Haskell

en.wikibooks.org

April 20, 2014

Contents

1 Haskell Basics

2 Getting set up

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

2.1 Installing Haskell

Haskell is a *programming language*, i.e. a language in which humans can express how computers should behave. It's like writing a cooking recipe: you write the recipe and the computer executes it.

To write Haskell programs, you need a program called a Haskell *compiler*. A compiler is a program that takes code written in Haskell and translates it into *machine code*, a more primitive language that the computer understands. Using the above analogy, the compiler is the oven that bakes your batter (code) into a cookie (executable file), and it's difficult to get the recipe from an executable once it's compiled.

To start learning Haskell, **download and install the Haskell platform**[1]. It will contain the "Glasgow Haskell Compiler", or GHC, and everything else you need.

If you're just trying out Haskell, or are averse to downloading and installing the full compiler, you might like to try Hugs[2], the lightweight Haskell interpreter (it also happens to be portable). You might also like to play around with TryHaskell[3], an interpreter hosted online. Note that all instructions will be for GHC.

Note:
UNIX users:
If you are a person who prefers 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[a]. In short, we strongly recommend downloading the Haskell Platform instead of compiling from source.

[a] http://hackage.haskell.org/trac/ghc/wiki/Building

[1] http://hackage.haskell.org/platform/
[2] http://www.haskell.org/hugs/
[3] http://www.tryhaskell.org/

2.2 Very first steps

After you have installed the Haskell Platform[4], it's now time to write your first Haskell code.

For that, you will use the program called **GHCi** (the 'i' stands for 'interactive'). Depending on your operating system, perform the following steps:

- On Windows: Click Start, then Run, then type 'cmd' and hit Enter, then type `ghci` and hit Enter once more.
- On MacOS: Open the application "Terminal" found in the "Applications/Utilities" folder, type the letters `ghci` into the window that appears and hit the Enter key.
- On Linux: Open a terminal and run the `ghci` program.

You should get output that looks something like the following:

```
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.6, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base ... linking ... done.
Prelude>
```

The first bit is GHCi's logo. It then informs you it's loading the base package, so you'll have access to most of the built-in functions and modules that come with GHC. Finally, the `Prelude>` bit is known as the *prompt*. This is where you enter commands, and GHCi will respond with their results.

Now you're ready to write your first Haskell code. In particular, let's try some basic arithmetic:

```
Prelude> 2 + 2
4
Prelude> 5 + 4 * 3
17
Prelude> 2 ^ 5
32
```

The operators are similar to what they are in other languages: `+` is addition, `*` is multiplication, and `^` is exponentiation (raising to the power of, or a^b). Note from the second example that Haskell follows standard order of operations.

Now you know how to use Haskell as a calculator. A key idea of the Haskell language is that it will always be like a calculator, except that it will become really powerful when we calculate not only with numbers, but also with other objects like characters, lists, functions, trees and even other programs (if you aren't familiar with these yet, don't worry).

GHCi is a very powerful development environment. As we progress, we'll learn how we can load source files into GHCi, and evaluate different bits of them.

[4] `http://hackage.haskell.org/platform/`

The next chapter will introduce some of the basic concepts of Haskell. Let's dive into that and have a look at our first Haskell functions.

3 Variables and functions

All the examples in this chapter can be typed into a Haskell source file and evaluated by loading that file into GHC or Hugs. Remember not to type the prompts at the beginning of input. If there are prompts at the beginning, then you can type it in an environment like GHCi. If not, then you should put it in a file and run it.

3.1 Variables

We've already seen how to use the GHCi program as a calculator. Of course, this is only practical for very short calculations. For longer calculations and for writing Haskell programs, we need to keep track of intermediate results.

Intermediate results can be stored in *variables*, to which we refer by their name. A variable contains a *value*, which is substituted for the variable name when you use a variable. For instance, consider the following calculation

```
ghci> 3.1416 * 5^2
78.53999999999999
```

This is the area of a circle with radius `5`, according to the formula $A = \pi r^2$. It is very cumbersome to type in the digits of $\pi \approx 3.1416$, or even to remember them at all. In fact, an important aspect of programming, if not the whole point, is to delegate mindless repetition and rote memorization to a machine. In this case, Haskell has already *defined* a variable named `pi` that stores over a dozen digits of π for us.

```
ghci> pi
3.141592653589793
ghci> pi * 5^2
78.53981633974483
```

Notice that the variable `pi` and its value, `3.141592653589793`, can be used interchangeably in calculations.

3.2 Haskell source files

Now, we want to define our own variables to help us in our calculations. This is done in a *Haskell source file*, which contains Haskell code.

Create a new file called `Varfun.hs` in your favourite text editor [1] (the file extension `.hs` stands for "Haskell") and type/paste in the following definition:

```
r = 5.0
```

Make sure that there are no spaces at the beginning of the line because Haskell is a whitespace sensitive language (more about indentation later).

Now, open GHCi, move to the directory (folder) where you saved your file with the `:cd YourDirectoryHere` command, and use the `:load YourFileHere` (or `:l YourFileHere`) command:

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

• Main>
```

Loading a Haskell source file will make all its definitions available in the GHCi prompt. Source files generally include *modules* (units of storage for code) to organize them or indicate where the program should start running (the `Main` module) when you use many files. In this case, because you did not indicate any `Main` module, it created one for you.

If GHCi gives an error like `Could not find module 'Varfun.hs'`, you probably are in the wrong directory.

Now you can use the newly defined variable `r` in your calculations.

```
• Main> r 5.0
• Main> pi * r^2 78.53981633974483
```

So, to calculate the area of a circle of radius 5, we simply define `r = 5.0` and then type in the well-known formula πr^2 for the area of a circle. There is no need to write the numbers out every time; that's very convenient!

Since this was so much fun, let's add another definition: Change the contents of the source file to

```
r = 5.0
area = pi * r ^ 2
```

Save the file and type the `:reload` (or `:r`) command in GHCi to load the new contents (note that this is a continuation of the last session):

```
• Main> :reload Compiling Main ( Varfun.hs, interpreted ) Ok, modules loaded: Main.
• Main>
```

Now we have two variables `r` and `area` available

[1] The Wikipedia article on text editors `^{http://en.wikipedia.org/wiki/text%20editor}` is a reasonable place to start if you need suggestions. Popular ones include Vim, Emacs and, on Windows, Notepad++ (*not* plain old Notepad). Proper programming text editors will provide *syntax highlighting*, a feature which colourises code in relevant ways so as to make it easier to read.

- `Main> area 78.53981633974483`
- `Main> area / r 15.707963267948966`

Note:
It is also possible to define variables directly at the GHCi prompt, without a source file. Skipping the details, the syntax for doing so uses the `let` *keyword* (a word with a special meaning) and looks like:

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

Although we will occasionally do such definitions for expediency in the examples, it will quickly become obvious, as we move into slightly more complex tasks, that this practice is not really convenient. That is why we are emphasizing the use of source files from the very start.

Note:
To experienced programmers: GHC can also be used as a compiler (that is, you could use GHC to convert your Haskell files into a stand-alone program that can be run without running the interpreter). How to do so will be explained much later.

3.3 Comments

Before we continue, it is good to understand that it is possible to include text in a program without having it treated as code. This is achieved by use of *comments*. In Haskell, a comment can be started with `--` and continues until the end of the line:

```
x = 5     -- The variable x is 5.
y = 6     -- The variable y is 6.
-- z = 7
```

In this case, `x` and `y` are defined, but `z` is not. Comments can also go anywhere using the alternative syntax `{- ... -}`:

```
x = {- Do this just because you can. -} 5
```

Comments are generally used for explaining parts of a program that may be somewhat confusing to readers otherwise.

Be careful about overusing comments. Too many comments make programs harder to read, and comments must be carefully updated whenever corresponding code is changed or else the comments may become outdated and incorrect.

3.4 Variables in imperative languages

If you are already familiar with imperative programming languages like C, you will notice that variables in Haskell are quite different from variables as you know them. We now explain why and how.

If you have no programming experience, you might like to skip this section and continue reading with Functions[2].

Unlike in imperative languages, variables in Haskell *do not vary.* Once defined, their value never changes; they are immutable. For instance, the following code does not work:

```
r = 5
r = 2
```

The variables in functional programming languages are more related to variables in mathematics than changeable locations in a computer's memory. As in Haskell, you would definitely never see a variable used this way in a math classroom (at least not in the same problem). The compiler will give an error due to "multiple declarations of `r`". People more familiar with *imperative programming*, which involves explicitly telling the computer what to do, may be accustomed to read this as first setting `r = 5` and then changing it to `r = 2`, but in functional programming languages, the program is in charge of figuring out what to do with memory.

Here's another example of a major difference from imperative languages

```
r = r + 1
```

Instead of "incrementing the variable `r`", this is actually a recursive definition of `r` in terms of itself (we will explain recursion[3] in detail later on; just remember that this is radically different from imperative languages).

Because you don't need to worry about changing values, variables can be defined in any order. For example, the following fragments of code do exactly the same thing:

```
y = x * 2
x = 3
```

```
x = 3
y = x * 2
```

We can write things in any order that we want, there is no notion of "`x` being declared before `y`" or the other way around. This is also why you can't declare something more than once; it would be ambiguous otherwise. Of course, using `y` will still require a value for `x`, but this is unimportant until you need a specific numeric value.

By now, you might be wondering how you can actually do anything at all in Haskell where variables don't change. But trust us; as we hope to show you in the rest of this book, you

[2] Chapter 3.5 on page 13

[3] Chapter 12 on page 81

can write every program under the sun without ever changing a single variable! In fact, variables that don't change make life so much easier because it makes programs much more predictable. It's a key feature of purely functional programming, which requires a very different approach from imperative programming and requires a different mindset.

3.5 Functions

Now, suppose that we have multiple circles with different radii whose areas we want to calculate. For instance, to calculate the area of another circle with radius 3, we would have to include new variables **r2** and **area2**[4] in our source file:

```
r  = 5
area  = pi*r^2
r2 = 3
area2 = pi*r2^2
```

Clearly, this is unsatisfactory because we are repeating the formula for the area of a circle verbatim. To eliminate this mindless repetition, we would prefer to write it down only once and then apply it to different radii. That's exactly what *functions* allow us to do.

A *function* takes an *argument* value (or *parameter*) and gives a result value, like a variable, that takes its place. (If you are already familiar with mathematical functions, they are essentially the same.) Defining functions in Haskell is simple: It is like defining a variable, except that we take note of the function argument that we put on the left hand side. For instance, the following is the definition of a function **area** which depends on a argument which we name **r**:

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

The syntax here is important to note: the function name comes first (in our example, that's "area"), followed by a space and then the argument ("r" in the example). Following the = sign, the function definition is a formula that uses the argument in context with other already defined terms.

Now, we can plug in different values for the argument in a *call* to the function. Load this definition into GHCi and try the following calls:

- Main> area 5 78.53981633974483
- Main> area 3 28.274333882308138
- Main> area 17 907.9202768874502

Thus, we can call this function with different radii to calculate the area of the corresponding circles.

You likely know functions from mathematics already. Our function here is defined mathematically as

[4] As you can see, the names of variables may also contain numbers. Variables *must* begin with a lowercase letter, but for the rest, any string consisting of letter, numbers, underscore (_) or tick (') is allowed.

$$A(r) = \pi \cdot r^2$$

In mathematics, the parameter is enclosed between parentheses, as in $A(5) = 78.54$ or $A(3) = 28.27$. While the Haskell code will still work with parentheses, they are normally omitted. As Haskell is a *functional* language, we will call a lot of functions, and whenever possible we want to minimize extra symbols.

Parentheses are still used for grouping *expressions* (any code that gives a value) to be evaluated together. Note how the following expressions are parsed differently:

```
area (5 + 3)     -- area (5 + 3)
area 5 + 3       -- (area 5) + 3
```

This shows that function calls take *precedence* over operators like + the same way multiplication is done before addition in mathematics.

3.5.1 Evaluation

Let us try to understand what exactly happens when you enter an expression into GHCi. After you press the enter key, GHCi will *evaluate* the expression you have given. This means that it will replace each function with its definition and calculate the results until a single value is left. For example, the evaluation of `area 5` proceeds as follows:

```
   area 5
=>    { replace the left-hand side  area r = ...  by the right-hand side  ... =
pi * r^2 }
   pi * 5^2
=>    { replace  pi  by its numerical value }
   3.141592653589793 * 5^2
=>    { apply exponentiation (^) }
   3.141592653589793 * 25
=>    { apply multiplication (*) }
   78.53981633974483
```

As this shows, to *apply* or *call* a function means to replace the left-hand side of its definition by its right-hand side. As a last step, GHCi prints the final result on the screen.

Here are some more functions:

```
double x    = 2*x
quadruple x = double (double x)
square x    = x*x
half   x     = x / 2
```

Exercises:

- Explain how GHCi evaluates `quadruple 5`.
- Define a function that subtracts 12 from half its argument.

3.5.2 Multiple parameters

Of course, functions can also have more than one argument. For example, here is a function for calculating the area of a rectangle given its length and its width:

```
areaRect l w = l * w
```

```
• Main> areaRect 5 10 50
```

Another example that calculates the area of a triangle $\left(A = \frac{bh}{2}\right)$:

```
areaTriangle b h = (b * h) / 2
```

```
• Main> areaTriangle 3 9 13.5
```

As you can see, multiple arguments are separated by spaces. That's also why you sometimes have to use parentheses to group expressions. For instance, to quadruple a value `x`, you can't write

```
quadruple x = double double x
```

because that would mean to apply a function named `double` to the two arguments `double` and `x`: *functions can be arguments to other functions* (and you will see why later). Instead, you have to put parentheses around the argument:

```
quadruple x = double (double x)
```

Arguments are always passed in the order given. For example:

```
subtract x y = x - y
```

```
• Main> subtract 10 5 5
• Main> subtract 5 10 -5
```

Here, `subtract 10 5` evaluates to `10 - 5`, but `subtract 5 10` evaluates to `5 - 10` because the order changes.

Exercises:

- Write a function to calculate the volume of a box.
- Approximately how many stones are the famous pyramids at Giza made up of? Use GHCi for your calculations.

3.5.3 Remark on combining functions

It goes without saying that you can use functions that you have already defined to define new functions, just like you can use the predefined functions like addition `(+)` or multiplication

`(*)` (operators are defined as functions in Haskell). For example, to calculate the area of a square, we can reuse our function that calculates the area of a rectangle

```
areaRect l w = l * w
areaSquare s = areaRect s s
```

- `Main> areaSquare 5 25`

After all, a square is just a rectangle with equal sides.

This principle may seem innocent enough, but it is really powerful, in particular when we start to calculate with other objects instead of numbers.

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.

3.6 Local definitions

3.6.1 `where` clauses

When defining a function, it is not uncommon to define intermediate results that are *local* to the function. For instance, consider Heron's formula[5] $A = \sqrt{s(s-a)(s-b)(s-c)}$ for calculating the area of a triangle with sides `a`, `b`, and `c`:

```
heron a b c = sqrt (s*(s-a)*(s-b)*(s-c))
    where
    s = (a+b+c) / 2
```

The variable `s` is half the perimeter of the triangle and it would be tedious to write it out four times in the argument of the square root function `sqrt`.

It would be wrong to just write the definitions in sequence

```
heron a b c = sqrt (s*(s-a)*(s-b)*(s-c))  -- s is not defined here
s = (a+b+c) / 2                           -- a, b, and c are not defined here
```

because the variables `a`, `b`, `c` are only available in the right-hand side of the function `heron`, but the definition of `s` as written here is not part of the right-hand side of `heron`. To make it part of the right-hand side, we have to use the `where` keyword.

Note that both the `where` and the local definitions are *indented* by 4 spaces, to distinguish them from subsequent definitions. Here is another example that shows a mix of local and top-level definitions:

[5] http://en.wikipedia.org/wiki/Heron%27s%20formula

```
areaTriangleTrig  a b c = c * height / 2   -- use trigonometry
    where
    cosa   = (b^2 + c^2 - a^2) / (2*b*c)
    sina   = sqrt (1 - cosa^2)
    height = b*sina
areaTriangleHeron a b c = result           -- use Heron's formula
    where
    result = sqrt (s*(s-a)*(s-b)*(s-c))
    s      = (a+b+c)/2
```

3.6.2 Scope

If you look closely at the previous example, you'll notice that we have used the variable names `a`, `b`, `c` twice, once for each of the area functions. How does that work?

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 `0` for the area because of the `let r = 0` definition getting in the way. That does not happen because when you defined `r` the second time 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 you are referring to. Programming has something similar to context, called **scope**[6].

We won't explain the technicalities behind scope (at least not now), just know that the value of a parameter is strictly what you pass in when you call the function, regardless of what the variable was called in the function's definition.

3.7 Summary

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

We also learned that comments allow non-code text to be stored in a source file.

[6] `http://en.wikipedia.org/wiki/Scope%20%28programming%29`

4 Truth values

4.1 Equality and other comparisons

So far we have seen how to use the equals sign to define variables and functions in Haskell. Writing

```
r = 5
```

in a source file will cause occurrences of `r` to be replaced by `5` in all places where it makes sense to do so according to the scope of the definition. Similarly,

```
f x = x + 3
```

causes occurrences of `f` followed by a number (which is taken as `f`'s argument) to be replaced by that number plus three.

In Mathematics, however, the equals sign is also used in a subtly different and equally important way. For instance, consider this simple problem:

Example:
Solve the following equation:$x+3=5$

When we look at a problem like this one, our immediate concern is not the ability to represent the value 5 as $x+3$, or vice-versa. Instead, we read the $x+3=5$ equation as a *proposition*, which says that some number x gives 5 as result when added to 3. Solving the equation means finding which, if any, values of x make that proposition true. In this case, using elementary algebra we can convert the equation into $x=5-3$ and finally to $x=2$, which is the solution we were looking for. The fact that it makes the equation true can be verified by replacing x with 2 in the original equation, leading us to $2+3=5$, which is of course true.

The ability of comparing values to see if they are equal turns out to be extremely useful in programming. Haskell allows us to write such tests in a very natural way that looks just like an equation. The main difference is that, since the equals sign is already used for defining things, we use a *double* equals sign, `==`. To see it at work, you can start GHCi and enter the proposition we wrote above like this:

```
Prelude> 2 + 3 == 5
True
```

GHCi returns "True" lending further confirmation of $2+3$ being equal to 5. As 2 is the only value that satisfies the equation, we would expect to obtain different results with other numbers.

```
Prelude> 7 + 3 == 5
False
```

Nice and coherent. Another thing to point out is that nothing stops us from using our own functions in these tests. Let us try it with the function `f` we mentioned at the start of the module:

```
Prelude> let f x = x + 3
Prelude> f 2 == 5
True
```

Just as expected, since `f 2` is just `2 + 3`.

In addition to tests for equality, we can just as easily compare two numerical values to see which one is larger. Haskell provides a number of tests including: `<` (less than), `>` (greater than), `<=` (less than or equal to) and `>=` (greater than or equal to), which work just like `==` (equal to). For a simple application, we could use `<` alongside the `area` function from the previous module to see whether a circle of a certain radius would have an area smaller than some value.

```
Prelude> let area r = pi * r^2
Prelude> area 5 < 50
False
```

4.2 Boolean values

At this point, GHCi might look like some kind of oracle (or not) which can tell you if propositions are true or false. That's all fine and dandy, but how could that help us to write programs? And what is actually going on when GHCi "answers" such "questions"?

To understand that, we will start from a different but related question. If we enter an arithmetical expression in GHCi the expression gets *evaluated*, and the resulting numerical value is displayed on the screen:

```
Prelude> 2 + 2
4
```

If we replace the arithmetical expression with an equality comparison, something similar seems to happen:

```
Prelude> 2 == 2
True
```

But *what* is that "True" that gets displayed? It certainly does not look like a number. We can think of it as something that tells us about the veracity of the proposition `2 == 2`. From that point of view, it makes sense to regard it as a *value* – except that instead of representing some kind of count, quantity, etc. it stands for the truth of a proposition. Such values are called **truth values**, or **boolean values**[1]. Naturally, there are only two possible boolean values – `True` and `False`.

4.2.1 An introduction to types

When we say `True` and `False` are values, we are not just making an analogy. Boolean values have the same status as numerical values in Haskell, and indeed you can manipulate them just as well. One trivial example would be equality tests on truth values:

```
Prelude> True == True
True
Prelude> True == False
False
```

`True` is indeed equal to `True`, and `True` is not equal to `False`. Now, quickly: can you answer whether `2` is equal to `True`?

```
Prelude> 2 == True

<interactive>:1:0:
      No instance for (Num Bool)
        arising from the literal `2' at <interactive>:1:0
      Possible fix: add an instance declaration for (Num Bool)
      In the first argument of `(==)', namely `2'
      In the expression: 2 == True
      In the definition of `it': it = 2 == True
```

The correct answer is you *can't*, because the question just does not make sense. It is impossible to compare a number with something that is not a number, or a boolean with something that is not a boolean. Haskell incorporates that notion, and the ugly error message we got is, in essence, stating exactly that. Ignoring all of the obfuscating clutter (which we will get to understand eventually) what the message tells us is that, since there was a number (`Num`) on the left side of the `==`, some kind of number was expected on the right side. But a boolean value (`Bool`) is not a number, and so the equality test exploded into flames.

[1] The term is a tribute to the mathematician and philosopher George Boole ^{`http://en.wikipedia.org/wiki/George%20Boole`} .

The general concept, therefore, is that values have **types**, and these types define what we can or cannot do with the values. In this case, for instance, `True` is a value of type `Bool`, just like `False` (as for the `2`, while there is a well-defined concept of number in Haskell the situation is slightly more complicated, so we will defer the explanation for a little while). Types are a very powerful tool because they provide a way to regulate the behaviour of values with rules which *make sense*, making it easier to write programs that work correctly. We will come back to the topic of types many times as they are very important to Haskell, starting with the very next module of this book.

4.3 Infix operators

What we have seen so far leads us to the conclusion that an equality test like `2 == 2` is an expression just like `2 + 2`, and that it also evaluates to a value in pretty much the same way. That fact is actually given a passing mention on the ugly error message we got on the previous example:

```
    In the expression: 2 == True
```

Therefore, when we type `2 == 2` in the prompt and GHCi "answers" `True` it is just evaluating an expression. But there is a deeper truth involved in this process. A hint is provided by the very same error message:

```
     In the first argument of `(==)', namely `2'
```

GHCi called `2` the first *argument* of `(==)`. In the previous module we used the term argument to describe the values we feed a function with so that it evaluates to a result. It turns out that `==` is just a function, which takes two arguments, namely the left side and the right side of the equality test. The only special thing about it is the syntax: Haskell allows two-argument functions with names composed only of non-alphanumeric characters to be used as *infix operators*, that is, placed between their arguments. The only caveat is that if you wish to use such a function in the "standard" way (writing the function name before the arguments, as a *prefix operator*) the function name must be enclosed in parentheses. So the following expressions are completely equivalent:

```
Prelude> 4 + 9 == 13
True
Prelude> (==) (4 + 9) 13
True
```

Writing the expression in this alternative style further drives the point that `(==)` is a function with two arguments just like `areaRect` in the previous module was. What's more, the same considerations apply to the other *relational operators* we mentioned (`<`, `>`, `<=`, `>=`) and to the arithmetical operators (`+`, `*`, etc.) – all of them are just functions. This generality is an illustration of one of the strengths of Haskell – there are few "special cases", and that helps to keep things simple. In general, we could say that all tangible things in Haskell are either values, variables or functions.[2]

4.4 Boolean operations

One nice and useful way of seeing both truth values and infix operators in action are the boolean operations, which allows us to manipulate truth values as in logic propositions. Haskell provides us three basic functions for that purpose:

- `(&&)` performs the *and* operation. Given two boolean values, it evaluates to `True` if both the first and the second are `True`, and to `False` otherwise.

```
Prelude> (3 < 8) && (False == False)
True
Prelude> (&&) (6 <= 5) (1 == 1)
False
```

- `(||)` performs the *or* operation. Given two boolean values, it evaluates to `True` if either the first or the second are `True` (or if both are true), and to `False` otherwise.

```
Prelude> (2 + 2 == 5) || (2 > 0)
True
Prelude> (||) (18 == 17) (9 >= 11)
False
```

- `not` performs the negation of a boolean value; that is, it converts `True` to `False` and vice-versa.

```
Prelude> not (5 * 2 == 10)
False
```

One relational operator we didn't mention so far in our discussions about comparison of values is the *not equal to* operator. It is also provided by Haskell as the `(/=)` function, but if we had to implement it a very natural way of doing so would be:

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

Note that it is perfectly legal syntax to write the operators infix, even when defining them. Another detail to note is that completely new operators can be created out of ASCII symbols (basically, those that are found on the keyboard).

[2] In case you found this statement to be quite bold, don't worry – we will go even further in due course.

4.5 Guards

Earlier on in this module we proposed two questions about the operations involving truth values: what was actually going on when we used them and how they could help us in the task of writing programs. While we now have a sound initial answer for the first question, the second one could well look a bit nebulous to you at this point, as we did little more than testing one-line expressions here. We will tackle this issue by introducing a feature that relies on boolean values and operations and allows us to write more interesting and useful functions: *guards*.

To show how guards work, we are going to implement the absolute value function. The absolute value of a number is the number with its sign discarded[3]; so if the number is negative (that is, smaller than zero) the sign is inverted; otherwise it remains unchanged. We could write the definition as:

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

The key feature of the definition is that the actual expression to be used for calculating $|x|$ depends on a set of propositions made about x. If $x \geq 0$ we use the first expression, but if $x < 0$ we use the second one instead. If we are going to implement the absolute value function in Haskell we need a way to express this decision process. That is exactly what guards help us to do. Using them, the implementation could look like this:[4]

Example:
The abs function.

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

Remarkably, the above code is almost as readable as the corresponding mathematical definition. In order to see how the guard syntax fits with the rest of the Haskell constructs, let us dissect the components of the definition:

- We start just like in a normal function definition, providing a name for the function, `abs`, and saying it will take a single parameter, which we will name `x`.
- Instead of just following with the `=` and the right-hand side of the definition, we entered a line break, and, following it, the two alternatives, placed in separate lines.[5] These alternatives are the *guards* proper. An important observation is that the whitespace is not there just for aesthetic reasons, but it is necessary for the code to be parsed correctly.

[3] Technically, that just covers how to get the absolute value of a *real* number, but let's ignore this detail for now.

[4] `abs` is also provided by Haskell, so in a real-world situation you don't need to worry about providing an implementation yourself.

[5] We *could* have joined the lines and written everything in a single line, but in this case it would be a lot less readable.

- Each of the guards begins with a pipe character, `|`. After the pipe, we put an expression which evaluates to a boolean (also called a boolean condition or a *predicate*), which is followed by the rest of the definition – the equals sign and the right-hand side which should be used if the predicate evaluates to `True`.
- The `otherwise` deserves some additional explanation. If none of the preceding predicates evaluate to `True`, the `otherwise` guard will be deployed by default. In this case, if `x` is not smaller than zero, it must be greater than or equal to zero, so the final predicate could have just as easily been `x >= 0`; `otherwise` is used here for the sake of convenience and readability.

Note:
There is no syntactical magic behind `otherwise`. It is defined alongside the default variables and functions of Haskell as simply

```
otherwise = True
```

This definition makes for a catch-all guard since evaluation of the guard predicates is sequential, and so the always true `otherwise` predicate will only be reached if none of the other ones evaluates to `True` (that is, assuming you place it as the last guard!). In general it is a good idea to always provide an `otherwise` guard, as if none of the predicates is true for some input a rather ugly runtime error will be produced.

Note:
You might be wondering why we wrote `0 - x` and not simply `-x` to denote the sign inversion. Truth is, we could have written the first guard as

```
    | x < 0    = -x
```

and it would have worked just as well. The only issue is that this way of expressing sign inversion is actually one of the few "special cases" in Haskell, in that this `-` is *not* a function that takes one argument and evaluates to `0 - x`, but just a syntactical abbreviation. While very handy, this shortcut occasionally conflicts with the usage of `(-)` as an actual function (the subtraction operator), which is a potential source of annoyance (for one of several possible issues, try writing three minus minus four without using any parentheses for grouping). In any case, the only reason we wrote `0 - x` explicitly on the example was so that we could have an opportunity to make this point clear in this brief digression.

4.5.1 `where` and Guards

`where` clauses are particularly handy when used with guards. For instance, consider this function, which computes the number of (real) solutions for a quadratic equation[6], $ax^2+bx+c=0$:

[6] http://en.wikipedia.org/wiki/Quadratic%20equation

```
numOfSolutions a b c
    | disc > 0  = 2
    | disc == 0 = 1
    | otherwise = 0
        where
        disc = b^2 - 4*a*c
```

The `where` definition is within the scope of all of the guards, sparing us from repeating the expression for `disc`.

5 Type basics

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

5.1 Introduction

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

$2+3$

What are 2 and 3? We can quite simply describe them as numbers. And what about the plus sign in the middle? That's certainly not a number, but it stands for an operation which we can do with two numbers – namely, addition.

Similarly, consider a program that asks you for your name and then greets you with a "Hello" message. Neither your name nor the word Hello are numbers. What are they then? We might refer to all words and sentences and so forth as text. It's normal in programming to use a slightly more esoteric word: *String*, which is short for "string of characters".

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

Databases illustrate clearly the concept of 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 in each entry 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 categorizing data, so let us see how we could classify the values in this example. The first three fields seem straightforward enough. "First Name" and "Last Name" contain text, so we say that the values are of type String, while "Telephone Number" is clearly a number.

At first glance one may be tempted to classify address as a String. However, the semantics behind an innocent address are quite complex. There are a whole lot of human conventions that dictate how we interpret it. For example, if the beginning of the address text contains a number it is likely the number of the house. If not, then it's probably the name of the house – except if it starts with "PO Box", in which case 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, as each part of the address has its own meaning. In principle there is nothing wrong with saying addresses are Strings, but when we describe something as a String, all that we are saying is that it is a sequence of letters, numbers, etc. Recognizing something as a specialized type, say, Address, is far more meaningful. If we know something is an Address, we instantly know much more about the piece of data – for instance, that we can interpret it using the "human conventions" that give meaning to addresses.

In retrospect, we might also apply this rationale to the telephone numbers. It could be a good idea to speak in terms of a TelephoneNumber type. Then, if we were to come across some arbitrary sequence of digits which happened to be of type TelephoneNumber we would have access to a lot more information than if it were just a Number – for instance, we could start looking for things such as area and country codes on the initial digits.

Another reason not to consider the telephone numbers as just Numbers is that doing arithmetics with them makes no sense. What is the meaning and expected effect of, say, adding 1 to a TelephoneNumber? It would not allow calling anyone by phone. That's a good reason for using a more specialized type than Number. Also, each digit comprising a telephone number is important; it's not acceptable to lose some of them by rounding it or even by omitting leading zeroes.

5.1.1 Why types are useful

So far, it seems that all what we've done was to describe and categorize things, and it may not be obvious why all of this talk would be so important for writing actual programs. Starting with this module, we will explore how Haskell uses types to the programmer's benefit, allowing us to incorporate the semantics behind, say, an address or a telephone number seamlessly in the code.

5.2 Using the interactive `:type` command

The best way to explore how types work in Haskell is from GHCi. The type of any expression can be checked with the immensely useful `:type` (or `:t`) command. Let us test it on the boolean values from the previous module:

Example:
Exploring the types of boolean values in GHCi

```
Prelude> :type True
True :: Bool
Prelude> :type False
False :: Bool
Prelude> :t (3 < 5)
(3 < 5) :: Bool
```

Usage of `:type` is straightforward: enter the command into the prompt followed by whatever you want to find the type of. On the third example, we use `:t`, which we will be using from now on. GHCi will then print the type of the expression. The symbol `::`, which will appear in a couple other places, can be read as simply "is of type", and indicates a *type signature*.

`:type` reveals that truth values in Haskell are of type `Bool`, as illustrated above for the two possible values, `True` and `False`, as well as for a sample expression that will evaluate to one of them. It is worthy to note at this point that boolean values are not just for value comparisons. `Bool` captures in a very simple way the semantics of a yes/no answer, and so it can be useful to represent any information of such kind – say, whether a name was found in a spreadsheet, or whether a user has toggled an on/off option.

5.2.1 Characters and strings

Now let us try `:t` on something new. Literal characters are entered by enclosing them with single quotation marks. For instance, this is the single letter H:

Example:
Using the :type command in GHCi on a literal character

```
Prelude> :t 'H'
'H' :: Char
```

Literal character values, then, have type `Char` (short for "character"). Single quotation marks, however, only work for individual characters. If we need to enter actual text – that is, a string of characters – we use *double* quotation marks instead:

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

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

Why did we get `Char` again? The difference is in the square brackets. `[Char]` means a number of characters chained together, forming a *list*. That is what text strings are in Haskell – lists of characters.[1]

Exercises:

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

A nice thing to be aware of is that Haskell allows for *type synonyms*, which work pretty much like synonyms in human languages (words that mean the same thing – say, 'fast' and 'quick'). In Haskell, type synonyms are alternative names for types. For instance, `String` is defined as a synonym of `[Char]`, and so we can freely substitute one with the other. Therefore, to say:

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

is also perfectly valid, and in many cases a lot more readable. From here on we'll mostly refer to text values as `String`, rather than `[Char]`.

5.3 Functional types

So far, we have seen how values (strings, booleans, characters, etc.) have types and how these types help us to categorize and describe them. Now, the big twist, and what makes the Haskell's type system truly powerful: Not only values, but *functions* have types as well[2]. Let's look at some examples to see how that works.

5.3.1 Example: `not`

Consider `not`, that negates boolean values (changing `True` to `False` and vice-versa). To figure out the type of a function we consider two things: the type of values it takes as its input and the type of value it returns. In this example, things are easy. `not` takes a `Bool` (the `Bool` to be negated), and returns a `Bool` (the negated `Bool`). The notation for writing that down is:

Example:
Type signature for `not`

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

[1] Lists, be they of characters or of other things, are very important entities in Haskell, and we will cover them in more detail in a little while.

[2] The deeper truth is that functions *are* values, just like all the others.

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

Using `:t` on a function will work just as expected:

```
Prelude> :t not
not :: Bool -> Bool
```

The description of a function's type is in terms of the types of argument(s) it takes and gives.

5.3.2 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 are 1s and 0s: computers work in binary. As working with binary numbers isn't at all 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 computer generally takes care of the conversion to binary numbers without our intervention.

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`[3]:

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

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

We already know what `Char` means. The new type on the signatures above, `Int`, amounts to integer numbers, and is one of quite a few different types of numbers.[4] The type signature of `chr` tells us that it takes an argument of type `Int`, an integer number, and evaluates to a result of type Char. The converse is the case with `ord`: It takes things of type Char and returns things of type Int. With the info from the type signatures, it becomes immediately clear which of the functions encodes a character into a numeric code (`ord`) and which does the decoding back to a character (`chr`).

To make things more concrete, here are a few examples of function calls to `chr` and `ord`. Notice that the two functions aren't available by default; so before trying them in GHCi you

[3] This isn't quite what `chr` and `ord` do, but that description fits our purposes well, and it's close enough.

[4] In fact, it is not even the only type for integers! We will meet its relatives in a short while.

need to use the `:module Data.Char` (or `:m Data.Char`) command to load the Data.Char module, where they are defined.

Example:
Function calls to <code>chr</code> and <code>ord</code>

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

5.3.3 Functions with more than one argument

The style of type signatures we have been using works fine enough for functions of one argument. But what would be the type of a function like this one?

Example:
A function with more than one argument

```
xor p q = (p || q) && not (p && q)
```

(`xor` is the exclusive-or function, which evaluates to `True` if either one or the other argument is `True`, *but not both*; and `False` otherwise.)

The general technique for forming the type of a function that accepts more than one argument is simply to write down all the types of the arguments in a row, in order (so in this case `p` first then `q`), then link them all with `->`. Finally, add the type of the result to the end of the row and stick a final `->` in just before it.[5] In this example, we have:

1. Write down the types of the arguments. In this case, the use of `(||)` and `(&&)` gives away that `p` and `q` have to be of type `Bool`:

```
Bool                 Bool
^^ p is a Bool       ^^ q is a Bool as well
```

2. Fill in the gaps with `->`:
```
Bool -> Bool
```
3. Add in the result type and a final `->`. In our case, we're just doing some basic boolean operations so the result remains a Bool.

```
Bool -> Bool -> Bool
                ^^ We're returning a Bool
            ^^ This is the extra -> that got added in
```

[5] This method might seem just a trivial hack by now, but actually there are very deep reasons behind it, which we'll cover in the chapter on Currying ^{Chapter19 on page 131}.

The final signature, then, is:

Example:
The signature of `xor`

```
xor :: Bool -> Bool -> Bool
```

5.3.4 Real-World Example: `openWindow`

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

As you'll learn in the Haskell in Practice section of the course, one popular group of Haskell libraries are the GUI (**G**raphical **U**ser **I**nterface) ones. These provide functions for dealing with all the parts of Windows, Linux, or Mac OS 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, 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]:

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 (which typically appears in a title bar at the very top of the window), and `WindowSize` specifies how big the window should be. The function then returns a value of type `Window` which represents the actual window.

One key point illustrated by this example, as well as the `chr`/`ord` one is that, even if you have never seen the function or don't know how it actually works, a type signature can immediately give you a good general idea of what the function is supposed to do. For that reason, a very useful habit to acquire is testing every new function you meet with `:t`. If you start doing so right now you'll not only learn about the standard library Haskell functions quite a bit quicker but also develop a useful kind of intuition. Curiosity pays off. :)

[6] This has been somewhat simplified to fit our purposes. Don't worry, the essence of the function is there.

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 'or', that takes two Bools and returns a third Bool which is True if either of the arguments were, and False otherwise.
3. A `monthLength` function which takes a Bool which is True if we are considering a leap year and False otherwise, and an Int which is the number of a month; and returns another Int which is the number of days in that month. For any functions hereafter involving numbers, you can just pretend the numbers are Ints.
4. `f x y = not x && y`
5. `g x = (2*x - 1)^2`

5.4 Type signatures in code

Now we've explored the basic theory behind types as well as how they apply to Haskell. The key way in which type signatures are used is for annotating functions in source files. Let us see what that looks like for `xor` function from an earlier example:

Example:
A function with its signature

```
xor :: Bool -> Bool -> Bool
xor p q = (p || q) && not (p && q)
```

That is all we have to do, really. Signatures are placed just before the corresponding functions, for maximum clarity.

The signatures we add in this way serve a dual role. They inform the type of the functions both to human readers of the code and to the compiler/interpreter.

5.4.1 Type inference

We just said that type signatures tell the interpreter (or compiler) what the function type is. However, up to now you wrote perfectly good Haskell code without any signatures, and it was accepted by GHC/GHCi. That shows that in general it is not mandatory to add type signatures. But that doesn't mean that if you don't add them Haskell simply forgets about types altogether! Instead, when you didn't tell Haskell the types of your functions and variables it *figured them out* through a process called *type inference*. In essence, the compiler performs inference by starting with the types of things it knows and then working out the types of the rest of the values. Let's see how that works with a general example.

Example:
Simple type inference

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

`isL` is a function that takes an argument `c` and returns the result of evaluating `c == 'l'`. If we don't provide a type signature the compiler, in principle, does not know the type of `c`, nor the type of the result. In the expression `c == 'l'`, however, it does know that `'l'` is a `Char`. Since `c` and `'l'` are being compared with equality with `(==)` and both arguments of `(==)` must have the same type[7], it follows that `c` must be a `Char`. Finally, since `isL c` is the result of `(==)` it must be a `Bool`. And thus we have a signature for the function:

Example:
`isL` with a type

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

And, indeed, if you leave out the type signature the Haskell compiler will discover it through this process. You can verify that by using `:t` on `isL` with or without a signature.

So, if type signatures are optional in most cases[8] why should we care so much about them? Here are a few reasons:

- **Documentation**: type signatures make your code easier to read. With most functions, the name of the function along with the type of the function is sufficient to guess what the function does. Of course, you should always comment your code properly too, but having the types clearly stated helps a lot, too.
- **Debugging**: if you annotate a function with a type signature and then make a typo in the body of the function which changes the type of a variable 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.

5.4.2 Types and readability

To understand better how signatures can help documentation, let us have a glance at a somewhat more realistic example. The piece of code quoted below is a tiny *module* (modules are the typical way of preparing a library), and this way of organizing code is

[7] As we discussed in "Truth values". That fact is actually stated by the type signature of `(==)` – if you are curious you can check it, although you will have to wait a little bit more for a full explanation of the notation used in it.

[8] There are a few situations in which the compiler lacks information to infer the type, and so the signature becomes obligatory; and, in some other cases, we can influence to a certain extent the final type of a function or value with a signature. That needn't concern us for the moment, however.

not too different from what you might find, say, when reading source code for the libraries bundled with GHC.

Note:
Do not go crazy trying to understand how the functions here actually work; that is beside the point as we still have not covered any of the features being used. Just keep reading the text and play along.

Example:
Module with type signatures

```
module StringManip where

import Data.Char

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

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

This tiny library provides three string manipulation functions. `uppercase` converts a string to upper case, `lowercase` to lower case, and `capitalize` capitalizes the first letter of every word. Each of these functions takes a `String` as argument and evaluates to another `String`. What is relevant to our discussion here is that, even if we do not understand how these functions work, looking at the type signatures allows us to immediately know the types of the arguments and return values. That information, when paired with sensible function names, can make it a lot easier to figure out how we can use the functions.

Note that when functions have the same type we have the option of writing just one signature for all of them, by separating their names with commas, as it was done with `uppercase` and `lowercase`.

5.4.3 Types prevent errors

The role of types in preventing errors is 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 pass the *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! In all likelihood the intention was to use the similar-looking 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, or has unforeseen consequences, 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": run the 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 having a strong type system like Haskell does.

6 Lists and tuples

Lists and tuples are the two most fundamental ways of manipulating several values together, by grouping them into a single value.

6.1 Lists

Functions are one of the two major building blocks of any Haskell program. The other is the list. So let's switch over to the interpreter and build lists:

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

The square brackets delimit the list, and individual elements are separated by commas. The only important restriction is that all elements in a list must be of the same type. Trying to define a list with mixed-type elements results in a typical type error:

```
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"]
```

6.1.1 Building lists

In addition to specifying the whole list at once using square brackets and commas, you can build them up piece by piece using the `(:)` operator. This process is often referred to as **consing**[1].

[1] You might object that "cons" isn't even a proper word. Well, it isn't. LISP programmers invented the verb "to cons" to refer to this specific task of appending an element to the front of a list. "cons" happens to be a mnemonic for "constructor". Later on we will see why that makes sense.

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. Therefore you can keep on consing for as long as you wish. It is important to note that the cons operator evaluates from right to left. Another (more general) way to think of it is that it takes the first value to its left and the whole expression to its right.

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 how lists are actually built, by consing all elements to the *empty list*, `[]`. The commas-and-brackets notation is just *syntactic sugar*, a more pleasant way to write code. So `[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 something like `True:False:[]` is perfectly good Haskell, `True:False` is *not*:

Example:
Whoops!

```
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
```

`True:False` produces a familiar-looking type error message, which tells us that the cons operator `(:)` (which is really just a function) expected a list as its second argument but we gave it another `Bool` instead. `(:)` only knows how to stick things onto lists.[2]

So, when using cons, remember:

- The elements of the list must have the same type.
- You can only cons `(:)` something onto a list, not the other way around (you cannot cons a list onto an element). So, the final item on the right must be a list, and the items on the left must be independent elements, not lists.

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` (at the beginning) on to it. Test it out on the following lists by doing:
   a) `cons8 []`
   b) `cons8 [1,2,3]`
   c) `cons8 [True,False]`
   d) `let foo = cons8 [1,2,3]`
   e) `cons8 foo`
3. Adapt the above function in a way that `8` is at the end of the list
4. 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 =`

6.1.2 Strings are just lists

As we briefly mentioned in the Type Basics module, strings in Haskell are just lists of characters. That means values of type String can be manipulated just like any other list. For instance, instead of entering strings directly as a sequence of characters enclosed in double quotation marks, they may also be constructed through a sequence of Char values, either linked with `(:)` and terminated by an empty list or using the commas-and-brackets notation.

```
Prelude>"hey" == ['h','e','y']
True
Prelude>"hey" == 'h':'e':'y':[]
True
```

Using double-quoted strings is just more syntactic sugar.

[2] At this point you might be justified in seeing types as an annoying thing. In a way they are, but more often than not they are actually a lifesaver. In any case, when you are programming in Haskell and something blows up, you'll probably want to think "type error".

6.1.3 Lists within lists

Lists can contain *anything*, just as long as they are all of the same type. Because lists are things too, lists can contain 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; the type `Int`, for example, is different from `[Int]`. 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.
   a) `[1,2,3,[]]`
   b) `[1,[2,3],4]`
   c) `[[1,2,3],[]]`
2. Which of these are valid Haskell, and which are not? Rewrite in comma and bracket notation.
   a) `[]:[[1,2,3],[4,5,6]]`
   b) `[]:[]`
   c) `[]:[]:[]`
   d) `[1]:[]:[]`
   e) `["hi"]:[1]:[]`
3. Can Haskell have lists of lists of lists? Why or why not?
4. Why is the following list invalid in Haskell?
   a) `[[1,2],3,[4,5]]`

Lists of lists can be useful because they allow one to express some kinds of 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 co-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.

6.2 Tuples

6.2.1 A different notion of many

Tuples are another way of storing multiple values in a single value. There are two key differences between tuples and lists:

- They have a *fixed* number of elements (*immutable*), and so you can't cons to a tuple. Therefore, it makes sense to use tuples when you *know* in advance how many values are to be stored. For example, we might want a type for storing 2D coordinates of a point. We know exactly how many values we need for each point (two – the x and y coordinates), so tuples are applicable.
- The elements of a tuple do not need to be all of the same type. For instance, in a phonebook application we might want to handle the entries by crunching three values into one: the name, phone number, and the address of each person. In such a case, the three values won't have the same type, so lists wouldn't help, but tuples would.

Tuples are created within parentheses with elements delimited by commas. 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 is False. The third example is a tuple consisting of *five* elements: the first is 4 (a number), the second is 5 (another number), the third is "Six" (a string), the fourth is True (a boolean value), and the fifth is 'b' (a character).

A quick note on nomenclature: In general you use *n-tuple* to denote 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, but if you were to logically extend the naming system, you'd have *quadruples*, *quintuples* and so on, hence the general term *tuple*.

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?
   a) `(4, 4)`
   b) `(4, "hello")`
   c) `(True, "Blah", "foo")`
3. Lists can be built by consing new elements onto 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.
   a) Why do you think that is?
   b) Say for the sake of argument, that there was such a function. What would you get if you "consed" something on a tuple?

Tuples are also handy when you want to return more than one value from a function. In many languages, returning two or more things at once often requires wrapping them up in a single-purpose data structure, maybe one that only gets used in that function. In Haskell, you have the very convenient alternative of just returning them as a tuple.

