evaluate it to normal form. You can think of anything you type into GHCi as being passed to `show`. In your average Haskell program, nothing at all will be evaluated until we come to perform the IO in `main`. So it makes no sense to say whether something is evaluated or not unless we know what it's being passed to, one level up.

So when we say "Does `f x` force `x`?" what we really mean is "Given that we're forcing `f x`, does `x` get forced as a result?". Now we can turn our attention back to `id`. If we force `id x` to normal form, then `x` will be forced to normal form, so we conclude that `id` is strict. `id` itself doesn't evaluate its argument, it just hands it on to the caller who will. One way to see this is in the following code:

```
-----
-- We evaluate the right-hand of the let-binding to WHNF by pattern-matching
-- against it.
let (x, y) = undefined in x -- Error, because we force undefined.
let (x, y) = id undefined in x -- Error, because we force undefined.
-----
```

`id` doesn't "stop" the forcing, so it is strict. Contrast this to a clearly lazy function, `const (3, 4)`:

```
-----
let (x, y) = undefined in x -- Error, because we force undefined.
let (x, y) = const (3, 4) undefined -- No error, because const (3, 4) is lazy.
-----
```

The denotational view on things

If you're familiar with denotational semantics (perhaps you've read the wikibook chapter on it?), then the strictness of a function can be summed up very succinctly:

$$f \perp = \perp \iff f \text{ is strict}$$

Assuming that you say that everything with type `forall a. a`, including `undefined`, `error "any string"`, `throw` and so on, has denotation \perp .

Lazy pattern matching

You might have seen pattern matches like the following in Haskell sources.

Example: A lazy pattern match

```
-----
-- From Control.Arrow
let (***) f g ~(x, y) = (f x, g y)
-----
```



The question is: what does the tilde sign (`~`) mean in the above pattern match? `~` makes a *lazy pattern* or *irrefutable pattern*. Normally, if you pattern match using a constructor as part of the pattern, you have to evaluate any argument passed into that function to make sure it matches the pattern. For example, if you had a function like the above, the third argument would be evaluated when you call the function to make sure the value matches the pattern. (Note that the first and second arguments won't be evaluated, because the patterns `f` and `g` match anything. Also it's worth noting that the *components* of the tuple won't be evaluated: just the 'top level'. Try `let f (Just x) = 1 in f (Just undefined)` to see the this.)

However, prepending a pattern with a tilde sign delays the evaluation of the value until the component parts are actually used. But you run the risk that the value might not match the pattern -- you're telling the compiler 'Trust me, I know it'll work out'. (If it turns out it doesn't match the pattern, you get a runtime error.) To illustrate the difference:

Example: How ~ makes a difference



```

Prelude> let f (x,y) = 1 in f undefined
!Undefined
Prelude> let f ~(x,y) = 1 in f undefined
!1

```

In the first example, the value is evaluated because it has to match the tuple pattern. You evaluate `undefined` and get `undefined`, which stops the proceedings. In the latter example, you don't bother evaluating the parameter until it's needed, which turns out to be never, so it doesn't matter you passed it `undefined`. To bring the discussion around in a circle back to `(***)`:

Example: How ~ makes a difference with (***)



```

Prelude> (const 1 *** const 2) undefined
!(1,2)

```

If the pattern weren't irrefutable, the example would have failed.

When does it make sense to use lazy patterns?

Essentially, when you only have the single constructor for the type, e.g. tuples. Multiple equations won't work nicely with irrefutable patterns. To see this, let's examine what would happen were we to make `head` have an irrefutable pattern:

Example: Lazier head



```

head' :: [a] -> a
head' ~[]      = undefined
head' ~(x:xs) = x

```

The fact we're using one of these patterns tells us not to evaluate even the top level of the argument until absolutely necessary, so we don't know whether it's an empty list or a cons cell. As we're using an *irrefutable* pattern for the first equation, this will match, and the function will always return `undefined`.

Exercises

- Why won't changing the order of the equations to `head'` help here?
- More to come...

Techniques with Lazy Evaluation

This section needs a better title and is intended to be the workhorse of this chapter.

Separation of concerns without time penalty

Examples:

```

or = foldr (||) False
isSubstringOf x y = any (isPrefixOf x) (tails y)
take n . quicksort
take n . mergesort
prune . generate

```

The more examples, the better!

What about the case of (large data -> small data) where lazy evaluation is space-hungry but doesn't take less reductions than eager evaluation? Mention it here? Elaborate it in Haskell/Strictness?

xs++ys

`xs ++ ys` is $O(\min(\text{length } xs, k))$ where k is the length of the part of the result which you observe. This follows directly from the definition of `(++)` and laziness.

```

[] ++ ys = ys           -- case 1
(x:xs) ++ ys = x : (xs ++ ys) -- case 2

```

Let's try it in a specific case, completely expanding the definition:

```

[1,2,3] ++ ys
= 1 : ([2,3] ++ ys)           -- by case 2
= 1 : (2 : ([3] ++ ys))      -- by case 2
= 1 : (2 : (3 : ([ ] ++ ys))) -- by case 2
= 1 : (2 : (3 : ys))         -- by case 1

```

Here, the length of the left list was 3, and it took 4 steps to completely reduce the definition of `(++)`. As you can see, the length and content of `ys` actually doesn't matter at all, as it just ends up being a tail of the resulting list. You can see fairly easily that it will take `length xs + 1` steps to completely expand the definition of `(++)` in `xs ++ ys` in general. However, this won't actually happen until you go about actually using those elements of the list. If only the first k elements of the list are demanded, where $k \leq \text{length } xs$, then they will be available after

only k steps, so indeed,

```
head (xs ++ ys)
```

(or getting any constant number of elements from the head) will evaluate in constant time.

`isSubstringOf`

TODO:rewrite introduction to this section / now redundant with main intro

Often code reuse is far better.

Here's a simple example:

Example: Laziness helps code reuse

```

-- From the Prelude
or = foldr (||) False
any p = or . map p

-- From Data.List
isPrefixOf [] _ = True
isPrefixOf _ [] = False
isPrefixOf (x:xs) (y:ys) = x == y && isPrefixOf xs ys

tails [] = [[]]
tails xss@(_:xs) = xss : tails xs

-- Our function
isSubstringOf x y = any (isPrefixOf x) (tails y)

```



Where `any`, `isPrefixOf` and `tails` are the functions taken from the `Data.List` library. This function determines if its first parameter, x occurs as a substring of its second, y . Read in a strict way, it forms the list of all the tails of y , then checks them all to see if any of them have x as a prefix. In a strict language, writing this function this way (relying on the already-written programs `any`, `isPrefixOf`, and `tails`) would be silly, because it would be far slower than it needed to be. You'd end up doing direct recursion again, or in an imperative language, a couple of nested loops. You might be able to get some use out of `isPrefixOf`, but you certainly wouldn't use `tails`. You might be able to write a usable shortcutting `any`, but it would be more work, since you wouldn't want to use `foldr` to do it.

Now, in a lazy language, all the shortcutting is done for you. You don't end up rewriting `foldr` to shortcut when you find a solution, or rewriting the recursion done in `tails` so that it will stop early again. You can reuse standard library code better. Laziness isn't just a constant-factor speed thing, it makes a qualitative impact on the code which it's reasonable to write. In fact, it's commonplace to define infinite structures, and then only use as much as is needed, rather than having to mix up the logic of constructing the data structure with code that determines whether any part is needed. Code modularity is increased, as laziness gives you more ways to chop up your code into small pieces, each of which does a simple task of generating, filtering, or otherwise manipulating data.

Why Functional Programming Matters (<http://www.md.chalmers.se/~rjmh/Papers/whyfp.html>) -- largely focuses

on examples where laziness is crucial, and provides a strong argument for lazy evaluation being the default.

Infinite Data Structures

Examples:

```
fibs = 1:1:zipWith (+) fibs (tail fibs)
"rock-scissors-paper" example from Bird&Wadler
prune . generate
```

Infinite data structures usually tie a knot, too, but the Sci-Fi-Explanation of that is better left to the next section. One could move the next section before this one but I think that infinite data structures are simpler than tying general knots

Tying the Knot

More practical examples?

```
repMin
```

Sci-Fi-Explanation: "You can borrow things from the future as long as you don't try to change them".

Advanced: the "Blueprint"-technique. Examples: the one from the haskellwiki, the one from the mailing list.

At first a pure functional language seems to have a problem with circular data structures. Suppose I have a data type like this:

```
data Foo a = Foo {value :: a; next :: Foo}
```

If I want to create two objects "x" and "y" where "x" contains a reference to "y" and "y" contains a reference to "x" then in a conventional language this is straightforward: create the objects and then set the relevant fields to point to each other:

```
-- Not Haskell code
x := new Foo;
y := new Foo;
x.value := 1;
x.next := y;
y.value := 2;
y.next := x;
```

In Haskell this kind of modification is not allowed. So instead we depend on lazy evaluation:

```
circularFoo :: Foo Int
circularFoo = x
  where
    x = Foo 1 y
    y = Foo 2 x
```

This depends on the fact that the "Foo" constructor is a function, and like most functions it gets evaluated lazily. Only when one of the fields is required does it get evaluated.

It may help to understand what happens behind the scenes here. When a lazy value is created, for example by a call to "Foo", the compiler generates an internal data structure called a "thunk" containing the function call and arguments. When the value of the function is demanded the function is called, as you would expect. But then the thunk data structure is replaced with the return value. Thus anything else that refers to that value gets it straight away without the need to call the function.