6.2.2 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, and all sorts of other combinations along the same lines.

Example:
Nesting tuples and lists

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

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 `("Hello",32)` and `(47,"World")` are fundamentally different. One is of type `(String,Int)`, 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?
   - `1:(2,3)`
   - `(2,4):(2,3)`
   - `(2,4):[]`
   - `[(2,4),(5,5),('a','b')]`
   - `([2,4],[2,2])`

6.3 Retrieving values

Up to now we have seen how to put values into lists and tuples. If they are to be of any use, though, there must be a way of getting back the stored values! In this section, we will explore the surface of this issue.

Let us begin with pairs (that is, 2-tuples). A very common use for them is to store the (*x*, *y*) coordinates of a point: imagine you have a chess board, and want to specify a specific square. You could do this by labelling all the rows from 1 to 8, and similarly with the columns. Then, a pair `(2, 5)` could 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 coordinates of all the pieces, then look at the row part and see whether it's equal to whatever row we're being asked to examine. This function would need, once it had the coordinate pair `(x, y)` of a piece, to extract the `x` (the row coordinate). To do that there are two functions, `fst` and `snd`, that retrieve[3] the first and second elements out of a pair, respectively. Let's see some examples:

Example:
Using <code>fst</code> and <code>snd</code>

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

Note that these functions *only* work on pairs. Why? Yet again, it has to do with types. Pairs and triples (and quadruples, etc.) have necessarily different types, and `fst` and `snd` only accept pairs as arguments.

As for lists, the functions `head` and `tail` are *roughly* analogous to `fst` and `snd`, in that they disassemble a list by taking apart what `(:)` joined: `head` evaluates to the first element of the list, while `tail` gives the rest of the list.

Example:
Using <code>head</code> and <code>tail</code>

```
Prelude> 2:[7,5,0]
[2,7,5,0]
Prelude> head [2,7,5,0]
2
Prelude> tail [2,7,5,0]
[7,5,0]
```

Note:
An important caveat: if we apply `head`, or `tail`, to an empty list...

```
Prelude> head []
• • • Exception: Prelude.head: empty list
```

... it blows up, as an empty list has no first element, nor any other elements at all.

[3] Or, more technically, which *project*. In math-speak, a function that gets some data out of a structure is called a projection.

6.3.1 Pending questions

The four functions introduced here do not appear to be enough to fully solve the problem we started this section with. While `fst` and `snd` provide a satisfactory solution for pairs, what about tuples with three or more elements? And with lists, it would be natural to wonder if we can't do any better than just breaking them after the first element - `head` and `tail` just don't seem to cut it. For the moment, we will have to leave these questions pending, but don't worry - the issues are perfectly solvable. Once we do some necessary groundwork we will return to this subject, considering it in detail and, in particular, dedicating a number of chapters to list manipulation. There, we will understand how separating head and tail allows us to do anything we want with lists.

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 and the columns a-h; and the column label is customarily given first. Could we label a specific point with a character and a number, like `('a', 4)`? What important difference with lists does this illustrate?
3. Write a function which returns the head and the tail of a list as the first and second elements of a tuple.
4. Use `head` and `tail` to write a function which gives the fifth element of a list. Then, make a critique of it, pointing out any annoyances and pitfalls you notice.

6.4 Polymorphic types

As you may have found out already by playing with `:t`, the type of a list depends on the types of its elements, and is denoted by enclosing it in square brackets:

```
Prelude>:t [True, False]
[True, False] :: [Bool]
Prelude>:t ["hey", "my"]
["hey", "my"] :: Char[4]
```

Therefore, lists of `Bool` have different types than lists of `[Char]` (that is, strings), of `Int` and so on. Since we have seen that functions only accept arguments of the types specified in the type of the function, that leads to some practical complication. For example, imagine a function that finds the length of a list. But since `[Int]`, `[Bool]` and `[String]` are different types it seems we would need separate functions for each case – `lengthInts :: [Int] -> Int`, as well as a `lengthBools :: [Bool] -> Int`, as well as a `lengthStrings :: [String] -> Int`, as well as a...

That would be horribly annoying, and frustrating too, because intuitively it would seem that counting how many things there are in a list should be independent of what the things actually are. Fortunately, it does not work like that: there is a single function `length`, which works on all lists. But how can that possibly work? As usual, checking the type of `length` provides a good hint that there is something different going on...

Example:
Our first polymorphic type
Prelude>:t length
length :: [a] -> Int

The `a` in the square brackets is *not* a type – remember that type names always start with uppercase letters. Instead, it is a **type variable**. 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 are called *monomorphic*, and things that use type variables to admit more than one type are *polymorphic*.

It is important to note that, in a single type signature, all cases of a certain type variable must be of the same type. For example,

```
f :: a -> a
```

means that `f` takes an argument of any type and gives something of the *same type as the argument*, as opposed to

```
f :: a -> b
```

which means that `f` takes an argument of any type and gives an argument of another type, which is *not necessarily the same type, but can be.*

6.4.1 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 already be developing the habit of wondering "what type is that function?" about every function you come across. Let's consider the cases of `fst` and `snd` . These two functions take a pair as their argument and return one element of this pair. First of all, the type of a pair depends on the type of its elements, just as with lists, so the functions need to be polymorphic. Also it is important to keep in mind that pairs, and tuples in general, don't have to be homogeneous with respect to types; their different parts can be different types. So if we were to say:

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

That would mean `fst` would only work if the first and second part of the pair given as input had the same type. So what is the correct type? Simply:

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

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

`b` is a second type variable, which stands for a type which may or may not be equal to the one which replaces `a`.

Note that if you knew nothing about `fst` and `snd` other than the type signatures you might guess that they return the first and second parts of a pair, respectively. Although that is correct, it is not necessarily true as all the signatures say is that they just have to return something *with the same type* of the first and second parts of the pair.

Exercises:
Give type signatures for the following functions:

1. The solution to the third exercise of the previous section ("... a function which returns the head and the tail of a list as the first and second elements of a tuple").
2. The solution to the fourth exercise of the previous section ("... a function which gives the fifth element of a list").
3. `h x y z = chr (x - 2)` (remember we discussed chr in the previous chapter).

6.5 Summary

We have introduced two new notions in this chapter, lists and tuples. Let us sum up the key similarities and differences between them:

1. Lists are defined by square brackets and commas : `[1,2,3]`.
   - They can contain *anything* as long as all the candidate 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
   - 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, but their criteria must still be fulfilled for the combinations to be valid.

7 Type basics II

Up to now we have shrewdly avoided number types in our examples. In one exercise, we even went as far as asking you to "pretend" the arguments to `(+)` had to be of type `Int`. So, from what are we hiding?

The main theme of this module will be how numerical types are handled in Haskell. While doing so, we will introduce some important features of the type system. Before diving into the text, though, pause for a moment and consider the following question: what should be the type of the function `(+)`[1]?

7.1 The `Num` class

As far as everyday Mathematics is concerned, there are very few restrictions on which kind of numbers we can add together. $2+3$ (two natural numbers), $(-7)+5.12$ (a negative integer and a rational number), $\frac{1}{7}+\pi$ (a rational and an irrational)... all of these are valid – indeed, any two real numbers can be added together. In order to capture such generality in the simplest way possible we would like to have a very general `Number` type in Haskell, so that the signature of `(+)` would be simply

```
(+) :: Number -> Number -> Number
```

That design, however, does not fit well with the way computers perform arithmetic. While integer numbers in programs can be quite straightforwardly handled as sequences of binary digits in memory, that approach does not work for non-integer real numbers[2], thus making it necessary for a more involved encoding to support them: floating point numbers[3]. While floating point provides a reasonable way to deal with real numbers in general, it has some inconveniences (most notably, loss of precision) which make using the simpler encoding worthwhile for integer values. We are thus left with at least two different ways of storing numbers, one for integers and another one for general real numbers, which should correspond to different Haskell types. Furthermore, computers are only able to perform operations like `(+)` on a pair of numbers if they are in the same format. That should put an end to our hopes of using a universal `Number` type – or even having `(+)` working with both integers and floating-point numbers...

[1] If you followed our recommendations in "Type basics", chances are you have already seen the rather exotic answer by testing with `:t`... if that is the case, consider the following analysis as a path to understanding the meaning of that signature.

[2] One of the reasons being that between any two real numbers there are infinitely many real numbers – and that can't be directly mapped into a representation in memory no matter what we do.

[3] `http://en.wikipedia.org/wiki/Floating%20point`

It is easy, however, to see reality is not that bad. We *can* use `(+)` with both integers and floating point numbers:

```
Prelude>3 + 4
7
Prelude>4.34 + 3.12
7.46
```

When discussing lists and tuples, we saw that functions can accept arguments of different types if they are made *polymorphic*. In that spirit, one possible type signature for `(+)` that would account for the facts above would be:

```
(+) :: a -> a -> a
```

`(+)` would then take two arguments of the same type `a` (which could be integers or floating-point numbers) and evaluate to a result of type `a`. There is a problem with that solution, however. As we saw before, the type variable `a` can stand for *any* type at all. If `(+)` really had that type signature we would be able to add up two `Bool`, or two `Char`, which would make no sense – and is indeed impossible. Rather, the actual type signature of `(+)` takes advantage of a language feature that allows us to express the semantic restriction that `a` can be any type *as long as it is a number type*:

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

`Num` is a **typeclass** - a group of types which includes all types which are regarded as numbers[4]. The `(Num a) =>` part of the signature restricts `a` to number types – or, more accurately, *instances* of `Num`.

7.2 Numeric types

But what are the *actual* number types – the instances of `Num` that `a` stands for in the signature? The most important numeric types are `Int`, `Integer` and `Double`:

- `Int` corresponds to the vanilla integer type found in most languages. It has fixed precision, and thus maximum and minimum values (in 32-bit machines the range goes from -2147483648 to 2147483647).
- `Integer` also is used for integer numbers, but unlike `Int` it supports arbitrarily large values – at the cost of some efficiency.
- `Double` is the double-precision floating point type, and what you will want to use for real numbers in the overwhelming majority of cases (there is also `Float`, the single-precision counterpart of `Double`, which in general is not an attractive option due to more loss of precision).

These types are available by default in Haskell, and are the ones you will generally deal with in everyday tasks.

[4] That is a very loose definition, but will suffice until we are ready to discuss typeclasses in more detail.

7.2.1 Polymorphic guesswork

There is one thing we haven't explained yet, though. If you tried the examples of addition we mentioned at the beginning you know that something like this is perfectly valid:

```
Prelude> (-7) + 5.12
-1.88
```

Here, it seems we are adding two numbers of different types – an integer and a non-integer. Shouldn't the type of `(+)` make that impossible?

To answer that question we have to see what the types of the numbers we entered actually are:

```
Prelude> :t (-7)
(-7) :: (Num a) => a
```

And, lo and behold, `(-7)` is neither `Int` nor `Integer`! Rather, it is a *polymorphic constant*, which can "morph" into any number type if need be. The reason for that becomes clearer when we look at the other number...

```
Prelude> :t 5.12
5.12 :: (Fractional t) => t
```

`5.12` is also a polymorphic constant, but one of the `Fractional` class, which is more restrictive than `Num` – every `Fractional` is a `Num`, but not every `Num` is a `Fractional` (for instance, `Int`s and `Integer`s are not).

When a Haskell program evaluates `(-7) + 5.12`, it must settle for an actual type for the numbers. It does so by performing type inference while accounting for the class specifications. `(-7)` can be any `Num`, but there are extra restrictions for `5.12`, so its type will define what `(-7)` will become. Since there is no other clues to what the types should be, `5.12` will assume the default `Fractional` type, which is `Double`; and, consequently, `(-7)` will become a `Double` as well, allowing the addition to proceed normally and return a `Double`[5].

There is a nice quick test you can do to get a better feel of that process. In a source file, define

```
x = 2
```

then load the file in GHCi and check the type of `x`. Then, change the file to add a `y` variable,

[5] *For seasoned programmers:* This appears to have the same effect of what programs in C (and many other languages) would manage with an *implicit cast* – in which case the integer literal would be silently converted to a double. The difference is that in C the conversion is done behind your back, while in Haskell it only occurs if the variable/literal is explicitly made a polymorphic constant. The difference will become clearer shortly, when we show a counter-example.

```
x = 2
y = x + 3
```

reload it and check the types of `x` and `y`. Finally, modify `y` to

```
x = 2
y = x + 3.1
```

and see what happens with the types of both variables.

7.2.2 Monomorphic trouble

The sophistication of the numerical types and classes occasionally leads to some complications. Consider, for instance, the common division operator `(/)`. It has the following type signature:

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

Restricting `a` to fractional types is a must because the division of two integer numbers in general will not result in an integer. Nevertheless, we can still write something like

```
Prelude> 4 / 3
1.3333333333333333
```

because the literals `4` and `3` are polymorphic constants and therefore assume the type `Double` at the behest of `(/)`. Suppose, however, we want to divide a number by the length of a list[6]. The obvious thing to do would be using the `length` function:

```
Prelude> 4 / length [1,2,3]
```

Unfortunately, that blows up:

```
<interactive>:1:0:
     No instance for (Fractional Int)
       arising from a use of `/' at <interactive>:1:0-17
     Possible fix: add an instance declaration for (Fractional Int)
     In the expression: 4 / length [1, 2, 3]
     In the definition of `it': it = 4 / length [1, 2, 3]
```

As usual, the problem can be understood by looking at the type signature of `length`:

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

[6] A reasonable scenario – think of computing an average of the values in a list.

The result of `length` is not a polymorphic constant, but an `Int`; and since an `Int` is not a `Fractional` it can't fit the signature of `(/)`.

There is a handy function which provides a way of escaping from this problem. Before following on with the text, try to guess what it does only from the name and signature:

```
fromIntegral :: (Integral a, Num b) => a -> b
```

`fromIntegral` takes an argument of some `Integral` type (like `Int` or `Integer`) and makes it a polymorphic constant. By combining it with `length` we can make the length of the list fit into the signature of `(/)`:

```
Prelude> 4 / fromIntegral (length [1,2,3])
1.3333333333333333
```

While this complication may look spurious at first, this way of doing things makes it easier to be rigorous when manipulating numbers. If you define a function that takes an `Int` argument you can be entirely sure that it will never be converted to an `Integer` or a `Double` unless you explicitly tell the program to do so (for instance, by using `fromIntegral`). As a direct consequence of the refinement of the type system, there is a surprising diversity of classes and functions for dealing with numbers in Haskell.

7.3 Classes beyond numbers

There are many other use cases for typeclasses beyond arithmetic. For example, the type signature of `(==)` is:

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

Like `(+)` or `(/)`, `(==)` is a polymorphic function. It compares two values of the same type, which must belong to the class `Eq`, and returns a `Bool`. `Eq` is simply the class of types of values which can be compared for equality, and includes all of the basic non-functional types[7].

Typeclasses are a very general language feature which adds a lot to the power of the type system. Later in the book we will return to this topic to see how to use them in custom ways, which will allow you to appreciate their usefulness in its fullest extent.

[7] Comparing two functions for equality is considered to be intractable

8 Building vocabulary

This chapter will be somewhat different from the surrounding ones. Think of it as an interlude, where the main goal is not to introduce new features, but to present important advice for studying (and using!) Haskell. Here, we will discuss the importance of acquiring a vocabulary of functions and how this book, along with other resources, can help you with that. First, however, we need to make a few quick points about function composition.

8.1 Function composition

Function composition is a really simple concept. It just means applying one function to a value and then applying another function to the result. Consider these two very simple functions:

Example:
Simple functions

```
f x = x + 3

square x = x2
```

We can compose them in two different ways, depending on which one we apply first:

```
Prelude> square (f 1)
16
Prelude> square (f 2)
25
Prelude> f (square 1)
4
Prelude> f (square 2)
7
```

The parentheses around the inner function are necessary; otherwise, the interpreter would think that you were trying to get the value of `square f`, or `f square`; and both have no meaning.

The composition of two functions is a function in its own right. If applying f and then square, or vice-versa, to a number were a frequent, meaningful or otherwise important operations in a program, a very natural next step would be defining:

Example:
Composed functions

```
squareOfF x = square (f x)


fOfSquare x = f (square x)
```

There is a second, nifty way of writing composed functions. It uses `(.)`, the function composition operator, and is as simple as putting a period between the two functions:

Example:
Composing functions with `(.)`

```
squareOfF x = (square . f) x


fOfSquare x = (f . square) x
```

Note that functions are still applied from right to left, so that `g(f(x)) == (g . f) x` [1].

8.2 The need for a vocabulary

Function composition allows us to define complicated functions using simpler ones as building blocks. One of the key qualities of Haskell is how simple it is to write composed functions, no matter if the base functions are written by ourselves or by someone else[2], and the extent that helps us in writing simple, elegant and expressive code.

In order to use function composition, though, we first need to have functions to compose. While naturally the functions we ourselves write will always be available, every installation of GHC comes with a vast assortment of libraries (that is, packaged code), which provide functions for various common tasks. For that reason, it is vital for *effective* Haskell programming to develop some familiarity with the essential libraries or, at least, knowing how to find useful functions in them.

We can look at this issue from a different perspective. We have gone through a substantial portion of the Haskell syntax already; and, by the time we are done with the Recursion[3] chapter (which is just around the corner), we could, in principle, write pretty much any list manipulation program we want; and, lists being such an important data structure, that corresponds to a very wide set of useful programs. Doing so with our current set of tools, however, would be terribly inefficient; not only due to the lack of some more advanced language features but, crucially, because we would end up rewriting large parts of the standard libraries. Rather, it is far preferable to have the libraries dealing as much

[1] `(.)` is modelled after the mathematical operator $\circ$, which works in the same way: $(g \circ f)(x) = g(f(x))$

[2] Such ease is not only due to the bits of syntax we mentioned above, but mainly due to features we will explain and discuss in depth later in the book, such as higher-order functions.

[3] Chapter 12 on page 81

as possible with trivial, or well-known and already solved, problems while we dedicate our brain cells to writing programs that solve the problems *we* are truly interested in. And, even in the latter case, many features of the libraries can help immensely with developing our own algorithms[4].

8.3 Prelude and the hierarchical libraries

It is time to mention a few basic facts about the standard libraries. First and foremost, there is *Prelude*, which is the core library loaded by default in every Haskell program. Alongside with the basic types and other essential functionality, it provides a set of ubiquitous and extremely useful functions. We will refer to Prelude and its functions all the time throughout these introductory chapters.

Alongside with Prelude, there are the *hierarchical libraries*, which provide a much wider range of functionality. Although they are provided by default with GHC, they are not loaded automatically like Prelude. Rather, they are distributed as *modules*, which must be *imported* into your program. Later on we will actually explain how that works; but for now all you need to know is that, if we mention that, for instance, the function `permutations` is in the module `Data.List`, you just have to add a line `import Data.List` to the top of your source file, and `permutations`, alongside with the rest of `Data.List`, will be available.

Example:
Importing a module in a source file

```
import Data.List


testPermutations = permutations "Prelude"
```

For quick GHCi tests, just enter `:m +Data.List` at the command line to load that module.

```
Prelude> :m +Data.List
Prelude Data.List> :t permutations
permutations :: [a] -> a5
```

8.4 One exhibit

Before continuing, let us see one (slightly histrionic, we admit) example of what familiarity with a few basic functions from Prelude can bring us[6]. Suppose we need a function which takes a string composed of words separated by spaces and returns that string with the order

[4] One simple example is provided by functions like `map`, `filter` and the folds, which we will cover in the chapters on list processing just ahead. Another would be the various monad libraries, which will be studied in depth later on.

[6] The example here is inspired by the Simple Unix tools ^{`http://www.haskell.org/haskellwiki/Simple_unix_tools`} demo in the HaskellWiki.

of the words reversed, so that `"Mary had a little lamb"` becomes `"lamb little a had Mary"`. Now, we *can* solve that problem using exclusively what we have seen so far about Haskell, plus a few insights that can be acquired by studying the Recursion chapter. Here is what it might look like:

Example:
There be dragons

```
monsterRevWords :: String -> String
monsterRevWords input = rejoinUnreversed (divideReversed input)
    where
    divideReversed s = go1 [] s
        where
        go1 divided [] = divided
        go1 [] (c:cs)
            | testSpace c = go1 [] cs
            | otherwise   = go1 [[]] (c:cs)
        go1 (w:ws) [c]
            | testSpace c = (w:ws)
            | otherwise   = ((c:w):ws)
        go1 (w:ws) (c:c':cs)
            | testSpace c =
                if testSpace c'
                    then go1 (w:ws) (c':cs)
                    else go1 ([c']:w:ws) cs
            | otherwise =
                if testSpace c'
                    then go1 ((c:w):ws) (c':cs)
                    else go1 ((c:w):ws) (c':cs)
    testSpace c = c == ' '
    rejoinUnreversed [] = []
    rejoinUnreversed [w] = reverseList w
    rejoinUnreversed strings = go2 (' ' : reverseList newFirstWord) (otherWords)
        where
        revStrings = reverseList strings
        newFirstWord = head revStrings
        otherWords = tail revStrings
        go2 rejoined ([]:[]) = rejoined
        go2 rejoined ([]:(w':ws')) = go2 (rejoined) ((' ':w'):ws')
        go2 rejoined ((c:cs):ws) = go2 (c:rejoined) (cs:ws)
    reverseList [] = []
    reverseList w = go3 [] w
        where
        go3 rev [] = rev
        go3 rev (c:cs) = go3 (c:rev) cs
```

There are too many problems with this *thing*; so let us consider just three of them:

- If we claimed that `monsterRevWords` does what is expected, you could either take our word for it, test it exhaustively on all sorts of possible inputs or attempt to understand it and get an awful headache (please don't).
- Furthermore, if we write a function this ugly and have to fix a bug or slightly modify it later on[7], we are set for an awful time.
- Finally, there is at least one easy to spot potential problem: if you have another glance at the definition, about halfway down there is a `testSpace` helper function which checks if a character is a space or not. The test, however, only includes the common space character (that is, `' '`), and not other whitespace characters (tabs, newlines, etc.)[8].

If, however, we are armed merely with knowledge of the following Prelude functions:

- `words`, which reliably breaks down a string in whitespace delimited words, returning a list of strings;
- `reverse`, which reverses a list (incidentally, that is exactly what the `reverseList` above does); and
- `unwords`, which does the opposite of `words`;

then function composition means our problem is instantly solved.

Example:
`revWords` done the Haskell way

```
revWords :: String -> String
revWords input = (unwords . reverse . words) input
```

Short, simple, readable and, since Prelude is reliable, bug-free[9]. The point here is: any time some program you are writing begins to look like `monsterRevWords`, look around and reach for your toolbox - the libraries.

8.5 Acquiring vocabulary

After the stern warnings above, you might have predicted we will continue with the book by diving deep into the standard libraries. That is not the route we will follow, however - at least not in the first part of the book. The main reason for that is the Beginner's Track is meant to cover most of the Haskell language functionality in a readable and reasonably compact account, and a linear, systematic study of the libraries would in all likelihood have us sacrificing either one attribute or the other.

[7] *Co-author's note*: "Later on? I wrote that half an hour ago and I'm not totally sure about how it works already..."

[8] A reliable way of checking whether a character is whitespace is with the `isSpace` function, which is in the module `Data.Char`.

[9] In case you are wondering, there are lots of other functions, either in Prelude or in `Data.List` which, in one way or another, would help to make `monsterRevWords` somewhat saner - just to name a few: `(++)`, `concat`, `groupBy`, `intersperse`. There is no need for them in this case, though, since nothing compares to the one-liner above.

In any case, even if we will not stop with the programme to investigate them, the libraries will remain close at hand as we advance in the course (that also means there is no need for you to pause in your way through the book just to study the libraries on your own!). In this final section, we will give some advice on how you can use this book to learn them and which other resources are available.

8.5.1 With this book

- For starters, once we enter Elementary Haskell, you will notice several of the exercises - mainly, among those about list processing - involve writing equivalent definitions for Prelude functions. For each of these exercises you do, one more function will be added to your repertoire.
- Furthermore, every now and then we will introduce a "new" library function; maybe within an example, or just with a mention in passing. Whenever we do so, take a minute to test the function and do some experiments. The idea is to extend that curiosity about types we mentioned in Type basics[10] to the functions themselves.
- While the first few chapters are quite tightly-knit, later parts of the book are more independent from each other. That is specially true of the third track, Haskell in Practice. There, among other things you can find chapters on the Hierarchical libraries[11]; and most of their content can be understood soon after having completed Elementary Haskell.
- As we reach the later parts of the Beginner's track, and specially when we get to monads, the concepts we will discuss will naturally lead to exploration of important parts of the hierarchical libraries.

8.5.2 Other resources

- First and foremost, there is the documentation. While it is probably too dry to be really useful right now, soon enough it will prove invaluable. You can read not only the Prelude specification[12] on-line but also the GHC hierarchical libraries[13] documentation, with nice navigation and source code just one click away.
- A fun way of searching through the documentation is provided by the Hoogle[14], a Haskell search engine which covers the libraries distributed with GHC. (There is also Hayoo![15]; it includes a wide range of libraries which are not installed by default).
- Finally, we will when appropriate give pointers to other useful learning resources, specially when we move towards intermediate and advanced topics.

[10] Chapter 5 on page 27

[11] Chapter 68 on page 465

[12] `http://www.haskell.org/onlinereport/standard-prelude.html`

[13] `http://www.haskell.org/ghc/docs/latest/html/libraries/index.html`

[14] `http://www.haskell.org/hoogle`

[15] `http://holumbus.fh-wedel.de/hayoo/hayoo.html`

9 Next steps

This chapter has two main goals. We will dedicate the bulk of the text to introducing *pattern matching*, which is an absolutely fundamental feature of Haskell. Besides, we will also mention two new pieces of syntax: `if` expressions and `let` bindings, which will prove handy in upcoming Simple input and output[1] chapter. In our context, though, they can be safely thought of as minor points; for, even though `if` and `let` are convenient sometimes, there is little in what they do that can't be achieved with the syntax we already know. There is plenty of time for you to master them; and furthermore there will be more examples a little ahead in the book.

9.1 if / then / else

Haskell syntax supports garden-variety conditional expressions of the form *if... then... (else ...)*. For instance, consider 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`. There is a predefined function which does that job already (it is called `signum`); for the sake of illustration, though, let's define a version of our own:

Example:
The signum function.

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

You can experiment with this as:

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

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

[1] Chapter 10 on page 69

In an if/then/else construct, first the condition (in this case `x < 0`) is evaluated. If it results `True`, the whole construct evaluates to the **`then`** expression; otherwise (if the condition is `False`), the construct evaluates to the **`else`** expression. All of that is pretty intuitive. If you have programmed in an imperative language before, however, it might seem surprising to know that we always need to have *both* a **`then`** *and* an **`else`** clause. That must be so because the construct has to result in a value in both cases - and a value of the same type in both cases, by the way.

if / then / else function definitions like the one above can be easily rewritten with the guards syntax presented in past modules:

Example:
From if to guards

```
mySignum x
    | x < 0     = -1
    | x > 0     = 1
    | otherwise = 0
```

And conversely, the absolute value function in Truth values[2] can be rendered as:

Example:
From guards to if

```
abs x =
    if x < 0
        then -x
        else x
```

9.2 Introducing pattern matching

Suppose we are writing a program which tracks statistics from a racing competition in which racers receive points based on their classification in each race, the scoring rules being:

- 10 points for the winner of the race;
- 6 for the second-placed;
- 4 for the third-placed;
- 3 for the fourth-placed;
- 2 for the fifth-placed;
- 1 for the sixth-placed; and
- no points for other racers.

[2] Chapter 4 on page 19

We can write a simple function which takes a classification (represented by an integer number: `1` for first place, etc.[3]) and returns how many points were earned. One possible solution uses if/then/else:

Example:
Computing points with if/then/else

```
pts :: Int -> Int
pts x =
    if x == 1
        then 10
        else if x == 2
            then 6
            else if x == 3
                then 4
                else if x == 4
                    then 3
                    else if x == 5
                        then 2
                        else if x == 6
                            then 1
                            else 0
```

Yuck! Admittedly, it wouldn't look this hideous had we used guards instead of if/then/else, but it still would be tedious to write (and read!) all those equality tests. We can do better, though:

Example:
Computing points with a piece-wise definition

```
pts :: Int -> Int
pts 1 = 10
pts 2 = 6
pts 3 = 4
pts 4 = 3
pts 5 = 2
pts 6 = 1
pts  = 0
```

Much better. However, even though defining `pts` in this style (which we will arbitrarily call *piece-wise definition* from now on) shows to a reader of the code what the function does in an awesomely clear way, the syntax looks odd given what we have seen of Haskell so far.

[3] Here we will not be much worried about what happens if a nonsensical value (say, `(-4)`) is passed to the function. In general, however, it is a good idea to give some thought to such "strange" cases.

Why are there seven equations for `pts`? What are those numbers doing in their left-hand sides? Where has the `x` gone?

The example above is our first encounter with a key feature of Haskell called *pattern matching*. When we call the second version of `pts`, the argument is *matched* against the numbers on the left side of each of the equations, which in turn are the *patterns*. This matching is done in the order we wrote the equations; so first of all the argument is matched against the `1` in the first equation. If the argument is indeed `1`, we have a match and the first equation is used; and so `pts 1` evaluates to `10` as expected. Otherwise, the other equations are tried in order following the same procedure. The final one, though, is rather different: the `_` is a special pattern that might be read as "whatever": it matches with anything; and therefore if the argument doesn't match any of the previous patterns `pts` will return zero.

As for why there is no `x` or any other variable standing for the argument, it is simply because we don't need that to write the definitions. All possible return values are constants; and since the point of specifying a variable name on the left side is using it to write the right side, the `x` is unnecessary in our function.

There is, however, an obvious way to make `pts` even more concise. The points given to a racer decrease regularly from third place to sixth place, at a rate of one point per position. After noticing that, we can eliminate three of the seven equations as follows:

Example:
Mixing styles

```
pts :: Int -> Int

pts 1 = 10

pts 2 = 6

pts x

    | x <= 6    = 7 - x

    | otherwise = 0
```

The first thing to point out here is that we can mix both styles of definitions. In fact, when we write `pts x` in the left side of an equation we are using pattern matching too! As a pattern, the `x` (or any other variable name) matches anything just like `_`; the only difference being that it also gives us a name to use on the right side (which, in this case, is necessary to write `7 - x`).

Exercises:
We cheated a little when moving from the second version of `pts` to the third one: they do not do exactly the same thing. Can you spot what the difference is?

Please do the exercise above before continuing: it is quick to do and really important.

Beyond integers, pattern matching works with values of various other types. One handy example are booleans. For instance, the `(||)` logical-or operator we met in Truth values[4] could be defined as:

[4] Chapter 4.5.1 on page 25

Example:
(||)

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

Or:

Example:
(||), done another way

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

When matching two or more arguments at once, the equation will only be used if all of them match.

To conclude this section, let us discuss a few things that might go wrong when using pattern matching:

- If we put a pattern which matches anything (such as the final patterns in each of the `pts` example) *before* the more specific ones the latter will be ignored. GHC(i) will typically warn us that "Pattern match(es) are overlapped" in such cases.
- If no patterns match, an error will be triggered. Generally, it is a good idea to ensure the patterns cover all cases, in the same way that the `otherwise` guard is not mandatory but highly recommended.
- Finally, while you can play around with various ways of (re)defining `(&&)`[5], here is one version that will *not* work:

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

The program won't test whether the arguments are equal just because we happened to use the same name for both. As far as the matching goes, we could just as well have written `_ && _` in the first case. And even worse: since we gave the same name to both arguments, GHC(i) will refuse the function due to "Conflicting definitions for `x'".

9.3 Tuple and list patterns

While the examples above show that pattern matching helps in writing more elegant code, that does not explain why it is so important. We will begin to grasp why it matters so much

[5] If you are going to experiment with it in GHCi, call your version something else to avoid a name clash; say, (&!&).

by considering the problem of writing a definition for `fst`, the function which extracts the first element of a pair. At this point, that appears to be an impossible task, as the only way of accessing the first value of the pair is by using `fst` itself... The following function, however, does the same thing as `fst` (confirm it in GHCi):

Example:
A definition for `fst`

```
fst' :: (a, b) -> a
fst' (x, ) = x
```

It's magic! Instead of using a regular variable in the left side of the equation, we specified the argument with the *pattern* of the 2-tuple - that is, `(,)` - filled with a variable and the `_` pattern. Then the variable was automatically associated with the first component of the tuple, and we used it to write the right side of the equation. The definition of `snd` is, of course, analogous.

Furthermore, the trick demonstrated above can be done with lists as well. Here are the actual definitions of `head` and `tail`:

Example:
`head`, `tail` and patterns

```
head             :: [a] -> a
head (x:)        =  x
head []          =  error ”Prelude.head : empty list”

tail             :: [a] − > [a]
tail (:xs)       =  xstail []          =  error ”Prelude.tail : empty list”
```

The only essential change in relation to the previous example was replacing `(,)` with the pattern of the cons operator `(:)`. These functions also have an equation using the pattern of the empty list, `[]`; however, since empty lists have no head or tail there is nothing to do other than use `error` to print a prettier error message.

Summarizing, the real power of pattern matching comes from how it can be used to access the parts of a complex value. Pattern matching on lists, in particular, will be extensively deployed in Recursion[6] and the chapters that follow it. Later on, we will also dedicate a whole chapter to explore what is behind such a seemingly magical feature.

9.4 `let` bindings

To conclude this chapter, a brief word about `let` bindings, which are an alternative to `where` clauses for making local declarations. For instance, take the problem of finding the roots of a polynomial of the form ax^2+bx+c (or, in other words, the solution to a second

[6] Chapter 12 on page 81

degree equation - think back of your middle school math courses). Its solutions are given by:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

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))
```

Writing the `sqrt(b*b - 4*a*c)` term in both cases is annoying, though; we can use a local binding instead, using either `where` or, as will be demonstrated below, a `let` declaration:

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

We put the `let` keyword before the declaration, and then use `in` to signal we are returning to the "main" body of the function. It is possible to put multiple declarations inside a single `let`...`in` block - just make sure they are indented the same amount, otherwise there will be syntax errors:

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

⚠ Warning

A general point: as indentation matters syntactically in Haskell you need to be careful about whether you are using tabs or spaces. By far the best solution is to configure your text editor to insert two or four spaces in place of tabs. If you keep tabs as distinct, at least ensure that your tabs have always the same length, or you're likely to run into trouble.

Note:
Still on indentation, the Indentation[a] chapter has a full account of indentation rules; so if doubts about that arise as you code you might want to give it a glance.

[a] Chapter 23 on page 149

10 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! Pure functional languages have always had a problem with input and output because IO requires *side effects*. Pure functions always have to return the same results for the same arguments. But how can such 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 procedure 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 pure functions, but they are, by definition, *not* pure. The item that reads a string from the keyboard cannot be a function, as it will not return the same `String` every time. If the first function simply returns `()` every time, then referential transparency tells us we should have no problem replacing it with a function `f _ = ()`, but clearly this does not have the desired result because the function has a *side effect*: it prints the argument.

10.1 Actions

The breakthrough for solving this problem came when Philip 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 to complement it. 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/[1].

[1] Chapter 29 on page 183

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`, and 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 right angled 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.

10.1.1 Left arrow clarifications

`<-` 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`."

`<-` can be used with any action but the last

On the flip side, there are also very few restrictions on which actions can have values obtained from 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.

10.1.2 Controlling actions

Normal Haskell constructions like **`if/then/else`** 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).

Note:
If you are using the **`If`**-structure keep in mind that the **`else`** branch is mandatory. Otherwise you get compile errors!

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. (As far as syntax goes there are a few different ways to do it; write at least a version using `if` / `then` / `else`.)

10.2 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.

10.2.1 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 down 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/[2] 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, `<-`.

[2] Chapter 5 on page 27

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.

10.2.2 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 does not produce one. 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, when those 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` function itself 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 anticlimactic `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**

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

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

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?

10.3 Learn more

At this point, you should have the fundamentals needed to do some fancier input/output. Here are some IO-related topics you may want to check in parallel with the main track of the course.

- You could continue the sequential track, by learning more about types[3] and eventually monads[4].
- Alternately: you could start learning about building graphical user interfaces in the ../GUI/[5] chapter
- For more IO-related functionality, you could also consider learning more about the System.IO library[6]

[3] Chapter 15 on page 107

[4] Chapter 29 on page 183

[5] Chapter 83 on page 553

[6] Chapter 73 on page 493

11 Elementary Haskell

12 Recursion

Recursion plays a central role in Haskell (and computer science and mathematics in general): recursion is the idea of defining a function in terms of itself. A function defined in this way is said to be **recursive**.

Recursion is merely a form of repetition, but sometimes it is taught in a confusing or obscure way. It is simple enough to understand as long as you separate the *meaning* of a recursive function from its *behaviour*.

Recursion, like other forms of repetition, require a stopping or termination condition. Like an infinite loop, a badly designed recursive function might lead to infinite regress; but if done properly it won't.

Generally speaking, a recursive definition has two parts. First, there are one or more *base cases* which express termination conditions, and the value of the function when the termination condition holds; it is essential that the base case does not call the function being defined. The *recursive case* is more general, and defines the function in terms of a 'simpler' call to itself. If we think of a function as being the solution to a computational problem, the recursive case expresses the relationship between the solution to a given problem and the solution of a somewhat smaller or simpler problem of the same kind. To make this idea more concrete, let's look at a few examples.

12.1 Numeric recursion

12.1.1 The factorial function

In mathematics, especially combinatorics, there is a function used fairly frequently called the **factorial** function[1]. It takes a single non-negative integer as an argument, finds all the positive integers less than or equal to "n", and multiplies them all together. For example, the factorial of 6 (denoted as 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 consecutive numbers

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

[1] In mathematics, $n!$ normally means the factorial of a non-negative integer, n, but that syntax is impossible in Haskell, so we don't use it here.

Notice how we've lined things up. You can see here that the 6! involves the 5!. In fact, 6! is just $6 \times 5!$. Let's look at another example:

Example:
Factorials of consecutive numbers

```
Factorial of 4 = 4 × 3 × 2 × 1
Factorial of 3 =     3 × 2 × 1
Factorial of 2 =         2 × 1
Factorial of 1 =             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[2]?). 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).

⚠ Warning

The `factorial` function written above should be defined in a file. Since it is a small function, however, it is not too unreasonable to write it in GHCi as a one-liner, by using braces (that is, `{` and `}`) and a semicolon:

```
    > let { factorial 0 = 1; factorial n = n * factorial (n - 1) }
```

[2] Actually, defining the factorial of 0 to be 1 is not just arbitrary; it's because the factorial of 0 represents an empty product ^{`http://en.wikipedia.org/wiki/empty%20product`} .

Using `let` statements without the braces in the interpreter will cause the function to be redefined with the last definition, losing the important first definition and thus leading to infinite recursion.

If this seems confusing to you, try to separate the meaning of the definition from the behaviour of the computer while executing a recursive function. The examples above demonstrate a very simple relationship between factorial of a number, n, and the factorial of a slightly smaller number, n-1. This relationship only needs to be understood at a single abstract level.

But understanding the relationship is only one side of the issue. We do need to understand how recursive functions behave. Think of a function call as delegation. The instructions for a recursive function delegate a sub-task. It just so happens that the delegate function uses the same instructions as the delegator. It's just the input data that changes. The only really confusing thing about the behaviour of a recursive function is the fact that each function call uses the same parameter names, and it can be tricky for a person to keep track of where they are.

Let's look at what happens when you execute `factorial 3`:

- 3 isn't 0, so we calculate the factorial of 2
  - 2 isn't 0, so we calculate the factorial of 1
    - 1 isn't 0, so we calculate the factorial of 0
      - 0 *is* 0, so we return 1.
    - To complete the calculation for factorial 1, we multiply the current number, 1, by the factorial of 0, which is 1, obtaining 1 (1 × 1).
  - To complete the calculation for factorial 2, we multiply the current number, 2, by the factorial of 1, which is 1, obtaining 2 (2 × 1 × 1).
- To complete the calculation for factorial 3, we multiply the current number, 3, by the factorial of 2, which is 2, obtaining 6 (3 × 2 × 1 × 1).

We can see how the result of the recursive call is calculated first, then combined using multiplication. Once you see how it can work, you rarely need to "unwind" the recursion like this when reading or composing recursive functions. Compilers have to implement the behaviour, but programmers can work at the abstract level, e.g., the relationship between `factorial n` and `factorial (n-1)`.

(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 * 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.

12.1.2 A quick aside

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

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 ( ; n > 1; n--)
    res *= n;
  return res;
}
```

This isn't directly possible in Haskell, since changing the value of the variables `res` and `n` (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 n 1 where

    factorialWorker n res | n > 1     = factorialWorker (n - 1) (res * n)

                          | otherwise = res
```

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 optimisation for recursion, including an 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.

12.1.3 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 × 4 = 5 × 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   -- recurse: multiply by one less, and add an
 extra copy
```

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 in some manner 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.)

12.2 List-based recursion

A *lot* of functions in Haskell turn out to be recursive, especially those concerning lists[3]. Let us begin by considering 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
```

Let us explain the algorithm in English to clarify how it works. The type signature of `length` tells us that it takes any sort of list and produces an `Int`. The next line says that the length of an empty list is 0; and that, naturally, is the base case. The final line is the recursive case: if a list consists of a first element, `x`, and `xs`, the rest of the list, the length of the list is one plus the length of `xs` (as in the `head`/`tail` example in ../Next steps[4], `x` and `xs` are set when the argument list matches the (:) pattern).

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 in this chapter.)

[3] 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).

[4] Chapter 9 on page 61

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. (Note that all of these functions are available in Prelude, so you will want to give them different names when testing your definitions in GHCi.)

1. `replicate :: Int -> a -> [a]`, which takes a count and an element 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 recursing *both* numerically and down a list[a].
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')]`.

[a] Incidentally, `(!!)` provides a reasonable solution for the problem of the fourth exercise in Lists and tuples/Retrieving values ^{Chapter6.3.1 on page 46}.

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.

12.3 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[5], but writing `factorial` in this way means you, the programmer, don't have to worry about it.

12.4 Summary

Recursion is the practice of defining a function in terms 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.

[5] Actually, it's using a function called `foldl`, which actually does the recursion.

13 More about lists

By now we have already met the basic tools for working with lists. We can build lists up from the cons operator `(:)` and the empty list `[]`, and we can take them apart by using a combination of recursion and pattern matching. In this chapter and the next we will consider in more depth techniques for list processing and, while doing so, discover a bit of new notation and have a first taste of characteristic Haskell features such as infinite lists, list comprehensions and higher-order functions.

Note:
Throughout this chapter you will read and write functions which sum, subtract and multiply elements of lists. For simplicity's sake we will pretend the list elements have to be of type `Integer`. However, by recalling the discussions on the Type basics II[a] chapter, you will realize that assumption is not necessary at all! Therefore, we suggest that, as an exercise of sorts, you figure out what would the type signatures of such functions look like if we made them polymorphic, allowing for the list elements to have any type in the class `Num`. To check your signatures, just omit them temporarily, load the functions into GHCi, use :t and let type inference guide you.

[a] Chapter 7 on page 49

13.1 Rebuilding lists

We'll start by writing and analysing a function to double every element of a list of integers. For our current purposes, it will be enough that its type is so that it takes a list of `Integer`s and evaluates to another list of `Integer`s:

```
doubleList :: [Integer] -> [Integer]
```

Then we must write the function definition. As usual in such cases, we will go for a recursive definition:

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

Here, the base case is for an empty list; and it just evaluates to an empty list. As for the general case, doubleList *builds up a new list* by using (:). The first element of this new list is twice the head of the argument, and the rest of the result is obtained by a recursive call

– the application of doubleList to the tail. If the tail happens to be an empty list, the base case will be invoked and recursion stops[1].

By understanding the recursive definition we can picture what actually happens when we evaluate an expression such as

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

We can work it 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]
```

The bottom line is that, in effect, we rebuilt the original list replacing every element by its double.

One important thing to notice is that in this longhand evaluation exercise the *moment* at which we choose to evaluate the multiplications does not affect the result[2]. Had we done them immediately after each recursive call of doubleList it would have made no difference. This reflects 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. Since Haskell is also a *lazy evaluation* language, 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 [3].

13.1.1 Generalizing

Suppose that we needed, while solving a given problem, not only a function to double a list but also one that tripled it. In principle, we could follow the same strategy and define:

```
tripleList :: [Integer] -> [Integer]
tripleList [] = []
tripleList (n:ns) = (3 * n) : tripleList ns
```

Both `doubleList` and `tripleList` have very limited applicability. Every time we needed multiplying the elements of a list by 4, 8, 17 etc. we would need to write a new list-multiplying function, and all of them would do nearly the same thing. An obvious improvement would be generalizing our function to allow multiplication by any number. Doing so requires a function that takes an `Integer` multiplicand as well as a list of `Integer`s. Here is a way to define it:

[1] Note that, had we forgotten the base case, once the recursion got to an empty list the (x:xs) pattern match would fail, and we would get an error.

[2] As long as none of them results in an error or nontermination. But let us sidestep that issue for now.

[3] One exception is the case of *infinite lists* (!) which we will consider in a short while

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

This example deploys _ as a "don't care" pattern. The multiplicand is not used for the base case, so instead of giving it a name (like `m`, `n` or `ns`) it is explicitly ignored.

`multiplyList` solves our current problem for any integer number:

```
Prelude> multiplyList 17 [1,2,3,4]
[17,34,51,68]
```

In particular, it is trivial to rewrite our earlier functions in terms of `multiplyList`:

```
doubleList :: [Integer] -> [Integer]
doubleList xs = multiplyList 2 xs

tripleList :: [Integer] -> [Integer]
tripleList xs = multiplyList 3 xs
```

Exercises:
Write the following functions and test them out. Don't forget the type signatures.

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].
5. diffs returns a list of the differences between adjacent items. So diffs [3,5,6,8] returns [2,1,2]. (Hints: one solution involves writing an auxiliary function which takes two lists and calculates the difference between corresponding. Alternatively, you might explore the fact that lists with at least two elements can be matched to a `(x:y:ys)` pattern.) The first three functions are in Prelude, under the names `take`, `drop`, and `sum`.

13.2 Generalizing even further

We just wrote a function which can multiply the elements of a list by any `Integer`. Not bad, but can we do any better?

Let us begin by rewriting `multiplyList` in a rather artificial manner:

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

The point of this rewrite is making it clear that while `multiplyList` is much more useful than `doubleList` it still resembles its predecessor in one aspect. Early on, the problem was being constrained to multiplying the elements by `2` or some other hard-coded number; now, we could justifiably complain about being constrained to apply `multiplyByM` to the list elements. What if we wanted to do `sumWithM` instead; or, for that matter, something entirely different (say, computing the square of each element)? We would be back to square one, having to rewrite the recursive function.

A key functionality of Haskell will save the day. Since the solution can be quite surprising, we will approach it in a somewhat roundabout way. Consider the type signature of `multiplyList`:

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

The first novelty is that the `->` arrow in type signatures is *right associative*. That means we can read this signature as:

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

How should we understand that? It tells us `multiplyList` is a function that takes *one* `Integer` argument and evaluates to *another function*, which in turn takes a list of `Integer`s and returns another list of `Integer`s. In practice, that means we can rewrite the following definition...

```
evens = multiplyList 2 [1,2,3,4] -- [2,4,6,8]
```

...like this:

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

The expression within parentheses, `(multiplyList 2)`, is just an `[Integer] -> [Integer]` function which is then applied to `[1,2,3,4]`. Now, `(multiplyList 2)` is of course the same as `doubleList`; and to drive the point home we can redefine that as:

```
doubleList :: [Integer] -> [Integer]
doubleList = multiplyList 2
```

The expression on the right side is perfectly well-formed and stands on its own, so writing the second argument of `multiplyList`, and thus the argument of `doubleList`, is strictly unnecessary[4].

The trickery above illustrates that functions in Haskell behave much like any other value - we can have them returned from other functions, and even define them without mentioning their arguments, as if they were a normal constant. What if we could have functions as *arguments* as well? That way, we could, based on the recursive skeleton of `multiplyList`, write a function that took, instead of the argument `m`, a function to replace `multiplyByM`. A function like this one:

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

[4] While this style of definition may look exotic, it is actually commonplace in Haskell code. It is called *point-free* style.

```
applyToIntegers _ [] = []
applyToIntegers f (n:ns) = (f n) : applyToIntegers f ns
```

This function does indeed work, and so we can apply *any* Integer -> Integer function to the elements of a list of Integers. Now, since the multiplication operator (*) is just a function with two arguments, the same tricks apply to it; in fact, had we defined the following function it would do exactly what its name suggests:

```
multiplyByM m = ((*) m)
```

And, finally, we can rewrite multiplyList in terms of applyToIntegers:

```
multiplyList :: Integer -> [Integer] -> [Integer]
multiplyList m = applyToIntegers ((*) m)
```

Or, equivalently:

```
multiplyList :: Integer -> [Integer] -> [Integer]
multiplyList m list = applyToIntegers ((*) m) list
```

13.3 The map function

While applyToIntegers has type (Integer -> Integer) -> [Integer] -> [Integer], there is nothing specific to integers in its algorithm. Therefore, we could define versions such as applyToChars, applyToStrings and applyToLists just by changing the type signature. That would be horribly wasteful, though: we did not climb all the way up to this point just to need a different function for each type! Furthermore, nothing prevents us from changing the signature to, for instance, (Integer -> String) -> [Integer] -> [String]; thus giving a function that takes a function Integer -> String and returns a function [Integer] -> [String] which applies the function originally passed as argument to each element of an Integer list.

The final step of generalization, then, is to make a fully polymorphic version of applyToIntegers, with signature (a -> b) -> [a] -> [b]. Such a function already exists in Prelude: it is called **map** and defined as:

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

Using it, we can effortlessly implement functions as different as...

```
multiplyList :: Integer -> [Integer] -> [Integer]
multiplyList m = map ((*) m)
```

... and...

```
heads :: [[a]] -> [a]
heads = map head
```

```
Prelude> heads [[1,2,3,4],[4,3,2,1],[5,10,15]]
[1,4,5]
```

`map` solves the problem of applying a function to each element of a list[5] in a completely general way by allowing us to choose which function to use. Functions like `map` which take other functions as arguments are called *higher-order functions*, and they are extremely useful. In particular, we will meet some other important higher-order functions used for list processing in the next chapter.

Exercises:

1. Use `map` to build functions that, given a list xs of Ints, return:
   - A list that is the element-wise negation of xs.
   - A list of lists of Ints xss that, for each element of xs, contains the divisors of xs. You can use the following function to get the divisors:
     `divisors p = [ f | f <- [1..p], p `mod` f == 0 ]`
   - The element-wise negation of xss.
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 `concat` and `group` functions might be helpful. In order to use `group`, you will need to import the Data.List module. You can access this by typing `:m Data.List` at the ghci prompt, or by adding `import Data.List` to your Haskell source code file.
   - 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?

13.4 Tips and Tricks

Before we carry on with more list processing, let us make a few miscellaneous useful observations about lists in Haskell.

13.4.1 Dot Dot Notation

Haskell has a convenient shorthand for writing ordered lists of regularly-spaced integers. Some examples to illustrate it:

```
Code          Result
```

[5] Our original `doubleList` problem was a very specific instance of that.

```
----            ------
[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 with characters as well, and even with floating point numbers - though the latter is not necessarily a good idea due to rounding errors. Try this:

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

Additionally, the notation only works with sequences with fixed differences between consecutive elements. For instance, you should not write...

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

... and expect to magically get back the rest of the Fibonacci series[6].

13.4.2 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 Ctrl-c).

The same effect could be achieved with a recursive function:

```
intsFrom n = n : intsFrom (n+1) -- note there is no base case!
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 unless they actually need to evaluate the end of the list it will all just work.

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.

[6] http://en.wikipedia.org/wiki/Fibonacci_number

13.4.3 A note about head and tail

Given the choice of using either the `( : )` pattern or `head`/`tail` to split lists, pattern matching is almost always preferable. While it may be tempting to use `head` and `tail` due to simplicity and terseness, it is all too easy to forget they fail on empty lists - and runtime crashes are never a good thing. While the Prelude function `null :: [a] -> Bool` provides a sane way of checking for empty lists without pattern matching (it returns `True` for empty lists and `False` otherwise), matching an empty list tends to be cleaner and clearer than the corresponding if-then-else expression.

Exercises:

1. With respect to your solutions to the first set of exercises in this chapter, is there any difference between `scanSum (takeInt 10 [1..])` and `takeInt 10 (scanSum [1..])`?
2. In the final block of exercises of the previous chapter you implemented the (!!) operator. Redo it, but this time using head and tail. (If by any chance you did it with head and tail back then, redo it with pattern matching.)
3. Write functions that when applied to lists give the *last* element of the list and the list with the last element dropped. That can be done either with pattern matching or by using head and tail only. We suggest you to try, and compare, both approaches.
   This functionality is provided by Prelude through the `last` and `init` functions.

14 List processing

14.1 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`.

14.1.1 foldr

The *right-associative* `foldr` folds up a list from the right, that is, it walks from the last to the first element of the list and applies the given function to each of the elements and the accumulator, the initial value of which has to be set:

```
foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr f acc []     = acc
foldr f acc (x:xs) = f x (foldr f acc 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 `acc` is `0`, and in `concat`, `f` is `(++)` and `acc` 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 acc xs` does is to replace each cons (:) in the list `xs` with the function `f`, and the empty list at the end with `acc`. That is,

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

becomes

```
f a (f b (f c acc))
```

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

```
    :                         f
   / \                       / \
  a   :       foldr f acc   a   f
     / \    ------------->     / \
    b   :                     b   f
       / \                       / \
      c  []                    c  acc
```

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

14.1.2 foldl

The *left-associative* `foldl` processes the list in the opposite direction, starting at the left side with the first element, and proceeding to the last one step by step:

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

The corresponding trees look like:

```
   :                          f
  / \                        / \
 a   :       foldl f acc    f   c
    / \    ------------->  / \
   b   :                  f   b
      / \                / \
     c  []              acc a
```

Note:
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` (which can be imported by adding `import Data.List` to the beginning of a source file). As a rule of thumb 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, unless the list is not too long, or memory usage isn't a problem.

14.1.3 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'.

14.1.4 folds 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 must call itself recursively until it reaches the end of the input list; this is inevitable, because the recursive call is not made in an argument to f. 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) []
```

(*Note:* This is very compact thanks to the `\x xs ->` syntax. Instead of defining a function somewhere else and passing it to foldr we provided the definition *in situ*; `x` and `xs` being the arguments and the right-hand side of the definition being what is after the `->`)

or, equally handily, 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! (If you try this in GHCi, remember you can stop an evaluation with Ctrl-c, but you have to be quick and keep an eye on the system monitor or your memory will be consumed in no time and your system will hang.)

As a final example, another thing that you might notice is that `map` itself can be implemented 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 likely want a fold.

Exercises:

1. 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.
2. 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).
3. Use a fold (*which one?*) to define `reverse :: [a] -> [a]`, which returns a list with the elements in reverse order.

Note that all of these are Prelude functions, so they will be always close at hand when you need them. (Also, that means you will want to use slightly different names for testing your answers in GHCi.)

14.2 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:

1. Write your own definition of `scanr`, first using recursion, and then using `foldr`. Do the same for `scanl` first using recursion then `foldl`.
2. 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*

14.3 filter

A very common operation performed on lists is filtering[1], which is generating a new list composed only of elements of the first list that meet a certain condition. One simple example of that would be taking a list of integers and making from it a list which only retains its even numbers.

```
retainEven :: [Int] -> [Int]
retainEven [] = []
retainEven (n:ns) =
-- mod n 2 computes the remainder for the integer division of n by 2, so if it
 is zero the number is even
  if ((mod n 2) == 0)
    then n : (retainEven ns)
    else retainEven ns
```

That works fine, but it is a slightly verbose solution. It would be nice to have a more concise way to write the filter function. Also, it would certainly be very useful to be able to generalize the filtering operation – that is, make it capable of filtering a list using any boolean condition we'd like. To help us with both of these issues Prelude provides a `filter` function. `filter` has the following type signature:

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

That means it evaluates to a list when given two arguments, namely an `(a -> Bool)` function which carries the actual test of the condition for the elements of the list and the list to be filtered. In order to write `retainEven` using `filter`, we need to state the condition as an auxiliary `(a -> Bool)` function, like this one:

```
isEven :: Int -> Bool
isEven n = ((mod n 2) == 0)
```

[1] http://en.wikipedia.org/wiki/Filter%20%28mathematics%29

And then retainEven becomes simply:

```
retainEven ns = filter isEven ns
```

It can be made even more terse by writing it in point-free style:

```
retainEven = filter isEven
```

This is just like what we demonstrated before for `map` and the folds. Like `filter`, those take another function as argument; and using them point-free emphasizes this "functions-of-functions" aspect.

14.4 List comprehensions

An additional tool for list processing is the list comprehension[2], a powerful, concise and expressive syntactic construct. One simple way we can use list comprehensions is as syntactic sugar[3] for filtering. So, instead of using the Prelude `filter`, we could write `retainEven` like this:

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

This compact syntax may look a bit intimidating at first, but it is simple to break down. One possible way to read it would be:

- (Starting from the middle) Take the list *es* and draw (the "<-") each of its elements as a value *n*.
- (After the comma) For each drawn *n* test the boolean condition `isEven n`.
- (Before the vertical bar) If (and only if) the boolean condition is satisfied, prepend *n* to the new list being created (note the square brackets around the whole expression).

Thus if `es` is equal to [1,2,3,4], then we would get back the list [2,4]. 1 and 3 were not drawn because `(isEven n) == False`.

The real power of list comprehensions, though, comes from the fact they are easily extensible. Firstly, we can use as many tests as we wish (even zero!). Multiple conditions are written as a comma-separated list of expressions (which should evaluate to a Boolean, of course). For a simple example, suppose we want to modify `retainEven` so that only numbers larger than 100 are retained:

```
retainLargeEvens :: [Int] -> [Int]
retainLargeEvens es = [ n | n <- es , isEven n, n > 100 ]
```

Furthermore, we are not limited to using `n` as the element to be prepended when generating a new list. Instead, we could place any expression before the vertical bar (if it is compatible with the type of the list, of course). For instance, if we wanted to subtract one from every even number, all it would take is:

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

[2] http://en.wikipedia.org/wiki/List%20comprehension

[3] http://en.wikipedia.org/wiki/Syntactic%20sugar

In effect, that means the list comprehension syntax incorporates the functionality of map as well as of filter. Now *that* is conciseness! (and conciseness that does not sacrifice readability, in that.)

To further sweeten things, the left arrow notation in list comprehensions can be combined with pattern matching. For example, suppose we had a list of (Int, Int) tuples and we would like to construct a list with the first element of every tuple whose second element is even. Using list comprehensions, we might write it as follows:

```
firstForEvenSeconds :: [(Int, Int)] -> [Int]
firstForEvenSeconds ps = [ fst p | p <- ps, isEven (snd p) ] -- here, p is for
 pairs.
```

Patterns can make what the function is doing more obvious:

```
firstForEvenSeconds ps = [ x | (x,y) <- ps, isEven y ]
```

As in other cases, arbitrary expressions may be used before the |. If we wanted a list with the double of those first elements:

```
doubleOfFirstForEvenSeconds :: [(Int, Int)] -> [Int]
doubleOfFirstForEvenSeconds ps = [ 2 * x | (x,y) <- ps, isEven y ]
```

Note that the function code is actually shorter than its descriptive name.

There are even more possible tricks:

```
allPairs :: [(Int, Int)]
allPairs = [ (x, y) | x <- [1..4], y <- [5..8] ]
```

This comprehension draws from *two* lists, and generates all 4 * 4 = 16 possible (x, y) pairs with the first element in [1..4] and the second in [5..8]. In the final list of pairs, the first elements will be those generated with the first element of the first list (here, 1), then those with the second element of the first list, and so on. In this example, the full list is (linebreaks added for clarity):

```
Prelude> [(x,y)|x<-[1..4],y<-[5..8]]
[(1,5),(1,6),(1,7),(1,8),
 (2,5),(2,6),(2,7),(2,8),
 (3,5),(3,6),(3,7),(3,8),
 (4,5),(4,6),(4,7),(4,8)]
```

Note that we didn't do any filtering here; but we could easily add a condition to restrict the combinations that go into the final list:

```
somePairs = [ (x, y) | x <- [1..4], y <- [5..8], x + y > 8 ]
```

This lists only has the pairs with the sum of elements larger than 8; starting with (1,8), then (2,7) and so forth.

Exercises:

1. Write a `returnDivisible :: Int -> [Int] -> [Int]` function which filters a list of integers retaining only the numbers divisible by the integer passed as first argument. For integers x and n, x is divisible by n if `(mod x n) == 0` (note that the test for evenness is a specific case of that).
2. - Write - using list comprehension syntax, a single function definition and *no* `if`, `case` and similar constructs - a `[[Int]] -> [[Int]]` which, from a list of lists of `Int`, returns a list of the tails of those lists using, as filtering condition, that the head of each `[Int]` must be larger than 5. Also, your function must not trigger an error when it meets an empty `[Int]`, so you'll need to add an additional test to detect emptiness.
   - Does order matter when listing the conditions for list comprehension? (You can find it out by playing with the function you wrote for the first part of the exercise.)
3. Over this section we've seen how list comprehensions are essentially syntactic sugar for `filter` and `map`. Now work in the opposite direction and define alternative versions of the `filter` and `map` using the list comprehension syntax.
4. Rewrite `doubleOfFirstForEvenSeconds` using `filter` and `map` instead of list comprehension.

15 Type declarations

You're not restricted to working with just the types provided by default with the language. Haskell allows you to define new types. Reasons for doing so include that:

- It allows for code to be written in terms of the problem being solved, making programs easier to design, write and understand.
- It allows for handling related pieces of data together in ways more convenient and meaningful than say, simply putting and getting values from lists or tuples.
- It makes it possible to use two powerful features of Haskell, pattern matching and the type system, to their fullest extent by making them work with your custom types.

How these things are achieved and why they are so important will gradually become clear as you progress on this course.

Haskell has three basic ways to declare a new type:

- The `data` declaration, which defines new data types.
- The `type` declaration for type synonyms.
- The `newtype` declaration, which is a cross between the other two.

In this chapter, we will discuss the most essential way, `data`. We'll also mention `type`, which is a convenience feature. You'll find out about `newtype` later on, but don't worry too much about it; it's there mainly for optimisation.

15.1 `data` and constructor functions

`data` is used to define new data types using existing ones as building blocks. Here's a data structure for elements in a simple list of anniversaries:

```
data Anniversary = Birthday String Int Int Int        -- name, year, month, day
                 | Wedding String String Int Int Int -- spouse name 1, spouse
 name 2, year, month, day
```

A type declaration like this has two effects:

- First, it declares a new data type `Anniversary`, which can be either a Birthday or a Wedding. A Birthday *contains* one string and three integers and a Wedding *contains* two strings and three integers. The definitions of the two possibilities are separated by the vertical bar. The text after the "--" are just comments, explaining to readers of the code what the fields actually mean.

- Moreover, it defines two *constructor functions* for `Anniversary`, called, appropriately enough, `Birthday` and `Wedding`. These functions provide a way to build a new `Anniversary`, be it a "Birthday" or a "Wedding".

Types defined by `data` declarations are often referred to as *algebraic data types*. We will eventually return to this topic to address the theory behind such a name and the practical implications of said theory.

As usual with Haskell the case of the first letter is important: type names and constructor functions must always start with capital letters. Other than this syntactic detail, constructor functions work pretty much like the "conventional" functions we met so far. In fact, if you use `:t` in ghci to query the type of, say, `Birthday`, you'll get:

- `Main> :t Birthday Birthday :: String -> Int -> Int -> Int -> Anniversary`

Meaning it's just a function which takes one String and three Int as arguments and *evaluates to* an `Anniversary`. This anniversary will contain the four arguments we passed as specified by the `Birthday` constructor.

Calling the constructor functions is no different from calling other 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 anniversaries can, for instance, be put in a list:

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

Or you could just as easily have called the constructors straight away when building the list (although the resulting code looks a bit cluttered).

```
anniversariesOfJohnSmith = [Birthday "John Smith" 1968 7 3, Wedding "John Smith"
 "Jane Smith" 1987 3 4]
```

15.2 Deconstructing types

So far we've seen how to define a new type and create values of that type. If the new data types are to be of any use, however, we must have a way to access its contents, that is, the values we used at construction. For instance, one very basic operation with the anniversaries defined above would be extracting the names and dates they contain as a String. Let us see how a function which does that, `showAnniversary`, could be written and analyse what's going on in it (you'll see that, for the sake of code clarity, we used an auxiliary `showDate` function but let's ignore it for a moment and focus on `showAnniversary`).

```
showDate :: Int -> Int -> Int -> String
showDate y m d = show y ++ "-" ++ show m ++ "-" ++ show d

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 ++ " on " ++ showDate year month day
```

This example shows what is truly special about constructor functions: they can also be used to deconstruct the values they build. `showAnniversary` takes a single argument of type `Anniversary`. Instead of just providing a name for the argument on the left side of the definition, however, we specify one of the constructor functions and give names to each argument of the constructor – which correspond to the contents of the Anniversary. A more formal way of describing this "giving names" process is to say we are *binding variables* – "binding" is being used in the sense of assigning a variable to each of the values so that we can refer to them on the right side of the function definition.

For such an approach to be able to handle both "Birthday" and "Wedding" Anniversaries, we needed to provide *two* function definitions, one for each constructor. When `showAnniversary` is called, if the argument is a `Birthday` Anniversary the first version is used and the variables `name`, `month`, `date` and `year` are bound to its contents. If the argument is a `Wedding` Anniversary then the second version is used and the variables are bound in the same way. This process of using a different version of the function depending on the type of constructor used to build the `Anniversary` is pretty much like what happens when we use a `case` statement or define a function piece-wise.

An important observation on syntax is that the parentheses around the constructor name and the bound variables are mandatory; otherwise the compiler or interpreter would not take them as a single argument. Also, it is important to have it absolutely clear that the expression inside the parentheses is *not* a call to the constructor function, even though it may look just like one.

Exercises:
Note: The solution of this exercise is given near the end of the chapter, so we recommend that you attempt it before getting there.
Reread the function definitions above, now having a closer look at the `showDate` helper function. In spite of us saying it was provided "for the sake of code clarity", there is a certain clumsiness in the way it is used. You have to pass three separate Int arguments to it, but these arguments are always linked to each other in that they are part of a single date. It would make no sense to do things like passing the year, month and day values of the Anniversary in a different order, or to pass the month value twice and omit the day.

- Could we use what we've seen in this chapter so far to reduce this clumsiness?
- Declare a `Date` type which is composed of three Int, corresponding to year, month and date. Then, rewrite `showDate` so that it uses the new data type, and picture the changes needed in `showAnniversary` and the `Anniversary` for them to make use of it as well.

15.3 type for making type synonyms

As mentioned in the introduction of this module, code clarity is one of the motivations for using custom types. In that spirit, it could be nice to make it clear that the Strings in the Anniversary type are being used as names while still being able to manipulate them like ordinary Strings. The `type` declaration allows us to do that.

```
type Name = String
```

The code above says that a `Name` is now a synonym for a `String`. Any function that takes a `String` will now take a `Name` as well (and vice-versa: functions that take `Name` will accept any `String`). 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]
```

We can do something similar for the list of anniversaries we made use of:

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

Since type synonyms are mostly a convenience feature for making the roles of types clearer or providing an alias to, for instance, a complicated list or tuple type, it is largely a matter of personal discretion to decide how they should be deployed. Abuse of synonyms might even, on occasion, make code confusing (for instance, picture a long program using multiple names for common types like Int or String simultaneously, all of course having the same functionality).

Incorporating the suggested type synonyms and the `Date` type we proposed in the exercise(*) of the previous section the code we've written so far looks like this:

((*) **last chance to try that exercise without looking at the spoilers.**)

```
type Name = String

data Anniversary =
   Birthday Name Date
   | Wedding Name Name Date

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

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

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

type AnniversaryBook = [Anniversary]

anniversariesOfJohnSmith :: AnniversaryBook
anniversariesOfJohnSmith = [johnSmith, smithWedding]

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

showAnniversary :: Anniversary -> String
showAnniversary (Birthday name date) =
   name ++ " born " ++ showDate date
```

```
showAnniversary (Wedding name1 name2 date) =
   name1 ++ " married " ++ name2 ++ " on " ++ showDate date
```

Even in a simple example like this one, there is a very noticeable gain in terms of simplicity and clarity when compared to what would be needed to do the same task using only Ints, Strings, and corresponding lists.

A final observation on syntax: note that the `Date` type has a constructor function which is called `Date` as well. That is perfectly valid and indeed giving the constructor the same name of the type when there is just one constructor is good practice, as a simple way of making the role of the function obvious.

Note:
After these initial examples, the mechanics of using constructor functions may look a bit unwieldy, particularly if you're familiar with analogous features in other languages. There are syntactical constructs that make dealing with constructors more convenient. These will be dealt with later on, when we return to the topic of constructors and data types to explore them in detail.

16 Pattern matching

In the previous modules of this book we introduced and then made occasional reference to pattern matching, pointing out common uses. Now that we have developed some familiarity with the language, it is time to take a proper, deeper look at pattern matching. We will kick-start the discussion with a condensed description, which we will expanded upon throughout the chapter:

In pattern matching, we attempt to ***match****values against* ***patterns****and, if so desired,* ***bind****variables to successful matches.*

Note:
Pattern matching on what?
Some languages like Perl and Python use the term *pattern matching* for matching regular expressions against strings. The pattern matching we are referring to in this chapter is something completely different. In fact, you're probably best off forgetting what you know about pattern matching for now.[a] Here, pattern matching is used in the same way as in other ML-like languages: to deconstruct values according to their type specification.

[a] If you came here looking for regex pattern matching, you might be interested in looking at the Haskell Text.Regex ^{http://hackage.haskell.org/packages/archive/regex-compat/0.95.1/doc/html/Text-Regex.html} library wrapper.

16.1 Analysing pattern matching

Pattern matching is virtually everywhere. To pick but one example, consider a definition like that of `map`:

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

Here, at surface level, there are four different patterns involved, two per equation. Let's explore each one in turn, beginning with the second equation:

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

In the `(x:xs)` pattern, `x` and `xs` can be seen as sub-patterns used to match the parts of the list. Just like `f`, they match anything - though it is evident that if there is a successful match and x has type a, xs will have type [a]. Finally, these considerations imply that xs will also match an empty list, and so a one-element list matches (x:xs).

From the above dissection, we can say pattern matching gives us a way to:

- *recognize values*. For instance, when map is called and the second argument matches [] the first equation for map is used instead of the second one.
- *bind variables* to the recognized values. In this case, the variables f, x, and xs are assigned to the values passed as arguments to map when the second equation is used, and so we can use these values through the variables in the right-hand side of =. As _ and [] show, binding is not an essential part of pattern matching, but just a side effect of using variable names as patterns.
- *break down values into parts*, as the (x:xs) pattern does by binding two variables to parts (head and tail) of a matched argument (the non-empty list).

16.2 The connection with constructors

Regardless of the detailed analysis above, the process of using (:) to break down a list, as if we were undoing the effects of the (:) operator, may look a little too magical. It will not, however, work with any arbitrary operator. For example, one might think, by analogy to how (:) is used to break down a list, of defining a function which uses (++) to chop off the first three elements of a list:

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

However, that *will not work*. The function (++) is not allowed in patterns, and in fact most other functions that act on lists are not allowed as well. Which functions, then, *are* allowed?

In one word, *constructors* – the functions used to build values of algebraic data types. Let us consider a random example:

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

Here Bar and Baz are constructors for the type Foo. And so you can use them for pattern matching Foo values, and bind variables to the Int value contained in a Foo constructed with Baz:

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

That was exactly what was going on back when we defined showAnniversary and showDate in the Type declarations module. For instance:

```
data Date = Date Int Int Int   -- Year, Month, Day
showDate :: Date -> String
showDate (Date y m d) = show y ++ "-" ++ show m ++ "-" ++ show d
```

The (Date y m d) pattern in the left-hand side of the showDate definition matches a Date (built with the Date constructor) and binds the variables y, m and d to the contents of the Date value.

16.2.1 Why does it work with lists?

As for lists, they are not different from data-defined algebraic data types as far as pattern matching is concerned. It works as if lists were defined with this data declaration (note that the following isn't actually valid syntax: lists are in reality deeply ingrained into Haskell):

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

So the empty list, [] and the (:) function are in reality constructors of the list datatype, and so you can pattern match with them. [] takes no arguments, and therefore no variables can be bound when it is used for pattern matching. (:) takes two arguments, the list head and tail, which may then have variables bound to them when the pattern is recognized.

```
Prelude> :t []
[] :: [a]
Prelude> :t (:)
(:) :: a -> [a] -> [a]
```

Furthermore, since [x, y, z] is just syntactic sugar for x:y:z:[], we can solve the dropThree problem using pattern matching alone:

```
dropThree :: [a] -> [a]
dropThree (_:_:_:xs) = xs
dropThree _          = []
```

The first pattern will match any list with at least three elements. The catch-all second definition provides a reasonable default[1] when lists fail to match the main pattern, and thus prevents runtime crashes due to pattern match failure.

Note:
From the fact that we *could* write a dropThree function with bare pattern matching it doesn't follow that we *should* do so! Even though the solution is simple, it is still a waste of effort to code something this specific when we could just use Prelude and settle it with drop 3 xs instead. Mirroring what was said before about baking bare recursive functions, we might say: *don't get too excited about pattern matching either...*

16.2.2 Tuple constructors

Analogous considerations are valid for tuples. Our access to their components via pattern matching...

```
fstPlusSnd :: (Num a) => (a, a) -> a
fstPlusSnd (x, y) = x + y

norm3D :: (Floating a) => (a, a, a) -> a
norm3D (x, y, z) = sqrt (x^2 + y^2 + z^2)
```

[1] Reasonable *for this particular task*, and just because it makes sense to expect that dropThree will give [] when applied to a list of, say, two elements. With a different problem, perhaps it would not be reasonable to return *any* list if the first match failed. Later in the book we will consider one simple way of dealing with such cases.

... is granted by the existence of tuple constructors. For pairs, the constructor is the comma operator, `(,)`; for larger tuples there are `(,,)`; `(,,,)` and so on. These operators are slightly unusual in that we can't use them infix in the regular way; so `5 , 3` is not a valid way to write `(5, 3)`. All of them, however, can be used prefix, which is occasionally useful.

```
Prelude> (,) 5 3
(5,3)
Prelude> (,,,) "George" "John" "Paul" "Ringo"
("George","John","Paul","Ringo")
```

16.3 Matching literal values

As discussed earlier in the book, a simple piece-wise function definition like this one

```
f :: Int -> Int
f 0 = 1
f 1 = 5
f 2 = 2
f _ = -1
```

is performing pattern matching as well, matching the argument of `f` with the `Int` literals 0, 1 and 2, and finally with `_` . In general, numeric and character literals can be used in pattern matching[2]. They can also be used together with constructor patterns. For instance, this function

```
g :: [Int] -> Bool
g (0:[]) = False
g (0:xs) = True
g _ = False
```

will evaluate to False for the [0] list, to True if the list has 0 as first element and a non-empty tail and to False in all other cases. Also, lists with literal elements like [1,2,3], or even "abc" (which is equivalent to ['a','b','c']) can be used for pattern matching as well, since these forms are only syntactic sugar for the (:) constructor.

It costs nothing to emphasize that the above considerations are only valid for literal values, so the following will **not** work:

```
k = 1
--again, this won't work as expected
h :: Int -> Bool
h k = True
h _ = False
```

[2] As perhaps could be expected, this kind of matching with literals is not constructor-based. Rather, there is an equality comparison behind the scenes.

Exercises:

1. Test the flawed `h` function above in GHCi, with arguments equal to and different from 1. Then, explain what goes wrong.
2. In this section about pattern matching with literal values we made no mention of the boolean values `True` and `False`, even though we can do pattern matching with them, as demonstrated in the Next steps[a] chapter. Can you guess why we omitted them? (Hint: is there anything distinctive about the way we write boolean values?)

[a] Chapter 9.2 on page 62

16.4 Syntax tricks

16.4.1 As-patterns

Sometimes, when matching a pattern with a value, it may be useful to bind a name also to the whole value being matched. As-patterns allow exactly this: they are of the form *`var@pattern`* and have the additional effect to bind the name `var` to the whole value being matched by `pattern`. For instance, here is a toy variation on the map theme:

```
contrivedMap :: ([a] -> a -> b) -> [a] -> [b]
contrivedMap f [] = []
contrivedMap f list@(x:xs) = f list x : contrivedMap f xs
```

`contrivedMap` passes to the parameter function `f` not only `x` but also the undivided list used as argument of each recursive call. Writing it without as-patterns would have been more cumbersome because we would have to needlessly *reconstruct* the original value of `list`, i.e. actually evaluate `x:xs` on the right side:

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

16.4.2 Introduction to records

For constructors with many elements, *records* provide useful syntactical help. Briefly, they are a way of naming values in a datatype, using the following syntax:

```
data Foo2 = Bar2 | Baz2 {bazNumber::Int, bazName::String}
```

Using records allows doing matching and binding only for the variables relevant to the function we're writing, making code much clearer:

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

Also, the {} pattern can be used for matching a constructor regardless of the datatype elements even if you don't use records in the data declaration:

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

The function g does not have to be changed if we modify the number or the type of elements of the constructors Bar or Baz.

There are even more potential advantages in using record syntax. We will cover records in more detail further down the road; in case you want to start using them right now it may be useful to have a look at the Named fields[3] section of the More on datatypes chapter.

16.5 Where we can use pattern matching

The short answer is that *wherever you can bind variables, you can pattern match.* Let us have a glance at such places we have seen before; a few more will be introduced in the following chapters.

16.5.1 Equations

The most obvious use case is the left-hand side of function definition equations, which were the subject of our examples so far.

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

In the map definition we're doing pattern matching on the left hand side of both equations, and also binding variables on the second one.

16.5.2 let expressions and where clauses

Both let and where are ways of doing local variable bindings. As such, you can also use pattern matching in them. A simple example:

```
y =
  let
    (x:_) = map ((*) 2) [1,2,3]
  in x + 5
```

Or, equivalently,

```
y = x + 5
  where
  (x:_) = map ((*) 2) [1,2,3]
```

[3] Chapter 24.2 on page 155

Here, x will be bound to the first element of map ((*) 2) [1,2,3]. y, therefore, will evaluate to $2+5=7$.

16.5.3 List comprehensions

After the | in list comprehensions you can pattern match. This is actually extremely useful, and adds a lot to the expressiveness of comprehensions. Let's see how that works with a slightly more sophisticated example. Prelude provides a Maybe type which has the following constructors:

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

It is typically used to hold values resulting from an operation which may or may not succeed; if the operation succeeds, the Just constructor is used and the value is passed to it; otherwise Nothing is used[4]. The utility function catMaybes (which is available from Data.Maybe library module) takes a list of Maybes (which may contain both "Just" and "Nothing" Maybes), and retrieves the contained values by filtering out the Nothing values and getting rid of the Just wrappers of the Just x. Writing it with list comprehensions is very straightforward:

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

Another nice thing about using a list comprehension for this task is that if the pattern match fails (that is, it meets a Nothing) it just moves on to the next element in ms, thus avoiding the need of explicitly handling constructors we are not interested in with alternate function definitions[5].

16.5.4 do blocks

Within a do block like the ones we used in the Simple input and output[6] chapter, we can pattern match with the left-hand side of the left arrow variable bindings:

```
putFirstChar = do
    (c:_) <- getLine
    putStrLn [c]
```

Furthermore, the let bindings in do blocks are, as far as pattern matching is concerned, just the same as the "real" let expressions.

[4] The canonical example of such an operation is looking up values in a *dictionary* - which might just be a [(a, b)] list with the tuples being key-value pairs, or a more sophisticated implementation. In any case, if we, given an arbitrary key, try to retrieve a value there is no guarantee we will actually find a value associated to the key.

[5] *Note for the curious*: The reason why it works this way instead of crashing out on a pattern matching failure has to do with the real nature of list comprehensions - namely, being nice wrappers for the list monad. Please don't lose sleep over this aside; we will explain what that means eventually.

[6] Chapter 10 on page 69

17 Control structures

Haskell offers several ways of expressing a choice between different values. We explored some of them through the Haskell Basics chapter. This section will bring together what we have seen thus far, discuss some finer points and introduce a new control structure.

17.1 if and guards revisited

We have already met these constructs. The syntax for if expressions is:

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

<condition> is an expression which evaluates to a boolean. 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 (which is converted to a value) – and not a statement (which is executed) like in many imperative languages[1]. As a consequence, in Haskell the else is *mandatory.* Since if is an expression, it must evaluate to a result whether the condition is true or false, and the else ensures this. Furthermore, <true-value> and <false-value> must evaluate to the same type, which will be the type of the whole if expression.

When if expressions are split across multiple lines, they are usually indented by aligning elses with thens, rather than with ifs. A common style looks like this:

```
describeLetter :: Char -> String
describeLetter c =
    if c >= 'a' && c <= 'z'
        then "Lower case"
        else if c >= 'A' && c <= 'Z'
            then "Upper case"
            else "Not an ASCII letter"
```

Guards and top-level if expressions are mostly interchangeable. The example above, for instance, does look neater with guards:

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

[1] If you programmed in C or Java, you will recognize Haskell's if/then/else as an equivalent to the ternary conditional operator ?: .

Remember that otherwise it is just an alias to True, and thus the last guard is a catch-all, playing the role of the final else of the if expression.

Guards are evaluated in the order they appear. Consider a set up similar to the following:

```
f (pattern1) | predicate1 = w
             | predicate2 = x

f (pattern2) | predicate3 = y
             | predicate4 = z
```

Here, the argument of f will be pattern-matched against pattern1. If it succeeds, then we proceed to the first set of guards: if predicate1 evaluates to True, then w is returned. If not, then predicate2 is evaluated; and if it is true x is returned. Again, if not, then we proceed to the next case and try to match the argument against pattern2, repeating the guards procedure with predicate3 and predicate4. (And of course if neither pattern matches or neither predicate is true for the matching pattern there will be a runtime error. Regardless of the chosen control structure, it is important to ensure all cases are covered.)

17.1.1 Embedding if expressions

A handy consequence of if constructs being *expressions* is that they can be placed anywhere a Haskell expression could be, allowing us to write code like this:

```
g x y = (if x == 0 then 1 else sin x / x) * y
```

Note that we wrote the if expression without line breaks for maximum terseness. Unlike if expressions, guard blocks are not expressions; and so a let or a where definition is the closest we can get to this style when using them. Needless to say, one-liner if expressions more complicated than the one in this example would be annoying to read, making let and where attractive options in such cases.

17.2 case expressions

One control structure we haven't talked about yet are case expressions. They are to piece-wise function definitions what if expressions are to guards. Take this simple piece-wise definition:

```
f 0 = 18
f 1 = 15
f 2 = 12
f x = 12 - x
```

It is equivalent to - and, indeed, syntactic sugar for:

```
f x =
    case x of
        0 -> 18
        1 -> 15
        2 -> 12
        _ -> 12 - x
```

Whatever definition we pick, the same happens when f is called: The argument x is matched against all of the patterns in order; and on the first match the expression on the right-hand side of the corresponding equal sign (in the piece-wise version) or arrow (in the case version) is evaluated. Note that in this case expression there is no need to write x in the pattern; the wildcard pattern _ gives the same effect[2].

Indentation is important when using case. The cases must be indented further to the right than the line where the of keyword is, and all cases must have the same indentation. For the sake of illustration, here are two other valid layouts for a case expression:

```
f x = case x of
    0 -> 18
    1 -> 15
    2 -> 12
    _ -> 12 - x
```

```
f x = case x of 0 -> 18
                1 -> 15
                2 -> 12
                _ -> 12 - x
```

Since the left hand side of any case branch is just a pattern, it can also be used for binding, exactly like in piece-wise function definitions[3]:

```
describeString :: String -> String
describeString str =
  case str of
    (x:xs) -> "The first character of the string is: " ++ [x] ++ "; and " ++
              "there are " ++ show (length xs) ++ " more characters in it."
    []     -> "This is an empty string."
```

This function describes some properties of str using a human-readable string. Of course, you could do that with an if-statement (with a condition of null str to pick the empty string case), but using a case binds variables to the head and tail of our list, which is convenient for what we are doing.

Finally, just like if expressions (and unlike piece-wise definitions), case expressions can be embedded anywhere another expression would fit:

```
data Colour = Black | White | RGB Int Int Int

describeBlackOrWhite :: Colour -> String
describeBlackOrWhite c =
  "This colour is"
  ++ case c of
       Black           -> " black"
       White           -> " white"
       RGB 0 0 0       -> " black"
       RGB 255 255 255 -> " white"
       _               -> "... uh... something else"
  ++ ", yeah?"
```

[2] To see why it is so, consider our discussion of matching and binding in the ../Pattern matching/ ^{Chapter16 on page 113} section

[3] That makes case statements a lot more versatile than most of the superficially similar switch/case statements in imperative languages, which are typically restricted to equality tests on integral primitive types.

The case block above fits in as any string would. Writing `describeBlackOrWhite` this way makes `let`/`where` unnecessary, though the resulting definition is not very readable.

Exercises:
Use a `case` statement to implement a `fakeIf` function which could be used as a replacement to the familiar `if` expressions.

17.3 Controlling actions, revisited

On the final part of this chapter we will take advantage of having just introduced `case` expressions to introduce a few extra points about control structures while revisiting the discussions in the "Simple input and output" chapter. There, on the Controlling actions[4] section, we used this function to show how to execute actions conditionally within a `do` block using `if` expressions:

```
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!"
```

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 a value of type `Ordering`, namely 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!"
```

The `do`s after the `->`s are necessary on the first two options, because we are sequencing actions within each case.

17.3.1 A note about `return`

Now, we are going to dispel a possible source of confusion. In a typical imperative language - say, C - an implementation of `doGuessing` might look like this (if you don't know C, don't worry with the details and just follow the if-else chain):

[4] Chapter 10.1.2 on page 73

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

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

This doGuessing first tests the equality case, which does not lead to a new call of doGuessing, with an if that has no accompanying else. If the guess was right, a return statement is used to exit the function at once, skipping the other cases. Now, going back to Haskell, action sequencing in do blocks looks a lot like imperative code, and furthermore there actually *is* a return in Prelude. Then, knowing that case statements (unlike if statements) do not force us to cover all cases, one might be tempted to write a literal translation of the C code above (try running it if you are curious)...

```
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 if guess == num
  if (read guess < num)
    then do print "Too low!";
            doGuessing num
    else do print "Too high!";
            doGuessing num
```

... but it won't work! If you guess correctly, the function will first print "You win!," but it *will not exit* at the return (); following to the if expression instead and checking 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. Things aren't any better with an incorrect guess: 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 (as usual, the incomplete case alone should be enough to raise suspicion).

The problem here is that return is not at all equivalent to the C (or Java etc.) statement with the same name. For our immediate purposes, we can say that return is a *function* [5] , and that return () evaluates to an action which does nothing - return *does not affect the*

[5] *Superfluous note*: somewhat closer to a proper explanation, we might say return is a function which takes a value and makes it into an action which, when evaluated, gives the original value. A return "strawberry" within one of the do blocks we are dealing with would have type IO String - the same one of getLine. Do not worry if that doesn't make sense for now; you will understand what return really does when we *actually* start discussing monads further ahead on the book.

control flow at all. In the correct guess case, the case statement evaluates to `return ()`, an action of type `IO ()`, and execution just follows along normally.

The bottom line is that while actions and `do` blocks resemble imperative code, they must be dealt with on their own terms - Haskell terms.

Exercises:

1. Redo the "Haskell greeting" exercise in Simple input and output/Controlling actions[a], this time using a `case` statement.
2. What does the following program print out? And why?

```
main =
 do x <- getX
    putStrLn x

getX =
 do return "My Shangri-La"
    return "beneath"
    return "the summer moon"
    return "I will"
    return "return"
    return "again"
```

[a] Chapter 10.1.2 on page 73

18 More on functions

As functions are absolutely essential to functional programming, there are several nice features that make using functions easier.

18.1 let and where revisited

First, a few extra words about `let` and `where`, which are useful to make local function definitions. A function like `addStr` from the List processing chapter...

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

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

... can be rewritten using local bindings in order to reduce clutter on the top level of the program (which makes a lot of sense assuming `addStr` is only used as part of `sumStr`). We can do that either with a `let` binding...

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

... or with a `where` clause...

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

... and the difference appears to be only one of style - bindings coming either before or after the rest of the definition. The relation between `let` and `where`, however, is similar to the one between `if` and guards, in that a `let...in` construct is an expression while a `where` clause isn't. That means we can embed a `let` binding, but not a `where` clause, in a complex expression in this way:

```
f x =
    if x > 0
        then (let lsq = (log x) ^ 2 in tan lsq) * sin x
        else 0
```

The expression within the outer parentheses is self-contained, and evaluates to the tangent of the square of the logarithm of `x`. Note that the scope of `lsq` does not extend beyond the parentheses; and therefore changing the then-branch to

```
        then (let lsq = (log x) ^ 2 in tan lsq) * (sin x + lsq)
```

wouldn't work - we would have to drop the parentheses around the `let`.

Still, `where` clauses *can* be incorporated into `case` expressions:

```
describeColour c =
   "This colour "
   ++ case c of
          Black -> "is black"
          White -> "is white"
          RGB red green blue -> " has an average of the components of " ++ show
 av
             where av = (red + green + blue) `div` 3
   ++ ", yeah?"
```

In this example, the indentation of the `where` clause sets the scope of the `av` variable so that it only exists as far as the `RGB red green blue` case is concerned. Placing it at the same indentation of the cases would make it available for all cases. Here is an example with guards:

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

Note that since there is one equals sign for each guard there is no place we could put a `let` expression which would be in scope of all guards, as the `where` clause is. Here we have a situation in which `where` is particularly convenient.

18.2 Anonymous Functions - lambdas

An alternative to creating a private 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 expression 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`, which evaluates to "x + read str". So, the `sumStr` presented just above is precisely the same as the one that used `addStr` in a let binding.

Lambdas are handy for writing one-off functions to be used with maps, folds and their siblings, especially where the function in question is simple - as cramming complicated expressions in a lambda can hurt readability.

Since variables are being bound in a lambda expression (to the arguments, just like in a regular function definition), pattern matching can be used in them as well. A trivial example would be redefining `tail` with a lambda:

```
tail' = (\(_:xs) -> xs)
```

18.3 Operators and sections

As noted in a number of occasions, operators such as the arithmetical ones can be used surrounded in parentheses and used prefix, like other functions:

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

Generalizing that point, we can now define the term "operator" clearly: as far as Haskell is concerned it's a function with two arguments and a name consisting entirely of non-alphanumeric characters. Unlike other functions, operators can be used infix straight away. We can define new operators in the usual way; 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 (\zs y -> delete y zs) 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 (\zs y -> delete y zs) xs ys
```

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

Sections are a nifty piece of syntactical sugar that can be used with operators. An operator within parentheses and flanked by one of its arguments...

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

... is a new function in its own right. `(2+)`, for instance, has the type `(Num a) => a -> a`. We can pass sections to other functions, e.g. `map (+2) [1..4] == [3..6]`. For another example, we can add an extra flourish to the `multiplyList` function we wrote back in More about lists:

```
multiplyList :: Integer -> [Integer] -> [Integer]
multiplyList m = map (m*)
```

If you have a "normal", 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*. 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 (that is, 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")``

19 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 section 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 section 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.

19.1 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 so, you can skip the following subsection and go straight to the next one.

19.1.1 The Idea Behind `quickSort`

The idea is very 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.

19.1.2 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! Even if you *have* met `quickSort` before, perhaps you thought recursion was a neat trick but difficult to implement.

19.2 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 (quickSort dictionary)` 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 `String`s are represented in a typical programming settings is by a list of Unicode characters. Unicode (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.

19.3 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.

19.4 `quickSort`, Take Two

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).

```haskell
-- no matter how we compare two things
-- the first two equations should not change
-- they need to accept the comparison function though
-- but the result doesn't need it
quickSort _ [] = []
quickSort _ [x] = [x]

-- we are in a more general setting now
-- but the changes are worth it!
-- c is the comparison function
quickSort c (x : xs) = (quickSort c less) ++ (x : equal) ++ (quickSort c more)
                             where less  = filter (\y -> y `c` x == LT) xs
                                   equal = filter (\y -> y `c` x == EQ) xs
                                   more  = filter (\y -> y `c` 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.

19.5 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"]`.

19.6 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

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

An example of how this function would be used might be

```
for 1 (<10) (+1) (print)
```

which prints the numbers 1 to 9 on the screen.
Starting from an initial value `i`, the `for` executes `job i`. It then modifies this value `f i` and checks to see if the modified value satisfies some condition. If it doesn't, 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
4. 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?
5. 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.
6. 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!

19.7 Currying

I hope you followed the reasoning of the preceding chapter closely enough. If you haven't, you should, so give it another try.

Currying is a technique[1] that lets you partially apply a multi-parameter function. When you do that, it remembers those given values, and waits for the remaining parameters.

Our `quickSort` takes two parameters, the comparison function, and the list. We can, by currying, construct variations of `quickSort` with a given comparison function. The variation just "remembers" the specific comparison, so you can apply it to the list, and it will sort the list using that comparison function.