(Note that the Haskell language standard makes no mention of thunks: they are an implementation mechanism. From the mathematical point of view this is a straightforward example of mutual recursion)

So when I call "circularFoo" the result "x" is actually a thunk. One of the arguments is a reference to a second thunk representing "y". This in turn has a reference back to the thunk representing "x". If I then use the value "next x" this forces the "x" thunk to be evaluated and returns me a reference to the "y" thunk. If I use the value "next \$ next x" then I force the evaluation of both thunks. So now both thunks have been replaced with the actual "Foo" structures, referring to each other. Which is what we wanted.

This is most often applied with constructor functions, but it isn't limited just to constructors. You can just as readily write:

```
x = f y
y = g x
```

The same logic applies.

Memoization, Sharing and Dynamic Programming

Dynamic programming with immutable arrays. DP with other finite maps, Hinze's paper "Trouble shared is Trouble halved". Let-floating `\x-> let z = foo x in \y -> ...`

Conclusions about laziness

Move conclusions to the introduction?

- Can make qualitative improvements to performance!
- Can hurt performance in some other cases.
- Makes code simpler.
- Makes hard problems conceivable.
- Allows for separation of concerns with regard to generating and processing data.

References

- Laziness on the Haskell wiki (<http://www.haskell.org/haskellwiki/Performance/Laziness>)
- Lazy evaluation tutorial on the Haskell wiki (http://www.haskell.org/haskellwiki/Haskell/Lazy_Evaluation)

Strictness

Difference between strict and lazy evaluation

Strict evaluation, or eager evaluation, is an evaluation strategy where expressions are evaluated as soon as they are bound to a variable. For example, with strict evaluation, when $x = 3 * 7$ is read, $3 * 7$ is immediately computed and 21 is bound to x . Conversely, with lazy evaluation values are only computed when they are needed. In the example $x = 3 * 7$, $3 * 7$ isn't evaluated until it's needed, like if you needed to output the value of x .

Why laziness can be problematic

Lazy evaluation often involves objects called thunks. A thunk is a placeholder object, specifying not the data itself, but rather how to compute that data. An entity can be replaced with a thunk to compute that entity. When an entity is copied, whether or not it is a thunk doesn't matter - it's copied as is (on most implementations, a pointer to the data is created). When an entity is evaluated, it is first checked if it is thunk; if it's a thunk, then it is executed, otherwise the actual data is returned. It is by the magic of thunks that laziness can be implemented.

Generally, in the implementation the thunk is really just a pointer to a piece of (usually static) code, plus another pointer to the data the code should work on. If the entity computed by the thunk is larger than the pointer to the code and the associated data, then a thunk wins out in memory usage. But if the entity computed by the thunk is smaller, the thunk ends up using more memory.

As an example, consider an infinite length list generated using the expression `iterate (+ 1) 0`. The size of the list is infinite, but the code is just an add instruction, and the two pieces of data, 1 and 0, are just two Integers. In this case, the thunk representing that list takes much less memory than the actual list, which would take infinite memory.

However, as another example consider the number generated using the expression `4 * 13 + 2`. The value of that number is 54, but in thunk form it is a multiply, an add, and three numbers. In such a case, the thunk loses in terms of memory.

Often, the second case above will consume so much memory that it will consume the entire heap and force the garbage collector. This can slow down the execution of the program significantly. And that, in fact, is the reason why laziness can be problematic.

Strictness annotations

seq

DeepSeq

References

- Strictness on the Haskell wiki (<http://www.haskell.org/haskellwiki/Performance/Strictness>)

Algorithm complexity

Complexity Theory is the study of how long a program will take to run, depending on the size of its input. There are many good introductory books to complexity theory and the basics are explained in any good algorithms book. I'll keep the discussion here to a minimum.

The idea is to say how well a program scales with more data. If you have a program that runs quickly on very small amounts of data but chokes on huge amounts of data, it's not very useful (unless you know you'll only be working with small amounts of data, of course). Consider the following Haskell function to return the sum of the elements in a list:

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

How long does it take this function to complete? That's a very difficult question; it would depend on all sorts of things: your processor speed, your amount of memory, the exact way in which the addition is carried out, the length of the list, how many other programs are running on your computer, and so on. This is far too much to deal with, so we need to invent a simpler model. The model we use is sort of an arbitrary "machine step." So the question is "how many machine steps will it take for this program to complete?" In this case, it only depends on the length of the input list.

If the input list is of length 0, the function will take either 0 or 1 or 2 or some very small number of machine steps, depending exactly on how you count them (perhaps 1 step to do the pattern matching and 1 more to return the value 0). What if the list is of length 1. Well, it would take however much time the list of length 0 would take, plus a few more steps for doing the first (and only element).

If the input list is of length n , it will take however many steps an empty list would take (call this value y) and then, for each element it would take a certain number of steps to do the addition and the recursive call (call this number x). Then, the total time this function will take is $nx + y$ since it needs to do those additions n many times. These x and y values are called *constant* values, *since they are independent of n , and actually dependent* only on exactly how we define a machine step, so we really don't want to consider them all that important. Therefore, we say that the complexity of this `sum` function is $\mathcal{O}(n)$ (read "order n "). Basically saying something is $\mathcal{O}(n)$ means that for some constant factors x and y , the function takes $nx + y$ machine steps to complete.

Consider the following sorting algorithm for lists (commonly called "insertion sort"):

```
sort [] = []
sort [x] = [x]
sort (x:xs) = insert (sort xs)
  where insert [] = [x]
        insert (y:ys) | x <= y    = x : y : ys
                      | otherwise = y : insert ys
```

The way this algorithm works is as follow: if we want to sort an empty list or a list of just one element, we return them as they are, as they are already sorted. Otherwise, we have a list of the form $x:xs$. In this case, we sort `xs` and then want to insert `x` in the appropriate location. That's what the `insert` function does. It traverses the now-sorted tail and inserts `x` wherever it naturally fits.

Let's analyze how long this function takes to complete. Suppose it takes $f(n)$ steps to sort a list of length n . Then, in order to sort a list of n -many elements, we first have to sort the tail of the list first, which takes $f(n - 1)$ time. Then, we have to insert x into this new list. If x has to go at the end, this will take $\mathcal{O}(n - 1) = \mathcal{O}(n)$ steps. Putting all of this together, we see that we have to do $\mathcal{O}(n)$ amount of work $\mathcal{O}(n)$ many times, which means that the entire complexity of this sorting algorithm is $\mathcal{O}(n^2)$. Here, the squared is not a constant value, so we cannot throw it out.

What does this mean? Simply that for really long lists, the `sum` function won't take very long, but that the `sort` function will take quite some time. Of course there are algorithms that run much more slowly than simply $\mathcal{O}(n^2)$ and there are ones that run more quickly than $\mathcal{O}(n)$. (Also note that a $\mathcal{O}(n^2)$ algorithm may actually be much faster than a $\mathcal{O}(n)$ algorithm in practice, if it takes much less time to perform a single step of the $\mathcal{O}(n^2)$ algorithm.)

Consider the random access functions for lists and arrays. In the worst case, accessing an arbitrary element in a list of length n will take $\mathcal{O}(n)$ time (think about accessing the last element). However with arrays, you can access any element immediately, which is said to be in *constant* time, or $\mathcal{O}(1)$, which is basically as fast as any algorithm can go.

There's much more in complexity theory than this, but this should be enough to allow you to understand all the discussions in this tutorial. Just keep in mind that $\mathcal{O}(1)$ is faster than $\mathcal{O}(n)$ is faster than $\mathcal{O}(n^2)$, etc.

Optimising

Profiling

Concurrency

Concurrency

If you need concurrency in Haskell, you should be able to simply consult the docs for `Control.Concurrent.*` and `Control.Monad.STM`.

Example

Example: Downloading files in parallel




```
downloadFile :: URL -> IO ()
downloadFile = undefined

downloadFiles :: [URL] -> IO ()
downloadFiles = mapM_ (forkIO . downloadFile)
```

Choosing data structures

Haskell/Choosing data structures

Libraries Reference

Hierarchical libraries

Haskell has a rich and growing set of function libraries. They fall into several groups:

- The Standard Prelude (often referred to as just "the Prelude") is defined in the Haskell 98 standard and imported automatically to every module you write. This defines standard types such as strings, lists and numbers and the basic functions on them, such as arithmetic, `map` and `foldr`
- The Standard Libraries are also defined in the Haskell 98 standard, but you have to import them when you need them. The reference manuals for these libraries are at <http://www.haskell.org/onlinereport/>
- Since 1998 the Standard Libraries have been gradually extended, and the resulting de-facto standard is known as the Base libraries. The same set is available for both HUGS and GHC.
- Other libraries may be included with your compiler, or can be installed using the Cabal mechanism.

When Haskell 98 was standardised modules were given a flat namespace. This has proved inadequate and a hierarchical namespace has been added by allowing dots in module names. For backward compatibility the standard libraries can still be accessed by their non-hierarchical names, so the modules `List` and `Data.List` both refer to the standard list library.

For details of how to import libraries into your program, see [Modules and libraries](#). For an explanation of the Cabal system for packaging Haskell software see [Distributing your software with the Cabal](#).

Haddock Documentation

Library reference documentation is generally produced using the Haddock tool. The libraries shipped with GHC are documented using this mechanism. You can view the documentation at <http://www.haskell.org/ghc/docs/latest/html/libraries/index.html>, and if you have installed GHC then there should also be a local copy.