```
descendingSort = quickSort descending
```

What is the type of descendingSort? `quickSort` was `(a -> a -> Ordering) -> [a] -> [a]`, and the comparison function `descending` was `a -> a -> Ordering`. Applying `quickSort` to `descending` (that is, applying it *partially*, we haven't specified the list in the definition) we get a function (our `descendingSort`) for which the first parameter is already given, so you can scratch that type out from the type of `quickSort`, and we are left with a simple `[a] -> [a]`. So we can apply this one to a list right away!

Note: This will not work in newer compilers without either applying a pragma or adding a definition: `descendingSort :: Ord a => [a] -> [a]`

```
descendingSort dictionary
```

gives us

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

It's rather neat. But is it useful? You bet it is. It is particularly useful as sections, you might notice. Currying often makes Haskell programs very concise. More than that, it often makes your intentions a lot clearer. Remember

```
less = filter (< x) xs
```

from the first version of `quickSort`? You can read it aloud like "keep those in `xs` that are less than `x`". Although `(< x)` is just a shorthand for `(\y -> y < x)`, try reading *that* aloud!

19.8 Function manipulation

We will close the chapter discussing a few examples of very common and very useful general-purpose higher-order functions, beginning with `flip`. `flip` is a handy little Prelude function that has the following type:

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

[1] named after the outstanding logician Haskell Brooks Curry, the same guy after whom the language is named.

It takes a function of two arguments and returns a version of the same function with the arguments swapped, so that:

```
Prelude> (flip (/)) 3 1
0.3333333333333333
Prelude> (flip map) [1,2,3] (*2)
[2,4,6]
```

We could have used flip to write the `descending` comparing function from the quickSort example point-free, as simply:

```
descending = flip compare
```

`flip` is particularly useful when we want to pass a function with two arguments of different types to another function and the arguments are in the wrong order with respect to the signature of the higher-order function.

The `(.)` composition operator is, as should be clear by now, just another higher-order function. It has the signature:

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

`(.)` takes two functions as argument and returns a new function, which applies the second argument and then the first.

Composition and higher-order functions provide a range of powerful tricks. For a very small sample of that, first consider the `inits` function, defined in the module `Data.List`. Quoting the documentation, it "returns all initial segments of the argument, shortest first", so that:

```
Prelude Data.List> inits [1,2,3]
[[],[1],[1,2],[1,2,3]]
```

With the higher order functions `flip`, `scanl`, `(.)` and `map` we can, using only functions in Prelude, provide the following one line implementation for `inits` (written point-free for extra dramatic effect):

```
myInits :: [a] -> [[a]]
myInits = map reverse . scanl (flip (:)) []
```

Swallowing a definition so condensed may look daunting at first; so slowly analyse it bit by bit, recalling what each function does and using the type signatures as a guide.

Note that the definition of `myInits` is not only very concise but also clean, in particular with parentheses usage kept to a bare minimum. Naturally, if one goes overboard with composition by writing mile-long `(.)` chains things are bound to get confusing, but when deployed reasonably this is a very attractive style. Furthermore, the implementation is quite "high level": we do not deal explicitly at all with details like pattern matching or recursion; the functions we deployed - both the higher-order ones and their functional arguments - take care of such plumbing.

Finally, we will present a very curious higher-order operator, `($)`. Its type is:

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

It takes a function as its first argument, and *all* it does is to apply the function to the second argument, so that, for instance, `(head $ "abc") == (head "abc")`.

By now, it would be entirely justified if you thought that `($)` was completely useless! However, there are two interesting points about it. First, `($)` has very low precedence[2], unlike regular function application which has very high precedence. In effect, that means we can write a non-point-free version of `myInits` without having to add any parentheses in spite of the intricate expression at the left of the argument:

```
myInits :: [a] -> [[a]]
myInits xs = map reverse . scanl (flip (:)) [] $ xs
```

Furthermore, as `($)` is just a function which happens to apply functions, and functions are just values, we can write intriguing expressions such as:

```
map ($ 2) [(2*), (4*), (8*)]
```

(Yes, that is a list of functions, and it is perfectly legal.)

Function manipulation through higher-order gives us a great deal of power. As we continue throughout this book we will see many examples of such power being harnessed.

[2] Precedence here is meant in the sense that + has lower precedence than *, and therefore `2 + 3 * 4` equals 14 and not 20.

20 Using GHCi effectively

GHCi offers several ways to make you work faster. This section will describe them and explain what they are for.

20.1 User interface

20.1.1 Tab completion

Whenever you are typing anything into the GHCi, you can hit Tab and you will be presented with a list of all possible names that start with what you've written so far. And if there is only one way how to continue, the string will be auto-completed. For example fol`<Tab>` will append letter d, another `Tab` will list four functions: fold foldl1 foldr and foldr1. Only functions and other defined objects (e.g. Data types) from modules that are currently imported are offered.

Tab completion works also when you are loading a file with your program into GHCi. After typing :l fi`<Tab>`, you will be presented with all files that start with "fi" and are present in the current directory (the one you were in when you launched GHCi).

The same also applies when you are importing modules, after typing :m +Da`<Tab>`, you will be presented with all modules that start with "Da" present in installed packages.

20.1.2 ": commands"

On GHCi command line, commands for the interpreter start with the character ":" (colon).

- :help -- prints a list of all available commands.
- :load myfile.hs -- loads a file (here, "myfile.hs") into GHCi.
- :reload -- reloads the file that has been loaded last time, so changes are visible from GHCi.
- :type <Haskell expression> -- prints the type of a given expression, e.g. a function that has been defined previously.
- :module +Data.List -- loads a module (in this example, `Data.List`). You can also *un*load a module with `:module -Data.List`.
- :browse Data.List -- gives the type signatures for all functions available from a module.

Many of these commands can be abbreviated - :l for :load, :r for :reload, :h for :help, :t for :type and :m for module.

Here again, you can use `Tab` to see the list of commands, type `:Tab` to see all possible commands.

20.1.3 Timing Functions in GHCi

GHCi provides a basic way to measure how much time a function takes to run, which can be useful for people who quickly want to find out which version of a function runs faster.

1. Type `:set +s` into the ghci command line.
2. run the function(s) you are testing. The time the function took to run will be displayed after GHCi outputs the results of the function.

20.1.4 Multi-line Input

If you are trying to define a function that takes up multiple lines, or if you want to type a do block into ghci, there is an easy way to do this:

1. Begin a new line with `:{`
2. Type in your code. Press enter when you need a new line.
3. Type `:}` to end the multiline input.

For example:

```
   *Main> :{
   *Main| let askname = do
   *Main|               putStrLn "What is your name?"
   *Main|               name <- getLine
   *Main|               putStrLn $ "Hello " ++ name
   *Main| :}
   *Main>
```

The same can be accomplished by using `:set +m` command (allow multiline commands). An empty line will end the block.

In addition, line breaks in ghci commands can be separated by `;`, like this:

```
   *Main> let askname1 = do ; putStrLn "what is your name?" ; name <- getLine ;
putStrLn $ "Hello " ++ name
```

21 Intermediate Haskell

22 Modules

Modules are the primary means of organizing Haskell code. We have already met them in passing, when using `import` statements to put library functions into scope. In this chapter, we will have a closer look at how modules work. Beyond allowing us to make better use of libraries, such knowledge will help us to shape our own programs and, in particular, to create standalone programs which can be executed independently of GHCi (incidentally, that is the topic of the very next chapter, ../Standalone programs/[1]).

22.1 Modules

Haskell modules[2] 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. the name of the module begins with a capital letter;
2. each file contains only one module.

The name of the file must be that of the module but with a `.hs` file extension. Any dots `'.'` in the module name are changed for directories.[3] So the module `YourModule` would be in the file `YourModule.hs` while a module `Foo.Bar` would be in the file `Foo/Bar.hs` or `Foo\Bar.hs`. Since the module name must begin with a capital letter, the file name must also start with a capital letter.

[1] http://en.wikibooks.org/wiki/..%2FStandalone%20programs%2F

[2] See the Haskell report for more details on the module system ^{http://www.haskell.org/onlinereport/modules.html} .

[3] In Haskell98, the last standardised version of Haskell before Haskell 2010, the module system was fairly conservative, but recent common practice consists of employing an hierarchical module system, using periods to section off namespaces.

22.2 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.

22.2.1 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
```

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 the latter code snippet, the function `remove_e` isn't even defined. Instead, we call the functions from the imported modules by prefixing them with the module's name. Note that MyModule.remove_e also works if the qualified keyword 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 the function composition operator (.). 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

22.2.2 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 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 multiple imported modules, you must use qualified names.

22.2.3 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. We can improve things by 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.

22.2.4 Combining renaming with limited import

Sometimes it is convenient to use the import directive twice for the same module. A typical scenario is as follows:

```
import qualified Data.Set as Set
import Data.Set (Set, empty, insert)
```

This give access to all of the Data.Set module via the alias "Set", and also lets you access a few selected functions (empty, insert, and the constructor) without using the "Set" prefix.

22.3 Exporting

In the examples at the start of this article, the words "import *everything exported* from MyModule" were used.[4] 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 cannot 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
```

[4] A module may export functions that it imports. Mutually recursive modules are possible but need some special treatment ^{http://www.haskell.org/ghc/docs/latest/html/users_guide/separate-compilation.html#mutual-recursion} .

In this case, the module declaration could be rewritten "MyModule2 (Tree(..))", declaring that all constructors are exported.

Maintaining an export list is good practice not only because it reduces namespace pollution, but also because it enables certain compile-time optimizations[5] which are unavailable otherwise.

[5] http://www.haskell.org/haskellwiki/Performance/GHC#Inlining

23 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.[1]

23.1 The golden rule of indentation

While 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 beginning of that expression (even if the expression is not the leftmost element of the line).

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. When you start the expression on a separate line, you only need to indent by one space (although more than one space is also acceptable).

```
let
 x = a
 y = b
```

Although the separation above makes things very clear, it is common to place the first line alongside the 'let' and indent the rest to line up:

wrong

```
let x = a
 y = b
```

wrong

```
let x = a
     y = b
```

right

```
let x = a
    y = b
```

[1] See section 2.7 of The Haskell Report (lexemes) ^{http://www.haskell.org/onlinereport/lexemes.html#sect2.7} on layout.

This tends to trip up a lot of beginners: All *grouped* expressions must be exactly aligned. (On the first line, Haskell counts everything to the left of the expression as indent, even though it is not whitespace).

Here are some more examples:

```
do
  foo
  bar
  baz

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 first subsidiary expression on the same line as the 'case' keyword, unlike common practice with 'do' and 'where' expression. Hence the subsidiary expressions in a case expression tend to be indented only one step further than the 'case' line. 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 doesn't start at the left-hand edge. In this case, it's safe to just indent further than the line containing the expression's beginning. So,

```
myFunction firstArgument secondArgument = do -- the 'do' doesn't start at the
left-hand edge
  foo                                        -- so indent these commands more
than the beginning of the line containing the 'do'.
  bar
  baz
```

Here are some alternative layouts which work:

```
myFunction firstArgument secondArgument =
  do foo
     bar
     baz

myFunction firstArgument secondArgument = do foo
                                             bar
                                             baz
myFunction firstArgument secondArgument =
  do
     foo
     bar
     baz
```

23.2 A mechanical translation

Indentation is actually optional if you instead separate things using semicolons and curly braces such as in "one-dimensional" languages like C. It may be occasionally useful to write code in this style, and understanding how to convert from one style to the other can help understand the indentation rules. To do so, 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):

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:
Answer in one word: what happens if you see something indented MORE?

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
```

23.3 Layout in action

wrong	wrong	right	right
do first thing second thing third thing	do first thing second thing third thing	do first thing second thing third thing	do first thing second thing third thing

23.3.1 Indent to the first

Remember that, due to the "golden rule of indentation" described above, 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-looking 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**
if foo then do first thing second thing third thing else do something else	if foo then do first thing second thing third thing else do something else

This is also the reason why you can write things like this

```
main = do
 first thing
 second thing
```

or

```
main =
 do
   first thing
   second thing
```

instead of

```
main =
 do first thing
    second thing
```

Either way is acceptable

23.3.2 `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**
-- why is this bad? do first thing if condition then foo else bar third thing	-- still bad, just explicitly so do { first thing ; if condition ; then foo ; else bar ; third thing }

Naturally, the Haskell compiler is confused because it thinks that you never finished writing your `if` expression, before writing a new statement. The 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**
-- whew, fixed it! do first thing if condition then foo else bar third thing	-- the fixed version without sugar do { first thing ; if condition then foo else bar ; third thing }

This little bit of indentation prevents the do block from misinterpreting your `then` as a brand new expression. Of course, you might as well prefer to always add indentation before `then` and `else`, even when it is not really necessary. That wouldn't hurt legibility, and would avoid bad surprises like this one.

Exercises:
The if-within-do issue has tripped up so many Haskellers that one programmer has posted a proposal[a] to the Haskell prime initiative to add optional semicolons between `if then else`. How would that help?

[a] `http://hackage.haskell.org/trac/haskell-prime/ticket/23`

24 More on datatypes

24.1 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 it's only an enumeration if none of the constructors have arguments. 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. As you will see further on when we discuss classes and derivation, this distinction is not only conceptual.

Incidentally, the definition of the `Bool` datatype is:

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

24.2 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 a convenient update method. 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}`.

Note:
Those of you familiar with object-oriented languages might be tempted to, after all of this talk about "accessor functions" and "update methods", think of the `y{x=z}` construct as a setter method, which modifies the value of x in a pre-existing y. It is **not** like that – remember that in Haskell variables are immutable[a]. Therefore, if, using the example above, you do something like `conf2 = changeDir conf1 "/opt/foo/bar"` conf2 will be defined as a Configuration which is just like conf1 except for having "/opt/foo/bar" as its currentdir, but conf1 will remain unchanged.

[a] Chapter 3.4 on page 12

24.2.1 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 }
```

The first way is much shorter, although the second is much clearer.

24.3 Parameterized Types

Parameterized types are similar to "generic" or "template" types in other languages. A parameterized 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 parameterize `type` and `newtype` declarations in exactly the same way. Furthermore you can combine parameterized types in arbitrary ways to construct new types.

24.3.1 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:

```
pairOff :: Int -> Either Int String
pairOff people
  | people < 0 = Right "Can't pair off negative number of people."
  | people > 30 = Right "Too many people for this activity."
  | even people = Left (people `div` 2)
  | otherwise = Right "Can't pair off an odd number of people."

groupPeople :: Int -> String
groupPeople people = case pairOff people of
                       Left groups -> "We have " ++ show groups ++ " group(s)."
                       Right problem -> "Problem! " ++ problem
```

In this example `pairOff` indicates how many groups you would have if you paired off a certain number of people for your activity. It can also let you know if you have too many people for your activity or if somebody will be left out. So `pairOff` will return either an Int representing the number of groups you will have, or a String describing the reason why you can't create your groups.

24.3.2 Kind Errors

The flexibility of Haskell parameterized 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 parameterized types wrong then the compiler will report a kind error.

25 Other data structures

25.1 Trees

Now let's look at one of the most important data structures: trees. A tree is an example of a recursive datatype. While there are several different kinds of trees, for this discussion we will adopt the following definition, which captures the essential features of a tree that will concern us here:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

As you can see, it's parametrised, so we can have trees of `Int`s, trees of `String`s, trees of `Maybe Int`s, 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.

25.1.1 Lists as Trees

As we have seen in More about lists[1] and List Processing[2], 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)`:

25.1.2 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) deriving (Show)
data [a]    = []     | (:)    a [a]
  -- (:) a [a] is the same as (a:[a]) with prefix instead of infix notation.
```

[1] Chapter 13 on page 89

[2] Chapter 14 on page 97

We consider map first, then folds.

Note:
Deriving is explained later on in the section Class Declarations[a]. For now, understand it as telling Haskell (and by extension your interpreter) how to display a Tree instance.

[a] Chapter 26 on page 169

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`. 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
```

25.2 Other datatypes

Map and fold functions can be defined for any kind of data type. In order to generalize the strategy applied for lists and trees, in this final section we will work out a map and a fold for a very strange, intentionally contrived, datatype:

```
data Weird a b = First a
               | Second b
               | Third [(a,b)]
               | Fourth (Weird a b)
```

It can be a useful exercise to write the functions as you follow the examples, trying to keep the coding one step ahead of your reading.

25.2.1 General Map

Again, we will begin with weirdMap. The first important difference in working with this Weird type is that it has *two* type parameters. For that reason, we will want the map function to take two functions as arguments, one to be applied on the elements of type a and another for the elements of type b. With that accounted for, we can write the type signature of weirdMap:

```
weirdMap :: (a -> c) -> (b -> d) -> Weird a b -> Weird c d
```

Next step is writing the definitions for weirdMap. The key point is that maps preserve the *structure* of a datatype, so the function must evaluate to a Weird which uses the same constructor as the one used for the original Weird. For that reason, we need one definition to handle each constructor, and these constructors are used as patterns for writing them. As before, to avoid repeating the weirdMap argument list over and over again a **where** clause comes in handy. A sketch of the function is below:

```
weirdMap :: (a -> c) -> (b -> d) -> Weird a b -> Weird c d
weirdMap fa fb = g
  where
    g (First x)          = --More to follow
    g (Second y)         = --More to follow
    g (Third z)          = --More to follow
    g (Fourth w)         = --More to follow
```

The first two cases are fairly straightforward, as there is just a single element of a or b type inside the Weird.

```
weirdMap :: (a -> c) -> (b -> d) -> Weird a b -> Weird c d
weirdMap fa fb = g
  where
    g (First x)          = First (fa x)
    g (Second y)         = Second (fb y)
    g (Third z)          = --More to follow
    g (Fourth w)         = --More to follow
```

Third is trickier because it contains another data structure (a list) whose elements are themselves data structures (the tuples). So we need to navigate the nested data structures, apply fa and fb on all elements of type a and b inside it and eventually (as a map must preserve structure) produce a list of tuples – [(c,d)] – to be used with the constructor. The simplest approach might seem to be just breaking down the list inside the Weird and playing with the patterns:

```
    g (Third []) = Third []
    g (Third ((x,y):zs)) = Third ( (fa x, fb y) : ( (\(Third z) -> z) (g (Third
 zs)) ) )
```

This appears to be written as a typical recursive function for lists. We start by applying the functions of interest to the first element in order to obtain the head of the new list, (fa x, fb y). But what we will cons it to? As g requires a Weird argument we need to make a Weird using the list tail in order to make the recursive call. But then g will give a Weird and not a list, so we have to retrieve the modified list from that – that's the role of the lambda function. And finally, there is also the empty list base case to be defined as well.

After all of that, we are left with a messy function. Every recursive call of g requires wrapping zs into a Weird, while what we really wanted to do was to build a list with (fa x, fb y) and the modified xs. The problem with this solution is that g can (thanks to pattern matching) act directly on the list head but (due to its type signature) can't be called directly on the list tail. For that reason, it would be better to apply fa and fb without breaking down the list with pattern matching (as far as g is directly concerned, at least). But there *was* a way to directly modify a list element-by-element...

```
    g (Third z) = Third ( map (\(x, y) -> (fa x, fb y) ) z)
```

...our good old map function, which modifies all tuples in the list z using a lambda function. In fact, the first attempt at writing the definition looked just like an application of the list map except for the spurious Weird packing and unpacking. We got rid of these by having the pattern splitting of z done by map, which works directly with regular lists. You could find it useful to expand the map definition inside g for seeing a clearer picture of that difference. Finally, you may prefer to write this new version in an alternative, very clean way using list comprehension syntax:

```
    g (Third z) = Third [ (fa x, fb y) | (x,y) <- z ]
```

Adding the Third function, we only have the Fourth one left to do:

```
weirdMap :: (a -> c) -> (b -> d) -> Weird a b -> Weird c d
weirdMap fa fb = g
  where
    g (First x)          = First (fa x)
    g (Second y)         = Second (fb y)
    g (Third z)          = Third ( map (\(x, y) -> (fa x, fb y) ) z)
    g (Fourth w)         = --More to follow
```

Dealing with the recursive Fourth constructor is actually really easy. Just apply g recursively!

```
weirdMap :: (a -> c) -> (b -> d) -> Weird a b -> Weird c d
weirdMap fa fb = g
  where
    g (First x)          = First (fa x)
    g (Second y)         = Second (fb y)
    g (Third z)          = Third ( map (\(x, y) -> (fa x, fb y) ) z)
    g (Fourth w)         = Fourth (g w)
```

25.2.2 General Fold

While we were able to define a map by specifying as arguments 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 – one for handling the internal structure of the datatype specified by each constructor. Next, we have an argument of the Weird a b type, and finally we want the whole fold function to evaluate to a value of some other, arbitrary, type. Additionally, each individual function we pass to weirdFold must evaluate to the same type `weirdFold` does. That allows us to make a mock type signature and sketch the definition:

```
weirdFold :: (something1 -> c) -> (something2 -> c) -> (something3 -> c) ->
 (something4 -> c) -> Weird a b -> c
weirdFold f1 f2 f3 f4 = g
  where
    g (First x)          = --Something of type c here
    g (Second y)         = --Something of type c here
    g (Third z)          = --Something of type c here
    g (Fourth w)         = --Something of type c here
```

Now we need to figure out to which types `something1`, `something2`, `something3` and `something4` correspond to. That is done by analysing the constructors, since the functions must take as arguments the elements of the datatype (whose types are specified by the constructor type signature). Again, the types and definitions of the first two functions are easy to find. The third one isn't difficult either, as for the purposes of folding the list of `(a,b)` tuples is no different from a simple type – unlike in the map example, its *internal* structure does not concern us now. The fourth constructor, however, is recursive, and we have to watch out. As in the case of `weirdMap`, we also need to recursively call the `g` function. 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 = g
  where
    g (First x)          = f1 x
    g (Second y)         = f2 y
    g (Third z)          = f3 z
    g (Fourth w)         = f4 (g w)
```

Note:
If you were expecting very complex expressions in the `weirdFold` above and are surprised by the immediacy of the solution, it might be helpful to have a look on what the common `foldr` would look like if we wrote it in this style and didn't have the special square-bracket syntax of lists to distract us:

```
-- List a is [a], Cons is (:) and Nil is []
data List a = Cons a (List a) | Nil

listFoldr :: (a -> b -> b) -> (b) -> List a -> b
listFoldr fCons fNil = g
  where
    g (Cons x xs) = fCons x (g xs)
    g Nil         = fNil
```

Now it is easier to see the parallels. The extra complications are that `Cons` (that is, `(:)`) takes two arguments (and, for that reason, so does `fCons`) and is recursive, requiring a call to `g`. Also, `fNil` is of course not really a function, as it takes no arguments.

Folds on recursive datatypes

As far as folds are concerned `Weird` was a fairly nice datatype to deal with. Just one recursive constructor, which isn't even nested inside other structures. What would happen if we added a truly complicated fifth constructor?

```
Fifth [Weird a b] a (Weird a a, Maybe (Weird a b))
```

A valid, and tricky, question. In general, the following rules apply:

- A function to be supplied to a fold has the same number of arguments as the corresponding constructor.
- The type of the arguments of such a function match the types of the constructor arguments, *except* if the constructor is recursive (that is, takes an argument of its own type).
- If a constructor is recursive, any recursive argument of the constructor will correspond to an argument of the type the fold evaluates to.[3]
- If a constructor is recursive, the complete fold function should be (recursively) applied to the recursive constructor arguments.
- If a recursive element appears inside another data structure, the appropriate map function for that data structure should be used to apply the fold function to it.

So `f5` would have the type:

```
f5 :: [c] -> a -> (Weird a a, Maybe c) -> c
```

as the type of `Fifth` is:

```
Fifth :: [Weird a b] -> a -> (Weird a a, Maybe (Weird a b)) -> Weird a b
```

The definition of `g` for the `Fifth` constructor will be:

```
g (Fifth list x (waa, mc)) = f5 (map g list) x (waa, maybeMap g mc)
  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 `g` gets called. What's up? This is a recursion, right? Well... not really. `Weird a a` and `Weird a b` are different types, so it isn't a real recursion. It isn't guaranteed that, for example, `f2` 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 as:

- It preserves structure.
- Only types are changed.

[3] This sort of recursiveness, in which the function used for folding can take the result of another fold as an argument, is what confers the folds of data structures such as lists and trees their "accumulating" functionality.

26 Classes and types

Back in Type basics II[1] we had a brief encounter with type classes, which were presented as the mechanism used by Haskell to deal with number types. As we hinted back then, however, classes have many other uses. Starting with this chapter, we will see how to define and implement type classes, and how to use them to our advantage.

Broadly speaking, the point of type classes is to ensure that certain operations will be available for values of chosen types. For example, if we know a type belongs to (or, to use the jargon, *instantiates*) the class `Fractional`, then we are guaranteed to, among other things, be able to perform real division with its values[2].

26.1 Classes and instances

Up to now we have seen how existing type classes appear in signatures, like that of `(==)`:

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

Now it is time to switch perspectives. First, we quote the definition of the `Eq` class from Prelude:

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

       -- Minimal complete definition:
       --      (==) or (/=)
   x /= y     =  not (x == y)
   x == y     =  not (x /= y)
```

The definition states that if a type `a` is to be made an *instance* of the class `Eq` it must support the functions `(==)` and `(/=)` - the *class methods* - both of them having type `a -> a -> Bool`. Additionally, the class provides default definitions for (==) and `(/=)` *in terms of each other.* As a consequence, there is no need for a type in `Eq` to provide both definitions - given one of them, the other will be generated automatically.

With a class defined, we proceed to make existing types instances of it. Here is an arbitrary example of an algebraic data type made into an instance of `Eq` by an *instance declaration*:

[1] Chapter 7 on page 49

[2] *To programmers coming from object-oriented languages*: A class in Haskell in all likelihood is *not* what you expect - don't let the terms confuse you. While some of the uses of type classes resemble what is done with abstract classes or Java interfaces, there are fundamental differences which will become clear as we advance.

```
data Foo = Foo {x :: Integer, str :: String}

instance Eq Foo where
   (Foo x1 str1) == (Foo x2 str2) = (x1 == x2) && (str1 == str2)
```

And now we can apply `(==)` and `(/=)` to `Foo` values in the usual way:

```
• Main> Foo 3 "orange" == Foo 6 "apple" False
• Main> Foo 3 "orange" /= Foo 6 "apple" True
```

A few important remarks:

- 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. That works both ways - you could just as easily create a new class `Bar` and then declare the type `Integer` to be an instance of it.
- *Classes are not types*, but categories of types; and so the instances of a class are not values, but types[3].
- The definition of `(==)` for `Foo` relies on the fact that `Integer` and `String` are also members of `Eq`, as we are using `(==)` with the values of the fields. In fact almost all types in Haskell (the most notable exception being functions) are members of `Eq`.
- Type synonyms defined with `type` keyword cannot be made instances of a class.

26.2 Deriving

Since equality tests between values are commonplace, in all likelihood 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 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 an automatically generated definition of `==` exactly equivalent to the one we just wrote, and also makes it an instance of `Ord` and `Show` for good measure.

You can only use `deriving` with a limited set of built-in classes, which are described *very* briefly below:

Eq

Equality operators `==` and `/=`

[3] That is a key difference from most OO languages, where a class is also itself a type.

Ord

Comparison operators `<` `<=` `>` `>=`; `min`, `max` and `compare`.

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`, which converts a value into a string, and other related functions.

Read

Defines the function `read`, which parses a string into a value of the type, and other related functions.

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" function synthesis 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.

26.3 Class inheritance

Classes can inherit from other classes. For example, here is the main part of the definition of `Ord` in Prelude:

```
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 `=>` notation in the first line, which mirrors the way classes appear in type signatures. Here, it means that for a type to be an instance of `Ord` it must also be an instance of `Eq`, and hence needs to implement the `==` and `/=` operations[4].

A class can inherit from several other classes: just put all the ancestor classes in the parentheses before the `=>`. Let us illustrate that with yet another Prelude quote:

[4] If you check the full definition in the Prelude ^{http://hackage.haskell.org/packages/archive/base/4.1.0.0/doc/html/Prelude.html} specification, the reason for that becomes clear: the default implementations involve applying `(==)` to the values being compared.

```
class  (Num a, Ord a) => Real a  where
    -- | the rational equivalent of its real argument with full precision
    toRational          ::  a -> Rational
```

26.4 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, while the non-bold text stands for the types that are instances of each class (`(->)` refers to functions and `[]`, to lists). The arrows linking classes indicate the inheritance relationships, pointing to the inheriting class.

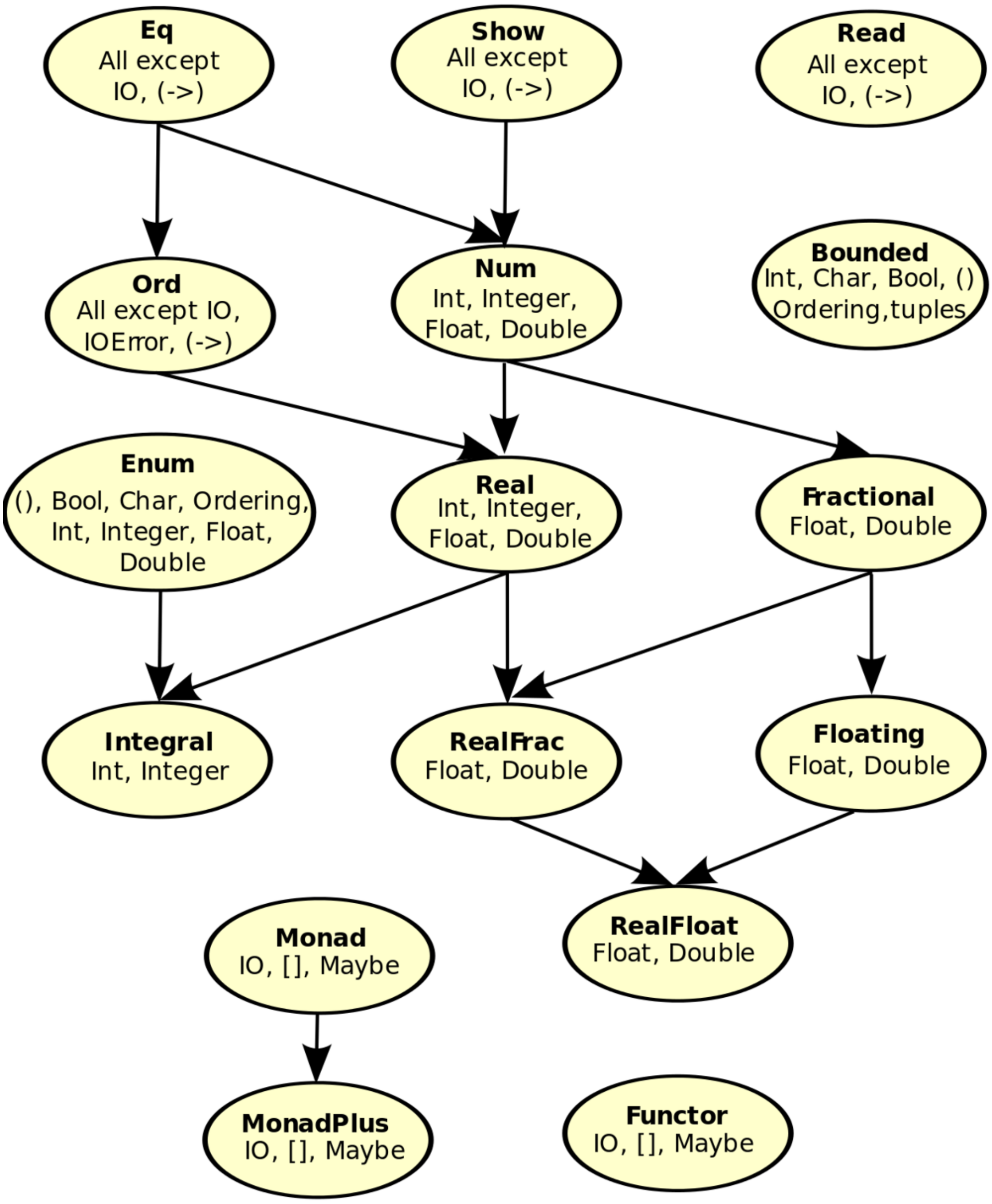


Figure 1

26.5 Type constraints

With all pieces in place, we can go full circle by returning to the very first example involving classes in this book:

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

`(Num a) =>` is a *type constraint*, which restricts the type `a` to instances of the class `Num`. In fact, `(+)` is a method of `Num`, along with quite a few other functions (notably, `(*)` and `(-)`; but not `(/)`).

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
```

Here, 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`. This example also displays clearly how constraints propagate from the functions used in a definition (in this case, `(+)` and `show`) to the function being defined.

26.5.1 Other uses

Beyond simple type signatures, type constraints can be introduced in a number of other places:

- `instance` declarations (typical with parametrized types);
- `class` declarations (constraints can be introduced in the method signatures in the usual way for any type variable other than the one defining the class[5]);
- `data` declarations[6], where they act as constraints for the constructor signatures.

[5] Constraints for the type defining the class should be set via class inheritance.

[6] And `newtype` declarations as well, but not `type`.

Note:
Type constraints in data declarations are less useful than it might seem at first. Consider:

```
data (Num a) => Foo a = F1 a | F2 a String
```

Here, Foo is a type with two constructors, both taking an argument of a type a which must be in Num. However, the (Num a) => constraint is only effective for the F1 and F2 constructors, and not for other functions involving Foo. Therefore, in the following example...

```
fooSquared :: (Num a) => Foo a -> Foo a
fooSquared (F1 x)   = F1 (x * x)
fooSquared (F2 x s) = F2 (x * x) s
```

... even though the constructors ensure a will be some type in Num we can't avoid duplicating the constraint in the signature of fooSquared[a].

[a] *Extra note for the curious*: This issue is related to some of the problems tackled by the advanced features discussed in the "Fun with types" chapter of the Advanced Track.

26.6 A concerted example

To provide a better view of how the interplay between types, classes and constraints, we will present a very simple and somewhat contrived example. In it, we define a Located class, a Movable class which inherits from it, and a function with a Movable constraint implemented using the methods of the parent class, i.e. Located.

```
-- Location, in two dimensions.
class Located a where
    getLocation :: a -> (Int, Int)

class (Located a) => Movable a where
    setLocation :: (Int, Int) -> a -> a

-- An example type, with accompanying instances.
data NamedPoint = NamedPoint
    { pointName :: String
    , pointX    :: Int
    , pointY    :: Int
    } deriving (Show)

instance Located NamedPoint where
    getLocation p = (pointX p, pointY p)

instance Movable NamedPoint where
    setLocation (x, y) p = p { pointX = x, pointY = y }

-- Moves a value of a Movable type by the specified displacement.
-- This works for any movable, including NamedPoint.
move :: (Movable a) => (Int, Int) -> a -> a
move (dx, dy) p = setLocation (x + dx, y + dy) p
    where
    (x, y) = getLocation p
```

26.6.1 A word of advice

Do not read too much into the `Movable` example just above; it is merely a demonstration of class-related language features. It would be a mistake to think that every single functionality which might be conceivably generalized, such as `setLocation`, needs a type class of its own. In particular, if all your `Located` instances should be able to be moved as well then `Movable` is unnecessary - and if there is just one instance there is no need for type classes at all! Classes are best used when there are several types instantiating it (or if you expect others to write additional instances) and you do not want users to know or care about the differences between the types. An extreme example would be `Show`: general-purpose functionality implemented by an immense number of types, about which you do not need to know a thing about before calling `show`. In the following chapters we will explore a number of important type classes in the libraries; they provide good examples of the sort of functionality which fits comfortably into a class.

27 The Functor class

This chapter serves a dual role. In it, we will not only introduce the very important `Functor` class but also use it as a simple example of how type classes can be useful tools for solving problems in a more general way.

27.1 Motivation

In Other data structures[1], we have seen how apply-to-all-elements operations (of which `map` is the prime example) are not specific to lists. One of the examples we worked through was that of the following tree datatype:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving (Show)
```

The map function we wrote for it was:

```
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)
```

Later in that chapter we also defined "maps" for `Weird` and (in a passage of the final example) `Maybe`; and could conceivably do the same for any arbitrary data structure.

The list-based `map` makes the apply-to-all-elements strategy general with regards to the function being applied[2]. Now, we can see that a further generalization is possible: mapping over types other than lists. Type classes provide a way of expressing this generality.

27.2 Introducing `Functor`

`Functor` is a Prelude class for types which can be mapped over. It has a single method, called `fmap`. The class is defined as follows:

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

The usage of the type variable `f` can look a little strange at first. Here, `f` is a parametrized data type; in the signature of `fmap`, it takes `a` as a type parameter in one of its appearances

[1] Chapter 25 on page 161

[2] Such generalization was one of the main themes of More about lists ^{Chapter13 on page 89}.

and b in the other. It becomes easier to see what is going on by considering an instance of Functor - say, Maybe. By replacing f with Maybe we get the following signature for fmap...

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

... which fits the natural definition:

```
instance  Functor Maybe  where
    fmap f Nothing    =  Nothing
    fmap f (Just x)   =  Just (f x)
```

(Incidentally, this definition is in Prelude; and thus we didn't really need to have implemented maybeMap for that example in "Other data structures".)

Unsurprisingly, the Functor instance for lists (also in Prelude) is just...

```
instance  Functor []  where
    fmap = map
```

... and replacing f for [] in the fmap signature gives us the familiar type of map.

Summing it all up, we can say that fmap is a generalization of map for any parametrized data type[3].

Naturally, we can provide Functor instances for our own data types. In particular, treeMap can be promptly relocated to an instance:

```
instance  Functor Tree  where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Branch left right) = Branch (fmap f left) (fmap f right)
```

To close this first stretch, a quick demo of fmap in action with the instances above (to reproduce it, you only need to load the data and instance declarations for Tree; the others are already in Prelude):

```
• Main> fmap (2*) [1,2,3,4] [2,4,6,8]
• Main> fmap (2*) (Just 1) Just 2
• Main> fmap (fmap (2*)) [Just 1, Just 2, Just 3, Nothing] [Just 2,Just 4,Just 6,Nothing]
• Main> fmap (2*) (Branch (Branch (Leaf 1) (Leaf 2)) (Branch (Leaf 3) (Leaf 4))) Branch (Branch
  (Leaf 2) (Leaf 4)) (Branch (Leaf 6) (Leaf 8))
```

Note:
Beyond the [] and Maybe examples mentioned here Prelude provides a number of other handy Functor instances. The full list can be found in GHC's documentation for the Control.Monad[a] module.

[a] http://hackage.haskell.org/packages/archive/base/latest/doc/html/Control-Monad.html

[3] Data structures provide the most intuitive examples; however, there are functors which cannot reasonably be seen as data structures. A commonplace metaphor consists in thinking of functors as containers; like all metaphors, however, it can be stretched only so far.

27.2.1 The functor laws

When providing a new instance of `Functor`, you should ensure it satisfies the two functor laws. There is nothing mysterious about these laws; their role is to guarantee `fmap` behaves sanely and actually performs a mapping operation (as opposed to some other nonsense). Indeed, a type whose `Functor` instance does not obey the functor laws cannot properly be called a functor[4]. The first law is:

```
fmap id  ==  id
```

`id` is the identity function, which returns its argument unaltered. The first law states that mapping `id` over a functor must return the functor unchanged.

Next, the second law:

```
fmap (f . g)  ==  fmap f . fmap g
```

It states that it should not matter whether we map a composed function or first map one function and then the other (assuming the application order remains the same in both cases).

27.3 What did we gain?

At this point, we can ask what benefit we get from the extra layer of generalization brought by the `Functor` class. There are two significant advantages:

- Starting from the more superficial one, the availability of the `fmap` method relieves us from having to recall, read and write a plethora of differently named mapping methods (`maybeMap`, `treeMap`, `weirdMap`, *ad infinitum*). As a consequence, code becomes both cleaner and easier to understand - on spotting a use of `fmap` we instantly have a general idea of what is going on[5].
- More significant, however, is the ability, granted by the type class system, to write `fmap`-based algorithms which work out of the box with *any* functor - be it `[]`, `Maybe`, `Tree` or whichever you need. Indeed, a number of useful classes in the hierarchical libraries inherit from `Functor`[6].

Type classes make it possible to create general solutions to whole categories of problems. While it is possible that, depending on what you will use Haskell for, you will not need to define new classes often, it is sure that you will be *using* type classes all the time. Many of the most powerful features and sophisticated capabilities of Haskell rely on type classes residing either in the standard libraries or elsewhere. And of course, classes will be

[4] The functor laws, and indeed the concept of a functor, are grounded on a branch of Mathematics called *Category Theory*. That need not be a concern for you at this point; in any case, we will have opportunities to explore related topics in the Advanced Track of this book.

[5] The situation is analogous to the gain in clarity provided by replacing explicit recursive algorithms on lists with implementations based on higher-order functions.

[6] *Note for the curious*: For one example, have a peek at Applicative Functors ^{Chapter44 on page 305} in the Advanced Track (for the moment you can ignore the references to monads there).

a permanent presence in this book from this point on - beginning with the all-important `Monad`.

28 Monads

29 Understanding monads

Monads are very useful in Haskell, but the concept is usually difficult for newcomers to grasp. Since they have so many applications, people often explain them from a particular point of view, which can confuse your understanding of monads in their full glory.

Historically, monads were introduced into Haskell to perform input/output. After all, lazy evaluation means that the order of evaluation is rather unpredictable, whereas a predetermined execution order is crucial for things like reading and writing files. Hence, a method for specifying a *sequence of operations* was needed, and monads were exactly the right abstraction for that.

But monads are by no means limited to input/output; they can model any imperative language. The choice of monad determines the semantics of this language, i.e., whether it supports exceptions, state, non-determinism, continuations, coroutines and so on[1].

The present chapter introduces the basic notions with the example of the `Maybe` monad, the simplest monad for handling exceptions. Beginning Haskell programmers will probably also want to understand the `IO` monad and then broaden their scope to the many other monads and the effects they represent; this chapter provides the corresponding pointers.

29.1 Definition

Let us dive in with both feet. A *monad* is defined by three things:

- a type constructor[2] `M`;
- a function **`return`**[3]; and
- an operator `(>>=)` which is pronounced "bind".

The function and operator are methods of the `Monad` type class, have types

```
return :: a -> M a
(>>=)  :: M a -> ( a -> M b ) -> M b
```

and are required to obey three laws[4] that will be explained later on.

For a concrete example, take the `Maybe` monad. The type constructor is `M = Maybe` so that **`return`** and `(>>=)` have types

[1] Indeed, thanks to the versatility of monads, in Haskell none of these constructs is a built-in part of the language; rather, they are defined by the standard libraries!

[2] Chapter 24.3 on page 158

[3] This `return` function has nothing to do with the `return` keyword found in imperative languages like C or Java; don't conflate these two.

[4] Chapter 29.3 on page 187

```
return :: a -> Maybe a
(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
```

They are implemented as

```
return x  = Just x
(>>=) m g = case m of
               Nothing -> Nothing
               Just x  -> g x
```

and our task is to explain how and why this definition is useful.

29.1.1 Motivation: Maybe

To see the usefulness of `(>>=)` and the `Maybe` monad, consider the following example: imagine a family database that provides two functions

```
father :: Person -> Maybe Person
mother :: Person -> Maybe Person
```

that look up the name of someone's father or mother, returning `Nothing` if they are not stored in the database. With them, we can query various grandparents. For instance, the following function looks up the maternal grandfather:

```
maternalGrandfather :: Person -> Maybe Person
maternalGrandfather p =
    case mother p of
        Nothing -> Nothing
        Just m  -> father m                         -- mother's father
```

Or consider a function that checks whether both grandfathers are in the database:

```
bothGrandfathers :: Person -> Maybe (Person, Person)
bothGrandfathers p =
    case father p of
        Nothing -> Nothing
        Just f  ->
            case father f of
                Nothing -> Nothing
                Just gf ->                          -- found first
grandfather
                    case mother p of
                        Nothing -> Nothing
                        Just m  ->
                            case father m of
                                Nothing -> Nothing
                                Just gm ->          -- found second one
                                    Just (gf, gm)
```

What a mouthful! Every single query might fail by returning `Nothing` and the whole function must fail with `Nothing` if that happens.

Clearly there has to be a better way to write that than repeating the case of `Nothing` again and again! Indeed, and that's what the `Maybe` monad is set out to do. For instance, the function retrieving the maternal grandfather has exactly the same structure as the `(>>=)` operator, and we can rewrite it as

```
maternalGrandfather p = mother p >>= father
```

With the help of lambda expressions and `return`, we can rewrite the mouthful of two grandfathers as well:

```
bothGrandfathers p =
   father p >>=
       (\f -> father f >>=
           (\gf -> mother p >>=
               (\m -> father m >>=
                   (\gm -> return (gf,gm) ))))
```

While these nested lambda expressions may look confusing to you, the thing to take away here is that `(>>=)` eliminates any mention of `Nothing`, shifting the focus back to the interesting part of the code.

29.1.2 Type class

In Haskell, the type class `Monad` is used to implement monads. It is provided by the Control.Monad[5] module and included in the Prelude. The class has the following methods:

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b

    (>>)   :: m a -> m b -> m b
    fail   :: String -> m a
```

Aside from return and bind, it defines two additional functions `(>>)` and `fail`.

The operator `(>>)` called "then" is a mere convenience and commonly implemented as

```
m >> n = m >>= \_ -> n
```

It is used for sequencing two monadic actions when the second does not care about the result of the first, which is common for monads like `IO`.

```
printSomethingTwice :: String -> IO ()
printSomethingTwice str = putStrLn str >> putStrLn str
```

The function `fail` handles pattern match failures in `do` notation[6]. It's an unfortunate technical necessity and doesn't really have anything to do with monads. You are advised to not call `fail` directly in your code.

29.2 Notions of Computation

While you probably agree now that `(>>=)` and `return` are very handy for removing boilerplate code that crops up when using `Maybe`, there is still the question of why this works and what this is all about.

[5] http://hackage.haskell.org/packages/archive/base/4.1.0.0/doc/html/Control-Monad.html

[6] Chapter 32 on page 199

To answer this, we shall write the example with the two grandfathers in a very suggestive style:

```
   bothGrandfathers p = do
          f  <- father p      -- assign p's father to f
          gf <- father f      -- assign f's father (p's paternal grandfather)
 to gf
          m  <- mother p      -- assign p's mother to m
          gm <- father m      -- assign m's father (p's maternal grandfather)
 to gm
          return (gf, gm)     -- return result pair
```

If this looks like a code snippet of an imperative programming language to you, that's because it is. In particular, this imperative language supports *exceptions* : `father` and `mother` are functions that might fail to produce results, i.e. raise an exception, and when that happens, the whole `do`-block will fail, i.e. terminate with an exception.

In other words, the expression `father p`, which has type `Maybe Person`, is interpreted as a statement of an imperative language that returns a `Person` as result. This is true for all monads: a value of type `M a` is interpreted as a statement of an imperative language that returns a value of type `a` as result; and the semantics of this language are determined by the monad `M`.

Now, the bind operator `(>>=)` is simply a function version of the semicolon. Just like a `let` expression can be written as a function application,

```
    let x  = foo in bar     corresponds to      (\x -> bar) foo
```

an assignment and semicolon can be written as the bind operator:

```
        x <- foo;   bar     corresponds to      foo >>= (\x -> bar)
```

The `return` function lifts a value `a` to a full-fledged statement `M a` of the imperative language.

Different semantics of the imperative language correspond to different monads. The following table shows the classic selection that every Haskell programmer should know. If the idea behind monads is still unclear to you, studying each of the examples in the following subchapters will not only give you a well-rounded toolbox but also help you understand the common abstraction behind them.

Semantics	**Monad**	**Wikibook chapter**
Exception (anonymous)	`Maybe`	Haskell/Understanding monads/Maybe[7]
Exception (with error description)	`Error`	Haskell/Understanding monads/Error[8]
Global state	`State`	Haskell/Understanding monads/State[9]

[7] Chapter 30 on page 191

[8] `http://en.wikibooks.org/wiki/Haskell%2FUnderstanding%20monads%2FError`

[9] Chapter 34 on page 209

Semantics	Monad	Wikibook chapter
Input/Output	`IO`	Haskell/Understanding monads/IO[10]
Nondeterminism	`[]` (lists)	Haskell/Understanding monads/List[11]
Environment	`Reader`	Haskell/Understanding monads/Reader[12]
Logger	`Writer`	Haskell/Understanding monads/Writer[13]

Furthermore, the semantics do not only occur in isolation but can also be mixed and matched. This gives rise to monad transformers[14].

Some monads, like monadic parser combinators[15] have loosened their correspondence to an imperative language.

29.3 Monad Laws

We can't just allow any junky implementation of `(>>=)` and `return` if we want to interpret them as the primitive building blocks of an imperative language. For that, an implementation has to obey the following three laws:

```
m >>= return     =  m                        -- right unit
return x >>= f   =  f x                      -- left unit

(m >>= f) >>= g  =  m >>= (\x -> f x >>= g)  -- associativity
```

In Haskell, every instance of the `Monad` type class is expected to obey them.

29.3.1 Return as neutral element

The behavior of `return` is specified by the left and right unit laws. They state that `return` doesn't perform any computation, it just collects values. For instance,

```
maternalGrandfather p = do
        m  <- mother p
        gm <- father m
        return gm
```

is exactly the same as

```
maternalGrandfather p = do
```

[10] Chapter 33 on page 205
[11] Chapter 31 on page 195
[12] http://en.wikibooks.org/wiki/Haskell%2FUnderstanding%20monads%2FReader
[13] http://en.wikibooks.org/wiki/Haskell%2FUnderstanding%20monads%2FWriter
[14] Chapter 37 on page 229
[15] Chapter 36 on page 227

```
        m  <- mother p
        father m
```

by virtue of the right unit law.

29.3.2 Associativity of bind

The law of associativity makes sure that – just like the semicolon – the bind operator (>>=) only cares about the order of computations, not about their nesting; e.g. we could have written bothGrandfathers like this (compare with our earliest version without do):

```
bothGrandfathers p =
   (father p >>= father) >>=
       (\gf -> (mother p >>= father) >>=
           (\gm -> return (gf,gm) ))
```

The associativity of the *then* operator (>>) is a special case:

```
(m >> n) >> o  =  m >> (n >> o)
```

It is easier to picture the associativity of bind by recasting the law as

```
(f >=> g) >=> h  =  f >=> (g >=> h)
```

where (>=>) is the monad composition operator, a close analogue of the (flipped) function composition operator (.), defined as

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

29.4 Monads and Category Theory

Monads originally come from a branch of mathematics called Category Theory. Fortunately, it is entirely unnecessary to understand category theory in order to understand and use monads in Haskell. However, the definition of monads in Category Theory uses a slightly different presentation. Translated into Haskell, this presentation gives an alternative yet equivalent definition of a monad which can give us some additional insight [16].

So far, we have defined monads in terms of >>= and return. The alternative definition, instead, starts with monads as functors with two additional combinators:

```
fmap   :: (a -> b) -> M a -> M b  -- functor

return :: a -> M a
join   :: M (M a) -> M a
```

[16] Deep into the Advanced Track, we will cover the theoretical side of the story in the chapter on Category Theory ^{Chapter57.3 on page 396}.

(Reminder: as discussed in the chapter on the functor class[17], a functor `M` can be thought of as container, so that `M a` "contains" values of type `a`, with a corresponding mapping function, i.e. `fmap`, that allows functions to be applied to values inside it.)

Under this interpretation, the functions behave as follows:

- `fmap` applies a given function to every element in a container
- `return` packages an element into a container,
- `join` takes a container of containers and flattens it into a single container.

With these functions, the bind combinator can be defined as follows:

```
m >>= g = join (fmap g m)
```

Likewise, we could give a definition of `fmap` and `join` in terms of `(>>=)` and `return`:

```
fmap f x = x >>= (return . f)
join x   = x >>= id
```

29.4.1 Is my Monad a Functor?

At this point we might, with good reason, deduce that all monads are by definition functors as well. While according to category theory that is indeed the case, GHC does it differently, and the `Monad` and `Functor` classes are unrelated. That will likely change in future versions of Haskell, so that every `Monad` instance will be guaranteed to have a matching `Functor` instance and the corresponding `fmap`. Meanwhile, `Control.Monad` defines `liftM`, a function with a strangely familiar type signature...

```
liftM :: (Monad m) => (a1 -> r) -> m a1 -> m r
```

As you might suspect, `liftM` is merely `fmap` implemented with `(>>=)` and `return`, just as we have done above. For a properly implemented monad with a matching `Functor` (that is, pretty much any *sensible* monad) `liftM` and `fmap` are interchangeable.

[17] Chapter 27 on page 177

30 The Maybe monad

We introduced monads using `Maybe` as an example. The `Maybe` monad represents computations which might "go wrong", in the sense of not returning a value; the definitions of `return` and `(>>=)` for `Monad` amounting to:[1]

```
return :: a -> Maybe a
return x  = Just x


(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) m g = case m of
              Nothing -> Nothing
              Just x  -> g x
```

Now, we will present two additional examples, similar in spirit to the grandparents one in the previous chapter, and then conclude with some general points.

30.1 Safe functions

The `Maybe` datatype provides a way to make a safety wrapper around functions which can fail to work for a range of arguments. `head` and `tail`, which only work with non-empty lists, would be a good example of that. Another typical case, which we will explore in this section, are mathematical functions like `sqrt` and `log`, which (as far as real numbers are concerned) are only defined for non-negative arguments. For instance, our implementation of a "safe" square root function could be:

```
safeSqrt :: (Floating a, Ord a) => a -> Maybe a
safeSqrt x
       | x >= 0    = Just (sqrt x)
       | otherwise = Nothing
```

We could now decide to write similar "safe functions" for all functions with limited domains, such as division, logarithm and inverse trigonometric functions (`safeDiv`, `safeLog`, `safeArcSin`, etc.). However, when we try to combine them we run into a problem: they output a `Maybe a`, but take an `a` as an input. As a result, before each application, we have to check whether the previous operation was successful:

```
safeLogSqrt :: (Floating a, Ord a) => a -> Maybe a
safeLogSqrt x = case safeSqrt x of
                  Just root -> safeLog root
                  Nothing   -> Nothing
```

[1] The definitions in the actual instance in `Data.Maybe` are written a little differently, but are fully equivalent to these.

You can see the problem: the code looks ugly already when using one concatenation. If you had multiple concatenations the code would get even worse. This is in stark contrast to the easy concatenation that we get with "unsafe" functions:

```
unsafeLogSqrt = log . sqrt
```

This, however, is precisely the sort of issue monads can tackle: through (>>=), they allow us to concatenate computations easily, so that the result of one is fed to the next. Maybe being a monad, we can achieve the desired effect very simply:

```
> return 1000 >>= safeSqrt >>= safeLog
Just 3.4538776394910684
> return (-1000) >>= safeSqrt >>= safeLog
Nothing
```

```
safeLogSqrt :: (Floating a, Ord a) => a -> Maybe a
safeLogSqrt x = return x >>= safeSqrt >>= safeLog
```

Every additional operation is just another >>= safeOperation term---no checks required, since (>>=) takes care of that. To make things even easier, we can write the combination in a style which mirrors composition of "unsafe" functions with (.) by using the (<=<) operator from the Control.Monad module:

```
import Control.Monad((<=<))
safeLogSqrt' = safeLog <=< safeSqrt
```

30.2 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 phone book application. An elementary 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[2]. Here's how the phone book lookup table might look like:

```
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 is fine if we try to look up one of "Bob", "Fred", "Alice" or "Jane" in our phone book, but what if we were to look up "Zoe"? Zoe isn't in our phone book, so the lookup would fail. Hence, the Haskell function to look up a value from the table is a Maybe computation (it is available from Prelude):

[2] Check the chapter about maps ^{Chapter72 on page 489} in Haskell in Practice for a different, and potentially more useful, implementation.

```
lookup :: Eq a => a  -- a key
       -> [(a, b)]   -- the lookup table to use
       -> Maybe b    -- the result of the lookup
```

Let us 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 phone book, 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
```

As you might have guessed by now, `comb` is just `(>>=)`; and 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 -> lookup registration taxDatabase)
```

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.*

30.3 Summary

The key features of the `Maybe` monad are that:

1. It represents computations that could fail.
2. It propagates failure.

Another trait of the `Maybe` monad is that it is "open": if we have a `Just` value, we can extract its associated value by pattern matching (which is what we did in the first implementation of `safeLogSqrt`). That is is not true for all monads; often, they will be designed so as to help you by hiding unnecessary details. It is perfectly possible to make a "no-exit" monad, from which it is never possible to extract "pure" values, the obvious example being the `IO` monad: in this way it is impossible not to notice that an operation involves input/output (and thereby side effects), because any I/O operation will carry around the `IO` monad, which functions as a warning sign.

31 The List monad

Lists are commonplace in Haskell, and you surely have used them extensively before getting to this chapter. What is novel here is that the list type, `[] a`, is a monad too. Taken as monads, lists are used to model *nondeterministic* computations which may return an arbitrary number of results. There is a certain parallel with how `Maybe` represented computations which could return zero or one value; the difference being that through lists zero, one or many values can be returned (the number of values being reflected in the length of the list).

31.1 Instantiation as monad

The implementation of the `return` function, that injects a generic value into a list, is quite simple:

```
return x = [x]
```

In other words, injecting a value into a list means making a list containing that value only. The type of the list `return` is `return :: a -> [a]`, or, equivalently, `return :: a -> [] a`. The latter style of writing it makes it more obvious that we are replacing the generic type constructor in the signature of `return` (which we had called `M` in Understanding monads[1]) by the list type constructor `[]` (do not confuse the latter with the empty list!).

The binding operator is a little less trivial. We will begin by considering its type, which for the case of lists should be:

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

That is, from a list of `a`-type values and a function producing lists of `b`-type values from each `a`-type one, we get a list of `b`'s. Now, `mapping` the function over the list of `a`'s would give `[[b]]`, a list of lists of `b`'s. By using `concat` to concatenate the elements of this list of lists we get a list of `b`'s - and a definition of `(>>=)`:

```
xs >>= f = concat (map f xs)
```

The bind operator is always key to understanding how a monad does its job, for it implements the chaining strategy which is responsible for making the monad useful. In the case of the list monad, the type of the function to the right of `(>>=)` brings non-determinism into play, as evaluating to a list for each value corresponds in effect to giving back a variable number of results for each value passed to the function. `map` in turn applies `f` to all values from the `xs` computation, leaving us with potentially many results for each element of `xs`.

[1] Chapter 29 on page 183

Finally, `concat` combines all of these results in a simple list of type `[b]`, thus ensuring that the type of the final computation matches the signature of `(>>=)` and therefore that we can proceed with chaining it to other list computations.

31.2 Bunny invasion

To begin with, a simple example showing that it is easy to incorporate the familiar list processing functions in monadic code. A rabbit couple can spawn six rabbits every month. Considering half of them will be females, we can model the number of female rabbits after a certain number of generations using a function like:

```
Prelude> let generation = replicate 3
Prelude> ["bunny"] >>= generation
["bunny","bunny","bunny"]
Prelude> ["bunny"] >>= generation >>= generation
["bunny","bunny","bunny","bunny","bunny","bunny","bunny","bunny","bunny"]
```

In this trivial example all elements are equal, but one could for example model radioactive decay[2] or chemical reactions, or any phenomena that produces a series of elements starting from a single one.

31.3 Noughts and crosses

Suppose 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). 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 monad instance for list described above. Here's how it might look, without using the list monad (`concatMap` is a shortcut for when you need to concat the results of a map: `concatMap f xs = concat (map f xs)`):

```
nextConfigs :: Board -> [Board]
nextConfigs = undefined -- details not important

tick :: [Board] -> [Board]
tick bds = concatMap nextConfigs bds
```

[2] http://en.wikipedia.org/wiki/Decay_chain

```
thirdConfigs :: Board -> [Board]
thirdConfigs bd = tick $ tick $ tick [bd]
```

`concatMap`, however, is just the bind operator for lists, only with the arguments in reversed order; and so we can write a literal translation of `thirdConfigs` using the list monad:

```
thirdConfigs :: Board -> [Board]
thirdConfigs bd = return bd >>= nextConfigs >>= nextConfigs >>= nextConfigs
```

Alternatively, we can write the above as a `do` block - compare the translation with what we did for the grandparents example in Understanding monads[3]:

```
thirdConfigs :: Board -> [Board]
thirdConfigs bd = do
  bd0 <- return bd       -- Initial configuration
  bd1 <- nextConfigs bd0 -- After one turn
  bd2 <- nextConfigs bd1 -- After two turns
  nextConfigs bd2        -- After three turns
```

Note how the left arrow in the list monad, in effect, means "do whatever follows with all elements of the list on the right of the arrow". That effect is due to the mapping performed by the bind operator.

We can simplify the monadic code a bit further by noting that using `return bd` to get a list with a single element and then immediately extracting that single element with the left arrow is redundant:

```
thirdConfigs :: Board -> [Board]
thirdConfigs bd = do
  bd1 <- nextConfigs bd
  bd2 <- nextConfigs bd1
  nextConfigs bd2
```

Short and sweet, with the plumbing formerly done by the `tick` function now wholly implicit.

31.4 List comprehensions

One thing that can be helpful in visualizing how the list monad works is its uncanny similarity to list comprehensions. If we slightly modify the `do` block we just wrote for `thirdConfigs` so that it ends with a `return`...

```
thirdConfigs bd = do
  bd1 <- nextConfigs bd
  bd2 <- nextConfigs bd1
  bd3 <- nextConfigs bd2
  return bd3
```

... it mirrors exactly the following list comprehension:

```
thirdConfigs bd = [ bd3 | bd1 <- nextConfigs bd, bd2 <- nextConfigs bd1, bd3 <-
 nextConfigs bd2 ]
```

[3] Chapter 29 on page 183

(In a list comprehension, it is perfectly legal to use the elements drawn from one list to define the following ones, like we did here.)

The resemblance is no coincidence: list comprehensions are, behind the scenes, also defined in terms of `concatMap`. For the correspondence to be complete, however, there should be a way to reproduce the filtering that list comprehensions are capable of. We will explain how that can be achieved a little later, in the Additive monads[4] chapter.

[4] Chapter 35 on page 221

32 do Notation

Among the initial examples of monads, there were some which used an alternative syntax with `do` blocks for chaining computations. Those examples, however, were not the first time we have seen `do`: back in Simple input and output[1] we had seen how code for doing input-output was written in an identical way. That is no coincidence: what we have been calling IO actions are just *computations in a monad* - namely, the `IO` monad. We will revisit `IO` soon; for now, though, let us consider exactly how the `do` notation translates into regular monadic code. Since the following examples all involve `IO`, we will refer to the computations/monadic values as *actions*, like in the earlier parts of the book. Still `do` works with any monad; there is nothing specific about `IO` in how it works.

32.1 Translating the *then* operator

The `(>>)` (*then*) operator is easy to translate between `do` notation and plain code, so we will see it first. For example, suppose we have a chain of actions like the following one:

```
putStr "Hello" >>
putStr " " >>
putStr "world!" >>
putStr "\n"
```

We can rewrite it in `do` notation as follows:

```
do putStr "Hello"
   putStr " "
   putStr "world!"
   putStr "\n"
```

This sequence of instructions is very similar to what you would see in any imperative language such as C. The actions being chained could be anything, as long as all of them are in the same monad. In the context of the `IO` monad, for instance, an action might be writing to a file, opening a network connection or asking the user for input. The general way we translate these actions from the `do` notation to standard Haskell code is:

```
do action1
   action2
   action3
```

which becomes

```
action1 >>
```

[1] Chapter 10 on page 69

```
do action2
   action3
```

and so on until the `do` block is empty.

32.2 Translating the *bind* operator

The `(>>=)` is a bit more difficult to translate from and to `do` notation, essentially because it involves passing a value, namely the result of an action, downstream in the binding sequence. These values can be stored using the `<-` notation, and used downstream in the `do` block.

```
do x1 <- action1
   x2 <- action2
   action3 x1 x2
```

`x1` and `x2` are the results of `action1` and `action2` (for instance, if `action1` is an `IO Integer` then `x1` will be bound to an `Integer`). They are passed as arguments to `action3`, whose return type is a third action. The `do` block is broadly equivalent to the following vanilla Haskell snippet:

```
action1 >>= \x1 -> action2 >>= \x2 -> action3 x1 x2
```

The second argument of `(>>=)` is a function specifying what to do with the result of the action passed as first argument; and so by chaining lambdas in this way we can pass results downstream. Remember that without extra parentheses a lambda extends all the way to the end of the expression, so `x1` is still in scope at the point we call `action3`. We can rewrite the chain of lambdas more legibly as:

```
action1 >>= \x1 ->
action2 >>= \x2 ->
action3 x1 x2
```

32.2.1 The *fail* method

Above we said the snippet with lambdas was "broadly equivalent" to the `do` block. It is not an exact translation because the `do` notation adds special handling of pattern match failures. `x1` and `x2` when placed at the left of either `<-` or `->` are patterns being matched. Therefore, if `action1` returned a `Maybe Integer` we could write a `do` block like this...

```
do Just x1 <- action1
   x2      <- action2
   action3 x1 x2
```

...and `x1` will be bound to an `Integer`. In such a case, however, what happens if `action1` returns `Nothing`? Ordinarily, the program would crash with an non-exhaustive patterns error, just like the one we get when calling `head` on an empty list. With `do` notation, however, failures will be handled with the `fail` method for the relevant monad. The translation of the first statement done behind the scenes is equivalent to:

```
action1 >>= f
where f (Just x1) = do x2 <- action2
                       action3 x1 x2
      f _         = fail "..." -- A compiler-generated message.
```

What `fail` actually does is up to the monad instance. While it will often just rethrow the pattern matching error, monads which incorporate some sort of error handling may deal with the failure in their own specific ways. For instance, `Maybe` has `fail _ = Nothing`; analogously, for the list monad `fail _ = []` [2].

All things considered, the `fail` method is an artefact of `do` notation. It is better not to call it directly from your code, and to only rely on automatic handling of pattern match failures when you are sure that `fail` will do something sensible for the monad you are using.

32.3 Example: user-interactive program

Note:
We are going to interact with the user, so we will use `putStr` and `getLine` alternately. To avoid unexpected results in the output remember to disable output buffering importing `System.IO` and putting `hSetBuffering stdout NoBuffering` at the top of your code. Otherwise you can explictly flush the output buffer before each interaction with the user (namely a `getLine`) using `hFlush stdout`. If you are testing this code with ghci you don't have such problems.

Consider this simple program that asks the user for his or her first and last names:

```
nameDo :: IO ()
nameDo = do putStr "What is your first name? "
            first <- getLine
            putStr "And your last name? "
            last <- getLine
            let full = first ++ " " ++ last
            putStrLn ("Pleased to meet you, " ++ full ++ "!")
```

The code in `do` notation is readable and easy to follow. The `<-` notation makes it possible to treat first and last names *as if* they were pure variables, though they never can be in reality: function `getLine` is not pure because it can give a different result every time it is run.

A possible translation into vanilla monadic code would be:

```
nameLambda :: IO ()
nameLambda = putStr "What is your first name? " >>
             getLine >>= \first ->
             putStr "And your last name? " >>
             getLine >>= \last ->
             let full = first ++ " " ++ last
             in putStrLn ("Pleased to meet you, " ++ full ++ "!")
```

[2] That explains why, as we pointed out in the "Pattern matching" chapter ^{Chapter16.5.3 on page 119}, pattern matching failures in list comprehensions are silently ignored.

In cases like this, in which we just want to chain several actions, the imperative style of do notation feels natural, and can be pretty convenient. In comparison, monadic code with explicit binds and lambdas is something of an acquired taste.

The example includes a let statement in the do block. They are translated to a regular let expression, with the in part being the translation of whatever follows it in the do block (in the example, that means the final putStrLn).

32.4 Returning values

The last statement in a do notation is the result of the do block. In the previous example, the result was of the type IO (), that is an empty value in the IO monad.

Suppose that we want to rewrite the example, but returning a IO String with the acquired name. All we need to do is add a return:

```
nameReturn :: IO String
nameReturn = do putStr "What is your first name? "
                first <- getLine
                putStr "And your last name? "
                last <- getLine
                let full = first ++ " " ++ last
                putStrLn ("Pleased to meet you, " ++ full ++ "!")
                return full
```

This example will "return" the full name as a string inside the IO monad, which can then be utilized downstream elsewhere:

```
greetAndSeeYou :: IO ()
greetAndSeeYou = do name <- nameReturn
                    putStrLn ("See you, " ++ name ++ "!")
```

Here, the name will be obtained from user input and the greeting will be printed as *side effects* of nameReturn. Its return value will then be used to prepare the goodbye message.

This kind of code is why it is so easy to misunderstand the nature of return: it does not only share a name with C's keyword, it *seems* to have the same function here. A small variation on the example, however, will dispel that impression:

```
nameReturnAndCarryOn = do putStr "What is your first name? "
                          first <- getLine
                          putStr "And your last name? "
                          last <- getLine
                          let full = first++" "++last
                          putStrLn ("Pleased to meet you, "++full++"!")
                          return full
                          putStrLn "I am not finished yet!"
```

The string in the extra line *will* be printed out, as return is not a final statement interrupting the flow like in C and other languages. Indeed, the type of nameReturnAndCarryOn is IO (), the type of the final putStrLn action, and after the function is called the IO String created by the return full will disappear without a trace.

32.5 It is just sugar

The do notation is just a syntactical convenience; it does not add anything essential. Keeping that in mind, we can raise a few points about style. First of all, do is never necessary for a single action; and so the Haskell "Hello world" is simply...

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

...without any do in sight. Additionally, snippets like this one are always redundant:

```
fooRedundant = do x <- bar
                  return x
```

Thanks to the monad laws[3], we can (and should!) write it as:

```
foo = bar
```

A subtler but crucial point is related to function composition. As we already know, the greetAndSeeYou action in the section just above could be rewritten as:

```
greetAndSeeYou :: IO ()
greetAndSeeYou = nameReturn >>= \name -> putStrLn ("See you, " ++ name ++ "!")
```

While you might find the lambda a little unsightly, suppose we had a printSeeYou function defined elsewhere:

```
printSeeYou :: String -> IO ()
printSeeYou name = putStrLn ("See you, " ++ name ++ "!")
```

Things suddenly look much nicer, and arguably even nicer than in the do version:

```
greetAndSeeYou :: IO ()
greetAndSeeYou = nameReturn >>= printSeeYou
```

Or, if we had a *non-monadic* seeYou function:

```
seeYou :: String -> String
seeYou name = "See you, " ++ name ++ "!"

-- Reminder: liftM f m == m >>= return . f == fmap f m
greetAndSeeYou :: IO ()
greetAndSeeYou = liftM seeYou nameReturn >>= putStrLn
```

Keep in mind this last example with liftM; we will soon return to the theme of using non-monadic functions in monadic code, and why it can be useful.

[3] http://en.wikibooks.org/wiki/Haskell%2FMonads%23Monad_Laws

33 The IO monad

As you should have picked up by now, Haskell is a *functional* and *lazy* language. This has some dire consequences for something apparently simple like input/output, but we can solve this with the `IO` monad.

33.1 The Problem: Input/Output and Purity

Haskell functions are in general *pure* functions: when given the same arguments, they return the same results. The reason for this paradigm is that pure functions are much easier to debug and to prove correct. Test cases can also be set up much more easily, since we can be sure that nothing other than the arguments will influence a function's result. We also require pure functions not to have *side effects* other than returning a value: a pure function must be self-contained, and cannot open a network connection, write a file or do anything other than producing its result. This allows the Haskell compiler to optimise the code very aggressively.

However, there are very useful functions that *cannot* be pure: an input function, say `getLine`, will return different results every time it is called; indeed, that's the point of an input function, since an input function returning always the same result would be pointless. Output operations have side effects, such as creating files or printing strings on the terminal: this is also a violation of purity, because the function is no longer self-contained.

Unwilling to drop the purity of standard functions, but unable to do without impure ones, Haskell places the latter ones in the `IO` monad. In other words, what we up to now have called "IO actions" are just values in the `IO` monad.

The `IO` monad is built in such a way as to be "closed", that is, it is not possible to make a `String` out of a `IO String`. Such an extraction would be possible for other monads, such as `Maybe String` or `[String]`, but not for `IO String`. In this way, any operation involving an impure function will be "tainted" with the `IO` monad, which then functions as a signal: if the `IO` monad is present in a function's signature, we know that that function may have side effects or may produce different results with the same inputs.

Another advantage of using monads is that, by concatenating I/O operations with the `(>>=)` or `(>>)` operators, we provide an order in which these I/O operations will be executed. This is important because Haskell is a lazy language, and can decide to evaluate functions whenever the compiler decides it is appropriate: however, this can work only for pure functions! If an operation with side effects (say, writing a log file) were to be written lazily, its entries could be in just about any order: clearly not the result we desire. Locking I/O operations inside a monad allows to define a clear operating sequence.

33.2 Combining Pure Functions and Input/Output

If all useful operations entail input/output, why do we bother with pure functions? The reason is that, thanks to monad properties, we can still have pure functions doing the heavy work, and benefit from the ease with which they can be debugged, tested, proved correct and optimised, while we use the `IO` monad to get our data and deliver our results.

Let's try a simple example: suppose we have a function converting a string to upper case:

```
> let shout = map Data.Char.toUpper
```

The type of this function is clearly pure:

```
> :t shout
shout :: [Char] -> [Char]
```

Suppose you apply this function to a string with many repeated characters, for example:

```
> shout "aaaaaaaaaaaaagh!"
"AAAAAAAAAAAAAGH!"
```

The Haskell compiler needs to apply the function only four times: for 'a', 'g', 'h' and '!'. A C compiler or a Perl interpreter, knowing nothing about purity or self-containment, would have to call the function for all 16 characters, since they cannot be sure that the function will not change its output somehow. The `shout` function is really trivial, but remember that this line of reasoning is valid for any pure function, and this optimisation capability will be extremely valuable for more complex operations: suppose, for instance, that you had a function to render a character in a particular font, which is a much more expensive operation.

To combine `shout` with I/O, we ask the user to insert a string (side effect: we are writing to screen), we read it (impure: result can be different every time), apply the (pure) function with `liftM` and, finally, write the resulting string (again, side effect).

```
> putStr "Write your string: " >> liftM shout getLine >>= putStrLn
Write your string: This is my string!
THIS IS MY STRING!
```

33.3 IO facts

- The `do` notation[1] is especially popular with the `IO` monad, since it is not possible to extract values from it, and at the same time it is very convenient to use statements with

[1] Chapter 32 on page 199

<- to store input values that have to be used multiple times after having been acquired. The above example could be written in do notation as:

```
do putStr "Write your string: "
   string <- getLine
   putStrLn (shout string)
```

- Every Haskell program starts from the main function, which has type IO (). Therefore, in reality, every line of Haskell code you have ever run has run in conjunction with the IO monad!
- A way of viewing the IO monad is thinking of an IO a value as a computation which gives a value of type a while changing *the state of the world* by doing input and output. This state of the world is hidden from you by the IO monad, and obviously you cannot set it. Seen this way, IO is roughly analogous to the State monad, which we will meet shortly. With State, however, the state being changed is made of normal Haskell values, and so we can manipulate it directly with pure functions.
- Actually, there is a "back door" out of the IO monad, System.IO.Unsafe.unsafePerformIO, which will transform, say, a IO String into a String. The naming of the function should point out clearly enough that it is usually *not* a good idea to use it.
- Naturally, the standard library has many useful functions for performing I/O, all of them involving the IO monad. Several important ones are presented in the IO chapter in Haskell in Practice[2].

33.4 Monadic control structures

Given that monads allow us to express sequential execution of actions in a wholly general way, we would hope to use them to implement common iterative patterns, such as loops, in an elegant manner. In this section, we will present a few of the functions from the standard libraries which allow us to do precisely that. While the examples are being presented amidst a discussion on IO for more immediate appeal, keep in mind that what we demonstrate here applies to *every* monad.

We begin by emphasizing that there is nothing magical about monadic values; we can manipulate them just like any other values in Haskell. Knowing that, we might try to, for instance, use the following to get five lines of user input:

```
fiveGetLines = replicate 5 getLine
```

That won't do, however (try it in GHCi!). The problem is that replicate produces, in this case, a list of actions, while we want an action which returns a list (that is, IO [String] rather than [IO String]). What we need is a fold which runs down the list of actions, executing them and combining the results into a single list. As it happens, there is a Prelude function which does that: sequence.

```
sequence :: (Monad m) => [m a] -> m [a]
```

[2] Chapter 73 on page 493

And so, we get the desired action.

```
fiveGetLines = sequence $ replicate 5 getLine
```

`replicate` and `sequence` form an appealing combination; and so Control.Monad[3] offers a `replicateM` function for repeating an action an arbitrary number of times. `Control.Monad` provides a number of other convenience functions in the same spirit - monadic zips, folds, and so forth.

```
fiveGetLinesAlt = replicateM 5 getLine
```

A particularly important combination is that of `map` and `sequence`; it allows us to make actions from a list of values, run them sequentially and collect the results. `mapM`, a Prelude function, captures this pattern:

```
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
```

Also common are the variants of the above functions with a trailing underscore in the name, such as `sequence_`, `mapM_` and `replicateM_`. They discard the return values, and so are appropriate when you are only interested in the side effects (they are to their underscore-less counterparts what `(>>)` is to `(>>=)`). `mapM_` for instance has the following type:

```
mapM_ :: (Monad m) => (a -> m b) -> [a] -> m ()
```

Finally, it is worth mentioning that `Control.Monad` also provides `forM` and `forM_`, which are flipped versions of `mapM` and `mapM_`. `forM_` happens to be the idiomatic Haskell counterpart to the imperative for-each loop; and the type signature suggests that neatly:

```
forM_ :: (Monad m) => [a] -> (a -> m b) -> m ()
```

Exercises:

1. Using the monadic functions we have just introduced, write a function which prints an arbitrary list of values.
2. Generalize the bunny invasion example[a] in the list monad chapter for an arbitrary number of generations.
3. What is the expected behaviour of `sequence` for the `Maybe` monad?

[a] Chapter 31.2 on page 196

[3] http://hackage.haskell.org/packages/archive/base/4.1.0.0/doc/html/Control-Monad.html

34 The State monad

If you programmed in any language before, chances are you wrote some functions that "kept state". In case you did not encounter the concept before, a *state* is one or more variables that are required to perform some computation, but are not among the arguments of the relevant function. In fact, object-oriented languages like C++ make extensive usage of state variables in objects in the form of member variables. Procedural languages like C use variables declared outside the current scope to keep track of state. In Haskell, however, such techniques can not be applied in a straightforward way, as they require mutable variables and thus clash with the default expectation of functional purity. We can very often keep track of state by passing parameters from function to function or by pattern matching of various sorts, but in some cases it is appropriate to find a more general or convenient solution. We will consider the common example of generation of pseudo-random numbers with pure functions, and find out how the `State` monad can make such a task easier.

34.1 Pseudo-Random Numbers

Generating actual random numbers[1] is a very complicated subject; we will consider *pseudo*-random numbers[2]. They are called "pseudo" because they are not really random, they only look like it. Starting from an initial state (commonly called the *seed*), pseudo-random number generators produce a sequence of numbers that have the appearance of being random.

Every time a pseudo-random number is requested, a global state is updated: that's the part we have problems with in Haskell, since it is a side effect from the point of view of the function requesting the number. Sequences of pseudo-random numbers can be replicated exactly if the initial seed and the algorithm is known.

34.1.1 Implementation in Haskell

Producing a pseudo-random number in most programming languages is very simple: there is usually a function, such as C or C++'s `rand()`, that provides a pseudo-random value (or a random one, depending on the implementation). Haskell has a similar one in the `System.Random` module:

```
> :m System.Random
> :t randomIO
randomIO :: Random a => IO a
> randomIO
-1557093684
```

[1] http://en.wikibooks.org/wiki/%3Awikipedia%3ARandom%20number%20generation

[2] http://en.wikibooks.org/wiki/%3Awikipedia%3APseudorandom%20number%20generator

Obviously, save eerie coincidences, the value you will obtain will be different. A disadvantage of `randomIO` is that it requires us to utilise the `IO` monad, which breaks purity requirements. Usage of the `IO` monad is dictated by the process of updating the global generator state, so that the next time we call `randomIO` the value will be different.

34.1.2 Implementation with Functional Purity

In general, we do not want to use the `IO` monad if we can help it, because of the loss of guarantees on no side effects and functional purity. Indeed, we can build a *local* generator (as opposed to the global generator, invisible to us, that `randomIO` uses) using `mkStdGen`, and use it as seed for the `random` function, which in turn returns a tuple with the pseudo-random number that we want and the generator to use the next time:

```
> :m System.Random
> let generator = mkStdGen 0           -- "0" is our seed
> generator
1 1
> random generator :: (Int, StdGen)
(2092838931,1601120196 1655838864)
```

While we have now regained functional purity, there are new problems to bother us. First and foremost, if we want to use `generator` to get random numbers, the obvious definition...

```
> let randInt = fst . random $ generator :: Int
> randInt
2092838931
```

...is useless, as it will always give back the same value, `2092838931`, no matter how many times `random` is called, for the same generator is always used. Of course we can take the second member of the tuple (i.e. the new generator) and feed it to a new call to `random`:

```
> let (randInt, generator') = random generator :: (Int, StdGen)
> randInt                          -- Same value
2092838931
> random generator' :: (Int, StdGen) -- Using new generator' returned from
 “random generator”
(-2143208520,439883729 1872071452)
```

That, however, keeps us from having a function which simply gives back a new value, without the fuss of having to pass the generator around. What we really need is a way to automate the extraction of the second member of the tuple (i.e. the new generator) and feed it to a new call to `random`; and that is where the `State` monad comes into the picture.

34.2 Definition of the State Monad

Note:
In this chapter we will use the state monad provided by the module `Control.Monad.Trans.State` of the `transformers` package. By reading Haskell code in the wild you will soon meet `Control.Monad.State`, a module of the closely related `mtl` package. The differences between these two modules need not concern us at the moment; everything we discuss here also applies to the `mtl` variant.

The Haskell type `State` is in essence a *function* that consumes state, and produces a result and the state after the result has been extracted. The function is wrapped by a data type definition, which comes along with a `runState` accessor so that pattern matching becomes unnecessary. For our current purposes, the definition is equivalent to:

```
newtype State s a = State { runState :: s -> (a, s) }
```

Here, `s` is the type of the state, and `a` the type of the produced result. The name `State` for the type is arguably a bit of a misnomer, as the wrapped value is not the state itself but a *state processor*.

34.2.1 Before You Ask...

What in the world did we mean by "for our current purposes" two paragraphs above? The subtlety is that the `transformers` package implements the `State` type in a somewhat different way. The differences do not affect how we use or understand `State`; as a consequence of them, however, `Control.Monad.Trans.State` does not export a `State` constructor. Rather, there is a `state` function,

```
state :: (s -> (a, s)) -> State s a
```

which does the same job. As for *why* the implementation is not the obvious one we presented above, we will get back to that a few chapters down the road.

34.2.2 newtype

You will also have noticed that we defined the data type with the `newtype` keyword, rather than the usual `data`. `newtype` can be used only for types with just one constructor and just one field; it ensures the trivial wrapping and unwrapping of the single field is eliminated by the compiler. For that reason, simple wrapper types such as `State` are usually defined with `newtype`. One might ask whether defining a synonym with `type` would be enough in such cases. `type`, however, does not allow us to define instances for the new data type, which is what we are about to do...

34.2.3 Instantiating the Monad

Note also that, in contrast to the monads we have met thus far, `State` has *two* type parameters. This means that, when we instantiate the monad, we are actually leaving the parameter for the state type:

```
instance Monad (State s) where
```

This means that the "real" monad could be `State String`, `State Int`, or `State SomeLargeDataStructure`, but not `State` on its own.

The `return` function is implemented as:

```
return :: a -> State s a
return x = state ( \st -> (x, st) )
```

In words, giving a value to `return` produces a function, wrapped in the `State` constructor: this function takes a state value, and returns it unchanged as the second member of a tuple, together with the specified result value.

Binding is a bit intricate:

```
(>>=) :: State s a -> (a -> State s b) -> State s b
processor >>= processorGenerator = state $ \st ->
                                   let (x, st') = runState processor st
                                   in runState (processorGenerator x) st'
```

The idea is that, given a state processor and a function that can generate another processor given the result of the first one, these two processors are combined to obtain a function that takes the *initial* state, and returns the *second* result and state (i.e. after the second function has processed them).

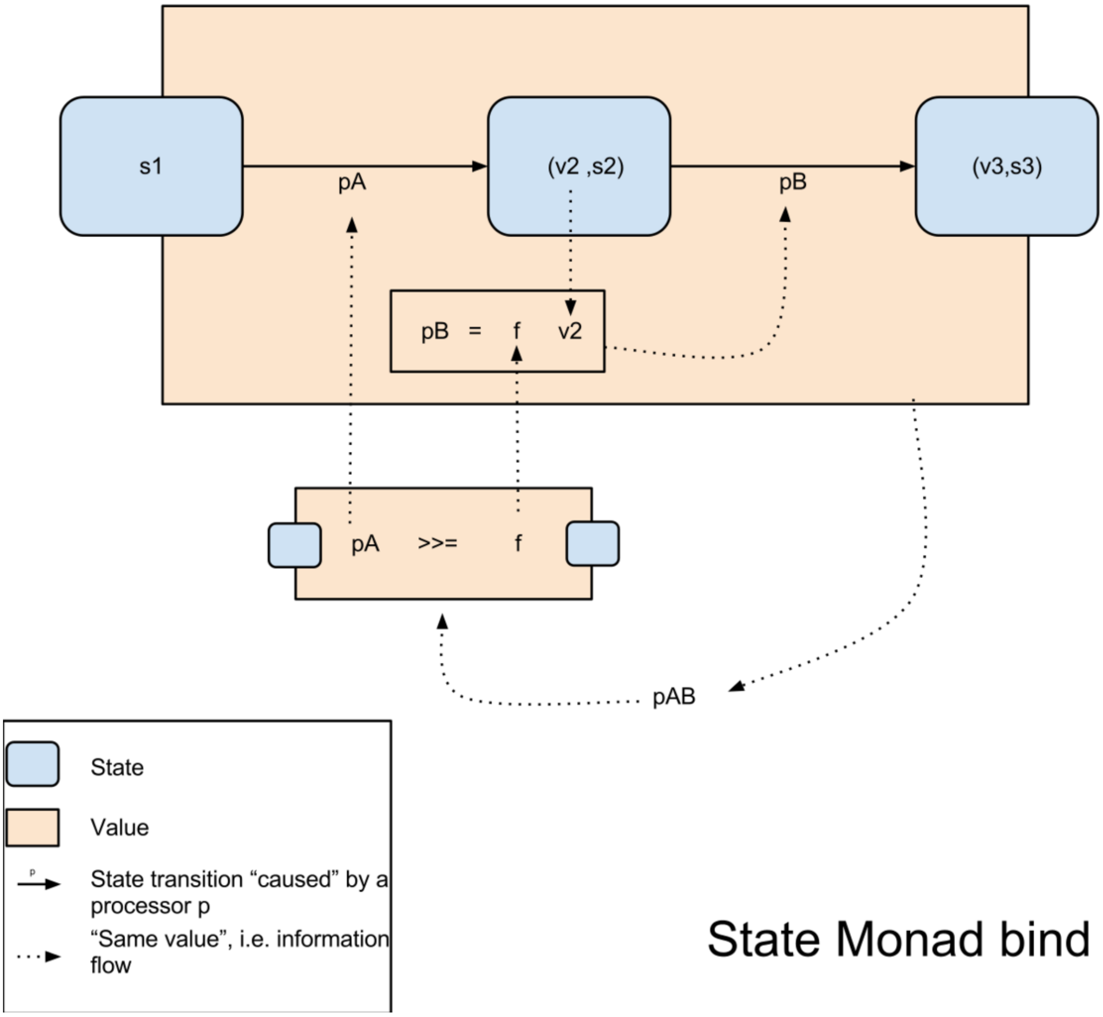

s1
pA
(v2 ,s2)
pB
(v3,s3)
pB = f v2
pA >>= f
pAB
State
Value
State transition “caused” by a processor p
p
“Same value”, i.e. information flow
State Monad bind

Figure 2 Loose schematic representation of how bind creates a new state processor (pAB) from the given state processor (pA) and the given generator (f). s1, s2 and s3 are actual states. v2 and v3 are values. pA, pB and pAB are state processors. The diagram ignores wrapping and unwrapping of the functions in the State wrapper.

The diagram shows this schematically, for a slightly different, but equivalent form of the ">>=" (bind) function, given below (where wpA and wpAB are wrapped versions of pA and pAB).

```
-- pAB = s1 --> pA --> (v2,s2) --> pB --> (v3,s3)
wpA >>= f = wpAB
    where wpAB = state $ \s1 -> let pA = runState wpA
                                    (v2, s2) = pA s1
                                    pB = runState $ f v2
                                    (v3,s3) = pB s2
                                in  (v3,s3)
```

34.2.4 Setting and Accessing the State

The monad instantiation allows us to manipulate various state processors, but you may at this point wonder where exactly the *original* state comes from in the first place. `State s` is also an instance of the `MonadState` class, which provides two additional functions:

```
put newState = state $ \_ -> ((), newState)
```

This function will generate a state processor given a state. The processor's input will be disregarded, and the output will be a tuple carrying the state we provided. Since we do not care about the result (we are discarding the input, after all), the first element of the tuple will be, so to say, "null".[3]

The specular operation is to read the state. This is accomplished by `get`:

```
get = state $ \st -> (st, st)
```

The resulting state processor is going to produce the input `st` in both positions of the output tuple - that is, both as a result and as a state, so that it may be bound to other processors.

34.2.5 Getting Values and State

From the definition of `State`, we know that `runState` is an accessor to apply to a `State a b` value to get the state-processing function; this function, given an initial state, will return the extracted value and the new state. Other similar, useful functions are `evalState` and `execState`, which work in a very similar fashion.

Function `evalState`, given a `State a b` and an initial state, will return the extracted value only, whereas `execState` will return only the new state; it is possibly easiest to remember them as defined as:

```
evalState :: State s a -> s -> a
evalState processor st = fst ( runState processor st )

execState :: State s a -> s -> s
execState processor st = snd ( runState processor st )
```

[3] The technical term for the type of `()` is *unit*.

34.3 Example: Rolling Dice

Figure 3 `randomRIO (1,6)`

Suppose we are coding a game in which at some point we need an element of chance. In real-life games that is often obtained by means of dice, which we will now try to simulate with Haskell code. For starters, we will consider the result of throwing two dice: to do that, we resort to the function `randomR`, which allows to specify an interval from which the pseudo-random values will be taken; in the case of a die, it is `randomR (1,6)`.

In case we are willing to use the `IO` monad, the implementation is quite simple, using the `IO` version of `randomR`:

```haskell
import Control.Monad
import System.Random

rollDiceIO :: IO (Int, Int)
rollDiceIO = liftM2 (,) (randomRIO (1,6)) (randomRIO (1,6))
```

Here, `liftM2` is used to make the non-monadic two-argument function `(,)` work within a monad, so that the two numbers will be returned (in `IO`) as a tuple.

Exercises:

1. Implement a function `rollNDiceIO :: Int -> IO [Int]` that, given an integer, returns a list with that number of pseudo-random integers between 1 and 6.

34.3.1 Getting Rid of the `IO` Monad

Now, suppose that for any reason we do not want to use the `IO` monad: we might want the function to stay pure, or need a sequence of numbers that is the same in every run, for repeatability.

To do that, we can produce a generator using the `mkStdGen` function in the `System.Random` library:

```
> mkStdGen 0
1 1
```

The argument to `mkStdGen` is an `Int` that functions as a seed. With that, we can generate a pseudo-random integer number in the interval between 1 and 6 with:

```
> randomR (1,6) (mkStdGen 0)
(6,40014 40692)
```

We obtained a tuple with the result of the dice throw (6) and the new generator (40014 40692). A simple implementation that produces a tuple of two pseudo-random integers is then:

```
clumsyRollDice :: (Int, Int)
clumsyRollDice = (n, m)
        where
        (n, g) = randomR (1,6) (mkStdGen 0)
        (m, _) = randomR (1,6) g
```

Figure 4 Boxcars![a]

[a] http://en.wikipedia.org/wiki/Boxcars_%28slang%29

When we run the function, we get:

```
> clumsyRollDice
(6, 6)
```

The implementation of `clumsyRollDice` works, but we have to manually write the passing of generator `g` from one `where` clause to the other. This is pretty easy now, but will become increasingly cumbersome if we want to produce large sets of pseudo-random numbers. It is also error-prone: what if we pass one of the middle generators to the wrong line in the `where` clause?

Exercises:

1. Implement a function `rollDice :: StdGen -> ((Int, Int), StdGen)` that, given a generator, return a tuple with our random numbers as first element and the last generator as the second.

34.3.2 Introducing `State`

We will now try to solve the clumsiness of the previous approach introducing the `State StdGen` monad. For convenience, we give it a name with a type synonym:

```
import Control.Monad.Trans.State
import System.Random
```

```
type GeneratorState = State StdGen
```

Remember, however, that a `GeneratorState Int` is in essence a `StdGen -> (Int, StdGen)` function, so it is not really the generator state, but a processor of the generator state. The generator state itself is produced by the `mkStdGen` function. Note that `GeneratorState` does not specify what type of values we are going to extract, only the type of the state.