Haddock produces hyperlinked documentation, so every time you see a function, type or class name you can

click on it to get to the definition. The sheer wealth of libraries available can be intimidating, so this tutorial will point out the highlights.

One thing worth noting with Haddock is that types and classes are cross-referenced by instance. So for example in the `Data.Maybe` (<http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Maybe.html>) library the `Maybe` data type is listed as an instance of `Ord`:

```
Ord a => Ord (Maybe a)
```

This means that if you declare a type `Foo` is an instance of `Ord` then the type `Maybe Foo` will automatically be an instance of `Ord` as well. If you click on the word `Ord` in the document then you will be taken to the definition of the `Ord` class and its (very long) list of instances. The instance for `Maybe` will be down there as well.

Hierarchical libraries/Lists

The **List** datatype is the fundamental data structure in Haskell — this is the basic building-block of data storage and manipulation. In computer science terms it is a singly-linked list. In the hierarchical library system the `List` module is stored in `Data.List`; but this module only contains utility functions. The definition of list itself is integral to the Haskell language.

Theory

A singly-linked list is a set of values in a defined order. The list can only be traversed in one direction (ie, you cannot move back and forth through the list like tape in a cassette machine).

The list of the first 5 positive integers is written as

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

We can move through this list, examining and changing values, from left to right, but not in the other direction. This means that the list

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

is not just a trivial change in perspective from the previous list, but the result of significant computation ($O(n)$ in the length of the list).

Definition

The polymorphic list datatype can be defined with the following recursive definition:

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

The "base case" for this definition is `[]`, the empty list. In order to put something into this list, we use the `(:)` constructor

```
emptyList = []
oneElem = 1:[]
```

The `(:)` (pronounced *cons*) is right-associative, so that creating multi-element lists can be done like

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

or even just

```
manyElems' = [1,2,3,4,5]
```

Basic list usage

Prepending

It's easy to hard-code lists without `cons`, but run-time list creation will use `cons`. For example, to push an argument onto a simulated stack, we would use:

```
push :: Arg -> [Arg] -> [Arg]
push arg stack = arg:stack
```

Pattern-matching

If we want to examine the top of the stack, we would typically use a `peek` function. We can try pattern-matching for this.

```
peek :: [Arg] -> Maybe Arg
peek [] = Nothing
peek (a:as) = Just a
```

The `a` before the `cons` in the pattern matches the head of the list. The `as` matches the tail of the list. Since we don't actually want the tail (and it's not referenced anywhere else in the code), we can tell the compiler this explicitly, by using a wild-card match, in the form of an underscore:

```
peek (a:_) = Just a
```

List utilities

FIXME: is this not covered in the chapter on list manipulation?

Maps

Folds, unfolds and scans

Length, head, tail etc.

Hierarchical libraries/Randoms

Random examples

Here are a handful of uses of random numbers by example

Example: Ten random integer numbers



```
import System.Random

main = do
  gen <- getStdGen
  let ns = randoms gen :: [Int]
      print $ take 10 ns
```

There exists a global random number generator which is initialized automatically in a system dependent fashion. This generator is maintained in the IO monad and can be accessed with `getStdGen`. Once obtained getting random numbers out of a generator does not require the IO monad, i.e. a generator can be used in pure functions.

Alternatively one can get a generator by initializing it with an integer, using `mkStdGen`:

Example: Ten random floats using `mkStdGen`



```
import System.Random

randomList :: (Random a) => Int -> [a]
randomList seed = randoms (mkStdGen seed)

main :: IO ()
main = do print $ take 10 (randomList 42 :: [Float])
```

Running this script results in output like this:

```
[0.110407025,0.8453985,0.3077821,0.78138804,0.5242582,0.5196911,0.20084688,0.4794773,0.3240164,6.1566383e-2]
```

Example: Unsorting a list (imperfectly)



```

import Data.List ( sort )
import Data.Ord ( comparing )
import System.Random ( Random, RandomGen, randoms, getStdGen )

main :: IO ()
main =
  do gen    <- getStdGen
     interact $ unlines . unsort gen . lines

unsort :: g -> [x] -> [x]
unsort g es = map snd $ sortBy (comparing fst) $ zip rs es
  where rs = randoms g :: [Integer]

```

There's more to random number generation than `randoms`. You can, for example, use `random` (sans 's') to generate a random number from a low to a high range. See below for more ideas.

The Standard Random Number Generator

The Haskell standard random number functions and types are defined in the `Random` module (or `System.Random` if you use hierarchical modules). The definition is at <http://www.haskell.org/onlinereport/random.html>, but its a bit tricky to follow because it uses classes to make itself more general.

From the standard:

```

----- The RandomGen class -----
class RandomGen g where
  genRange :: g -> (Int, Int)
  next     :: g -> (Int, g)
  split    :: g -> (g, g)

----- A standard instance of RandomGen -----
data StdGen = ... -- Abstract

```

OK. This basically introduces `StdGen`, the standard random number generator "object". Its an instance of the `RandomGen` class just in case anyone wants to implement a different random number generator.

If you have `r :: StdGen` then you can say:

```
(x, r2) = next r
```

This gives you a random `Int` `x` and a new `StdGen` `r2`. The 'next' function is defined in the `RandomGen` class, and you can apply it to something of type `StdGen` because `StdGen` is an instance of the `RandomGen` class, as below.

From the Standard:

```

instance RandomGen StdGen where ...
instance Read     StdGen where ...
instance Show     StdGen where ...

```

This also says that you can convert a `StdGen` to and from a string, which is there as a handy way to save the state of the generator. (The dots are not Haskell syntax. They simply say that the Standard does not define an

implementation of these instances.)

From the Standard:

```
mkStdGen :: Int -> StdGen
```

This is the factory function for StdGen objects. Put in a seed, get out a generator.

The reason that the 'next' function also returns a new random number generator is that Haskell is a functional language, so no side effects are allowed. In most languages the random number generator routine has the hidden side effect of updating the state of the generator ready for the next call. Haskell can't do that. So if you want to generate three random numbers you need to say something like

```
let
  (x1, r2) = next r
  (x2, r3) = next r2
  (x3, r4) = next r3
```

The other thing is that the random values (x1, x2, x3) are random integers. To get something in the range, say, (0,999) you would have to take the modulus yourself, which is silly. There ought to be a library routine built on this, and indeed there is.

From the Standard:

```
----- The Random class -----
class Random a where
  randomR :: RandomGen g => (a, a) -> g -> (a, g)
  random  :: RandomGen g => g -> (a, g)

  randomRs :: RandomGen g => (a, a) -> g -> [a]
  randoms  :: RandomGen g => g -> [a]

  randomRIO :: (a,a) -> IO a
  randomIO  :: IO a
```

Remember that StdGen is the only instance of type RandomGen (unless you roll your own random number generator). So you can substitute StdGen for 'g' in the types above and get this:

```
randomR :: (a, a) -> StdGen -> (a, StdGen)
random  :: StdGen -> (a, StdGen)
```

```
randomRs :: (a, a) -> StdGen -> [a]
randoms  :: StdGen -> [a]
```

But remember that this is all inside *another* class declaration "Random". So what this says is that any instance of Random can use these functions. The instances of Random in the Standard are:

```
instance Random Integer where ...
instance Random Float   where ...
instance Random Double  where ...
instance Random Bool    where ...
instance Random Char    where ...
```

So for any of these types you can get a random range. You can get a random integer with:

```
(x1, r2) = randomR (0,999) r
```

And you can get a random upper case character with

```
(c2, r3) = randomR ('A', 'Z') r2
```

You can even get a random bit with

```
(b3, r4) = randomR (False, True) r3
```

So far so good, but threading the random number state through your entire program like this is painful, error prone, and generally destroys the nice clean functional properties of your program.

One partial solution is the "split" function in the RandomGen class. It takes one generator and gives you two generators back. This lets you say something like this:

```
(r1, r2) = split r
x = foo r1
```

In this case we are passing r1 down into function foo, which does something random with it and returns a result "x", and we can then take "r2" as the random number generator for whatever comes next. Without "split" we would have to write

```
(x, r2) = foo r1
```

But even this is often too clumsy, so you can do it the quick and dirty way by putting the whole thing in the IO monad. This gives you a standard global random number generator just like any other language. But because its just like any other language it has to do it in the IO monad.