We can now produce a function that, given a `StdGen` generator, outputs a number between 1 and 6:

```
rollDie :: GeneratorState Int
rollDie = do generator <- get
             let (value, newGenerator) = randomR (1,6) generator
             put newGenerator
             return value
```

The `do` notation is in this case much more readable; let's go through each of the steps:

1. First, we take out the pseudo-random generator with `<-` in conjunction with `get`. `get` overwrites the monadic value (The 'a' in 'm a') with the state, and thus generator is bound to the state. (If in doubt, recall the definition of `get` and `>>=` above).
2. Then, we use the `randomR` function to produce an integer between 1 and 6 using the generator we took; we also store the new generator graciously returned by `randomR`.
3. We then set the state to be the `newGenerator` using the `put` function, so that the next call will use a different pseudo-random generator;
4. Finally, we inject the result into the `GeneratorState` monad using `return`.

We can finally use our monadic die:

```
> evalState rollDie (mkStdGen 0)
6
```

At this point, a legitimate question is why we have involved monads and built such an intricate framework only to do exactly what `fst $ randomR (1,6)` does. The answer is illustrated by the following function:

```
rollDice :: GeneratorState (Int, Int)
rollDice = liftM2 (,) rollDie rollDie
```

We obtain a function producing *two* pseudo-random numbers in a tuple. Note that these are in general different:

```
> evalState rollDice (mkStdGen 666)
(6,1)
```

That is because, under the hood, the monads are passing state to each other. This used to be very clunky using `randomR (1,6)`, because we had to pass state manually; now, the monad is taking care of that for us. Assuming we know how to use the lifting functions, constructing intricate combinations of pseudo-random numbers (tuples, lists, whatever) has suddenly become much easier.

Exercises:

1. Similarly to what was done for `rollNDiceIO`, implement a function `rollNDice :: Int -> GeneratorState [Int]` that, given an integer, returns a list with that number of pseudo-random integers between 1 and 6.

34.4 Pseudo-random values of different types

Until now, absorbed in the die example, we considered only `Int` as the type of the produced pseudo-random number. However, already when we defined the `GeneratorState` monad, we noticed that it did not specify anything about the type of the returned value. In fact, there is one implicit assumption about it, and that is that we can produce values of such a type with a call to `random`.

Values that can be produced by `random` and similar function are of types that are instances of the `Random` class (capitalised). There are default implementations for `Int`, `Char`, `Integer`, `Bool`, `Double` and `Float`, so you can immediately generate any of those.

Since we noticed already that the `GeneratorState` is "agnostic" in regard to the type of the pseudo-random value it produces, we can write down a similarly "agnostic" function, analogous to `rollDie`, that provides a pseudo-random value of unspecified type (as long as it is an instance of `Random`):

```
getRandom :: Random a => GeneratorState a
getRandom = do generator <- get
               let (value, newGenerator) = random generator
               put newGenerator
               return value
```

Compared to `rollDie`, this function does not specify the `Int` type in its signature and uses `random` instead of `randomR`; otherwise, it is just the same. What is notable is that `getRandom` can be used for any instance of `Random`:

```
> evalState getRandom (mkStdGen 0) :: Bool
True
> evalState getRandom (mkStdGen 0) :: Char
'\64685'
> evalState getRandom (mkStdGen 0) :: Double
0.9872770354820595
> evalState getRandom (mkStdGen 0) :: Integer
2092838931
```

Indeed, it becomes quite easy to conjure all these at once:

```
allTypes :: GeneratorState (Int, Float, Char, Integer, Double, Bool, Int)
allTypes = liftM (,,,,,,) getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
```

Here we are forced to used the ap function, defined in Control.Monad, since there exists no liftM7. As you can see, its effect is to fit multiple computations into an application of the (lifted) 7-element-tuple constructor, (,,,,,,). To understand what ap does, look at its signature:

```
>:type ap
ap :: (Monad m) => m (a -> b) -> m a -> m b
```

remember then that type a in Haskell can be a function as well as a value, and compare to:

```
>:type liftM (,,,,,,) getRandom
liftM (,,,,,) getRandom :: (Random a1) =>
                          State StdGen (b -> c -> d -> e -> f -> (a1, b, c, d,
e, f))
```

The monad m is obviously State StdGen (which we "nicknamed" GeneratorState), while ap's first argument is function b -> c -> d -> e -> f -> (a1, b, c, d, e, f). Applying ap over and over (in this case 6 times), we finally get to the point where b is an actual value (in our case, a 7-element tuple), not another function. To sum it up, ap applies a function-in-a-monad to a monadic value (compare with liftM, which applies a function *not* in a monad to a monadic value).

So much for understanding the implementation. Function allTypes provides pseudo-random values for all default instances of Random; an additional Int is inserted at the end to prove that the generator is not the same, as the two Ints will be different.

```
 > evalState allTypes (mkStdGen 0)

(2092838931,9.953678e-4,'\825586',-868192881,0.4188001483955421,False,316817438)
```

Exercises:

1. If you are not convinced that State is worth using, try to implement a function equivalent to evalState allTypes without making use of monads, i.e. with an approach similar to clumsyRollDice above.

35 Additive monads (MonadPlus)

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 *all* valid solutions into a single result. Taking the list monad as an example, given two lists of valid solutions we can simply concatenate the lists together to get all valid solutions. In such a context, it is also useful, especially when working with folds, to require a value corresponding to "zero results" (i.e. failure). For lists, the empty list represents zero results. `MonadPlus` is a type class which captures these features in a general way.

35.1 Definition

`MonadPlus` defines two methods. `mzero` is the monadic value standing for zero results; while `mplus` is a binary function which combines two computations.

```
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[1], then `(Either e)` becomes an instance:

```
instance (Error e) => MonadPlus (Either e) where
```

[1] http://hackage.haskell.org/packages/archive/mtl/latest/doc/html/Control-Monad-Error.html

```
  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" (often, but not necessarily, a string). Typically, Left s means a failed computation carrying an error message s, and Right x means a successful computation with result x.

35.2 Example: parallel parsing

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 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
```

Parser libraries often make use of MonadPlus in this way. If you are curious, check the (+++) operator in Text.ParserCombinators.ReadP[2], or (<|>) in Text.ParserCombinators.Parsec.Prim[3].

35.3 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. The most common (but not universal) are that mzero and mplus form a *monoid*. By that, we mean:

[2] http://hackage.haskell.org/packages/archive/base/latest/doc/html/Text-ParserCombinators-ReadP.html

[3] http://hackage.haskell.org/packages/archive/parsec/latest/doc/html/Text-ParserCombinators-Parsec-Prim.html

```
-- mzero is a neutral element
mzero `mplus` m  =  m
m `mplus` mzero  =  m
-- mplus is associative
--  (not all instances obey this law, because it makes some infinite structures
 impossible)
m `mplus` (n `mplus` o)  =  (m `mplus` n) `mplus` o
```

There is nothing fancy about "forming a monoid": in the above, "neutral element" and "associative" are meant with exactly the same sense that addition of integer numbers is said to be associative and to have zero as neutral element. In fact, this analogy gives the names of `mzero` and `mplus`.

The Haddock documentation[4] for Control.Monad quotes additional laws:

```
mzero >>= f  =  mzero
m >> mzero   =  mzero
```

And the HaskellWiki page[5] cites another (with controversy):

```
(m `mplus` n) >>= k  =  (m >>= k) `mplus` (n >>= k)
```

There are even more sets of laws available, and therefore you'll sometimes see monads like IO being used as a MonadPlus. Consult All About Monads[6] and the Haskell Wiki page[7] on MonadPlus for extra more information about such issues.

35.4 Useful functions

Beyond the basic `mplus` and `mzero` themselves, there are two other important general-purpose functions involving `MonadPlus`:

35.4.1 msum

A very common task when working with instances of MonadPlus is to take a list of monadic values, e.g. `[Maybe a]` or `[[a]]`, and fold it down 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.

[4] http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Monad.html#t%3AMonadPlus

[5] http://www.haskell.org/haskellwiki/MonadPlus

[6] http://www.haskell.org/haskellwiki/All_About_Monads#Zero_and_Plus

[7] http://www.haskell.org/haskellwiki/MonadPlus

35.4.2 guard

When discussing the list monad[8] we noted how similar it was to list comprehensions, but we left the question of how to mirror list comprehension filtering in the list monad hanging. The `guard` function allows us to do exactly that. Our example for this section will be the following comprehension, which retrieves all pythagorean triples[9] (that is, trios of integer numbers which taken together are the lengths of the sides for some right triangle) in the obvious, brute-force way. It uses a boolean condition for filtering; namely, Pythagoras' theorem:

```
pythags = [ (x, y, z) | z <- [1..], x <- [1..z], y <- [x..z], x^2 + y^2 == z^2 ]
```

The translation of the comprehension above to the list monad is:

```
pythags = do
  z <- [1..]
  x <- [1..z]
  y <- [x..z]
  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 `x^2 + y^2 == z^2` is `False`. Let's look at the expansion of the above `do`-block to see how it works:

```
pythags =
  [1..] >>= \z ->
  [1..z] >>= \x ->
  [x..z] >>= \y ->
  guard (x^2 + y^2 == z^2) >>= \_ ->
  return (x, y, z)
```

Replacing `>>=` and `return` with their definitions for the list monad (and using some let-bindings to keep it readable), we obtain:

[8] Chapter 31 on page 195

[9] http://en.wikipedia.org/wiki/Pythagorean_triple

```
pythags =
 let ret x y z = [(x, y, z)]
     gd  z x y = concatMap (\_ -> ret x y z) (guard $ x^2 + y^2 == z^2)
     doY z x   = concatMap (gd  z x) [x..z]
     doX z     = concatMap (doY z  ) [1..z]
     doZ       = concatMap (doX    ) [1..]
 in doZ
```

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 `z`, then a branch from each of these branches for every value of `x`, then from each of these, a branch for every value of `y`. So the tree looks like this:

```
   start
   |__________________...
   | |     |
z  1 2     3
   | |____ |__________
   | |   | |     |   |
x  1 1   2 1     2   3
   | |__ | |____ |__ |
   | | | | | | | | | |
y  1 1 2 2 1 2 3 2 3 3
```

Each 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.

35.5 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 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 hexidecimal character (0-9 or a-f). Try writing this function (hint: `map digit [0..9] :: [String -> Maybe Int]`).
3. More to come...

35.6 Relationship with monoids

When discussing the MonadPlus laws, we alluded to the mathematical concept of monoids. It turns out that there is a `Monoid` class in Haskell, defined in Data.Monoid[10].[11] A minimal definition of `Monoid` implements two methods; namely, a neutral element (or 'zero') and an associative binary operation (or 'plus').

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

For example, lists form a simple monoid:

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

Sounds familiar, doesn't it? In spite of the uncanny resemblance to `MonadPlus`, there is a subtle yet key difference: note the usage of `[a]`, not `[]`, in the instance declaration. Monoids are not necessarily "containers" of anything, or parametrically polymorphic. The integer numbers, for instance, form a monoid under addition, with `0` as neutral element.

In any case, `MonadPlus` instances look very similar to monoids, as both feature concepts of zero and plus. Indeed, we could even make `MonadPlus` a subclass of `Monoid` if there was a real need to take the trouble:

```
 instance MonadPlus m => Monoid (m a) where
   mempty  = mzero
   mappend = mplus
```

Note:
Due to the "free" type variable `a` in the instance definition, the snippet above is not valid Haskell 98. If you want to test it, you will have to enable the GHC *language extension* `FlexibleInstances`:

- If you are testing with GHCi, start it with the command line option `-XFlexibleInstances`.
- Alternatively, if you are running a compiled program, add `{-# LANGUAGE FlexibleInstances #-}` to the top of your source file.

However, they work at different levels. As noted, there is no requirement for monoids to be parameterized in relation to "contained" or related type. More formally, monoids have kind *, but instances of `MonadPlus`, as they are monads, have kind * -> *.

[10] `http://hackage.haskell.org/packages/archive/base/latest/doc/html/Data.Monoid.html`

[11] A fuller presentation of it will be given in a later chapter ^{Chapter45 on page 311}.

36 Monadic parser combinators

Monads provide a clean means of embedding a domain specific parsing language directly into Haskell without the need for external tools or code generators.

- Functional Pearls: Monadic Parsing in Haskell[1]
- Practical Monads: Parsing Monads[2] which shows an example using Parsec[3], a popular, efficient monadic recursive descent parser library

[1] http://eprints.nottingham.ac.uk/223/1/pearl.pdf

[2] Chapter 38.1 on page 239

[3] http://legacy.cs.uu.nl/daan/parsec.html

37 Monad transformers

By this point you should have grasped the concept of monad, and what different monads are used for: `IO` for impure functions, `Maybe` for values that can be there or not, and so on. With monads providing such useful general-purpose functionalities, it is very natural that sometimes we would like to use the capabilities of *several* monads at once - for instance, a function which uses both I/O and `Maybe` exception handling. Sure, we can use a type like `IO (Maybe a)`, but that forces us to do pattern matching within the `do` blocks to extract the values you want: the point of monads was also to get rid of that.

Enter **monad transformers**: special types that allow us to roll two monads into a single one that shares the behaviour of both. We will begin with an example to illustrate why transformers are useful and show a simple example of how they work.

37.1 Password validation

Consider a common real-life problem for IT staff worldwide: to get their users to select passwords that are not easy to guess. A typical strategy is to force the user to enter a password with a minimum length, and at least one letter, one number and similar irritating requirements.

A Haskell function to acquire a password from a user could look like:

```
getPassword :: IO (Maybe String)
getPassword = do s <- getLine
                 if isValid s then return $ Just s
                              else return Nothing

-- The validation test could be anything we want it to be.
isValid :: String -> Bool
isValid s = length s >= 8 && any isAlpha s && any isNumber s && any
 isPunctuation s
```

First and foremost, `getPassword` is an `IO` function, as it needs to get input from the user, and so it will not always return. We also use `Maybe`, as we intend to return `Nothing` in case the password does not pass the `isValid`. Note, however, that we aren't actually using `Maybe` as a monad here: the `do` block is in the `IO` monad, and we just happen to `return` a `Maybe` value into it.

The true motivation for monad transformers is not only to make it easier to write `getPassword` (which it nevertheless does), but rather to simplify all the code instances in which we use it. Our password acquisition program could continue like this:

```
askPassword :: IO ()
askPassword = do putStrLn "Insert your new password:"
                 maybe_value <- getPassword
```

```
          if isJust maybe_value
              then do putStrLn "Storing in database..."
              -- ... other stuff, including 'else'
```

We need one line to generate the maybe_value variable, and then we have to do some further checking to figure out whether our password is OK or not.

With monad transformers, we will be able to extract the password in one go, without any pattern matching or equivalent bureaucracy like isJust. The gains for our simple example might seem small, but will scale up for more complex situations.

37.2 A simple monad transformer: MaybeT

To simplify getPassword function and all the code that uses it, we will define a *monad transformer* that gives the IO monad some characteristics of the Maybe monad; we will call it MaybeT, following the convention that monad transformers have a "T" appended to the name of the monad whose characteristics they provide.

MaybeT is a wrapper around m (Maybe a), where m can be any monad (in our example, we are interested in IO):

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

This data type definition specifies a MaybeT type constructor, parametrized over m, with a term constructor, also called MaybeT, and a convenient accessor function runMaybeT, with which we can access the underlying representation.

The whole point of monad transformers is that *they are monads themselves*; and so we need to make MaybeT m an instance of the Monad class:

```
instance Monad m => Monad (MaybeT m) where
    return  = MaybeT . return . Just
```

return is implemented by Just, which injects into the Maybe monad, a generic return that injects into m (whatever it is), and the MaybeT constructor. It would also have been possible (though arguably less readable) to write return = MaybeT . return . return.

```
    x >>= f = MaybeT $ do maybe_value <- runMaybeT x
                          case maybe_value of
                               Nothing    -> return Nothing
                               Just value -> runMaybeT $ f value
```

Like for all monads, the bind operator is the heart of the transformer, and the most important snippet for understanding how it works. As always, it is useful to keep the type signature in mind. The type of the MaybeT bind is:

```
-- The signature of (>>=), specialized to MaybeT m
(>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
```

Now, let us consider what it does, step by step, starting from the first line of the do block.

- First, it uses the runMaybeT accessor to unwrap x into a m (Maybe a) computation. That gives away the do block is in m.

- Still in the first line, `<-` extracts a `Maybe a` value from the unwrapped computation.
- The `case` statement tests `maybe_value`:
  - if it is `Nothing`, it returns `Nothing` into `m`;
  - if it is a `Just`, it applies `f` to the `value` inside it. Since `f` has `MaybeT m b` as result type, we need an extra `runMaybeT` to put the result back into the `m` monad.
- Finally, the `do` block as a whole has `m (Maybe b)` type; so it is wrapped with the `MaybeT` constructor.

It may look a bit complicated; but other than for the copious amounts of wrapping and unwrapping, the implementation does the same as the familiar bind operator of `Maybe`:

```
-- (>>=) for the Maybe monad
    maybe_value >>= f = case maybe_value of
                            Nothing -> Nothing
                            Just value -> f value
```

You may wonder why we are using the `MaybeT` constructor before the `do` block, when inside it we use the accessor `runMaybeT`: however, the `do` block must be in the `m` monad, not in `MaybeT m`, since for the latter we have not yet defined the bind operator.

Note:
The chained functions in the definition of `return` suggest an analogy, which you may find either useful or confusing, between the combined monad and a *sandwich.* In this example, `Maybe`, the "base" monad, would be the bottom layer; the inner monad `m`, the filling; and the transformer `MaybeT`, the top layer. There is some danger in the analogy in that it might suggest there are three layers of monads in action, while in fact there are only two: the inner monad and the combined monad (there are no binds or returns done in the base monad; it only appears as part of the implementation of the transformer). A better way of interpreting the analogy is thinking of the transformer and the base monad as two parts of the same thing - the *bread* - which wraps the inner monad.

Technically, this is all we need; however, it is convenient to make `MaybeT` an instance of a few other classes:

```
instance Monad m => MonadPlus (MaybeT m) where
    mzero     = MaybeT $ return Nothing
    mplus x y = MaybeT $ do maybe_value <- runMaybeT x
                            case maybe_value of
                                 Nothing    -> runMaybeT y
                                 Just _     -> return maybe_value

instance MonadTrans MaybeT where
    lift = MaybeT . (liftM Just)
```

The latter class, `MonadTrans`, implements the `lift` function, which is very useful to take functions from the `m` monad and bring them into the `MaybeT m` monad, so that we can use them in `do` blocks inside the `MaybeT m` monad. As for `MonadPlus`, since `Maybe` is an instance of that class it makes sense to make the `MaybeT` an instance too.

37.2.1 Application to the password example

With all this done, here is what the previous example of password management looks like:

```
getValidPassword :: MaybeT IO String
getValidPassword = do s <- lift getLine
                      guard (isValid s) -- MonadPlus provides guard.
                      return s

askPassword :: MaybeT IO ()
askPassword = do lift $ putStrLn "Insert your new password:"
                 value <- getValidPassword
                 lift $ putStrLn "Storing in database..."
```

The code is now simpler, especially in the user function `askPassword`. Most importantly, we do not have to manually check whether the result is `Nothing` or `Just`: the bind operator takes care of that for us.

Note how we use `lift` to bring the functions `getLine` and `putStrLn` into the `MaybeT IO` monad. Also, since `MaybeT IO` is an instance of `MonadPlus`, checking for password validity can be taken care of by a `guard` statement, which will return `mzero` (i.e. `IO Nothing`) in case of a bad password.

Incidentally, with the help of `MonadPlus` it also becomes very easy to ask the user *ad infinitum* for a valid password:

```
askPassword :: MaybeT IO ()
askPassword = do lift $ putStrLn "Insert your new password:"
                 value <- msum $ repeat getValidPassword
                 lift $ putStrLn "Storing in database..."
```

37.3 A plethora of transformers

The modules of the `transformers` package provide transformers for many common monads (`MaybeT`, for instance, can be found in Control.Monad.Trans.Maybe[1]). They are defined consistently with their non-transformer versions; that is, the implementation is basically the same, only with the extra wrapping and unwrapping needed to thread the other monad. From this point on, we will refer to the non-transformer monad on which the transformer is based as the **base monad**, and to the other monad, on which the transformer is applied, as the **inner monad**.

To pick an arbitrary example, `ReaderT Env IO String` is a computation which involves reading values from some environment of type `Env` (the semantics of `Reader`, the base monad) and performing some `IO` in order to give a value of type `String`. Since the bind operator and `return` for the transformer mirror the semantics of the base monad, a `do` block of type `ReaderT Env IO String` will, from the outside, look a lot like a `do` block of the `Reader` monad; the main difference being that `IO` actions become trivial to embed by using `lift`.

[1] http://hackage.haskell.org/packages/archive/transformers/latest/doc/html/Control-Monad-Trans-Maybe.html

37.3.1 Type juggling

We have seen that the type constructor for `MaybeT` is a wrapper for a `Maybe` value in the inner monad, and so the corresponding accessor `runMaybeT` gives us a value of type `m (Maybe a)` - that is, a value of the base monad returned in the inner monad. In a similar fashion, we have

```
runListT :: ListT m a -> m [a]
```

and

```
runErrorT :: ErrorT e m a -> m (Either e a)
```

for the list and error transformers.

Not all transformers are related to their base monads in this way, however. The `Writer`, `Reader`, `State` and `Cont` monads have in common that, unlike the base monads in the examples above, they have neither multiple constructors nor constructors with multiple arguments. For that reason, they have `run...` functions, which act as simple unwrappers analogous to the `run...T` of the transformer versions. The table below shows the result types of the `run...` and `run...T` functions in each case, which may be thought of as the types wrapped by the base and transformed monads respectively.[2]

Base Monad	**Transformer**	**Original Type ("wrapped" by base)**	**Combined Type ("wrapped" by transformed)**
Writer	WriterT	`(a, w)`	`m (a, w)`
Reader	ReaderT	`r -> a`	`r -> m a`
State	StateT	`s -> (a, s)`	`s -> m (a, s)`
Cont	ContT	`(a -> r) -> r`	`(a -> m r) -> m r`

The first thing to notice is the base monad is nowhere to be seen in the combined types. That is very natural, as without interesting constructors (like the ones for `Maybe` or lists) there is no reason to retain the base monad type after unwrapping the transformed monad. Besides that, in the three latter cases we have function types being wrapped. `StateT`, for instance, turns state-transforming functions of the form `s -> (a, s)` into state-transforming functions of the form `s -> m (a, s)`; so that only the result type of the wrapped function goes into the inner monad. `ReaderT` is analogous, but not `ContT`: due to the semantics of `Cont` (the *continuation* monad) the result types of both the wrapped function and its functional argument must be the same, and so the transformer puts both into the inner monad. What these examples show is that 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.

[2] The wrapping interpretation is only literally true for versions of the `mtl` package older than 2.0.0.0 .

37.4 Lifting

The `lift` function, which we have first presented in the introduction, is very important in day-to-day usage of transformers; and so we will dedicate a few more words to it.

37.4.1 liftM

Let us begin by considering the more familiar `liftM` function. It has the following type:

```
liftM :: Monad m => (a1 -> r) -> m a1 -> m r
```

`liftM` applies a function `(a1 -> r)` to a value within a monad `m`. If you prefer the point-free interpretation, it converts a regular function into one that acts within `m` - and *that* is what is meant by lifting.

To recapitulate, here is a simple example of `liftM` usage. The following pieces of code all mean the same thing.

do notation

```
do x <- monadicValue
   return (f x)
```

liftM

```
liftM f monadicValue
```

liftM as an operator

```
f `liftM` monadicValue
```

The third example, in which we use `liftM` as an operator, suggests an interesting way of viewing `liftM`: it is a monadic analogue of `($)`!

non monadic

```
f $ value
```

monadic

```
f `liftM` monadicValue
```

37.4.2 lift

When using combined monads created with monad transformers, we avoid having to manage the inner monad types explicitly, 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.

With `liftM` we have seen how the essence of lifting is, to paraphrase the documentation, promoting something into a monad. The `lift` function, available for all monad transformers, performs a different kind of lifting: it promotes a computation from the inner monad into the combined monad. `lift` is defined as the single method of the `MonadTrans` class in Control.Monad.Trans.Class[3].

```
class MonadTrans t where
    lift :: (Monad m) => m a -> t m a
```

There is a variant of `lift` specific to `IO` operations called `liftIO`, which is the single method of the `MonadIO` class in Control.Monad.IO.Class[4].

```
class (Monad m) => MonadIO m where
   liftIO :: IO a -> m a
```

`liftIO` can be convenient when having multiple transformers stacked into a single combined monad. In such cases, `IO` is always the innermost monad, and so more than one lift would be typically needed to bring `IO` values to the top of the stack. `liftIO`, however, is defined for the instances in a way that allows us to bring n `IO` value from any depth while writing the function a single time.

37.4.3 Implementing `lift`

Implementing `lift` is usually pretty straightforward. Consider the transformer `MaybeT`:

```
instance MonadTrans MaybeT where
    lift m = MaybeT (m >>= 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. Note that, just as in the implementation of the `Monad` class, both the bind operator and the generic `return` are 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.

37.5 Implementing transformers

In order to develop a better feel for the workings of transformers, we will discuss two more implementations in the standard libraries.

[3] http://hackage.haskell.org/packages/archive/transformers/latest/doc/html/Control-Monad-Trans-Class.html

[4] http://hackage.haskell.org/packages/archive/transformers/latest/doc/html/Control-Monad-IO-Class.html

37.5.1 The List transformer

Just as with the `Maybe` transformer, we start by creating a datatype with a constructor that takes one argument:

```
newtype ListT m a = ListT { runListT :: m [a] }
```

The implementation of the `ListT m` monad is also strikingly similar to its "cousin", the list monad. We do the same operations done for `[]`, but with a little extra support to operate within the inner monad `m`, and to pack and unpack the monadic sandwich.

List

```
instance Monad [] where
    return x = [x]
    xs >>= f =
        let yss = map f xs
        in concat yss
```

ListT

```
instance (Monad m) => Monad (ListT m) where
    return x = ListT $ return [x]
    tm >>= f = ListT $ runListT tm
               >>= \xs -
> mapM (runListT . f) xs
               >>= \yss -
> return (concat yss)
```

Exercises:

1. Dissect the bind operator for the (ListT m) monad. For example, why do we now have `mapM` and `return`?
2. `Identity` is a trivial monad defined, in `Data.Functor.Identity`, as:
   ```
   newtype Identity a = Identity { runIdentity :: a }
   instance Monad Identity where
       return a = Identity a
       m >>= k = k (runIdentity m)
   ```
   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?

37.5.2 The State transformer

Previously, we have pored over the implementation of one simple monad transformer, `MaybeT`, and reviewed the implementation of another, `ListT`, taking a detour along the way to talk about lifting from 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[5] before continuing.

[5] Chapter 34 on page 209

Just as the State monad might have been built upon the definition `newtype State s a = State { runState :: (s -> (a,s)) }`[6] the StateT transformer is built upon the definition

```
newtype StateT s m a = StateT { runStateT :: (s -> m (a,s)) }
```

`State s` is an instance of both the `Monad` class and the `MonadState s` class (which provides `get` and `put`), 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

```
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'
```

StateT

```
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
    (v,s') <- x s          --
 get new value and state
    runStateT (f v) s'     --
 pass them to f
```

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:

[6] In the `transformers` and `mtl` packages, `State s` is implemented as a type synonym for `StateT s Identity`. Incidentally, that explains why there was a `state` function instead of the `State` constructor back in the chapter about `State` ^{Chapter34.2.1 on page 211}, and why we had to defer the explanation until now.

```
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, if for instance we apply StateT to the List monad, 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.

37.6 Acknowledgements

This module uses a number of excerpts from All About Monads[7], with permission from its author Jeff Newbern.

ru:Haskell/Monad transformers[8]

[7] http://www.haskell.org/haskellwiki/All_About_Monads

[8] http://ru.wikibooks.org/wiki/Haskell%2FMonad%20transformers

38 Practical monads

In this chapter, we will present some very diverse examples of monads being used for practical tasks. Consider it as bonus material, and go through it at your own pace. You can always come back later if some parts (e.g. the final example about concurrency) seem too alien right now.

38.1 Parsing monads

This section is based on the "Parsing" chapter of Jonathan Tang's Write Yourself a Scheme in 48 Hours[1].

In the previous chapters, we saw how monads were used for IO, and started working more extensively with some of the more rudimentary monads like `Maybe`, `List` or `State`. Now let us try something quintessentially "practical": writing a simple parser. Monads provide a clean way of embedding a domain specific parsing language directly into Haskell without the need for external tools or code generators. For a brief and accessible presentation of the subject, we suggest the paper Functional Pearls: Monadic Parsing in Haskell[2], by Graham Hutton and Erik Meijer. Right now, however, is time to get our hands dirty; and for that we will be using the Parsec[3] library, version 3 or greater.

We need an extension for this code: FlexibleContexts. This allows us to write class constraints such as `(Stream s u Char) =>`, where one of the type variables is defined instead of polymorphic.

```
{-# LANGUAGE FlexibleContexts #-}
```

Start by adding this line to the import section:

```
import Control.Monad
import Control.Monad.Identity (Identity)
import System.Environment (getArgs)
import Text.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. In addition, the Identity monad is made available so that we can use ParsecT on Identity.

[1] https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours

[2] http://eprints.nottingham.ac.uk/223/1/pearl.pdf

[3] http://hackage.haskell.org/package/parsec-3.1.1

Now, we'll define a parser that recognizes one of the symbols allowed in Scheme identifiers:

```
symbol :: Stream s m Char => ParsecT s u m 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 3 uses a monad transformer to take care of all of that for us. We need only use the Parsec library function oneOf (see Text.Parsec.Char[4]), and it'll recognize a single one of any of the characters in the string passed to it. And as you're about to see, you can compose primitive parsers into more sophisticated productions.

The type of the function is somewhat confusing. Stream s m Char defines a "stream" of Char's of type s, wrapped around monad m. Examples are of s would be String or ByteString. Accommodating both String and ByteString is the main reason for defining our function to be polymorphic around String. Parsec contains a type called Parser, but its not as polymorphic as we would normally like - it explicitly requires a stream type of String.

ParsecT defines a parser for a stream type s, state type u (we don't really need to use state, but its useful to define our functions to be polymorphic on state), inner monad m (usually Identity if we don't want to use it as a transformer) and result type Char, which is the "normal" type argument to Monads.

Let's define a function to call our parser and handle any possible errors:

```
readExpr :: Stream s Identity Char => s -> 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 Stream (String or ByteString, most of the time) 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[5].

Parse 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[6] 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".

[4] http://hackage.haskell.org/packages/archive/parsec/3.1.1/doc/html/Text-Parsec-Char.html

[5] http://hackage.haskell.org/packages/archive/parsec/3.1.1/doc/html/Text-Parsec-Prim.html#v:parse

[6] http://www.haskell.org/onlinereport/standard-prelude.html#\protect\char"0024\relax{}tEither

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 -package mtl" 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"
```

38.1.1 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[7]" in that library, but it doesn't quite do what we want it to. (For that matter, there's also a parser called lexeme[8] that does exactly what we want, but we'll ignore that for pedagogical purposes.)

```
spaces :: Stream s m Char => ParsecT s u m ()
spaces = skipMany1 space
```

[7] http://research.microsoft.com/users/daan/download/parsec/parsec.html#spaces

[8] http://research.microsoft.com/users/daan/download/parsec/parsec.html#lexeme

Just as functions can be passed to functions, so can actions. Here we pass the Parser action space[9] to the Parser action skipMany1[10], 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:

```
readExpr input = case parse (spaces >> symbol) "lisp" input of
    Left err -> "No match: " ++ show err
    Right val -> "Found value"
```

We touched briefly on the >> ("then") operator in lesson 1, 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, then has completely different semantics in the Parser and IO monads. In the Parser monad, then means "Attempt to match the first parser, then attempt to match the second with the remaining input, and fail if either fails." In general, then 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 listing3.2.hs]
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
```

38.1.2 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:

[9] http://research.microsoft.com/users/daan/download/parsec/parsec.html#space

[10] http://research.microsoft.com/users/daan/download/parsec/parsec.html#skipMany1

```
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 :: Stream s m Char => ParsecT s u m 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[11] (noneOf[12] "\"")) 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.

[11] http://research.microsoft.com/users/daan/download/parsec/parsec.html#many

[12] http://research.microsoft.com/users/daan/download/parsec/parsec.html#noneOf

We then apply the built-in function return[13] 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[14].

Now let's move on to Scheme variables. An atom[15] is a letter or symbol, followed by any number of letters, digits, or symbols:

```
parseAtom :: Stream s m Char => ParsecT s u m 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 <|>[16]. 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 :: Stream s m Char => ParsecT s u m LispVal
parseNumber = liftM (Number . read) $ many1 digit
```

[13] http://www.haskell.org/onlinereport/standard-prelude.html#\protect\char"0024\relax{}tMonad

[14] http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.4

[15] http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-5.html#%_sec_2.1

[16] http://research.microsoft.com/users/daan/download/parsec/parsec.html#or

It's easiest to read this backwards, since both function application ($) and function composition (.) associate to the right. The parsec combinator many1[17] 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[18] 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.

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 :: Stream s m Char => ParsecT s u m 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 listing3.3.hs]
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
```

[17] http://www.cs.uu.nl/~daan/download/parsec/parsec.html#many1

[18] http://www.haskell.org/onlinereport/standard-prelude.html#\protect\char"0024\relax{}vread

```
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
   a) do-notation
   b) explicit sequencing with the >>=[a] operator
2. Our strings aren't quite R5RS compliant[b], 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[c]. You may find the readOct and readHex[d] functions useful.
5. Add a Character constructor to LispVal, and create a parser for character literals[e] as described in R5RS.
6. Add a Float constructor to LispVal, and support R5RS syntax for decimals[f]. The Haskell function readFloat[g] may be useful.
7. Add data types and parsers to support the full numeric tower[h] of Scheme numeric types. Haskell has built-in types to represent many of these; check the Prelude[i]. 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).

[a] http://www.haskell.org/onlinereport/standard-prelude.html#tMonad
[b] http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.3.5
[c] http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.2.4
[d] http://www.haskell.org/onlinereport/numeric.html#sect14
[e] http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.3.4
[f] http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.2.4
[g] http://www.haskell.org/onlinereport/numeric.html#sect14
[h] http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.2.1
[i] http://www.haskell.org/onlinereport/standard-prelude.html#\protect\char"0024\relax{}tNum

38.1.3 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 :: Stream s m Char => ParsecT s u m 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[19], 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 :: Stream s m Char => ParsecT s u m 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 :: Stream s m Char => ParsecT s u m 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 :: Stream s m Char => ParsecT s u m LispVal
parseExpr = parseAtom
        <|> parseString
        <|> parseNumber
        <|> parseQuoted
```

```
         <|> do char '('
                x <- (try parseList) <|> parseDottedList
                char ')'
                return x
```

[19] http://research.microsoft.com/users/daan/download/parsec/parsec.html#sepBy

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[20] 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 listing3.4.hs]
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[a] syntactic sugar: the Scheme standard details what it should expand into (quasiquote/unquote).
2. Add support for vectors[b]. The Haskell representation is up to you: GHC does have an Array[c] 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

[a] http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-7.html#%_sec_4.2.6

[b] http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.3.6

[c] http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Array.html

[20] http://research.microsoft.com/users/daan/download/parsec/parsec.html#try

38.2 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.

38.3 Stateful monads for concurrent applications

You're going to have to know about Monad transformers [21] before you can do these things. Although the example came up because of Concurrency [22], if you realize a TVar is a mutable variable of some kind, why this example came up might make some sense to you.

This is a little trick that I find makes writing stateful concurrent applications easier, especially for network applications. Lets look at an imaginary stateful server.

Each currently connected client has a thread allowing the client to update the state.

The server also has a main logic thread which also transforms the state.

So you want to allow the client to update the state of the program

It's sometimes really simple and easy to expose the whole state of the program in a TVar, but I find this can get really messy, especially when the definition of the state changes!

Also it can be very annoying if you have to do anything conditional.

[21] Chapter 37 on page 229

[22] Chapter 47 on page 319

So to help tidy things up (Say your state is called World)

38.3.1 Make a monad over state

First, make a monad over the World type

```
import Control.Monad.State.Lazy
```

```
-- heres yer monad
-- it can liftIO too
type WorldM
 = StateT World IO
```

```
data World =
  World { objects :: [ WorldObject ] }
```

Now you can write some accessors in WorldM

```
-- maybe you have a bunch of objects each with a unique id
import Data.Unique
import Data.Maybe
import Prelude hiding ( id )
```

```
data WorldObject =
   WorldObject { id :: Unique }
```

```
-- check Control.Monad.State.Lazy if you are confused about get and put
addObject :: WorldObject -> WorldM ( )
addObject wO = do
   wst <- get
   put $ World $ wO : ( objects wst )
```

```
-- presuming unique id
getObject :: Unique -> WorldM ( Maybe WorldObject )
getObject id1 = do
   wst <- get
   return $ listToMaybe $ filter ( \ wO -> id wO == id1 )
                                 ( objects wst )
```

now heres a type representing a change to the World

```
  data WorldChange = NewObj WorldObject |
                     UpdateObj WorldObject | -- use the unique ids as replace
match
                     RemoveObj Unique -- delete obj with named id
```

What it looks like all there's left to do is to

```
  type ChangeBucket = TVar [ WorldChange ]
```

```
mainLoop :: ChangeBucket -> WorldM ( )
mainLoop cB =
   -- do some stuff
      -- it's probably fun
         -- using your cheeky wee WorldM accessors
   mainLoop cB -- recurse on the shared variable
```

Remember, your main thread is a transformer of World and IO so it can run 'atomically' and read the changeBucket.

Now, presuming you have a function that can incorporate a WorldChange into the existing WorldM your 'wait-for-client-input' thread can communicate with the main thread of the program, and it doesn't look too nasty.

38.3.2 Make the external changes to the state monadic themselves

However! Since all the state inside your main thread is now hidden from the rest of the program and you communicate through a one way channel --- data goes from the client to the server, but the mainLoop keeps its state a secret --- your client thread is never going to be able to make conditional choices about the environment - the client thread runs in IO but the main thread runs in WorldM.

So the REAL type of your shared variable is

```
type ChangeBucket =
   TVar [ WorldM ( Maybe WorldChange ) ]
```

This can be generated from the client-input thread, but you'll be able to include conditional statements inside the code, which is only evaluated against the state when it is run from your main thread

It all sounds a little random, but it's made my life a lot easier. Heres some real working code, based on this idea

- this takes commands from a client, and attempts change the object representing the client inside the game's state
- the output from this function is then written to a ChangeBucket (using the ChangeBucket definition in this section, above) and run inside the DState of the game's main loop.

(you might want to mentally substitute DState for WorldM)

```
  -- cmd is a command generated from parsing network input
  mkChange :: Unique -> Command -> DState ( Maybe WorldChange )
  mkChange oid cmd = do
     mp <- getObject oid -- this is maybe an object, as per the getObject
definition earlier in the article
     -- enter the maybe monad
     return $ do p <- mp -- if its Nothing, the rest will be nothing
                 case cmd of
                    -- but it might be something
```

```
                   Disconnect ->
                      Just $ RemoveObject oid
                   Move x y ->
                      Just $ UpdateObject $ DO ( oid )
                                               ( name p )
                                               ( speed p )
                                               ( netclient p )
                                               ( pos p )
                                               [ ( x , y ) ]
                                               ( method p )
```

A note and some more ideas.

Another design might just have

```
type ChangeBucket = TVar [ WorldM ( ) ]
```

And so just update the game world as they are run. I have other uses for the WorldM (Maybe Change) type.

So I conclude - All I have are my monads and my word so go use your monads imaginatively and write some computer games ;)

39 Advanced Haskell

40 Arrows

This chapter used to be the initial presentation of arrows, leading to the more detailed discussion in ../Understanding arrows/[1]. It has been replaced in that role by ../Arrow tutorial/[2].

40.1 Introduction

Arrows are a generalization of monads: every monad gives rise to an arrow, but not all arrows give rise to monads. They serve much the same purpose as monads -- providing a common structure for libraries -- but are more general. In particular they allow notions of computation that may be partially static (independent of the input) or may take multiple inputs. If your application works fine with monads, you might as well stick with them. But if you're using a structure that's very like a monad, but isn't one, maybe it's an arrow.

40.2 `proc` and the arrow tail

Let's begin by getting to grips with the arrows notation. We'll work with the simplest possible arrow there is (the function) and build some toy programs strictly in the aims of getting acquainted with the syntax.

Fire up your text editor and create a Haskell file, say toyArrows.hs:

```
{-# LANGUAGE Arrows #-}

import Control.Arrow (returnA)

idA :: a -> a
idA = proc a -> returnA -< a

plusOne :: Int -> Int
plusOne = proc a -> returnA -< (a+1)
```

These are our first two arrows. The first is the identity function in arrow form, and second, slightly more exciting, is an arrow that adds one to its input. Load this up in GHCi, using the -XArrows extension and see what happens.

```
% ghci -XArrows toyArrows.hs
   ---         --- -
```

```
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.4.1, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base-1.0 ... linking ... done.
Compiling Main             ( toyArrows.hs, interpreted )
Ok, modules loaded: Main.
*Main> idA 3
3
*Main> idA "foo"
"foo"
*Main> plusOne 3
4
*Main> plusOne 100
101
```

Thrilling indeed. Up to now, we have seen three new constructs in the arrow notation:

- the keyword `proc`
- `-<`
- the imported function `returnA`

Now that we know how to add one to a value, let's try something twice as difficult: adding TWO:

```
 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?

40.3 `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
```

40.4 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

41 Understanding arrows

We have permission to import material from the Haskell arrows page[1]. *See the talk page for details.*

41.1 The factory and conveyor belt metaphor

In this tutorial, we shall present arrows from the perspective of stream processors, using a factory metaphor as 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 -> 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 b c`, which we can pronounce as an arrow `a` from `b` to `c`.

41.2 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 (>> and >>=).

41.2.1 `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 -> c`, and adds conveyor belts to form an `a` arrow from `b` to `c`.

[1] `http://www.haskell.org/arrows`

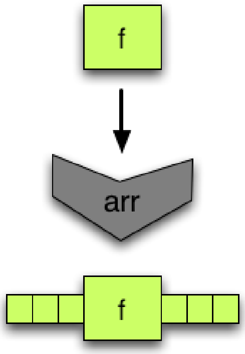


Figure 5 the `arr` robot

41.2.2 `(>>>)`

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.

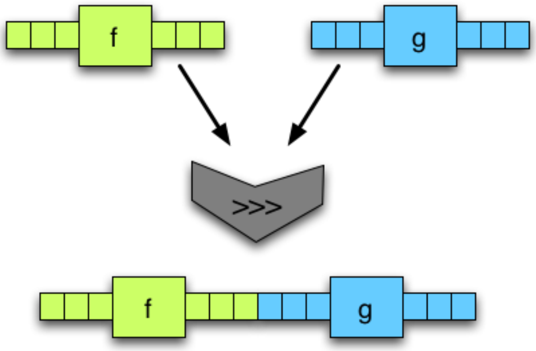


Figure 6 the `(>>>)` robot

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 b c`, the second arrow must be of type `a c d`. Here is the same diagram as above, but with things on the conveyor belts to help you see the issue with types.

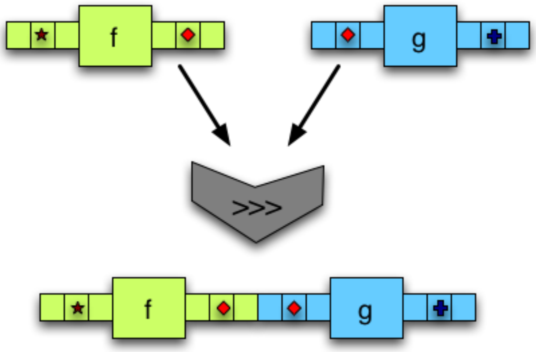


Figure 7 running the combined arrow

Exercises:
What is the type of the combined arrow?

41.2.3 `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.

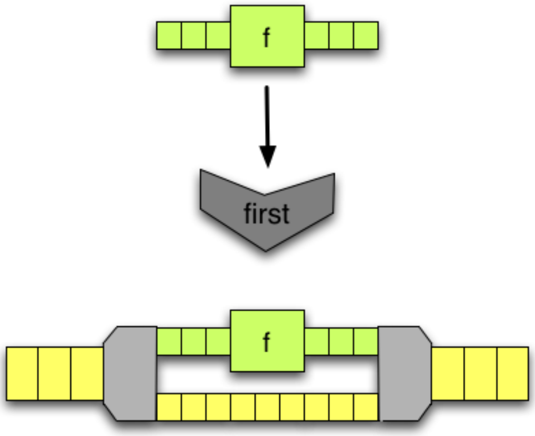


Figure 8 The `first` robot

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`.

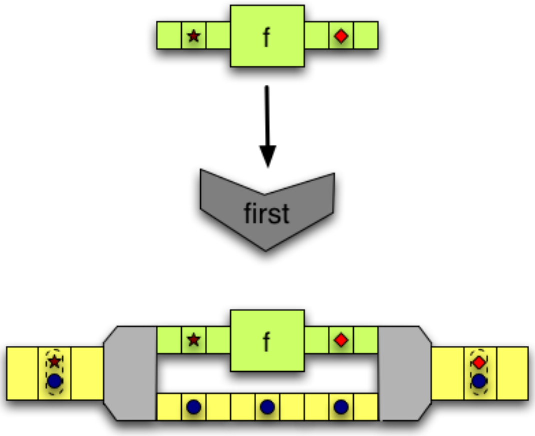


Figure 9 The combined arrow from the `first` robot

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 b 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?

41.2.4 `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.

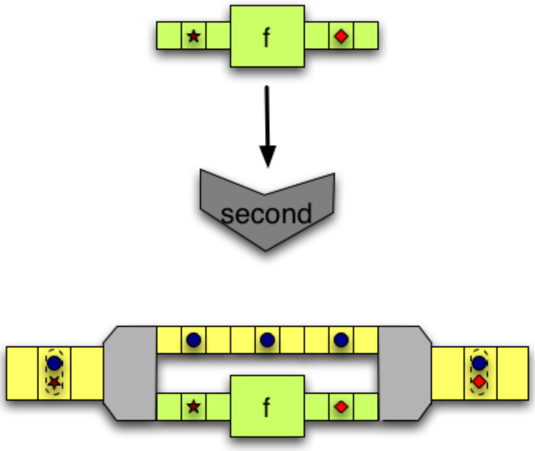


Figure 10 the `second` robot with things running

What makes the `second` robot interesting is that it can be derived from the previous robots! Strictly speaking, the only robots you need 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

41.2.5 ***

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

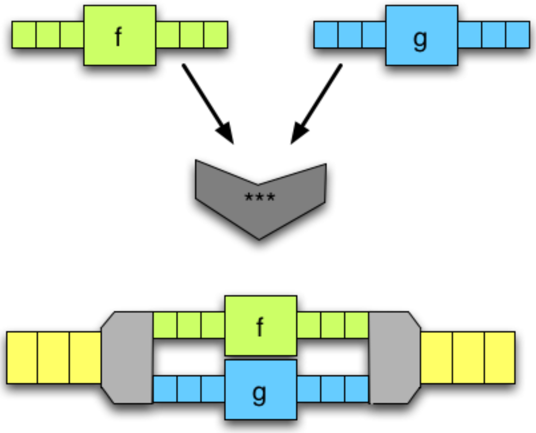


Figure 11 The (***) robot.

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?

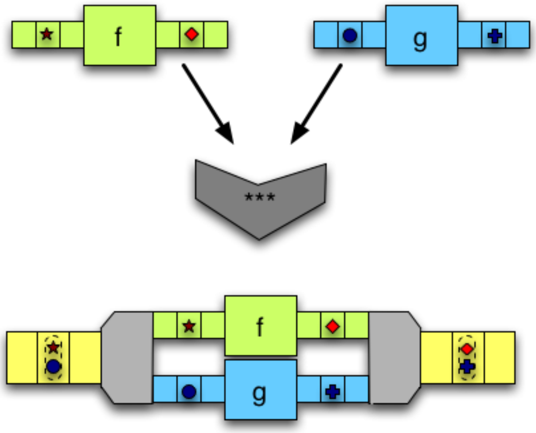


Figure 12 The (***) robot: running the combined arrow

Exercises:

1. What is the type of the `(***)` robot?
2. Given the `(>>>)`, `first` and `second` robots, implement the `(***)` robot.

41.2.6 `&&&`

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?

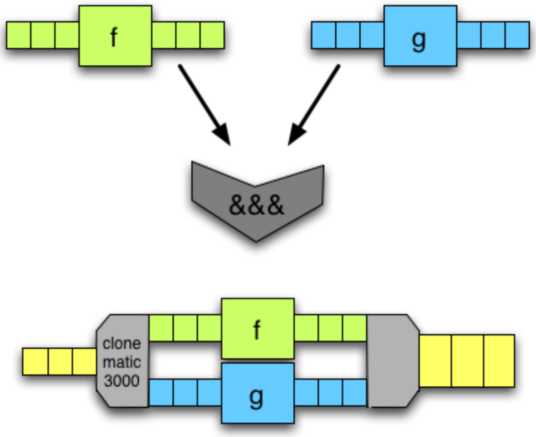


Figure 13 The `&&&` robot

The answer is simple: we clone the input and feed a copy into each machine!

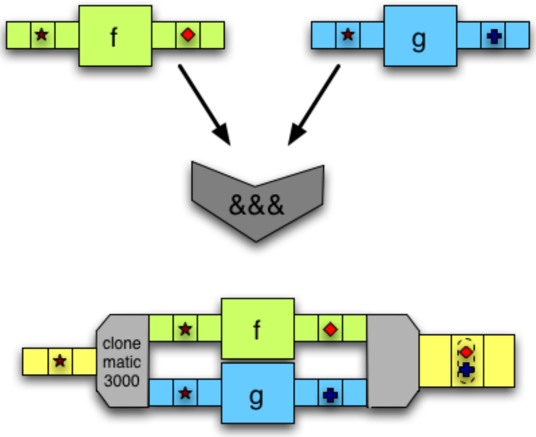


Figure 14 The `&&&` robot: the resulting arrow with inputs

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)`

41.3 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 implementation 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     -> (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[2] if you want the real deal.

[2] `http://darcs.haskell.org/packages/base/Control/Arrow.hs`

41.4 The arrow notation

In the companion ../Arrow tutorial/[3], 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 monad do-notation, but let's have a look.

- proc (arrow abstraction) is a kind of lambda, except that it constructs an arrow instead of a function.
- -< (arrow application) feeds the value of an expression into an arrow.

```
addA :: Arrow a => a b Int -> a b Int -> a b Int
addA f g = proc x -> do
                y <- f -< x
                z <- g -< x
                returnA -< y + z

addA :: Arrow a => a b Int -> a b Int -> a b Int
addA f g = arr (\ x -> (x, x)) >>>
           first f >>> arr (\ (y, x) -> (x, y)) >>>
           first g >>> arr (\ (z, y) -> y + z)

addA :: Arrow a => a b Int -> a b Int -> a b Int
addA f g = f &&& g >>> arr (\ (y, z) -> y + z)
```

TODO: Incorporate Arrows1[4] Arrows2[5]; harmonise with Stephen's Arrow Tutorial[6].

41.5 Maybe functor

It turns out that any monad can be made into an arrow. We'll go into that later on, but for now, *FIXME: transition*

41.6 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.

41.6.1 Stream processing

41.6.2 Avoiding leaks

Arrows were originally motivated by an efficient parser design found by Swierstra & Duponcheel[7].

[3] http://en.wikibooks.org/wiki/..%2FArrow%20tutorial%2F

[4] http://www.haskell.org/arrows/syntax.html

[5] Chapter 40 on page 257

[6] http://en.wikibooks.org/wiki/Haskell%2FStephensArrowTutorial

[7] Swierstra, Duponcheel. *Deterministic, error correcting parser combinators.* http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.2760

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 -> 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`.

⚠ **Warning**

The rest of this section needs to be verified

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) -> (Int,String)`, where the `Int` represents a count of the characters parsed so far. The table below 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 -> b`), and it consumes a bit of the input string, (informally, `[s] -> [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) -> (x,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.

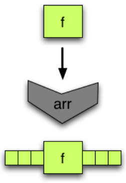
Anyway, let us put this together. Here is our S+D style parser for a single character:

```
charA :: Char -> Parser Char Char Char
charA c = P (SP False [c]) (DP (\(_,x:xs) -> (x,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 to 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)))
```

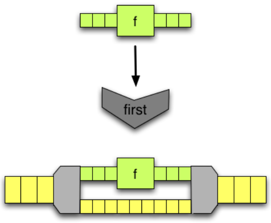


Figure 16

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 (DP (\((b,d),s) -> let (c, s') = p (b,s) in
((c,d),s'))))
```

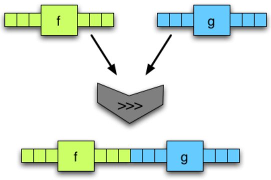


Figure 17

On the other hand, the implementation of `(>>>)` requires a little more thought. We want to take two parsers, and return 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?

If you look back at our Parser type and blank out the static parser section, you might notice that this looks a lot like the arrow instance for functions.

```
arr f = \(b, s) -> (f b, s)
first p = \((b, d), s) ->
            let (c, s') = p (b, s)
            in  ((c, d), s'))
p1 >>> p2 = p2 . p1
```

There's the odd s variable out for the ride, which makes the definitions look a little strange, but you can roughly see the outline of the conveyor belts and computing machines. Actually, what you see here is roughly the arrow instance for the State monad (let *f :: b -> c*, *p :: b -> State s c* and *.* actually be *<=<*.

That's fine, but we could have done that with bind in classic monadic style, and *first* would have just been an odd helper function that you could have easily pattern matched. But remember, our Parser type is not just the dynamic parser; it also contains the static parser.

```
arr f = SP True []
first sp = sp
(SP empty1 start1) >>> (SP empty2 start2) = (SP (empty1 && empty2)
        (if not empty1 then start1 else start1 `union` start2))
```

This is not at all a function, it's just pushing around some data types. But the arrow metaphor works for it too, and we wrap it up with the same names. And when we combine the two types, we get a two-for-one deal; the static parser data structure goes along for the ride along with the dynamic parser. The Arrow interface lets us transparently simultaneously compose and manipulate the two parsers as a unit, which we can then run as a traditional, unified function.

41.7 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 with them" versus "I chatted with them." 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.

41.8 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[8]) uses arrows for processing XML. There is a Wiki page in the Haskell Wiki with a somewhat Gentle Introduction to HXT[9].

41.9 See also

- Generalising Monads to Arrows - John Hughes
- `http://www.haskell.org/arrows/biblio.html`

[8] `http://www.fh-wedel.de/~si/HXmlToolbox/index.html`

[9] `http://www.haskell.org/haskellwiki/HXT`

41.10 Acknowledgements

This module uses text from *An Introduction to Arrows* by Shae Erisson, originally written for The Monad.Reader 4

42 Continuation passing style (CPS)

Continuation Passing Style (CPS for short) is a style of programming in which functions do not return values; rather, they pass control onto a *continuation*, which specifies what happens next. In this chapter, we are going to consider how that plays out in Haskell and, in particular, how CPS can be expressed with a monad.

42.1 What are continuations?

To dispel puzzlement, we will have a second look at an example from way back in the book, when we introduced the `($)` operator[1]:

```
> map ($ 2) [(2*), (4*), (8*)]
[4,8,16]
```

There is nothing out of ordinary about the expression above, except that it is a little quaint to write that instead of `map (*2) [2, 4, 8]`. The `($)` section makes the code appear backwards, as if we are applying a value to the functions rather than the other way around. And now, the catch: such an innocent-looking reversal is at heart of continuation passing style!

From a CPS perspective, `($ 2)` is a *suspended computation*: a function with general type `(a -> r) -> r` which, given another function as argument, produces a final result. The a -> r argument is the *continuation*; it specifies how the computation will be brought to a conclusion. In the example, the functions in the list are supplied as continuations via `map`, producing three distinct results. Note that suspended computations are largely interchangeable with plain values: `flip ($)` [2] converts any value into a suspended computation, and passing `id` as its continuation gives back the original value.

42.1.1 What are they good for?

There is more to continuations than just a parlour trick to impress Haskell newbies. They make it possible to explicitly manipulate, and dramatically alter, the control flow of a program. For instance, returning early from a procedure can be implemented with continuations. Exceptions and failure can also be handled with continuations - pass in a continuation for success, another continuation for fail, and invoke the appropriate continuation. Other possibilities include "suspending" a computation and returning to it at another time, and

[1] Chapter 19.8 on page 136

[2] That is, `\x -> ($ x)`, fully spelled out as `\x -> \k -> k x`

implementing simple forms of concurrency (notably, one Haskell implementation, Hugs, uses continuations to implement cooperative concurrency).

In Haskell, continuations can be used in a similar fashion, for implementing interesting control flow in monads. Note that there usually are alternative techniques for such use cases, especially in tandem with laziness. In some circumstances, CPS can be used to improve performance by eliminating certain construction-pattern matching sequences (i.e. a function returns a complex structure which the caller will at some point deconstruct), though a sufficiently smart compiler *should* be able to do the elimination [3].

42.2 Passing continuations

An elementary way to take advantage of continuations is to modify our functions so that they return suspended computations rather than ordinary values. We will illustrate how that is done with two simple examples.

42.2.1 `pythagoras`

Example:
A simple module, no continuations

```
-- We assume some primitives add and square for the example:

add :: Int -> Int -> Int
add x y = x + y

square :: Int -> Int
square x = x * x

pythagoras :: Int -> Int -> Int
pythagoras x y = add (square x) (square y)
```

Modified to return a suspended computation, `pythagoras` looks like this:

[3] attoparsec ^{`http://hackage.haskell.org/package/attoparsec-0.10.4.0/docs/Data-Attoparsec-ByteString.html`} is an example of performance-driven usage of CPS.

Example:
A simple module, using continuations

```
-- We assume CPS versions of the add and square primitives,
-- (note: the actual definitions of add_cps and square_cps are not
-- in CPS form, they just have the correct type)

add_cps :: Int -> Int -> ((Int -> r) -> r)
add_cps x y = \k -> k (add x y)

square_cps :: Int -> ((Int -> r) -> r)
square_cps x = \k -> k (square x)

pythagoras_cps :: Int -> Int -> ((Int -> r) -> r)
pythagoras_cps x y = \k ->
 square_cps x $ \x_squared ->
 square_cps y $ \y_squared ->
 add_cps x_squared y_squared $ k
```

How the `pythagoras_cps example` works:

1. square x and throw the result into the (\x_squared -> ...) continuation
2. square y and throw the result into the (\y_squared -> ...) continuation
3. add x_squared and y_squared and throw the result into the top level/program continuation `k`.

We can try it out in GHCi by passing `print` as the program continuation:

```
*Main> pythagoras_cps 3 4 print
25
```

If we look at the type of `pythagoras_cps` without the optional parentheses around `(Int -> r) -> r` and compare it with the original type of `pythagoras`, we note that the continuation was in effect added as an extra argument, thus justifying the "continuation passing style" moniker.

42.2.2 `thrice`

Example:
A simple higher order function, no continuations

```
thrice :: (a -> a) -> a -> a
thrice f x = f (f (f x))
```

```
*Main> thrice tail "foobar"
"bar"
```

A higher order function such as `thrice`, when converted to CPS, takes as arguments functions in CPS form as well. Therefore, `f :: a -> a will becomef`_cps :: a -> ((a -> r) -> r), and the final type will be `thrice_cps :: (a -> ((a -> r) -> r)) -> a -> ((a -> r) -> r)`. The rest of the definition follows quite naturally from the types - we replace `f` by the CPS version, passing along the continuation at hand.

Example:
A simple higher order function, with continuations

```
thrice_cps :: (a -> ((a -> r) -> r)) -> a -> ((a -> r) -> r)
thrice_cps f_cps x = \k ->
 f_cps x $ \fx ->
 f_cps fx $ \ffx ->
 f_cps ffx $ k
```

42.3 The `Cont` monad

Having continuation-passing functions, the next step is providing a neat way of composing them, preferably one which does not require the long chains of nested lambdas we have seen just above. A good start would be a combinator for applying a CPS function to a suspended computation. A possible type for it would be:

```
chainCPS :: ((a -> r) -> r) -> (a -> ((b -> r) -> r)) -> ((b -> r) -> r)
```

(You may want to try implementing it before reading on. Hint: start by stating that the result is a function which takes a `b -> r` continuation; then, let the types guide you.)

And here is the implementation:

```
chainCPS s f = \k -> s $ \x -> f x $ k
```

We supply the original suspended computation `s` with a continuation which makes a new suspended computation (produced by `f`) and passes the final continuation `k` to it. Unsurprisingly, it mirrors closely the nested lambda pattern of the previous examples.

Doesn't the type of `chainCPS` look familiar? If we replace `(a -> r) -> r` with (Monad m) => m a and `(b -> r) -> r` with (Monad m) => m b we get the `(>>=)` signature. Furthermore, our old friend `flip ($)` plays a `return`-like role, in that it makes a suspended computation out of a value in a trivial way. Lo and behold, we have a monad! All we need now [4] is a `Cont r a` type to wrap suspended computations, with the usual wrapper and unwrapper functions.

```
cont :: ((a -> r) -> r) -> Cont r a
runCont :: Cont r a -> (a -> r) -> r
```

The monad instance for `Cont` follows directly from our presentation, the only difference being the wrapping and unwrapping cruft:

```
instance Monad (Cont r) where
    return x = cont ($ x)
```

[4] Beyond verifying that the monad laws hold, which is left as an exercise to the reader.

```
s >>= f  = cont $ \c -> runCont s $ \x -> runCont (f x) c
```

The end result is that the monad instance makes the continuation passing (and thus the lambda chains) implicit. The monadic bind applies a CPS function to a suspended computation, and `runCont` is used to provide the final continuation. For a simple example, the Pythagoras example becomes:

Example:
The `pythagoras` example, using the Cont monad

```
-- Using the Cont monad from the transformers package.
import Control.Monad.Trans.Cont

add_cont :: Int -> Int -> Cont r Int
add_cont x y = return (add x y)

square_cont :: Int -> Cont r Int
square_cont x = return (square x)

pythagoras_cont :: Int -> Int -> Cont r Int
pythagoras_cont x y = do
    x_squared <- square_cont x
    y_squared <- square_cont y
    add_cont x_squared y_squared
```

42.4 callCC

While it is always pleasant to see a monad coming forth naturally, a hint of disappointment might linger at this point. One of the promises of CPS was precise control flow manipulation through continuations. And yet, after converting our functions to CPS we promptly hid the continuations behind a monad. To rectify that, we shall introduce `callCC`, a function which gives us back explicit control of continuations - but only where we want it.

`callCC` is a very peculiar function; one that is best introduced with examples. Let us start with a trivial one:

Example:
`square` using `callCC`

```
-- Without callCC
square :: Int -> Cont r Int
square n = return (n ^ 2)

-- With callCC
squareCCC :: Int -> Cont r Int
squareCCC n = callCC $ \k -> k (n ^ 2)
```

The argument passed to `callCC` is a function, whose result is a suspended computation (general type `Cont r a`) which we will refer to as "the `callCC` computation". *In principle*, the `callCC` computation is what the whole `callCC` expression evaluates to. The caveat, and what makes `callCC` so special, is due to `k`, the argument to the argument. It is a function which acts as an *eject button*: calling it anywhere will lead to the value passed to it being made into a suspended computation, which then is inserted into control flow at the point of the `callCC` invocation. That happens unconditionally; in particular, whatever follows a `k` invocation in the `callCC` computation is summarily discarded. From another perspective, `k` captures *the rest of the computation* following the `callCC`; calling it throws a value into the continuation at that particular point ("callCC" stands for "call with current continuation"). While in this simple example the effect is merely that of a plain `return`, `callCC` opens up a number of possibilities, which we are now going to explore.

42.4.1 Deciding when to use `k`

`callCC` gives us extra power over what is thrown into a continuation, and when that is done. The following example begins to show how we can use this extra power.

Example:
Our first proper `callCC` function

```
foo :: Int -> Cont r String
foo x = callCC $ \k -> do
    let y = x ^ 2 + 3
    when (y > 20) $ k "over twenty"
    return (show $ y - 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 `callCC` computation (and, in this case, from the whole function) immediately, throwing the string `"over twenty"` into the continuation that will be passed to `foo`. If not, then we subtract four from our previous computation, `show` it, and throw it into the continuation. Remarkably, `k` here is used just like the 'return' *statement* from an imperative language, that immediately exits the function. And yet, this being Haskell, `k` is just an ordinary first-class function, so you can pass it to other functions like `when`, 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 effect, that makes `k` a lot like an '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

```
quux :: Cont r Int
quux = callCC $ \k -> do
    let n = 5
    k n
    return 25
```

`quux` will return `5`, and not `25`, because we pop out of `quux` before getting to the `return 25` line.

42.4.2 Behind the scenes

We have deliberately broken a trend here: normally when we introduce a function we give its type straight away, but in this case we chose not to. The reason is simple: the type is pretty complex, and it does not immediately give insight into what the function does, or how it works. After the initial presentation of `callCC`, however, we are in a better position to tackle it. Take a deep breath...

```
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
```

We can make sense of that based on what we already know about `callCC`. The overall result type and the result type of the argument have to be the same (i.e. `Cont r a`), as in the absence of an invocation of `k` the corresponding result values are one and the same. Now, what about the type of `k`? As mentioned above, `k`'s argument is made into a suspended computation inserted at the point of the `callCC` invocation; therefore, if the latter has type `Cont r a` `k`'s argument must have type `a`. As for `k`'s result type, interestingly enough it doesn't matter as long as it is in the same `Cont r` monad; in other words, the `b` stands for an arbitrary type. That happens because the suspended computation made out of the `a` argument will receive whatever continuation follows the `callCC`, and so the continuation taken by `k`'s result is irrelevant.

Note:
The arbitrariness of `k`'s result type explains why the following variant of the useless line example leads to a type error:

```
quux :: Cont r Int
quux = callCC $ \k -> do
   let n = 5
   when True $ k n
   k 25
```

`k`'s result type *could* be anything; however, the `when` constrains it to `Cont r ()`, and so the closing `k 25` does not match the result type of `quux`. The solution is very simple: replace the final `k` by a plain old `return`.

To conclude this section, here is the implementation of `callCC`. Can you identify `k` in it?

```
callCC f = cont $ \h -> runCont (f (\a -> cont $ \_ -> h a)) h
```

The code is far from obvious. However, the amazing fact is that the implementations of `callCC`, `return` and `(>>=)` for `Cont` can be produced automatically from their type signatures - Lennart Augustsson's Djinn `http://lambda-the-ultimate.org/node/1178` is a program that will do this for you. See Phil Gossett's Google tech talk: `http://www.youtube.com/watch?v=h0OkptwfX4g` for background on the theory behind Djinn; and Dan Piponi's article: `http://www.haskell.org/wikiupload/1/14/TMR-Issue6.pdf` which uses Djinn in deriving continuation passing style.

42.5 Example: a complicated control structure

We will now look at some more realistic examples of control flow manipulation. The first one, presented below, was originally taken from the "The Continuation monad" section of the All about monads tutorial[5], 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                            -- define "exit1"
        when (n < 10) (exit1 (show n))
        let ns = map digitToInt (show (n `div` 2))
        n' <- callCC $ \exit2 -> do                         -- define "exit2"
            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
```

`fun` is a function that takes an integer `n`. The implementation uses `Cont` and `callCC` to set up a control structure using `Cont` and `callCC` that does different things based on the range that `n` falls in, as stated by the comment at the top. Let us dissect it:

[5] `http://www.haskell.org/haskellwiki/All_about_monads`

1. Firstly, the (`runCont` id) at the top just means that we run the Cont block that follows with a final continuation of id (or, in other words, we extract the value from the suspended computation unchanged). That 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:
    a) If n is less than 10, we exit straight away, just showing n.
    b) If not, we proceed. We construct a list, ns, of digits of n `div` 2.
    c) n' (an Int) gets bound to the result of the following inner callCC do-block.
        i. 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.
        ii. If n `div` 2 has less than 5 digits, we pop out of the inner do-block returning the original n.
        iii. 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).
        iv. Otherwise, we end the inner do-block, returning the sum of the digits of n `div` 2.
    d) 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.

42.6 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 (more precisely, a function which takes an error-throwing function and results in the computation) and an error handler as the second parameter. This example takes advantage of the generic `MonadCont` class [6] which covers both `Cont` and the corresponding `ContT` transformer by default, as well as any other continuation monad which instantiates it.

Example:
General `try` using continuations.

```
import Control.Monad.Cont

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
```

And here is our `try` in action:

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
```

In this example, error throwing means escaping from an enclosing `callCC`. The `throw` in `sqrtIO` jumps out of `tryCont`'s inner `callCC`.

42.7 Example: coroutines

[6] Found in the `mtl` package, module Control.Monad.Cont ^{`http://hackage.haskell.org/packages/archive/mtl/2.1.2/doc/html/Control-Monad-Cont.html`} .

43 Zippers

43.1 Theseus and the Zipper

43.1.1 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 nodded. "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 had been a hero in the labyrinth back then on Crete[1]. 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)
```

[1] Ian Stewart. *The true story of how Theseus found his way out of the labyrinth.* Scientific American, February 1991, page 137.

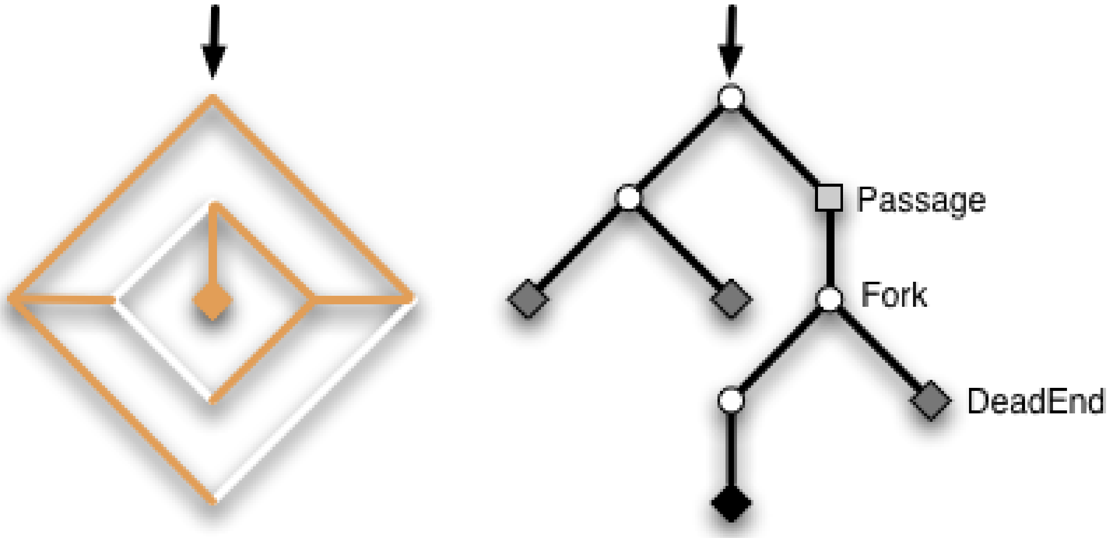


Figure 18 An example labyrinth and its representation as tree.

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]
```

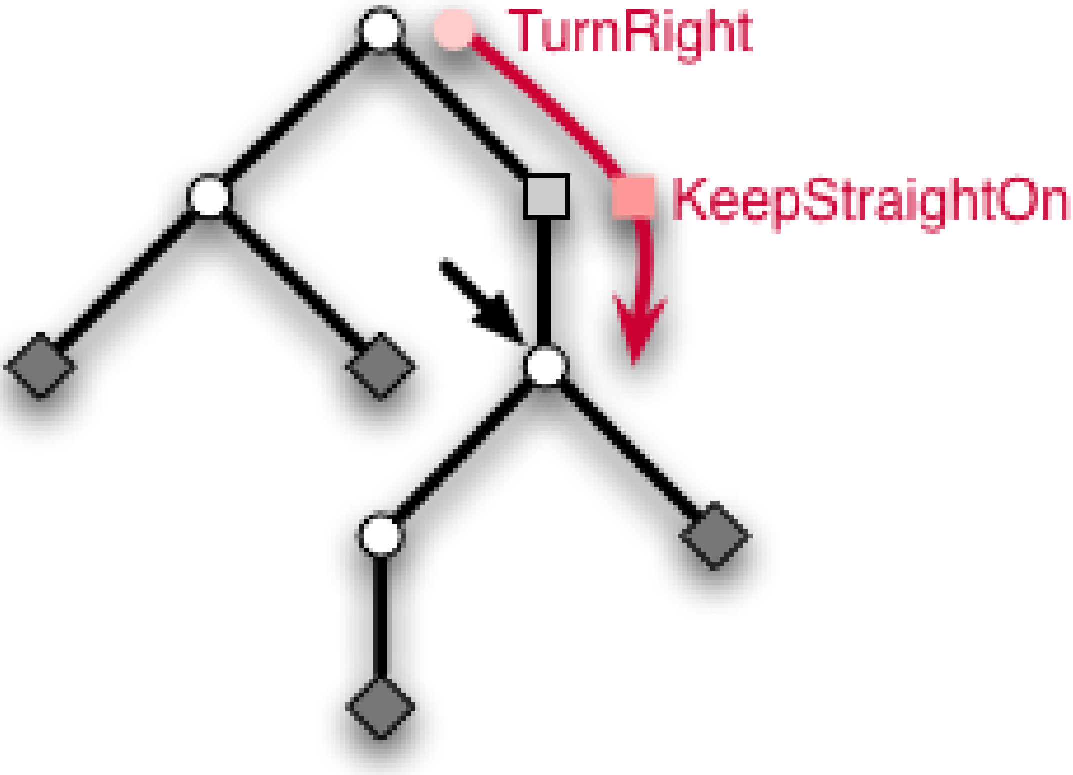


Figure 19 Representation of the player's position by Ariadne's thread.

"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?"

43.1.2 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 hope!" 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[2]. "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[3]. 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

[2] 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`}

[3] 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 ^{Chapter43.2 on page 296}.

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[4][5]. 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)
```

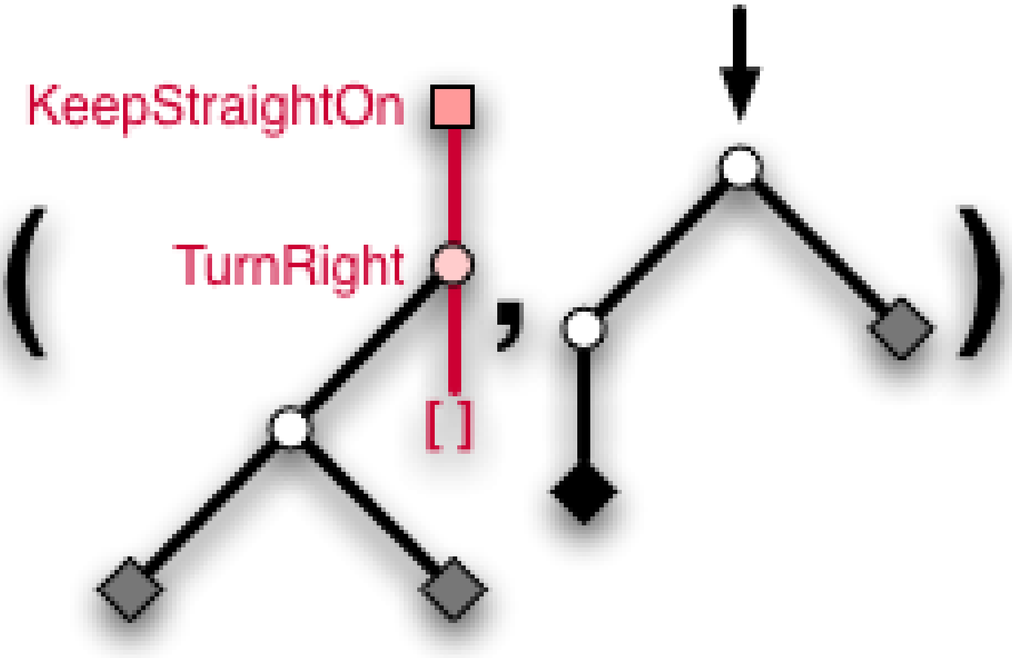


Figure 20 The zipper is a pair of Ariadne's thread and the current sub-labyrinth that the player stands on top. The main thread is colored red and has sub-labyrinths attached to it, such that the whole labyrinth can be reconstructed from the pair.

[4] 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.

[5] 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).

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
```

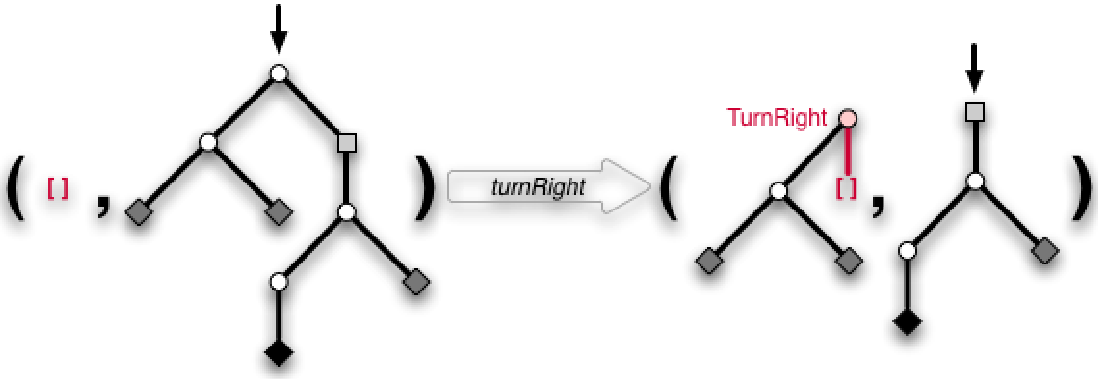


Figure 21 Taking the right subtree from the entrance. Of course, the thread is initially empty. Note that the thread runs backwards, i.e. the topmost segment is the most recent.

"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
```

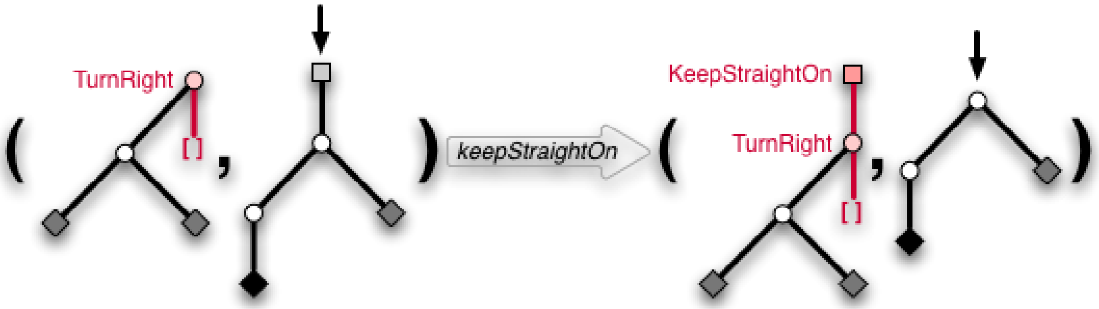


Figure 22 Now going down a passage.

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 were 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.

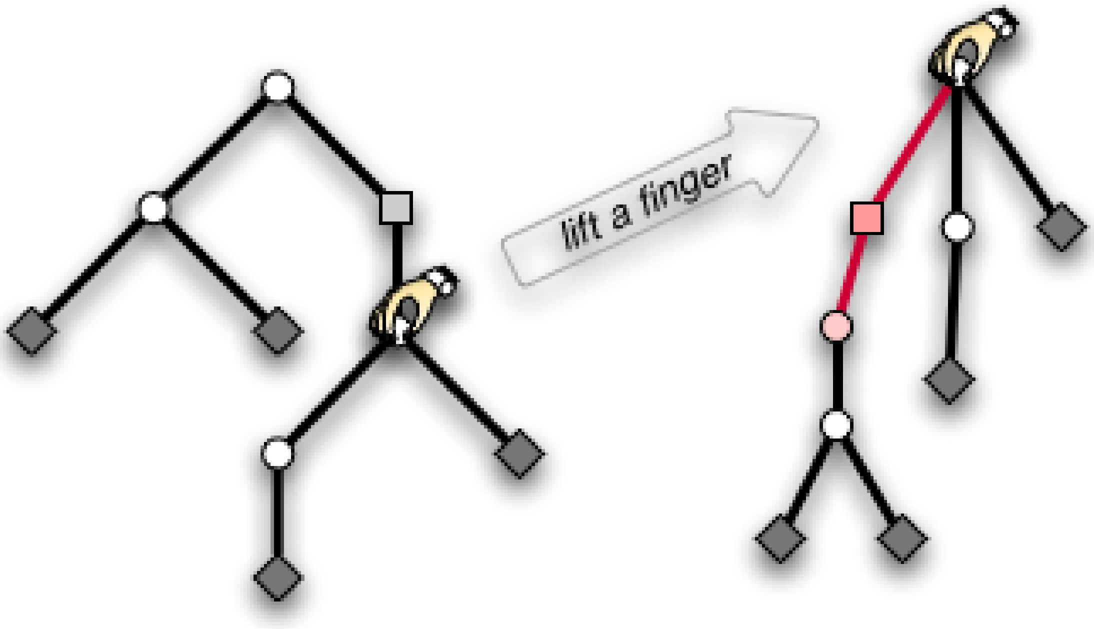


Figure 23 Grab the focus with your finger, lift it in the air and the hanging branches will form new tree with your finger at the top, ready to be structured by an algebraic data type.

"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.

43.2 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`.

43.2.1 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[6]. 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/[7] 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

$$\mathit{Tree2} = 1 + \mathit{Tree2} \times \mathit{Tree2}$$

.

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

$$\mathit{Branch2} = \mathit{Tree2} + \mathit{Tree2} \cong 2 \times \mathit{Tree2}$$

.

Similarly, the thread for a ternary tree

$$\mathit{Tree3} = 1 + \mathit{Tree3} \times \mathit{Tree3} \times \mathit{Tree3}$$

[6] Conor Mc Bride. *The Derivative of a Regular Type is its Type of One-Hole Contexts.* Available online. PDF ^{http://strictlypositive.org/diff.pdf}

[7] http://en.wikibooks.org/wiki/..%2FGeneric%20Programming%2F

has branches of type

$$Branch3 = 3 \times Tree3 \times Tree3$$

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.

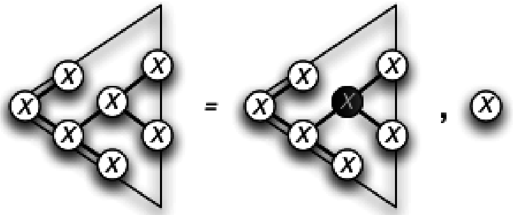


Figure 24 Removing a value of type X from a $Tree\,X$ leaves a hole at that position.

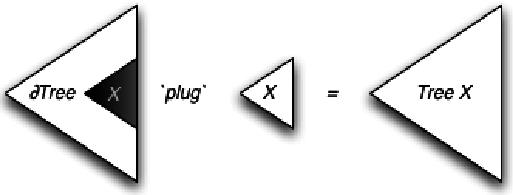
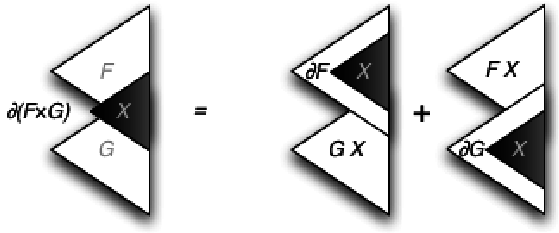
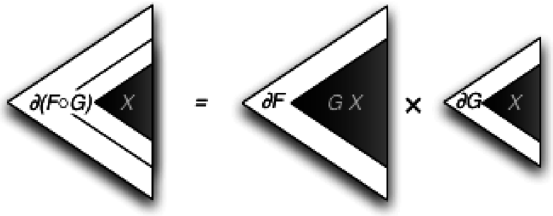


Figure 25 A more abstract illustration of plugging X into a one-hole context.

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[8]. 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.

[8] This phenomenon already shows up with generic tries.

One-hole context		Illustration
$\partial(F+G)$	$=\partial F+\partial G$	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.
$\partial(F\times G)$	$=F\times\partial G+\partial F\times G$	 ∂(FxG) F X G = ∂F X G X + F X ∂G X **Figure 26** The hole in a one-hole context of a pair is either in the first or in the second component.
$\partial(F\circ G)$	$=(\partial F\circ G)\times\partial G$	 **Figure 27** **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 $(\partial F\,X)\times X\to F\,X$.

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.

43.2.2 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

$$\mathit{Zipper}_F = \mu F \times \mathit{Context}_F$$

.

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

$$\mathit{Context}_F = \mathit{List}\,(\partial F\,(\mu F))$$

.

or equivalently

$$\mathit{Context}_F = 1 + \partial F\,(\mu F) \times \mathit{Context}_F$$

.

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

$$\mathit{Node}\,A = \mu X.\,\mathit{NodeF}_A\,X$$

of the functor

$$\mathit{NodeF}_A\,X = A + A \times X + A \times X \times X$$

.

In other words, we have

$$\mathit{Node}\,A \cong \mathit{NodeF}_A\,(\mathit{Node}\,A) \cong A + A \times \mathit{Node}\,A + A \times \mathit{Node}\,A \times \mathit{Node}\,A$$

.

The derivative reads

$$\partial_X(\mathit{NodeF}_A\,X) \cong A + 2 \times A \times X$$

and we get

$$\partial\mathit{NodeF}_A\,(\mathit{Node}\,A) \cong A + 2 \times A \times \mathit{Node}\,A$$

.

Thus, the context reads

$$\mathit{Context}_{\mathit{NodeF}} \cong \mathit{List}\,(\partial\mathit{NodeF}_A\,(\mathit{Node}\,A)) \cong \mathit{List}\,(A + 2 \times A \times (\mathit{Node}\,A))$$

.

Comparing with

```
data Branch a  = KeepStraightOn a
               | TurnLeft  a (Node a)
               | TurnRight a (Node a)
type 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?

43.2.3 Differentation 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 $\mathit{Node}\,A$:

$$\begin{array}{rcc} \partial_A(\mathit{Node}\,A) & = & \partial_A(A + A \times \mathit{Node}\,A + A \times \mathit{Node}\,A \times \mathit{Node}\,A) \\ & \cong & 1 + \mathit{Node}\,A + \mathit{Node}\,A \times \mathit{Node}\,A \\ & & +\partial_A(\mathit{Node}\,A) \times (A + 2 \times A \times \mathit{Node}\,A). \end{array}$$

Of course, expanding $\partial_A(\mathit{Node}\,A)$ further is of no use, but we can see this as a fixed point equation and arrive at

$$\partial_A(\mathit{Node}\,A) = \mu X.\,T\,A + S\,A \times X$$

with the abbreviations

$$T\,A = 1 + \mathit{Node}\,A + \mathit{Node}\,A \times \mathit{Node}\,A$$

and

$$S\,A = A + 2 \times A \times \mathit{Node}\,A$$

.

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:

$$\partial_A(Node\,A) \cong T\,A \times (\mu X.\,1 + S\,A \times X) = T\,A \times List\,(S\,A)$$

.

Comparing with the zipper we derived in the last paragraph, we see that the list type is our context

$$List\,(S\,A) \cong Context_{NodeF}$$

and that

$$A \times T\,A \cong Node\,A$$

.

In the end, we have

$$Zipper_{NodeF} \cong \partial_A(Node\,A) \times A$$

.

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?

43.2.4 Differentation with respect to functions of the argument

When finding the type of a one-hole context one does d f(x)/d x. It is entirely possible to solve expressions like d f(x)/d g(x). For example, solving d x^4 / d x^2 gives 2x^2 , a two-hole context of a 4-tuple. The derivation is as follows let u=x^2 d x^4 / d x^2 = d u^2 /d u = 2u = 2 x^2 .

43.2.5 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

$$\mathit{Tree}\,A = \mu X.\, A + X \times X$$

.

The zipper can focus on subtrees whose top is `Bin` or `Leaf` but the hole of one-hole context of $\mathit{Tree}\,A$ may only focus a `Leaf`s because this is where the items of type A reside. The derivative of $\mathit{Node}\,A$ only turned out to be the zipper because every top of a subtree is always decorated with an A.

Exercises:

1. Surprisingly, $\partial_A(\mathit{Tree}\,A) \times A$ and the zipper for $\mathit{Tree}\,A$ 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.

43.2.6 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

$$\partial_X F\,X \to (X \multimap F\,X)$$

where $A \multimap B$ denotes a linear function, one that does not duplicate or ignore its argument, as 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 $X \to F\,X$.

43.3 See Also

w:Zipper (data structure)[9]

- Zipper[10] on the haskell.org wiki
- Generic Zipper and its applications[11]
- Zipper-based file server/OS[12]
- Scrap Your Zippers: A Generic Zipper for Heterogeneous Types[13]

[9] http://en.wikipedia.org/wiki/Zipper%20%28data%20structure%29

[10] http://www.haskell.org/haskellwiki/Zipper

[11] http://okmij.org/ftp/Computation/Continuations.html#zipper

[12] http://okmij.org/ftp/Computation/Continuations.html#zipper-fs

[13] http://www.michaeldadams.org/papers/scrap_your_zippers/

44 Applicative Functors

Applicative functors are functors with some extra properties; the most important one is that it allows you to apply functions inside the functor (hence the name) to other values. First we do a quick recap of functors, next we will see the applicative functors, and for what structure they are used. After that, we'll see that we can get an applicative functor out of every monad, and then we'll see an example of an applicative functor that is not a monad, but still very useful.

44.1 Functors

Functors, or instances of the typeclass `Functor`, are some of the most often used structures in Haskell. Typically, they are structures that "can be mapped over". Here is the class definition of `Functor`:

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

The most well-known functor is the list, where `fmap` corresponds to `map`. Another example is `Maybe`:

```
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap _ Nothing  = Nothing
```

Typically, for all tree-like structures you can write an instance of `Functor`.

Exercises:
Define instances of `Functor` for the following types:

1. The rose tree type `Tree`, defined by:
   `data Tree a = Node a [Tree a]`
2. `Either e` for a fixed `e`.
3. The function type `((->) t)`. In this case, `f a` will be `(t -> a)`

44.2 Applicative Functors

Applicative functors are functors with extra properties: you can apply functions inside a functor to values that can be inside the functor or not. We will first look at the definition, then at some instances of applicative functors and their most important use.

44.2.1 Definition

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

The `pure` function lifts any value inside the functor. `(<*>)` changes a function inside the functor to a function over values of the functor. The functor should satisfy some laws:

```
pure id <*> v = v                            -- Identity
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- Composition
pure f <*> pure x = pure (f x)               -- Homomorphism
u <*> pure y = pure ($ y) <*> u              -- Interchange
```

And the `Functor` instance should satisfy the following law:

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

44.2.2 Instances

As we have seen the `Functor` instance of `Maybe`, let's try to make it `Applicative` as well.

The definition of `pure` is easy. It is `Just`. Now the definition of `(<*>)`. If any of the two arguments is `Nothing`, the result should be `Nothing`. Otherwise, we extract the function and its argument from the two `Just` values, and return `Just` the function applied to its argument:

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

Exercises:

1. Check that the Applicative laws hold for this instance for `Maybe`
2. Write `Applicative` instances for
   1. `Either e`, for a fixed `e`
   2. `((->) t)`, for a fixed `t`

44.2.3 Using Applicative Functors

The use of `(<*>)` may not be immediately clear, so let us look at some example that you may have come across yourself.

Suppose we have the following function:

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

But instead of `Int`s, we want to apply this function to values of type `Maybe Int`. Because you've seen this problem before, you decide to write a function `fmap2`:

```
fmap2 :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
fmap2 f (Just x) (Just y) = Just (f x y)
fmap2 _ _        _        = Nothing
```

You are happy for a while, but then you find that `f` really needs another `Int` argument. But now, `fmap2` isn't sufficient anymore. You need another function to accommodate for this extra parameter:

```
fmap3 :: (a -> b -> c -> d) -> Maybe a -> Maybe b -> Maybe c -> Maybe d
fmap3 f (Just x) (Just y) (Just z) = Just (f x y z)
fmap3 _ _        _        _        = Nothing
```

This is all good as such, but what if `f` suddenly needs 4 arguments, or 5, or 10?

Here is where `(<*>)` comes in. Look at what happens if you write `fmap f`:

```
 f      :: (a -> b -> c)
 fmap   :: Functor f => (d -> e) -> f d -> f e
 fmap f :: Functor f =>             f a -> f (b -> c)    -- Identify d with a,
and e with (b -> c)
```

Now the use of `(<*>)` becomes clear. Once we have the final `f (b -> c)`, we can use it to get a `(f b -> f c)`. And indeed:

```
fmap2 f a b = f `fmap` a <*> b
fmap3 f a b c = f `fmap` a <*> b <*> c
fmap4 f a b c d = f `fmap` a <*> b <*> c <*> d
```

To make it look more pretty, the Control.Applicative[1] library defines `(<$>)` as a synonym of `fmap`. The ultimate result is that the code is much cleaner to read and easier to adapt and reuse.