From the Standard:

```
----- The global random generator -----
newStdGen    :: IO StdGen
setStdGen    :: StdGen -> IO ()
getStdGen    :: IO StdGen
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

So you could write:

```
foo :: IO Int
foo = do
  r1 <- getStdGen
  let (x, r2) = randomR (0,999) r1
      setStdGen r2
  return x
```

This gets the global generator, uses it, and then updates it (otherwise every random number will be the same). But having to get and update the global generator every time you use it is a pain, so its more common to use

`getStdRandom`. The argument to this is a function. Compare the type of that function to that of `'random'` and `'randomR'`. They both fit in rather well. To get a random integer in the IO monad you can say:

```
x <- getStdRandom $ randomR (1,999)
```

The `'randomR (1,999)'` has type `"StdGen -> (Int, StdGen)"`, so it fits straight into the argument required by `getStdRandom`.

Using QuickCheck to Generate Random Data

Only being able to do random numbers in a nice straightforward way inside the IO monad is a bit of a pain. You find that some function deep inside your code needs a random number, and suddenly you have to rewrite half your program as IO actions instead of nice pure functions, or else have an `StdGen` parameter tramp its way down there through all the higher level functions. Something a bit purer is needed.

If you have read anything about Monads then you might have recognized the pattern I gave above:

```
let
  (x1, r2) = next r
  (x2, r3) = next r2
  (x3, r4) = next r3
```

The job of a monad is to abstract out this pattern, leaving the programmer to write something like:

```
do -- Not real Haskell
  x1 <- random
  x2 <- random
  x3 <- random
```

Of course you can do this in the IO monad, but it would be better if random numbers had their own little monad that specialised in random computations. And it just so happens that such a monad exists. It lives in the `Test.QuickCheck` library, and it's called `"Gen"`. And it does lots of very useful things with random numbers.

The reason that `"Gen"` lives in `Test.QuickCheck` is historical: that is where it was invented. The purpose of `QuickCheck` is to generate random unit tests to verify properties of your code. (Incidentally its very good at this, and most Haskell developers use it for testing). See the `QuickCheck` (<http://www.cs.chalmers.se/~rjmh/QuickCheck>) homepage for more details. This tutorial will concentrate on using the `"Gen"` monad for generating random data.

Most Haskell compilers (including GHC) bundle `QuickCheck` in with their standard libraries, so you probably won't need to install it separately. Just say

```
import Test.QuickCheck
```

in your source file.

The `"Gen"` monad can be thought of as a monad of random computations. As well as generating random numbers it provides a library of functions that build up complicated values out of simple ones.

So lets start with a routine to return three random integers between 0 and 999:


```
randomTriple :: Gen (Integer, Integer, Integer)
randomTriple = do
  x1 <- choose (0,999)
  x2 <- choose (0,999)
  x3 <- choose (0,999)
  return (x1, x2, x3)
```

"choose" is one of the functions from QuickCheck. Its the equivalent to randomR. The type of "choose" is

```
choose :: Random a => (a, a) -> Gen a
```

In other words, for any type "a" which is an instance of "Random" (see above), "choose" will map a range into a generator.

Once you have a "Gen" action you have to execute it. The "generate" function executes an action and returns the random result. The type is:

```
generate :: Int -> StdGen -> Gen a -> a
```

The three arguments are:

1. The "size" of the result. This isn't used in the example above, but if you were generating a data structure with a variable number of elements (like a list) then this parameter lets you pass some notion of the expected size into the generator. We'll see an example later.
2. A random number generator.
3. The generator action.

So for example:

```
let
  triple = generate 1 (mkStdGen 1) randomTriple
```

will generate three arbitrary numbers. But note that because the same seed value is used the numbers will always be the same (which is why I said "arbitrary", not "random"). If you want different numbers then you have to use a different StdGen argument.

A common pattern in most programming languages is to use a random number generator to choose between two courses of action:

```
-- Not Haskell code
r := random (0,1)
if r == 1 then foo else bar
```

QuickCheck provides a more declaritive way of doing the same thing. If "foo" and "bar" are both generators returning the same type then you can say:

```
oneof [foo, bar]
```

This has an equal chance of returning either "foo" or "bar". If you wanted different odds, say that there was a 30% chance of "foo" and a 70% chance of "bar" then you could say

```
frequency [ (30, foo), (70, bar) ]
```

"oneof" takes a simple list of Gen actions and selects one of them at random. "frequency" does something similar, but the probability of each item is given by the associated weighting.

```
oneof :: [Gen a] -> Gen a
frequency :: [(Int, Gen a)] -> Gen a
```

General Practices

Applications

So you want to build a simple application -- a piece of standalone software -- with Haskell.

The Main module

The basic requirement behind this is to have a module **Main** with a main function **main**

```
-- thingamie.hs
module Main where

main = do
  putStrLn "Bonjour, world!"
```

Using GHC, you may compile and run this file as follows:

```
$ ghc --make -o bonjourWorld thingamie.hs
$ ./bonjourWorld
Bonjour, world!
```

Voilà! You now have a standalone application built in Haskell.

Other modules?

Invariably your program will grow to be complicated enough that you want to split it across different files. Here is an example of an application which uses two modules.

```
-- hello.hs
module Hello where

hello = "Bonjour, world!"
```

```
-- thingamie.hs
module Main where

import Hello

main = do
  putStrLn hello
```

We can compile this fancy new program in the same way. Note that the `--make` flag to `ghc` is rather handy because it tells `ghc` to automatically detect dependencies in the files you are compiling. That is, since `thingamie.hs` imports a module 'Hello', `ghc` will search the haskell files in the current directory for files that implement `Hello` and also compile that. If `Hello` depends on yet other modules, `ghc` will automatically detect those dependencies as well.

```
$ ghc --make -o bonjourWorld thingamie.hs
$ ./bonjourWorld
Bonjour, world!
```

If you want to search in other places for source files, including a nested structure of files and directories, you can add the starting point for the dependency search with the `-i` flag. This flag takes multiple, colon-separated directory names as its argument.

As a contrived example, the following program has three files all stored in a `src/` directory. The directory structure looks like:

```
HaskellProgram/
  src/
    Main.hs
    GUI/
      Interface.hs
    Functions/
      Mathematics.hs
```

The `Main` module imports its dependencies by searching a path analogous to the module name — so that **import GUI.Interface** would search for **GUI/Interface** (with the appropriate file extension).

To compile this program from within the `HaskellProgram` directory, invoke `ghc` with:

```
$ ghc --make -i src -o sillyprog Main.hs
```

Debugging/

Haskell/Debugging/

Testing

Quickcheck

Consider the following function:

```

getList = find 5 where
  find 0 = return []
  find n = do
    ch <- getChar
    if ch `elem` ['a'..'e'] then do
      tl <- find (n-1)
      return (ch : tl) else
    find n

```

How would we effectively test this function in Haskell? The solution we turn to is refactoring and QuickCheck.

Keeping things pure

The reason your `getList` is hard to test, is that the side effecting monadic code is mixed in with the pure computation, making it difficult to test without moving entirely into a "black box" IO-based testing model. Such a mixture is not good for reasoning about code.

Let's untangle that, and then test the referentially transparent parts simply with QuickCheck. We can take advantage of lazy IO firstly, to avoid all the unpleasant low-level IO handling.

So the first step is to factor out the IO part of the function into a thin "skin" layer:

```

-- A thin monadic skin layer
getList :: IO [Char]
getList = fmap take5 getContents

-- The actual worker
take5 :: [Char] -> [Char]
take5 = take 5 . filter (`elem` ['a'..'e'])

```

Testing with QuickCheck

Now we can test the 'guts' of the algorithm, the `take5` function, in isolation. Let's use QuickCheck. First we need an Arbitrary instance for the Char type -- this takes care of generating random Chars for us to test with. I'll restrict it to a range of nice chars just for simplicity:

```

import Data.Char
import Test.QuickCheck

instance Arbitrary Char where
  arbitrary = choose ('\32', '\128')
  coarbitrary c = variant (ord c `rem` 4)

```

Let's fire up GHCi (or Hugs) and try some generic properties (it's nice that we can use the QuickCheck testing framework directly from the Haskell REPL). An easy one first, a `[Char]` is equal to itself:

```

>*A> quickCheck ((\s -> s == s) :: [Char] -> Bool)
OK, passed 100 tests.

```

What just happened? QuickCheck generated 100 random `[Char]` values, and applied our property, checking the result was True for all cases. QuickCheck *generated the test sets for us!*

A more interesting property now: reversing twice is the identity:

```
*A> quickCheck ((\s -> (reverse.reverse) s == s) :: [Char] -> Bool)
OK, passed 100 tests.
```

Great!

Testing take5

The first step to testing with QuickCheck is to work out some properties that are true of the function, for all inputs. That is, we need to find *invariants*.

A simple invariant might be: $\forall s . \text{length} (\text{take5 } s) = 5$

So let's write that as a QuickCheck property:

```
\s -> length (take5 s) == 5
```

Which we can then run in QuickCheck as:

```
*A> quickCheck (\s -> length (take5 s) == 5)
Falsifiable, after 0 tests:
""
```

Ah! QuickCheck caught us out. If the input string contains less than 5 filterable characters, the resulting string will be less than 5 characters long. So let's weaken the property a bit: $\forall s . \text{length} (\text{take5 } s) \leq 5$

That is, take5 returns a string of at most 5 characters long. Let's test this:

```
*A> quickCheck (\s -> length (take5 s) <= 5)
OK, passed 100 tests.
```

Good!

Another property

Another thing to check would be that the correct characters are returned. That is, for all returned characters, those characters are members of the set ['a','b','c','d','e'].

We can specify that as: $\forall s . \forall e . e \in \text{take5 } s \rightarrow e \in [abcde]$

And in QuickCheck:

```
*A> quickCheck (\s -> all (`elem` ['a'..'e']) (take5 s))
OK, passed 100 tests.
```

Excellent. So we can have some confidence that the function neither returns strings that are too long, nor includes invalid characters.

Coverage

One issue with the default QuickCheck configuration, when testing [Char], is that the standard 100 tests isn't enough for our situation. In fact, QuickCheck never generates a String greater than 5 characters long, when using the supplied Arbitrary instance for Char! We can confirm this:

```
*A> quickCheck (\s -> length (take5 s) < 5)
OK, passed 100 tests.
```

QuickCheck wastes its time generating different Chars, when what we really need is longer strings. One solution to this is to modify QuickCheck's default configuration to test deeper:

```
deepCheck p = check (defaultConfig { configMaxTest = 10000}) p
```

This instructs the system to find at least 10000 test cases before concluding that all is well. Let's check that it is generating longer strings:

```
*A> deepCheck (\s -> length (take5 s) < 5)
Falsifiable, after 125 tests:
";:iD^*NNi~Y\\RegMob\DEL@krsx/=dcf7kub|EQi\DELD*"
```

We can check the test data QuickCheck is generating using the 'verboseCheck' hook. Here, testing on integers lists:

```
*A> verboseCheck (\s -> length s < 5)
#0: []
#1: [0]
#2: []
#3: []
#4: []
#5: [1,2,1,1]
#6: [2]
#7: [-2,4,-4,0,0]
Falsifiable, after 7 tests:
#[-2,4,-4,0,0]
```

More information on QuickCheck

- http://haskell.org/haskellwiki/Introduction_to_QuickCheck
- http://haskell.org/haskellwiki/QuickCheck_as_a_test_set_generator

HUnit

Sometimes it is easier to give an example for a test and to define one from a general rule. HUnit provides a unit testing framework which helps you to do just this. You could also abuse QuickCheck by providing a general rule which just so happens to fit your example; but it's probably less work in that case to just use HUnit.

TODO: give an example of HUnit test, and a small tour of it

More details for working with HUnit can be found in its user's guide (<http://hunit.sourceforge.net/HUnit-1.0/Guide.html>) .



At least part of this page was imported from the Haskell wiki article [Introduction to QuickCheck](http://www.haskell.org/haskellwiki/Introduction_to_QuickCheck) (http://www.haskell.org/haskellwiki/Introduction_to_QuickCheck) , in accordance to its Simple Permissive License. If you wish to modify this page and if your changes will also be useful on that wiki, you might consider modifying that source page instead of this one, as changes from that page may propagate here, but not the other way around. Alternately, you can explicitly dual license your contributions under the Simple Permissive License.

Packaging

A guide to the best practice for creating a new Haskell project or program.

Recommended tools

Almost all new Haskell projects use the following tools. Each is intrinsically useful, but using a set of common tools also benefits everyone by increasing productivity, and you're more likely to get patches.

Revision control

Use darcs (<http://darcs.net>) unless you have a specific reason not to. It's much more powerful than most competing systems, it's written in Haskell, and it's the standard for Haskell developers. See the wikibook [Understanding darcs](#) to get started.

Build system

Use Cabal (<http://haskell.org/cabal>) . You should read at least the start of section 2 of the Cabal User's Guide (<http://www.haskell.org/ghc/docs/latest/html/Cabal/index.html>) .

Documentation

For libraries, use Haddock (<http://haskell.org/haddock>) . We recommend using recent versions of haddock (0.8 or above).

Testing

Pure code can be tested using QuickCheck (<http://www.md.chalmers.se/~rjmh/QuickCheck/>) or SmallCheck (<http://www.mail-archive.com/haskell@haskell.org/msg19215.html>) , impure code with HUnit (<http://hunit.sourceforge.net/>) .

To get started, try Haskell/Testing. For a slightly more advanced introduction, Simple Unit Testing in Haskell (<http://blog.codersbase.com/2006/09/01/simple-unit-testing-in-haskell/>) is a blog article about creating a testing framework for QuickCheck using some Template Haskell.

Structure of a simple project

The basic structure of a new Haskell project can be adopted from HNop (<http://semantic.org/hnop/>), the minimal Haskell project. It consists of the following files, for the mythical project "haq".

- Haq.hs -- the main haskell source file
- haq.cabal -- the cabal build description
- Setup.hs -- build script itself
- _darcs -- revision control
- README -- info
- LICENSE -- license

You can of course elaborate on this, with subdirectories and multiple modules.

Here is a transcript on how you'd create a minimal darcs-using and cabalised Haskell project, for the cool new Haskell program "haq", build it, install it and release.

The new tool 'mkcabal' automates all this for you, but it's important that you understand all the parts first.

We will now walk through the creation of the infrastructure for a simple Haskell executable. Advice for libraries follows after.

Create a directory

Create somewhere for the source:

```
┌-----┐  
|$ mkdir haq  
|$ cd haq  
└-----┘
```

Write some Haskell source

Write your program:

```
┌-----┐  
|$ cat > Haq.hs  
|--  
|-- Copyright (c) 2006 Don Stewart - http://www.cse.unsw.edu.au/~dons  
|-- GPL version 2 or later (see http://www.gnu.org/copyleft/gpl.html)  
|--  
|import System.Environment  
|--  
|-- 'main' runs the main program  
|main :: IO ()  
|main = getArgs >>= print . haqify . head  
|haqify s = "Haq! " ++ s  
└-----┘
```

Stick it in darcs

Place the source under revision control:


```

-----
i$ darcs init
i$ darcs add Haq.hs
i$ darcs record
i$ addfile ./Haq.hs
iShall I record this change? (1/?) [ynWsfqadjkc], or ? for help: y
iYunk ./Haq.hs 1
i+---
i+--- Copyright (c) 2006 Don Stewart - http://www.cse.unsw.edu.au/~dons
i+--- GPL version 2 or later (see http://www.gnu.org/copyleft/gpl.html)
i+---
i+import System.Environment
i+
i+-- | 'main' runs the main program
i+main :: IO ()
i+main = getArgs >>= print . haqify . head
i+
i+haqify s = "Haq! " ++ s
iShall I record this change? (2/?) [ynWsfqadjkc], or ? for help: y
iWhat is the patch name? Import haq source
iDo you want to add a long comment? [yn]n
iFinished recording patch 'Import haq source'
-----

```

And we can see that darcs is now running the show:

```

-----
i$ ls
iHaq.hs _darcs
-----

```

Add a build system

Create a `.cabal` file describing how to build your project:

```

-----
i$ cat > haq.cabal
iName:             haq
iVersion:          0.0
iDescription:      Super cool mega lambdas
iLicense:          GPL
iLicense-file:     LICENSE
iAuthor:           Don Stewart
iMaintainer:       dons@cse.unsw.edu.au
iBuild-Depends:    base
i
iExecutable:      haq
iMain-is:         Haq.hs
ighc-options:     -O
-----

```

(If your package uses other packages, e.g. `haskell198`, you'll need to add them to the `Build-Depends:` field.) Add a `Setup.lhs` that will actually do the building:

```

-----
i$ cat > Setup.lhs
i#! /usr/bin/env runhaskell
i
i> import Distribution.Simple
i> main = defaultMain
-----

```

Cabal allows either `Setup.hs` or `Setup.lhs`, but we recommend writing the setup file this way so that it can be executed directly by Unix shells.

Record your changes:

```
⌘$ darcs add haq.cabal Setup.lhs
⌘$ darcs record --all
What is the patch name? Add a build system
Do you want to add a long comment? [yn]n
Finished recording patch 'Add a build system'
```

Build your project

Now build it!

```
⌘$ runhaskell Setup.lhs configure --prefix=$HOME --user
⌘$ runhaskell Setup.lhs build
⌘$ runhaskell Setup.lhs install
```

Run it

And now you can run your cool project:

```
⌘$ haq me
"Haq! me"
```

You can also run it in-place, avoiding the install phase:

```
⌘$ dist/build/haq/haq you
"Haq! you"
```

Build some haddock documentation

Generate some API documentation into dist/doc/*

```
⌘$ runhaskell Setup.lhs haddock
```

which generates files in dist/doc/ including:

```
⌘$ w3m -dump dist/doc/html/haq/Main.html
haq Contents Index
Main
Synopsis
main :: IO ()
Documentation
main :: IO ()
main runs the main program
Produced by Haddock version 0.7
```

No output? Make sure you have actually installed haddock. It is a separate program, not something that comes with the Haskell compiler, like Cabal.

Add some automated testing: QuickCheck

We'll use QuickCheck to specify a simple property of our Haq.hs code. Create a tests module, Tests.hs, with

some QuickCheck boilerplate:

```

-----
$ cat > Tests.hs
import Char
import List
import Test.QuickCheck
import Text.Printf

main = mapM_ (\(s,a) -> printf "%-25s: " s >> a) tests

instance Arbitrary Char where
  arbitrary      = choose ('\0', '\128')
  coarbitrary c = variant (ord c `rem` 4)
-----

```

Now let's write a simple property:

```

-----
$ cat >> Tests.hs
-- reversing twice a finite list, is the same as identity
prop_reversereverse s = (reverse . reverse) s == id s
  where _ = s :: [Int]
-- and add this to the tests list
tests = [("reverse.reverse/id", test prop_reversereverse)]
-----

```

We can now run this test, and have QuickCheck generate the test data:

```

-----
$ runhaskell Tests.hs
reverse.reverse/id      : OK, passed 100 tests.
-----

```

Let's add a test for the 'haqify' function:

```

-----
-- Dropping the "Haq! " string is the same as identity
prop_haq s = drop (length "Haq! ") (haqify s) == id s
  where haqify s = "Haq! " ++ s
tests = [("reverse.reverse/id", test prop_reversereverse)
        ,("drop.haq/id",      test prop_haq)]
-----

```

and let's test that:

```

-----
$ runhaskell Tests.hs
reverse.reverse/id      : OK, passed 100 tests.
drop.haq/id            : OK, passed 100 tests.
-----

```

Great!

Running the test suite from darcs

We can arrange for darcs to run the test suite on every commit:

```

-----
$ darcs setpref test "runhaskell Tests.hs"
Changing value of test from '' to 'runhaskell Tests.hs'
-----

```

will run the full set of QuickChecks. (If your test requires it you may need to ensure other things are built too eg: darcs setpref test "alex Tokens.x;happy Grammar.y;runhaskell Tests.hs").

Let's commit a new patch:

```
$ darcs add Tests.hs
$ darcs record --all
What is the patch name? Add testsuite
Do you want to add a long comment? [yn]n
Running test...
reverse.reverse/id      : OK, passed 100 tests.
drop.haq/id             : OK, passed 100 tests.
Test ran successfully.
Looks like a good patch.
Finished recording patch 'Add testsuite'
```

Excellent, now patches must pass the test suite before they can be committed.

Tag the stable version, create a tarball, and sell it!

Tag the stable version:

```
$ darcs tag
What is the version name? 0.0
Finished tagging patch 'TAG 0.0'
```

Tarballs via Cabal

Since the code is cabalised, we can create a tarball with Cabal directly:

```
$ runhaskell Setup.lhs sdist
Building source dist for haq-0.0...
Source tarball created: dist/haq-0.0.tar.gz
```

This has the advantage that Cabal will do a bit more checking, and ensure that the tarball has the structure expected by HackageDB. It packages up the files needed to build the project; to include other files (such as `Test.hs` in the above example), we need to add:

```
extra-source-files: Tests.hs
```

to the `.cabal` file to have everything included.

Tarballs via darcs

Alternatively, you can use darcs:

```
$ darcs dist -d haq-0.0
Created dist as haq-0.0.tar.gz
```

And you're all set up!

Summary

The following files were created:

```
$ ls
Haq.hs      Tests.hs   dist       haq.cabal
Setup.lhs   _darcs    haq-0.0.tar.gz
```

Libraries

The process for creating a Haskell library is almost identical. The differences are as follows, for the hypothetical "ltree" library:

Hierarchical source

The source should live under a directory path that fits into the existing module layout guide (<http://www.haskell.org/~simonmar/lib-hierarchy.html>) . So we would create the following directory structure, for the module Data.LTree:

```
$ mkdir Data
$ cat > Data/LTree.hs
module Data.LTree where
```

So our Data.LTree module lives in Data/LTree.hs

The Cabal file

Cabal files for libraries list the publically visible modules, and have no executable section:

```
$ cat ltree.cabal
Name:          ltree
Version:       0.1
Description:   Lambda tree implementation
License:       BSD3
License-file:  LICENSE
Author:        Don Stewart
Maintainer:    dons@cse.unsw.edu.au
Build-Depends: base
Exposed-modules: Data.LTree
ghc-options:   -Wall -O
```

We can thus build our library:

```
$ runhaskell Setup.lhs configure --prefix=$HOME --user
$ runhaskell Setup.lhs build
Preprocessing library ltree-0.1...
Building ltree-0.1...
[1 of 1] Compiling Data.LTree      ( Data/LTree.hs, dist/build/Data/LTree.o )
/usr/bin/ar: creating dist/build/libHSLtree-0.1.a
```

and our library has been created as a object archive. On *nix systems, you should probably add the --user flag to the configure step (this means you want to update your local package database during installation). Now install it:

```
$ runhaskell Setup.lhs install
Installing: /home/dons/lib/ltree-0.1/ghc-6.6 & /home/dons/bin ltree-0.1...
Registering ltree-0.1...
Reading package info from ".installed-pkg-config" ... done.
Saving old package config file... done.
Writing new package config file... done.
```

And we're done! You can use your new library from, for example, ghci:

```
$ ghci -package ltree
Prelude> :m + Data.LTree
Prelude Data.LTree>
```

The new library is in scope, and ready to go.

More complex build systems

For larger projects it is useful to have source trees stored in subdirectories. This can be done simply by creating a directory, for example, "src", into which you will put your src tree.

To have Cabal find this code, you add the following line to your Cabal file:

```
hs-source-dirs: src
```

Cabal can set up to also run configure scripts, along with a range of other features. For more information consult the Cabal documentation (<http://www.haskell.org/ghc/docs/latest/html/Cabal/index.html>) .

Automation

A tool to automatically populate a new cabal project is available (beta!):

```
darcs get http://www.cse.unsw.edu.au/~dons/code/mkcabal
```

N.B. This tool does not work in Windows. The Windows version of GHC does not include the readline package that this tool needs.

Usage is:

```
$ mkcabal
Project name: haq
What license ["GPL","LGPL","BSD3","BSD4","PublicDomain","AllRightsReserved"] ["BSD3"]:
What kind of project [Executable,Library] [Executable]:
Is this your name? - "Don Stewart " [Y/n]:
Is this your email address? - "<dons@cse.unsw.edu.au>" [Y/n]:
Created Setup.lhs and haq.cabal
$ ls
Haq.hs    LICENSE  Setup.lhs _darcs   dist     haq.cabal
```

which will fill out some stub Cabal files for the project 'haq'.

To create an entirely new project tree:

```
$ mkcabal --init-project
Project name: haq
What license ["GPL","LGPL","BSD3","BSD4","PublicDomain","AllRightsReserved"] ["BSD3"]:
What kind of project [Executable,Library] [Executable]:
Is this your name? - "Don Stewart " [Y/n]:
Is this your email address? - "<dons@cse.unsw.edu.au>" [Y/n]:
Created new project directory: haq
$ cd haq
$ ls
Haq.hs    LICENSE  README  Setup.lhs haq.cabal
```

Licenses

Code for the common base library package must be BSD licensed or Freer. Otherwise, it is entirely up to you as the author.

Choose a licence (inspired by this (<http://www.dina.dk/~abraham/rants/license.html>)). Check the licences of things you use, both other Haskell packages and C libraries, since these may impose conditions you must follow.

Use the same licence as related projects, where possible. The Haskell community is split into 2 camps, roughly, those who release everything under BSD, GPLers, and LGPLers. Some Haskellers recommend specifically avoiding the LGPL, due to cross module optimisation issues. Like many licensing questions, this advice is controversial. Several Haskell projects (wxHaskell, HaXml, etc) use the LGPL with an extra permissive clause to avoid the cross-module optimisation problem.

Releases

It's important to release your code as stable, tagged tarballs. Don't just rely on darcs for distribution (<http://awayrepl.blogspot.com/2006/11/we-dont-do-releases.html>) .

- **darcs dist** generates tarballs directly from a darcs repository

For example:

```
-----  
$ cd fps  
$ ls  
Data      LICENSE  README   Setup.hs  TODO      _darcs   cbits  dist      fps.cabal  tests  
$ darcs dist -d fps-0.8  
Created dist as fps-0.8.tar.gz  
-----
```

You can now just post your fps-0.8.tar.gz

You can also have darcs do the equivalent of 'daily snapshots' for you by using a post-hook.

put the following in `_darcs/prefs/defaults`:

```
-----  
: apply posthook darcs dist  
: apply run-posthook  
-----
```

Advice:

- Tag each release using **darcs tag**. For example:

```
-----  
$ darcs tag 0.8  
Finished tagging patch 'TAG 0.8'  
-----
```

Then people can `darcs pull --partial -t 0.8`, to get just the tagged version (and not the entire history).

Hosting

A Darcs repository can be published simply by making it available from a web page. If you don't have an

account online, or prefer not to do this yourself, source can be hosted on darcs.haskell.org (you will need to email Simon Marlow (<http://research.microsoft.com/~simonmar/>) to do this). haskell.org itself has some user accounts available.

There are also many free hosting places for open source, such as

- Google Project Hosting (<http://code.google.com/hosting/>)
- SourceForge (<http://sourceforge.net/>) .

Example

A complete example (<http://www.cse.unsw.edu.au/~dons/blog/2006/12/11#release-a-library-today>) of writing, packaging and releasing a new Haskell library under this process has been documented.



At least part of this page was imported from the Haskell wiki article [How to write a Haskell program](http://www.haskell.org/haskellwiki/How_to_write_a_Haskell_program) (http://www.haskell.org/haskellwiki/How_to_write_a_Haskell_program) , in accordance to its Simple Permissive License. If you wish to modify this page and if your changes will also be useful on that wiki, you might consider modifying that source page instead of this one, as changes from that page may propagate here, but not the other way around. Alternately, you can explicitly dual license your contributions under the Simple Permissive License. Note also that the original tutorial contains extra information about announcing your software and joining the Haskell community, which may be of interest to you.

Specialised Tasks

GUI

Haskell has at least three toolkits for programming a graphical interface:

- wxHaskell - provides a Haskell interface to the wxWidgets toolkit
- Gtk2Hs (<http://haskell.org/gtk2hs/>) - provides a Haskell interface to the GTK+ library
- hoc (<http://hoc.sourceforge.net/>) - provides a Haskell to Objective-C binding which allows users to access to the Cocoa library on MacOS X

In this tutorial, we will focus on the wxHaskell toolkit, as it allows you to produce a native graphical interface on all platforms that wxWidgets is available on, including Windows, Linux and MacOS X.

Getting and running wxHaskell

To install wxHaskell, you'll need to use the GHC (<http://haskell.org/ghc/>) . Then, find your wxHaskell package on the wxHaskell download page (<http://wxhaskell.sourceforge.net/download.html>) .

The latest version of GHC is 6.6.1, but wxHaskell hasn't been updated for versions higher than 6.4. You can either downgrade GHC to 6.4, or build wxHaskell yourself. Instructions on how to do this can be found on the building page (<http://wxhaskell.sourceforge.net/building.html>) .

Follow the installation instruction provided on the wxHaskell download page. Don't forget to register wxHaskell with GHC, or else it won't run. To compile source.hs (which happens to use wxHaskell code), open a command line and type:

```
ghc -package wx source.hs -o bin
```

Code for GHCi is similar:

```
ghci -package wx
```

You can then load the files from within the GHCi interface. To test if everything works, go to `$wxHaskellDir/samples/wx` (`$wxHaskellDir` is the directory you installed it in) and load (or compile) `HelloWorld.hs`. It should show a window with title "Hello World!", a menu bar with File and About, and a status bar at the bottom, that says "Welcome to wxHaskell".

If it doesn't work, you might try to copy the contents of the `$wxHaskellDir/lib` directory to the ghc install directory.

Hello World

Here's the basic Haskell "Hello World" program:

```
module Main where
main :: IO ()
main = putStrLn "Hello World!"
```

It will compile just fine, but it isn't really fancy. We want a nice GUI! So how to do this? First, you must import `Graphics.UI.WX`. This is the wxHaskell library. `Graphics.UI.WXCore` has some more stuff, but we won't be needing that now.

To start a GUI, use (guess what) `start gui`. In this case, `gui` is the name of a function which we'll use to build the interface. It must have an IO type. Let's see what we have:

```

module Main where

import Graphics.UI.WX

main :: IO ()
main = start gui

gui :: IO ()
gui = do
  --GUI stuff

```

To make a frame, we use `frame`. Check the type of `frame`. It's `[Prop (Frame ())] -> IO (Frame ())`. It takes a list of "frame properties" and returns the corresponding frame. We'll look deeper into properties later, but a property is typically a combination of an attribute and a value. What we're interested in now is the title. This is in the `text` attribute and has type `(Textual w) => Attr w String`. The most important thing here, is that it's a `String` attribute. Here's how we code it:

```

gui :: IO ()
gui = do
  frame [text := "Hello World!"]

```

The operator `(:=)` takes an attribute and a value, and combines both into a property. Note that `frame` returns an `IO (Frame ())`. You can change the type of `gui` to `IO (Frame ())`, but it might be better just to add `return ()`. Now we have our own GUI consisting of a frame with title "Hello World!". Its source:

```

module Main where

import Graphics.UI.WX

main :: IO ()
main = start gui

gui :: IO ()
gui = do
  frame [text := "Hello World!"]
  return ()

```



The result should look like the screenshot. (It might look slightly different on Linux or MacOS X, on which `wxhaskell` also runs)

Controls



From here on, it's good practice to keep a browser window or tab open with the wxHaskell documentation (<http://wxhaskell.sourceforge.net/doc/>). It's also available in `$wxHaskellDir/doc/index.html`.

A text label

Simply a frame doesn't do much. In this chapter, we're going to add some more elements. Let's start with something simple: a label. wxHaskell has a `label`, but that's a layout thing. We won't be doing layout until next chapter. What we're looking for is a `staticText`. It's in `Graphics.UI.WX.Controls`. As you can see, the `staticText` function takes a window as argument, and a list of properties. Do we have a window? Yup! Look at

`Graphics.UI.WX.Frame`. There we see that a `Frame` is merely a type-synonym of a special sort of window. We'll change the code in `gui` so it looks like this:

```
gui :: IO ()
gui = do
  f <- frame [text := "Hello World!"]
  staticText f [text := "Hello StaticText!"]
  return ()
```

Again, `text` is an attribute of a `staticText` object, so this works. Try it!

A button

Now for a little more interaction. A button. We're not going to add functionality to it until the chapter about events, but at least something visible will happen when you click on it.

A button is a control, just like `staticText`. Look it up in `Graphics.UI.WX.Controls`.

Again, we need a window and a list of properties. We'll use the `frame` again. `text` is also an attribute of a button:

```
gui :: IO ()
gui = do
  f <- frame [text := "Hello World!"]
  staticText f [text := "Hello StaticText!"]
  button f [text := "Hello Button!"]
  return ()
```

Load it into GHCi (or compile it with GHC) and... hey!? What's that? The button's been covered up by the label! We're going to fix that next, in the layout chapter.

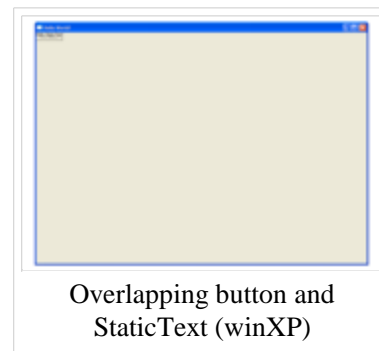
Layout

The reason that the label and the button overlap, is that we haven't set a *layout* for our frame yet. Layouts are created using the functions found in the documentation of `Graphics.UI.WXCore.Layout`. Note that you don't have to import `Graphics.UI.WXCore` to use layouts.

The documentation says we can turn a member of the widget class into a layout by using the `widget` function. Also, windows are a member of the widget class. But, wait a minute... we only have one window, and that's the frame! Nope... we have more, look at `Graphics.UI.WX.Controls` and click on any occasion of the word *Control*. You'll be taken to `Graphics.UI.WXCore.WxcClassTypes` and it is here we see that a `Control` is also a type synonym of a special type of window. We'll need to change the code a bit, but here it is.

```
gui :: IO ()
gui = do
  f <- frame [text := "Hello World!"]
  st <- staticText f [text := "Hello StaticText!"]
  b <- button f [text := "Hello Button!"]
  return ()
```

Now we can use `widget st` and `widget b` to create a layout of the `staticText` and the button. `layout` is an attribute of the frame, so we'll set it here:



```

gui :: IO ()
gui = do
  f <- frame [text := "Hello World!"]
  st <- staticText f [text := "Hello StaticText!"]
  b <- button f [text := "Hello Button!"]
  set f [layout := widget st]
  return ()

```

The `set` function will be covered in the chapter about attributes. Try the code, what's wrong? This only displays the `staticText`, not the button. We need a way to combine the two. We will use *layout combinators* for this. `row` and `column` look nice. They take an integer and a list of layouts. We can easily make a list of layouts of the button and the `staticText`. The integer is the spacing between the elements of the list. Let's try something:

```

gui :: IO ()
gui = do
  f <- frame [text := "Hello World!"]
  st <- staticText f [text := "Hello StaticText!"]
  b <- button f [text := "Hello Button!"]
  set f [layout :=
    row 0 [widget st, widget b]
  ]
  return ()

```

Play around with the integer and see what happens, also change `row` into `column`. Try to change the order of the elements in the list to get a feeling of how it works. For fun, try to add `widget b` several more times in the list. What happens?

Here are a few exercises to spark your imagination. Remember to use the documentation!

Exercises

1. Add a checkbox control. It doesn't have to do anything yet, just make sure it appears next to the `staticText` and the button when using row-layout, or below them when using column layout. `text` is also an attribute of the checkbox.
2. Notice that `row` and `column` take a list of *layouts*, and also generates a layout itself. Use this fact to make your checkbox appear on the left of the `staticText` and the button, with the `staticText` and the button in a column.
3. Can you figure out how the radiobox control works? Take the layout of the previous exercise and add a radiobox with two (or more) options below the checkbox, `staticText` and button. Use the documentation!
4. Use the `boxed` combinator to create a nice looking border around the four controls, and another one around the `staticText` and the button. (Note: the `boxed` combinator might not be working on MacOS X - you might get widgets that can't be interacted with. This is likely just



StaticText with layout
(winXP)



A row layout (winXP)



Column layout with a spacing
of 25 (winXP)

a bug in wxhaskell.)

After having completed the exercises, the end result should look like this:

You could have used different spacing for `row` and `column`, or the options of the radiobox are displayed horizontally.

Attributes

After all this, you might be wondering things like: "Where did that `set` function suddenly come from?", or "How would *I* know if `text` is an attribute of something?". Both answers lie in the attribute system of `wxHaskell`.



Answer to exercises

Setting and modifying attributes

In a `wxHaskell` program, you can set the properties of the widgets in two ways:

1. during creation: `f <- frame [text := "Hello World!"]`
2. using the `set` function: `set f [layout := widget st]`

The `set` function takes two arguments: one of any type `w`, and the other is a list of properties of `w`. In `wxHaskell`, these will be the widgets and the properties of these widgets. Some properties can only be set during creation, like the `alignment` of a `textEntry`, but you can set most others in any IO-function in your program, as long as you have a reference to it (the `f` in `set f [--stuff]`).

Apart from setting properties, you can also get them. This is done with the `get` function. Here's a silly example:

```
gui :: IO ()
gui = do
  f <- frame [ text := "Hello World!" ]
  st <- staticText f []
  ftext <- get f text
  set st [ text := ftext ]
  set f [ text := ftext ++ " And hello again!" ]
```

Look at the type signature of `get`. It's `w -> Attr w a -> IO a`. `text` is a `String` attribute, so we have an `IO String` which we can bind to `ftext`. The last line edits the text of the frame. Yep, destructive updates are possible in `wxHaskell`. We can overwrite the properties using `(:=)` anytime with `set`. This inspires us to write a modify function:

```
modify :: w -> Attr w a -> (a -> a) -> IO ()
modify w attr f = do
  val <- get w attr
  set w [ attr := f val ]
```

First it gets the value, then it sets it again after applying the function. Surely we're not the first one to think of that...

And nope, we aren't. Look at this operator: `(:~)`. You can use it in `set`, because it takes an attribute and a

function. The result is an property, in which the original value is modified by the function. This means we can write:

```
gui :: IO ()
gui = do
  f <- frame [ text := "Hello World!" ]
  st <- staticText f []
  ftext <- get f text
  set st [ text := ftext]
  set f [ text := ++ " And hello again!" ]
```

This is a great place to use anonymous functions with the lambda-notation.

There are two more operators we can use to set or modify properties: `(:=)` and `(:~)`. They do the same as `(:=)` and `(:~)`, except a function of type `w -> orig` is expected, where `w` is the widget type, and `orig` is the original "value" type (a in case of `(:=)`, and `a -> a` in case of `(:~)`). We won't be using them now, though, as we've only encountered attributes of non-IO types, and the widget needed in the function is generally only useful in IO-blocks.

How to find attributes

Now the second question. Where did I read that `text` is an attribute of all those things? The easy answer is: in the documentation. Now where in the documentation to look for it?

Let's see what attributes a button has, so go to `Graphics.UI.WX.Controls`, and click the link that says "Button". You'll see that a `Button` is a type synonym of a special kind of `Control`, and a list of functions that can be used to create a button. After each function is a list of "Instances". For the normal `button` function, this is *Commanding -- Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint*. This is the list of classes of which a button is an instance. Read through the `Class_Declarations` chapter. It means that there are some class-specific functions available for the button. `Textual`, for example, adds the `text` and `appendText` functions. If a widget is an instance of the `Textual` class, it means that it has a `text` attribute!

Note that while `StaticText` hasn't got a list of instances, it's still a `Control`, which is a synonym for some kind of `Window`, and when looking at the `Textual` class, it says that `Window` is an instance of it. This is an error on the side of the documentation.

Let's take a look at the attributes of a frame. They can be found in `Graphics.UI.WX.Frame`. Another error in the documentation here: It says `Frame` instantiates `HasImage`. This was true in an older version of `wxHaskell`. It should say `Pictured`. Apart from that, we have `Form`, `Textual`, `Dimensions`, `Colored`, `Able` and a few more. We're already seen `Textual` and `Form`. Anything that is an instance of `Form` has a `layout` attribute.

`Dimensions` adds (among others) the `clientSize` attribute. It's an attribute of the `Size` type, which can be made with `sz`. Please note that the `layout` attribute can also change the size. If you want to use `clientSize` you should set it after the `layout`.

`Colored` adds the `color` and `bgcolor` attributes.

`Able` adds the Boolean `enabled` attribute. This can be used to enable or disable certain form elements, which is often displayed as a greyed-out option.

There are lots of other attributes, read through the documentation for each class.

Events

There are a few classes that deserve special attention. They are the `Reactive` class and the `Commanding` class. As you can see in the documentation of these classes, they don't add attributes (of the form `Attr w a`), but *events*. The `Commanding` class adds the `command` event. We'll use a button to demonstrate event handling.

Here's a simple GUI with a button and a `staticText`:

```
gui :: IO ()
gui = do
  f <- frame [ text := "Event Handling" ]
  st <- staticText f [ text := "You haven't clicked the button yet." ]
  b <- button f [ text := "Click me!" ]
  set f [ layout := column 25 [ widget st, widget b ] ]
```



We want to change the `staticText` when you press the button. We'll need the `on` function:

```
b <- button f [ text := "Click me!"
               , on command := --stuff
               ]
```

The type of `on: Event w a -> Attr w a. command` is of type `Event w (IO ())`, so we need an IO-function. This function is called the *Event handler*. Here's what we get:

```
gui :: IO ()
gui = do
  f <- frame [ text := "Event Handling" ]
  st <- staticText f [ text := "You haven't clicked the button yet." ]
  b <- button f [ text := "Click me!"
                 , on command := set st [ text := "You have clicked the button!" ]
                 ]
  set f [ layout := column 25 [ widget st, widget b ] ]
```



Insert text about event filters here

Database

Haskell/Database

Web programming

An example web application, using the HAppS framework, is `hpaste` (<http://hpaste.org>), the Haskell paste bin. Built around the core Haskell web framework, HAppS, with HaXML for page generation, and `binary/zlib` for state serialisation.

The HTTP and Browser modules (<http://homepages.paradise.net.nz/warrickg/haskell/http/>) exist, and might be useful.

XML

There are several Haskell libraries for XML work, and additional ones for HTML. For more web-specific work, you may want to refer to the Haskell/Web programming chapter.

Libraries for parsing XML

- The Haskell XML Toolbox (hxt) (<http://www.fh-wedel.de/~si/HXmlToolbox/>) is a collection of tools for parsing XML, aiming at a more general approach than the other tools.
- HaXml (<http://www.cs.york.ac.uk/fp/HaXml/>) is a collection of utilities for parsing, filtering, transforming, and generating XML documents using Haskell.
- HXML (<http://www.flightlab.com/~joe/hxml/>) is a non-validating, lazy, space efficient parser that can work as a drop-in replacement for HaXml.

Libraries for generating XML

- HSXML represents XML documents as statically typesafe s-expressions.

Other options

- tagsoup (<http://www.cs.york.ac.uk/fp/darcs/tagsoup/tagsoup.htm>) is a library for parsing unstructured HTML, i.e. it does not assume validity or even well-formedness of the data.

Getting acquainted with HXT

In the following, we are going to use the Haskell XML Toolbox for our examples. You should have a working installation of GHC, including GHCi, and you should have downloaded and installed HXT according to the instructions (<http://www.fh-wedel.de/~si/HXmlToolbox/#install>) .

With those in place, we are ready to start playing with HXT. Let's bring the XML parser into scope, and parse a simple XML-formatted string:

```
-----  
Prelude> :m + Text.XML.HXT.Parser  
Prelude Text.XML.HXT.Parser> xread "<foo>abc<bar/>def</foo>"  
[NTree (XTag (QN {namePrefix = "", localPart = "foo", namespaceUri = ""}) [])  
 [NTree (XText "abc") [],NTree (XTag (QN {namePrefix = "", localPart = "bar",  
 namespaceUri = ""}) []) [],NTree (XText "def") []]]  
-----
```

We see that HXT represents an XML document as a list of trees, where the nodes can be constructed as an XTag containing a list of subtrees, or an XText containing a string. With GHCi, we can explore this in more detail:


```

Prelude Text.XML.HXT.Parser Text.XML.HXT.DOM> :i NTree
data NTree a = NTree a (NTrees a)
              -- Defined in Data.Tree.NTree.TypeDefs
Prelude Text.XML.HXT.Parser Text.XML.HXT.DOM> :i NTrees
type NTrees a = [NTree a]
              -- Defined in Data.Tree.NTree.TypeDefs

```

As we can see, an `NTree` is a general tree structure where a node stores its children in a list, and some more browsing around will tell us that XML documents are trees over an `XNode` type, defined as:

```

data XNode
= XText String
| XCharRef Int
| XEntityRef String
| XCmt String
| XCdata String
| XPi QName XmlTrees
| XTag QName XmlTrees
| XDTD DTDElem Attributes
| XAttr QName
| XError Int String

```

Returning to our example, we notice that while HXT successfully parsed our input, one might desire a more lucid presentation for human consumption. Lucky for us, the DOM module supplies this. Notice that `xread` returns a list of trees, while the formatting function works on a single tree.

```

Prelude Text.XML.HXT.Parser> :m + Text.XML.HXT.DOM
Prelude Text.XML.HXT.Parser Text.XML.HXT.DOM> putStrLn $ formatXmlTree $ head $ xread "<foo>abc<bar/>def</foo>"
---XTag "foo"
|
+---XText "abc"
|
+---XTag "bar"
|
+---XText "def"

```

This representation makes the structure obvious, and it is easy to see the relationship to our input string. Let's proceed to extend our XML document with some attributes (taking care to escape the quotes, of course):

```

Prelude Text.XML.HXT.Parser> xread "<foo a1=\"my\" b2=\"oh\">abc<bar/>def</foo>"
[NTree (XTag (QN {namePrefix = "", localPart = "foo", namespaceUri = ""}) [NTree (XAttr (QN {namePrefix = "", localPart = "a1", namespaceUri = ""}) [NTree (XText "my") []],NTree (XAttr (QN {namePrefix = "", localPart = "b2", namespaceUri = ""}) [NTree (XText "oh") []])) [NTree (XText "abc") []],NTree (XTag (QN {namePrefix = "", localPart = "bar", namespaceUri = ""}) [] [NTree (XText "def") []])

```

Notice that attributes are stored as regular `NTree` nodes with the `XAttr` content type, and (of course) no children. Feel free to pretty-print this expression, as we did above.

For a trivial example of data extraction, consider this small example using `XPath` (<http://en.wikipedia.org/wiki/XPath>):

```

Prelude> :set prompt "> "
> :m + Text.XML.HXT.Parser Text.XML.HXT.XPath.XPathEval
> let xml = "<foo><a>A</a><c>C</c></foo>"
> let xmltree = head $ xread xml
> let result = getXPath "//a" xmltree
> result
> [NTree (XTag (QN {namePrefix = "", localPart = "a", namespaceUri = ""}) []) [NTree (XText "A") []]]
> :t result
> result :: NTrees XNode

```

Retrieved from "http://en.wikibooks.org/wiki/Haskell/Print_version"

- This page was last modified 23:52, 17 January 2007.
 - All text is available under the terms of the GNU Free Documentation License (see **Copyrights** for details).
- Wikibooks® is a registered trademark of the Wikimedia Foundation, Inc.