```
fmap2 f a b = f <$> a <*> b
fmap3 f a b c = f <$> a <*> b <*> c
fmap4 f a b c d = f <$> a <*> b <*> c <*> d
```

Anytime you feel the need to define different higher order functions to accommodate for function-arguments with a different number of arguments, think about how defining a proper instance of `Applicative` can make your life easier.

[1] http://hackage.haskell.org/packages/archive/base/4.1.0.0/doc/html/Control-Applicative.html

Of course, Control.Applicative[2] provides the above functions as convenience, under the names of `liftA` to `liftA3`.

44.3 Monads and Applicative Functors

The type of `pure` may look familiar. Indeed, if you change its typeclass restriction from:

```
Applicative f => a -> f a
```

to

```
Monad m       => a -> m a
```

it has exactly the same type as `return`.

As a matter of fact, every instance of `Monad` can also be made an instance of `Applicative`. Here are the definitions that can be used:

```
pure  = return
(<*>) = ap
```

Here, `ap` is defined as:

```
ap f a = do
  f' <- f
  a' <- a
  return (f' a')
```

It is also defined in Control.Monad[3]

Exercises:
Check that the `Applicative` instance of `Maybe` can be obtained by the above transformation.

44.4 ZipLists

Let's come back now to the idea that `Applicative` can make life easier. Perhaps the best known example of this are the different `zipWithN` functions of Data.List[4]. It looks exactly like the kind of pattern that an applicative functor would be useful for.

[2] http://hackage.haskell.org/packages/archive/base/4.1.0.0/doc/html/Control-Applicative.html

[3] http://hackage.haskell.org/packages/archive/base/4.1.0.0/doc/html/Control-Monad.html

[4] http://hackage.haskell.org/packages/archive/base/4.1.0.0/doc/html/Data-List.html

For technical reasons, we can not define a `Applicative` instance for `[]`, as it already has one defined. This instance does something completely different. `fs <*> xs` takes all functions from `fs` and applies them to all values in `xs`. To remedy this, we create a wrapper:

```
newtype ZipList a = ZipList [a]

instance Functor ZipList where
  fmap f (ZipList xs) = ZipList (map f xs)
```

To make this an instance of `Applicative` with the expected behaviour, we shall first look at the definition of `(<*>)`, as it follows quite straightforward from what we want to do. The `(<*>)` operator takes a list of functions and a list of values, and it should apply the functions to the values in a pairwise way. This sounds a lot like `zipWith ($)`, we just need to add the `ZipList` wrapper:

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

Now we only need to define `pure`. If we define it like this:

```
pure x = ZipList [x]
```

it won't satisfy the Identity law, `pure id <*> v = v`, since `v` can contain more than one element, and `zipWith` only returns a list of the smaller of the two input sizes. Since we don't know how many elements `v` has, the safest way is to let `pure` return a list of infinite elements. Now our instance declaration is complete:

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

44.5 References

Control.Applicative[5]

[5] http://hackage.haskell.org/packages/archive/base/4.1.0.0/doc/html/Control-Applicative.html

45 Monoids

In earlier parts of the book, we have made a few passing allusions to monoids and the `Monoid` type class (most notably when discussing MonadPlus[1]). Now it is time to give them a more detailed look, and show what makes them useful.

45.1 Introduction

A **monoid** (`m`, `mappend`, `mempty`) is a type `m` together with an associative operation `mappend :: m -> m -> m` (also called `(<>)` in Haskell) that combines two elements and a zero element `mempty :: m` which is the neutral element of `mappend`: Before describing them more formally let's see monoids in action.

45.1.1 Examples

As an information introduction, let's take a look at a common pattern:

```
> (5 + 6) + 10 == 5 + (6 + 10)
True
> (5 * 6) * 10 == 5 * (6 * 10)
True
> ("Hello" ++ " ") ++ "world!" == "Hello" ++ (" " ++ "world!")
True
```

This property is called *associativity* and doesn't just hold for those selected values but for all integers under addition, all integers under multiplication and all lists under concatenation, let's look at a few more examples:

```
> 255 + 0 == 255 && 0 + 255 == 255
True
> 255 * 1 == 255 && 1 * 255 == 255
True
> [1,2,3] ++ [] == [1,2,3] && [] ++ [1,2,3] == [1,2,3]
True
```

Here `0` is the identity element when adding integers, `1` is the identity when multiplying them and `[]` is the identity when appending two lists: This should not surprise you. At this point you may have realised the connection.

- Integers form a monoid under addition where 0 is the unit: (`Integer`, `(+)`, `0`)
- Integers form a monoid under multiplication where 1 is the unit: (`Integer`, `(*)`, `1`)
- Lists form a monoid under concatenation: (`[a]`, `(++)`, `[]`)

[1] Chapter 35.6 on page 226

Since Integers form monoids under two distinct operations we instead create two new datatypes: `Sum` for addition and `Product` for multiplication.

```
> import Data.Monoid
> Sum 5 <> Sum 6 <> Sum 10
Sum {getSum = 21}
> mconcat [Sum 5, Sum 6, Sum 10]
Sum {getSum = 21}
> getSum $ mconcat $ map Sum [5, 6, 10]
21
> getProduct $ mconcat $ map Product [5, 6, 10]
300
> mconcat ["5", "6", "10"]
"5610"
```

But how is this useful?

45.1.2 Generalizing functions

We can generalize a function that concatenates three lists:

```
threeConcat :: [a] -> [a] -> [a] -> [a]
threeConcat a b c = a ++ b ++ c
```

with:

```
threeConcat' :: Monoid m => m -> m -> m -> m
threeConcat' a b c = a <> b <> c

-- > threeConcat' "Hello" " " "world!"
-- "Hello world!"
-- > threeConcat' (Sum 5) (Sum 6) (Sum 10)
-- Sum {getSum = 21}
```

Other functions like `fold :: (Foldable t, Monoid m) => t m -> m` from `Data.Foldable` use properties of monoids to reduce any foldable structure containing monoids into a single monoidal value:

```
> fold ["Hello", " ", "world!"]
"Hello world!"
> fold (Just (Sum 10))
Sum {getSum = 10}
> fold Nothing :: Sum Integer
Sum {getSum = 0}
```

This way if we create our own container (like trees) containing our own monoidal data type, we can immediately make use of previously defined functions like `fold`!

Exercises:

1. There is a *second* possible instance of Monoid for Integer, using multiplication instead of addition as the combining operation. Implement that instance.[a]
2. It is possible to define mempty and mappend in terms of mconcat. Write the definition of this. Figure out whether or not your definition automatically fullfills the above laws.

[a] If you check the documentation for Data.Monoid, you will find that both instances are defined for types in Num, through the means of two newtype wrappers.

45.2 Haskell definition and laws

The Monoid type class (from Data.Monoid[2]) captures this general concept:

```
class Monoid a where
    mempty  :: a
    mappend :: a -> a -> a
    mconcat :: [a] -> a
    mconcat = foldr mappend mempty
```

and lists are defined as:

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

and the third method mconcat is provided as bonus with a default implementation of mconcat = foldr (++) [] which is equivalent to concat for lists.

It is legitimate to think of monoids as types which support *appending*, though some poetic license with respect to what is meant by "appending" is advisable, as the Monoid definition is extremely generic and not at all limited to data structures. Integer numbers, for instance, form a monoid with addition as "appending" and 0 as, well, zero:

```
-- | Monoid under addition.
newtype Sum a = Sum { getSum :: a }

-- | Monoid under multiplication.
newtype Product a = Product { getProduct :: a }

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

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

Monoids also have laws, which the instances of Monoid must follow. They are really simple, though; corresponding to the neutral element and associativity properties we mentioned:

[2] http://hackage.haskell.org/package/base-4.6.0.1/docs/Data-Monoid.html

```
mempty <> x = x
x <> mempty = x
x <> (y <> z) = (x <> y) <> z
```

The `Monoid` class is defined in Data.Monoid[3], along with instances for many common types.

45.3 Examples

For a class with such a pompous name, monoids might seem just too simple, or just plain boring. Do not let any such impression mislead you, though: they have many interesting applications. We will mention some of them in this section.

The `Writer` monad

A computation of type `Writer w a` returns a value of type `a` while producing accumulating output of type `w`. `w` is an instance of `Monoid`, and the bind operator of the monad uses `mappend` to accumulate the output. It is easy to picture `w` as a list type used for some kind of logging; we can use any monoid instance, though, which opens up many possibilities thanks to the generality of the class.

The `Foldable` class

`Foldable`, available from Data.Foldable[4], provide a simple way to generalize list folding (as well as many related `Data.List`) functions to other data structures.[5] The implementations in `Data.Foldable` take advantage of monoids by generating monoidal values from the elements of the data structure, which can then be trivially combined with `mappend`. Another role of the `Foldable` is in the definition of the `Traversable` class (in Data.Traversable[6]), which generalizes `sequence` from `Control.Monad`.

Finger trees

Moving on from operations on data structures to data structure implementations, monoids can be used to implement *finger trees*, an efficient and very versatile data structure. Its implementation makes use of monoidal values as tags for the tree nodes; and different data structures - such as sequences, priority queues and search trees - can be obtained simply by changing the involved `Monoid` instance. This blog post[7], based on a paper by Ralf Hinze and Ross Patterson[8][9], contains a brief and accessible explanation on how monoids are used in finger trees.

Options and settings

[3] http://hackage.haskell.org/packages/archive/base/latest/doc/html/Data-Monoid.html

[4] http://hackage.haskell.org/packages/archive/base/latest/doc/html/Data-Foldable.html

[5] Note that this kind of folding is different from the one discussed in Haskell/Other data structures ^{Chapter25 on page 161}.

[6] http://hackage.haskell.org/packages/archive/base/latest/doc/html/Data-Traversable.html

[7] http://apfelmus.nfshost.com/articles/monoid-fingertree.html

[8] http://www.soi.city.ac.uk/~ross/papers/FingerTree.html

[9] Ralf Hinze and Ross Paterson. Finger Trees: A Simple General-purpose Data Structure. Journal of Functional Programming 16, 2 (2006), pages 197-217.

In a wholly different context, monoids can be a handy way of treating application options and settings. Two examples are Cabal, the Haskell packaging system ("Package databases are monoids. Configuration files are monoids. Command line flags and sets of command line flags are monoids. Package build information is a monoid."`http://www.haskell.org/pipermail/haskell-cafe/2009-January/053602.html`) and XMonad[10], a tiling window manager implemented in Haskell ("xmonad configuration hooks are monoidal."`http://www.haskell.org/pipermail/haskell-cafe/2009-January/053603.html`).

45.4 Homomorphisms

A function `f :: a -> b` between two monoids `a` and `b` is called a **homomorphism** if it preserves the monoid structure, so that:

```
f mempty          = mempty
f (x `mappend` y) = f x `mappend` f y
```

For instance, `length` is an homomorphism between ([],++) and (Int,+)

```
length []         = 0
length (xs ++ ys) = length x + length y
```

An interesting example "in the wild" of monoids and homomorphisms was identified by Chris Kuklewicz[11] amidst the Google Protocol Buffers API documentation. Based on the quotes provided in the referenced comment, we highlight that the property that

```
    MyMessage message;
    message.ParseFromString(str1 + str2);
```

is equivalent to

```
    MyMessage message, message2;
    message.ParseFromString(str1);
    message2.ParseFromString(str2);
    message.MergeFrom(message2);
```

means that `ParseFromString` is a homomorphism. Translating it to Haskell and monoids, the following equations hold:

```
parse :: String -> Message
-- these are just equations, not actual code.
parse []         = mempty
parse (xs ++ ys) = parse xs `merge` parse ys
```

Well, it doesn't hold *perfectly*, because parse can fail, but roughly so.

[10] `http://xmonad.org`

[11] `http://www.haskell.org/pipermail/haskell-cafe/2009-January/053709.html`

45.5 Further reading

- Dan Pipone (Sigfpe) on monoids: a blog post overview[12]; a comment about intuition on associativity[13].
- Many monoid related links[14]
- Additional comment on finger trees: FingerTrees[15].
- Additional comments on `Monoid` usage in Cabal: `http://www.haskell.org/pipermail/haskell-cafe/2009-January/053626.html`; `http://www.haskell.org/pipermail/haskell-cafe/2009-January/053721.html`.

[12] http://sigfpe.blogspot.com/2009/01/haskell-monoids-and-their-uses.html

[13] http://www.haskell.org/pipermail/haskell-cafe/2009-January/053798.html

[14] http://groups.google.com/group/bahaskell/browse_thread/thread/4cf0164263e0fd6b/42b621f5a4da6019

[15] http://www.haskell.org/pipermail/haskell-cafe/2009-January/053689.html

46 Mutable objects

As one of the key strengths of Haskell is its *purity*: all side-effects are encapsulated in a monad[1]. 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.

46.1 The ST and IO monads

Recall the The State Monad[2] and The IO monad[3] 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?

46.2 State references: STRef and IORef

46.3 Mutable arrays

46.4 Examples

```
import Control.Monad.ST
import Data.STRef
import Data.Map(Map)
import qualified Data.Map as M
import Data.Monoid(Monoid(..))

memo :: (Ord a) => (a -> b) -> ST s (a -> ST s b)
memo f = do m <- newMemo
            return (withMemo m f)

newtype Memo s a b = Memo (STRef s (Map a b))

newMemo :: (Ord a) => ST s (Memo s a b)
newMemo = Memo `fmap` newSTRef mempty

withMemo :: (Ord a) => Memo s a b -> (a -> b) -> (a -> ST s b)
```

[1] http://en.wikibooks.org/wiki/Haskell%2FMonads

[2] http://en.wikibooks.org/wiki/Haskell%2FMonads%23The_State_monad

[3] http://en.wikibooks.org/wiki/Haskell%2FMonads%23The_.28I.29O_monad

```
withMemo (Memo r) f a = do m <- readSTRef r
                           case M.lookup a m of
                             Just b -> return b
                             Nothing -> do let b = f a
                                           writeSTRef r (M.insert a b m)
                                           return b
```

47 Concurrency

47.1 Concurrency

Concurrency in Haskell is mostly done with Haskell threads. Haskell threads are user-space threads that are implemented in the runtime. Haskell threads are much more efficient in terms of both time and space than Operating System threads. Apart from traditional synchronization primitives like semaphores, Haskell offers Software Transactional Memory which greatly simplifies concurrent access to shared memory.

The modules for concurrency are Control.Concurrent.* and Control.Monad.STM.

47.2 When do you need it?

Perhaps more important than **when** is **when not**. Concurrency in Haskell is not used to utilize multiple processor cores; you need another thing, "parallelism[1]", for that. Instead, concurrency is used for when a single core must divide its attention between various things, typically IO.

For example, consider a simple "static" webserver (i.e. serves only static content such as images). Ideally, such a webserver should consume few processing resources; instead, it must be able to transfer data as fast as possible. The bottleneck should be I/O, where you can throw more hardware at the problem. So you must be able to efficiently utilize a single processor core among several connections.

In a C version of such a webserver, you'd use a big loop centered around `select()` on each connection and on the listening socket. Each open connection would have an attached data structure specifying the state of that connection (i.e. receiving the HTTP header, parsing it, sending the file). Such a big loop would be difficult and error-prone to code by hand. However, using Concurrent Haskell, you would be able to write a much smaller loop concentrating solely on the listening socket, which would spawn a new "thread" for each accepted connection. You can then write a new "thread" in the IO monad which, in sequence, receives the HTTP header, parses it, and sends the file.

Internally, the Haskell compiler will then convert the spawning of the thread to an allocation of a small structure specifying the state of the "thread", congruent to the data structure you would have defined in C. It will then convert the various threads into a single big loop.

[1] `http://en.wikibooks.org/wiki/Haskell%2FParallelism`

Thus, while you write as if each thread is independent, internally the compiler will convert it to a big loop centered around `select()` or whatever alternative is best on your system.

47.3 Example

Example:
Downloading files in parallel

```
downloadFile :: URL -> IO ()
downloadFile = undefined

downloadFiles :: [URL] -> IO ()
downloadFiles = mapM_ (forkIO . downloadFile)
```

47.4 Software Transactional Memory

Software Transactional Memory (STM) is a mechanism that allows transactions on memory similar to database transactions. It greatly simplifies access to shared resources when programming in a multithreaded environment. By using STM, you no longer have to rely on locking.

To use STM, you have to include Control.Monad.STM. To change into the STM-Monad the atomically function is used. STM offers different primitives (TVar, TMVar, TChan and TArray) that can be used for communication.

The following example shows how to use a TChan to communicate between two threads. The channel is created in the main function and handed over to the reader/writerThread functions. The readerThread waits on the TChan for new input and prints it. The writerThread writes some Int-values to the channel and terminates.

Example:
Communication with a TChan

```
import Control.Monad.STM
import Control.Concurrent
import Control.Concurrent.STM.TChan

oneSecond = 1000000

writerThread :: TChan Int -> IO ()
writerThread chan = do
        atomically $ writeTChan chan 1
        threadDelay oneSecond
        atomically $ writeTChan chan 2
        threadDelay oneSecond
        atomically $ writeTChan chan 3
        threadDelay oneSecond

readerThread :: TChan Int -> IO ()
readerThread chan = do
        newInt <- atomically $ readTChan chan
        putStrLn $ "read new value: " ++ show newInt
        readerThread chan

main = do
        chan <- atomically $ newTChan
        forkIO $ readerThread chan
        forkIO $ writerThread chan
        threadDelay $ 5 * oneSecond
```

48 Fun with Types

49 Polymorphism basics

49.1 Parametric Polymorphism

Section goal = short, enables reader to read code (ParseP) with ∀ and use libraries (ST) without horror. Question Talk:Haskell/The_Curry-Howard_isomorphism#Polymorphic types[1] would be solved by this section.

Link to the following paper: Luca Cardelli: On Understanding Types, Data Abstraction, and Polymorphism[2].

49.1.1 forall a

As you may know, a **polymorphic** function is a function that works for many different types. For instance,

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

can calculate the length of any list, be it a string `String = [Char]` or a list of integers `[Int]`. The **type variable** `a` indicates that `length` accepts any element type. Other examples of polymorphic functions are

```
fst :: (a, b) -> a
snd :: (a, b) -> b
map :: (a -> b) -> [a] -> [b]
```

Type variables always **begin in lowercase** whereas concrete types like `Int` or `String` always start with an uppercase letter, that's how we can tell them apart.

There is a more **explicit** way to indicate that `a` can be any type

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

In other words, "for all types `a`, the function `length` takes a list of elements of type `a` and returns an integer". You should think of the old signature as an abbreviation for the new one with the `forall`[3]. That is, the compiler will internally insert any missing `forall` for you. Another example: the types signature for `fst` is really a shorthand for

[1] http://en.wikibooks.org/wiki/Talk%3AHaskell%2FThe_Curry-Howard_isomorphism%23Polymorphic%20types

[2] http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf

[3] Note that the keyword `forall` is not part of the Haskell 98 standard, but any of the language extensions `ScopedTypeVariables`, `Rank2Types` or `RankNTypes` will enable it in the compiler. A future Haskell standard will incorporate one of these.

```
fst :: forall a. forall b. (a,b) -> a
```

or equivalently

```
fst :: forall a b. (a,b) -> a
```

Similarly, the type of `map` is really

```
map :: forall a b. (a -> b) -> [a] -> [b]
```

The idea that something is applicable to every type or holds for everything is called **universal quantification**[4]. In mathematical logic, the symbol ∀[5] (an upside-down A, read as "forall") is commonly used for that, it is called the **universal quantifier**.

49.1.2 Higher rank types

With explicit `forall`, it now becomes possible to write functions that expect **polymorphic arguments**, like for instance

```
foo :: (forall a. a -> a) -> (Char,Bool)
foo f = (f 'c', f True)
```

Here, `f` is a polymorphic function, it can be applied to anything. In particular, `foo` can apply it to both the character `'c'` and the boolean `True`.

It is not possible to write a function like `foo` in Haskell98, the type checker will complain that `f` may only be applied to values of either the type `Char` or the type `Bool` and reject the definition. The closest we could come to the type signature of `foo` would be

```
bar :: (a -> a) -> (Char, Bool)
```

which is the same as

```
bar :: forall a. ((a -> a) -> (Char, Bool))
```

But this is very different from `foo`. The `forall` at the outermost level means that `bar` promises to work with any argument `f` as long as `f` has the shape `a -> a` for some type `a` unknown to `bar`. Contrast this with `foo`, where it's the argument `f` who promises to be of shape `a -> a` for all types `a` at the same time , and it's `foo` who makes use of that promise by choosing both `a = Char` and `a = Bool`.

Concerning nomenclature, simple polymorphic functions like `bar` are said to have a rank-1 type while the type `foo` is classified as **rank-2 type**. In general, a **rank-n type** is a function that has at least one rank-(n-1) argument but no arguments of even higher rank.

[4] `http://en.wikipedia.org/wiki/Universal%20quantification`

[5] The `UnicodeSyntax` extension allows you to use the symbol ∀ instead of the `forall` keyword in your Haskell source code.

The theoretical basis for higher rank types is **System F**[6], also known as the second-order lambda calculus. We will detail it in the section System F[7] in order to better understand the meaning of `forall` and its placement like in `foo` and `bar`.

Haskell98 is based on the Hindley-Milner[8] type system, which is a restriction of System F and does not support `forall` and rank-2 types or types of even higher rank. You have to enable the `RankNTypes`[9] language extension to make use of the full power of System F.

But of course, there is a good reason that Haskell98 does not support higher rank types: type inference for the full System F is undecidable, the programmer would have to write down all type signatures himself. Thus, the early versions of Haskell have adopted the Hindley-Milner type system which only offers simple polymorphic function but enables complete type inference in return. Recent advances in research have reduced the burden of writing type signatures and made rank-n types practical in current Haskell compilers.

49.1.3 `runST`

For the practical Haskell programmer, the ST monad[10] is probably the first example of a rank-2 type in the wild. Similar to the IO monad, it offers mutable references

```
newSTRef   :: a -> ST s (STRef s a)
readSTRef  :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

and mutable arrays. The type variable `s` represents the state that is being manipulated. But unlike IO, these stateful computations can be used in pure code. In particular, the function

```
runST :: (forall s. ST s a) -> a
```

sets up the initial state, runs the computation, discards the state and returns the result. As you can see, it has a rank-2 type. Why?

The point is that mutable references should be local to one `runST`. For instance,

```
v   = runST (newSTRef "abc")
foo = runST (readSTRef v)
```

is wrong because a mutable reference created in the context of one `runST` is used again in a second `runST`. In other words, the result type `a` in `(forall s. ST s a) -> a` may not be a reference like `STRef s String` in the case of `v`. But the rank-2 type guarantees exactly that! Because the argument must be polymorphic in `s`, it has to return one and the same type `a` for all states `s`; the result `a` may not depend on the state. Thus, the unwanted code snippet above contains a type error and the compiler will reject it.

[6] http://en.wikipedia.org/wiki/System%20F

[7] Chapter 49.2 on page 328

[8] http://en.wikipedia.org/wiki/Hindley-Milner

[9] Or enable just `Rank2Types` if you only want to use rank-2 types

[10] http://www.haskell.org/haskellwiki/Monad/ST

You can find a more detailed explanation of the ST monad in the original paper Lazy functional state threads[11][12].

49.1.4 Impredicativity

- *predicative* = type variables instantiated to *monotypes*. *impredicative* = also *polytypes*. Example: `length [id :: forall a . a -> a]` or `Just (id :: forall a. a -> a)`. Subtly different from higher-rank.

- relation of polymorphic types by their generality, i.e. `isInstanceOf`.
- haskell-cafe: RankNTypes short explanation.[13]

49.2 System F

Section goal = a little bit lambda calculus foundation to prevent brain damage from implicit type parameter passing.

- System F = Basis for all this ∀-stuff.
- Explicit type applications i.e. `map Int (+1) [1,2,3]`. ∀ similar to the function arrow ->.
- Terms depend on types. Big Λ for type arguments, small λ for value arguments.

49.3 Examples

Section goal = enable reader to judge whether to use data structures with ∀ in his <u>own</u> code.

- Church numerals, Encoding of arbitrary recursive types (positivity conditions): `&forall x. (F x -> x) -> x`
- Continuations, Pattern-matching: `maybe`, `either` and `foldr`

I.e. ∀ can be put to good use for implementing data types in Haskell.

49.4 Other forms of Polymorphism

Section goal = contrast polymorphism in OOP and stuff. how type classes fit in.

- *ad-hoc polymorphism* = different behavior depending on type s. => Haskell type classes.
- *parametric polymorphism* = ignorant of the type actually used. => ∀
- *subtyping*

[11] http://www.dcs.gla.ac.uk/fp/papers/lazy-functional-state-threads.ps.Z

[12] John Launchbury; Simon Peyton Jones 1994-??-??. Lazy functional state threads- ACM Press". pp. 24-35 http://

[13] http://thread.gmane.org/gmane.comp.lang.haskell.cafe/40508/focus=40610

49.5 Free Theorems

Secion goal = enable reader to come up with free theorems. no need to prove them, intuition is enough.

- free theorems for parametric polymorphism.

49.6 See also

- Luca Cardelli. On Understanding Types, Data Abstraction, and Polymorphism[14].

[14] http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf

50 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 `-XExistentialQuantification`, or put `{-# LANGUAGE ExistentialQuantification #-}` at the top of your sources that use existentials.

50.1 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, $\exists x$ means that whatever follows is true for at least one value of x. $\forall x$ means that what follows is true for every x you could imagine. For example, $\forall x, x^2 \geq 0$ and $\exists x, x^3 = 27$.

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]
```

So we see that for any `a` and `b` we can imagine, `map` takes the type `(a -> b) -> [a] -> [b]`. For example, we might choose `a = Int` and `b = String`. Then it's valid to say that map has the type (Int -> String) -> [Int] -> [String]. We are *instantiating* the general type of map to a more specific type.

However, in Haskell, as we know, 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 dive right into the deep end by seeing an example of the power of existential types in action.

50.2 Example: heterogeneous lists

The premise behind Haskell's typeclass system is grouping types that all share a common property. 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 elements of Int can be compared for equality.

Suppose we have a group of values, and we don't know if they are all the same type, but we do know they all instantiate some class, i.e. we know all the values have a certain property. 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

heteroList :: [ShowBox]
heteroList = [SB (), SB 5, SB True]
```

We won't explain precisely what we mean by that datatype definition, but its meaning should be clear to your intuition. The important thing is that we're calling the constructor on three values of different types, and we place them all into a list, so we must end up with the same type for each one. Essentially this is because our use of the forall keyword gives our constructor the type SB :: forall s. Show s => s -> ShowBox. If we were now writing a function to which we intend to pass heteroList, we couldn't apply a function such as not to the values inside the SB, for their type might not be Bool. But we do know something about each of the elements: 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        -- (*) see the comment in the text below

f :: [ShowBox] -> IO ()
f xs = mapM_ print xs

main = f heteroList
```

Let's expand on this a bit more. In the definition of `show` for `ShowBox` – the line marked with `(*) see the comment in the text below` – we don't know the type of `s`. But as we mentioned, 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 `f`, 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.

50.3 Explaining the term *existential*

Note:
Since you can get existential types with `forall`, Haskell forgoes the use of an `exists` keyword, which would just be redundant.

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, ⊥, is a member of every type!), Integer is the set of integers (and bottom), String is 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, which must be {⊥}, 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 appear in, for example, String? Bottom is the only value common to all types.

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 intersecting 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).

We see that most intersections over types just lead to combinations of bottoms in some ways, because types don't have a lot of values in common.

Recall that in the last section, we developed a heterogeneous list using a 'type hider'. Ideally, we'd like the type of a heterogeneous list to be `[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, so that `[exists a. a]` is the type of a list where all elements could take any type at all (and the types of different elements needn't be the same).

But we got almost the same behaviour above using 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 hetereogeneous 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[1], 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 heterogenous setup

```
heteroList' = [MkT' 5, MkT' (), MkT' True, MkT' "Sartre"]
main = mapM_ (\(MkT' x) -> print x) heteroList'

{- prints:
5
()
True
"Sartre"
-}
```

[1] 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`, `seq`. So technically, we can't do anything *useful* with its elements, except reduce them to WHNF.

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'. Whenever you want existential types, you must wrap them up in a datatype constructor, they can't exist "out in the open" like with `[exists a. a]`.

50.4 Example: `runST`

One monad that you may not have 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/[2] 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 in 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)
```

[2] Chapter 29 on page 183

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.

50.5 Quantification as a primitive

Universal quantification is useful for defining data types that aren't already defined. Suppose there was no such thing as pairs built into haskell. Quantification could be used to define them.

```
newtype Pair a b=Pair (forall c.(a->b->c)->c)
```

50.6 Further reading

- GHC's user guide contains useful information[3] on existentials, including the various limitations placed on them (which you should know about).
- *Lazy Functional State Threads*[4]*, by Simon Peyton-Jones and John Launchbury, is a paper which explains more fully the ideas behind ST.*

[3] http://haskell.org/ghc/docs/latest/html/users_guide/data-type-extensions.html#existential-quantification

[4] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.3299

51 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.

51.1 Multi-parameter type classes

Multi-parameter type classes are a generalisation of the single parameter type classes[1], 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:

Example:
The `Collection` type class (wrong)

```
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:

Example:
The `Collection` type class (right)

```
{-# LANGUAGE MultiParamTypeClasses #-}
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
```

[1] http://en.wikibooks.org/wiki/Class_Declarations

51.2 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

Example:
A functional dependency

```
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.

51.2.1 Examples

Matrices and vectors

Suppose you want to implement some code to perform simple linear algebra:

Example:
The `Vector` and `Matrix` datatypes

```
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:

Example:
Instance declarations for `Vector` and `Matrix`

```
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:

Example:
What we need

```
(*) :: 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:

Example:
An ineffective attempt (too general)

```
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:

Example:
Ambiguities lead to more verbose code

```
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:

Example:
A nonsensical instance of `Mult`

```
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:

Example:
The correct definition of `Mult`

```
class Mult a b c | a b -> c where
  (*) :: a -> b -> c
```

This tells Haskell that `c` is uniquely determined from `a` and `b`.

52 Phantom types

Phantom types are a way to embed a language with a stronger type system than Haskell's.

52.1 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)))

type D2 = Succ (Succ Zero)
type D3 = Succ (Succ (Succ Zero))

data Vector n a = Vector [a] deriving (Eq, Show)

vector2d :: Vector D2 Int
vector2d = Vector [1,2]

vector3d :: Vector D3 Int
vector3d = Vector [1,2,3]
```

```
-- vector2d == vector3d raises a type error
-- at compile-time:

--   Couldn't match expected type `Zero'
--               with actual type `Succ Zero'
--   Expected type: Vector D2 Int
--     Actual type: Vector D3 Int
--   In the second argument of `(==)', namely `vector3d'
--   In the expression: vector2d == vector3d

-- while vector2d == Vector [1,2,3] works
```

53 Generalised algebraic data-types (GADT)

w:Generalized algebraic data type[1]

53.1 Introduction

Generalized Algebraic Datatypes are, as the name suggests, a generalization of Algebraic Data Types that you are already familiar with. Basically, they allow you to explicitly write down the types of the constructors. In this chapter, you'll learn why this is useful and how to declare your own.

We begin with an example of building a simple embedded domain specific language (EDSL) for simple arithmetical expressions, which is put on a sounder footing with GADTs. This is followed by a review of the syntax for GADTs, with simpler illustrations, and a different application to construct a safe list type for which the equivalent of `head []` fails to typecheck and thus does not give the usual runtime error: `*** Exception: Prelude.head: empty list`.

53.2 Understanding GADTs

So, what are GADTs and what are they useful for? GADTs are mainly used to implement domain specific languages and this section will introduce them with a corresponding example.

53.2.1 Arithmetic expressions

Let's consider a small language for arithmetic expressions, given by the data type

```
data Expr = I Int            -- integer constants
          | Add Expr Expr    -- add two expressions
          | Mul Expr Expr    -- multiply two expressions
```

In other words, this data type corresponds to the abstract syntax tree, an arithmetic term like `(5+1)*7` would be represented as ``(I 5 `Add` I 1) `Mul` I 7 :: Expr``.

[1] http://en.wikipedia.org/wiki/Generalized%20algebraic%20data%20type

Given the abstract syntax tree, we would like to do something with it; we want to compile it, optimize it and so on. For starters, let's write an evaluation function that takes an expression and calculates the integer value it represents. The definition is straightforward:

```
eval :: Expr -> Int
eval (I n)       = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

53.2.2 Extending the language

Now, imagine that we would like to extend our language with other types than just integers. For instance, let's say we want to represent equality tests, so we need booleans as well. We augment the `Expr` type to become

```
data Expr = I Int
          | B Bool           -- boolean constants
          | Add Expr Expr
          | Mul Expr Expr
          | Eq  Expr Expr    -- equality test
```

The term 5+1 == 7 would be represented as (I 5 `Add` I 1) `Eq` I 7.

As before, we want to write a function eval to evaluate expressions. But this time, expressions can now represent either integers or booleans and we have to capture that in the return type

```
eval :: Expr -> Either Int Bool
```

The first two cases are straightforward

```
eval (I n) = Left n
eval (B b) = Right b
```

but now we get in trouble. We would like to write

```
eval (Add e1 e2) = eval e1 + eval e2  -- ???
```

but this doesn't type check: the addition function + expects two integer arguments, but eval e1 is of type Either Int Bool and we'd have to extract the Int from that.

Even worse, what happens if e1 actually represents a *boolean*? The following is a valid expression

```
B True `Add` I 5 :: Expr
```

but clearly, it doesn't make any sense; we can't add booleans to integers! In other words, evaluation may return integers or booleans, but it may also *fail* because the expression makes no sense. We have to incorporate that in the return type:

```
eval :: Expr -> Maybe (Either Int Bool)
```

Now, we could write this function just fine, but that would still be unsatisfactory, because what we *really* want to do is to have Haskell's type system rule out any invalid expressions; we don't want to check types ourselves while deconstructing the abstract syntax tree.

Exercise: Despite our goal, it may still be instructional to implement the `eval` function; do this.

Starting point:

```
data Expr = I Int
       | B Bool           -- boolean constants
       | Add Expr Expr
       | Mul Expr Expr
       | Eq  Expr Expr    -- equality test

eval :: Expr -> Maybe (Either Int Bool)
-- Your implementation here.
```

53.2.3 Phantom types

The so-called *phantom types* are the first step towards our goal. The technique is to augment the `Expr` with a type variable, so that it becomes

```
data Expr a = I Int
            | B Bool
            | Add (Expr a) (Expr a)
            | Mul (Expr a) (Expr a)
            | Eq  (Expr a) (Expr a)
```

Note that an expression `Expr a` does not contain a value `a` at all; that's why `a` is called a *phantom type*, it's just a dummy variable. Compare that with, say, a list `[a]` which does contain a bunch of `a`'s.

The key idea is that we're going to use `a` to track the type of the expression for us. Instead of making the constructor

```
Add :: Expr a -> Expr a -> Expr a
```

available to users of our small language, we are only going to provide a *smart constructor* with a more restricted type

```
add :: Expr Int -> Expr Int -> Expr Int
add = Add
```

The implementation is the same, but the types are different. Doing this with the other constructors as well,

```
i :: Int  -> Expr Int
i = I
```

```
b :: Bool -> Expr Bool
b = B
```

the previously problematic expression

```
b True `add` i 5
```

no longer type checks! After all, the first arguments has the type `Expr Bool` while `add` expects an `Expr Int`. In other words, the phantom type `a` marks the intended type of the expression. By only exporting the smart constructors, the user cannot create expressions with incorrect types.

As before, we want to implement an evaluation function. With our new marker `a`, we might hope to give it the type

```
eval :: Expr a -> a
```

and implement the first case like this

```
eval (I n) = n
```

But alas, this does not work: how would the compiler know that encountering the constructor `I` means that `a = Int`? Granted, this will be case for all the expression that were created by users of our language because they are only allowed to use the smart constructors. But internally, an expression like

```
I 5 :: Expr String
```

is still valid. In fact, as you can see, `a` doesn't even have to be `Int` or `Bool`, it could be anything.

What we need is a way to restrict the return types of the constructors themselves, and that's exactly what generalized data types do.

53.2.4 GADTs

The obvious notation for restricting the type of a constructor is to write down its type, and that's exactly how GADTs are defined:

```
data Expr a where
    I   :: Int  -> Expr Int
    B   :: Bool -> Expr Bool
    Add :: Expr Int -> Expr Int -> Expr Int
    Mul :: Expr Int -> Expr Int -> Expr Int
    Eq  :: Expr Int -> Expr Int -> Expr Bool
```

In other words, we simply list the type signatures of all the constructors. In particular, the marker type `a` is specialised to `Int` or `Bool` according to our needs, just like we would have done with smart constructors.

And the great thing about GADTs is that we now *can* implement an evaluation function that takes advantage of the type marker:

```
eval :: Expr a -> a
eval (I n) = n
eval (B b) = b
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Eq  e1 e2) = eval e1 == eval e2
```

In particular, in the first case

```
eval (I n) = n
```

the compiler is now able infer that `a=Int` when we encounter a constructor `I` and that it is legal to return the `n :: Int`; similarly for the other cases.

To summarise, GADTs allows us to restrict the return types of constructors and thus enable us to take advantage of Haskell's type system for our domain specific languages. Thus, we can implement more languages and their implementation becomes simpler.

53.3 Summary

53.3.1 Syntax

Here a quick summary of how the syntax for declaring GADTs works.

First, consider the following ordinary algebraic datatypes: the familiar `List` and `Maybe` types, and a simple tree type, `RoseTree`:

Maybe

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

List

```
data List a =
    Nil |
    Cons a (List a)
```

Rose Tree

```
data RoseTree a =
     RoseTree a [RoseTree a]
```

Remember that the constructors introduced by these declarations can be used both for pattern matches to deconstruct values and as functions to construct values. (`Nothing` and `Nil` are functions with "zero arguments".) We can ask what the types of the latter are:

Maybe

List

Rose Tree

```
> :t Nothing
Nothing :: Maybe a
> :t Just
Just :: a -> Maybe a
```

```
> :t Nil
Nil :: List a
> :t Cons
Cons :: a -> List a -
> List a
```

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

It is clear that this type information about the constructors for `Maybe`, `List` and `RoseTree` respectively is equivalent to the information we gave to the compiler when declaring the datatype in the first place. In other words, it's also conceivable to declare a datatype by simply listing the types of all of its constructors, and that's exactly what the GADT syntax does:

Maybe

```
data Maybe a where
   Nothing  :: Maybe a
   Just :: a -> Maybe a
```

List

```
data List a where
   Nil  :: List a
   Cons :: a -> List a -
> List a
```

Rose Tree

```
data RoseTree a where
   RoseTree ::
      a ->  [RoseTree a] -
> RoseTree a
```

This syntax is made available by the language option `{-#LANGUAGE GADTs #-}`. It should be familiar to you in that it closely resembles the syntax of type class declarations. It's also easy to remember if you already like to think of constructors as just being functions. Each constructor is just defined by a type signature.

53.3.2 New possibilities

Note that when we asked the `GHCi` for the types of `Nothing` and `Just` it returned `Maybe a` and `a -> Maybe a` as the types. In these and the other cases, the type of the final output of the function associated with a constructor is the type we were initially defining - `Maybe a`, `List a` or `RoseTree a`. In general, in standard Haskell, the constructor functions for `Foo a` have `Foo a` as their final return type. If the new syntax were to be strictly equivalent to the old, we would have to place this restriction on its use for valid type declarations.

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 blah` 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 FooInGadtClothing a where
 MkFooInGadtClothing :: a -> FooInGadtClothing a

--which is no different from:  data Haskell98Foo a = MkHaskell98Foo a ,

--by contrast, consider:

data TrueGadtFoo a where
  MkTrueGadtFoo :: a -> TrueGadtFoo Int

--This has no Haskell 98 equivalent.
```

But note that you can only push the generalization so far... if the datatype you are declaring is a `Foo`, the constructor functions *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
  BarNone :: Bar -- This is ok

data Foo where
  MkFoo :: Bar Int-- This will not typecheck
```

53.4 Examples

53.4.1 Safe Lists

Prerequisite: *We assume in this section that you know how a List tends to be represented in functional languages*

Note: *The examples in this section additionally require the extensions EmptyDataDecls and KindSignatures to be enabled*

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 a Empty
-- or SafeList a NonEmpty
data SafeList a b 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 a NonEmpty -> a
```

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* list-of-a types, `SafeList a Empty` for the `Nil` constructor, and `SafeList a NonEmpty` for the `Cons` constructor:

```
data SafeList a b where
  Nil  :: SafeList a Empty
  Cons :: a -> SafeList a b -> SafeList a 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 all together, along with the actual definition of `SafeHead`:

Example:
safe lists via GADT

```
{-#LANGUAGE GADTs, EmptyDataDecls #-}
-- (the EmptyDataDecls pragma must also appear at the very top of the module,
-- in order to allow the Empty and NonEmpty datatype declarations.)

data Empty
data NonEmpty

data SafeList a b where
     Nil :: SafeList a Empty
     Cons:: a -> SafeList a b -> SafeList a NonEmpty

safeHead :: SafeList a NonEmpty -> a
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
```

The 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, that also sets up a pitfall in potential. Consider the following function. What do you think its type is?

Example:
Trouble with GADTs

```
silly False = Nil
silly True  = Cons () 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 () Empty
     Inferred type: SafeList () NonEmpty
   In the application `Cons () Nil'
   In the definition of `silly': silly True = Cons () Nil
```

The cases in the definition of `silly` evaluate to marked lists of different types, leading to a type error. The extra constraints imposed through the GADT make it impossible for a function to produce both empty and non-empty lists.

If we are really keen on defining `silly`, we can do so by liberalizing the type of `Cons`, so that it can construct both safe and unsafe lists.

Example:
A different approach

```
{-#LANGUAGE GADTs, EmptyDataDecls, KindSignatures #-}
-- here we add the KindSignatures pragma,
-- which makes the GADT declaration a bit more elegant.

-- Note the subtle yet revealing change in the phantom type names.
data NotSafe
data Safe


data MarkedList             ::  * -> * -> * where
  Nil                       ::  MarkedList t NotSafe
  Cons                      ::  a -> MarkedList a b -> MarkedList a c


safeHead                    ::  MarkedList a Safe -> a
safeHead (Cons x _)          =  x

-- This function will never produce anything that can be consumed by safeHead,
-- no matter that the resulting list is not necessarily empty.
silly                       :: Bool -> MarkedList () NotSafe
silly False                  =  Nil
silly True                   =  Cons () Nil
```

There is a cost to the fix above: by relaxing the constraint on `Cons` we throw away the knowledge that it cannot produce an empty list. Based on our first version of the safe list we could, for instance, define a function which took a `SafeList a Empty` argument and be sure anything produced by `Cons` would not be accepted by it. That does not hold for the analogous `MarkedList a NotSafe`; arguably, the type is less useful exactly because it is less restrictive. While in this example the issue may seem minor, given that not much can be done with an empty list, in general it is worth considering.

Exercises:

1. Could you implement a `safeTail` function? Both versions introduced here would count as valid starting material, as well as any other variants in similar spirit.

53.4.2 A simple expression evaluator

Insert the example used in Wobbly Types paper... I thought that was quite pedagogical

This is already covered in the first part of the tutorial.

53.5 Discussion

More examples, thoughts

From FOSDEM 2006, I vaguely recall that there is some relationship between GADT and the below... what?

53.5.1 Phantom types

See ../Phantom types/[2].

53.5.2 Existential types

If you like ../Existentially quantified types/[3], 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
```

53.5.3 Witness types

[2] Chapter 52 on page 343
[3] Chapter 50 on page 331

54 Type constructors & Kinds

54.1 Kinds for C++ users

- * is any concrete type, including functions. These all have kind *:

```
type MyType = Int
type MyFuncType = Int -> Int
myFunc :: Int -> Int
```

```
typedef int MyType;
typedef int (*MyFuncType)(int);
int MyFunc(int a);
```

- * -> * is a template that takes one type argument. It is like a function from types to types: you plug a type in and the result is a type. Confusion can arise from the two uses of MyData (although you can give them different names if you wish) - the first is a type constructor, the second is a data constructor. These are equivalent to a class template and a constructor respectively in C++. Context resolves the ambiguity - where Haskell expects a type (e.g. in a type signature) MyData is a type constructor, where a value, it is a data constructor.

```
data MyData t -- type constructor with kind * -> *
              = MyData t -- data constructor with type a -> MyData a
*Main> :k MyData
MyData :: * -> *
*Main> :t MyData
MyData :: a -> MyData a
```

```
template <typename t> class MyData
{
   t member;
};
```

- * -> * -> * is a template that takes two type arguments

```
data MyData t1 t2 = MyData t1 t2
```

```
template <typename t1, typename t2> class MyData
{
   t1 member1;
   t2 member2;
   MyData(t1 m1, t2 m2) : member1(m1), member2(m2) { }
};
```

- (* -> *) -> * is a template that takes one template argument of kind (* -> *)

```
data MyData tmpl = MyData (tmpl Int)
```

```
template <template <typename t> class tmpl> class MyData
{
   tmpl<int> member1;
   MyData(tmpl<int> m) : member1(m) { }
};
```

55 Wider Theory

56 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[1] page or the #haskell IRC channel so that the style of exposition can be improved.

56.1 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.[2]

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 concisely with denotational semantics alone, no reference to an execution model is necessary. They will be put to good use in Graph Reduction[3], but it is this chapter that will familiarize the reader with the denotational definition and involved notions such as ⊥ ("Bottom"). The reader only interested in strictness may wish to poke around in section Bottom and Partial Functions[4] and quickly head over to Strict and Non-Strict Semantics[5].

56.1.1 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 (denotation), that represents the *meaning* of

[2] 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.

[3] Chapter 63 on page 433

[4] Chapter 56.2 on page 364

[5] Chapter 56.4 on page 373

the program in question. As an example, the mathematical object for the Haskell programs `10`, `9+1`, `2*5` and `sum [1..4]` can be represented by the integer *10*. We say that all those programs **denote** the integer *10*. The collection of such mathematical objects is called the **semantic domain**.

The mapping from program code to a semantic domain is commonly written down with double square brackets ("Oxford brackets") around program code. For example,

$$[[\texttt{2*5}]] = 10.$$

Denotations are *compositional*, i.e. the meaning of a program like `1+9` only depends on the meaning of its constituents:

$$[[\texttt{a+b}]] = [[\texttt{a}]] + [[\texttt{b}]].$$

The same notation is used for types, i.e.

$$[[\texttt{Integer}]] = \mathbb{Z}.$$

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 $\mathbb{Z} \to \mathbb{Z}$ between integers. While we will see that this expression needs refinement generally, 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 tightly tied to **operational semantics** which describes their way of execution 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 operational nature and sometimes must be extended in comparison to the denotational semantics for functional languages.[6] In contrast, the meaning of purely functional languages is *by default* completely independent from their way of execution. The Haskell98 standard even goes as far as to specify only Haskell's non-strict denotational semantics, 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[7], which transforms programs into

[6] Monads are one of the most successful ways to give denotational semantics to imperative programs. See also Haskell/Advanced monads ^{`http://en.wikibooks.org/wiki/Haskell%2FAdvanced%20monads`} .

[7] `http://en.wikibooks.org/wiki/Haskell%2FEquational%20reasoning`

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[8].

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. On the other hand, the implementation has to respect the semantics, and to a certain extent, it is the semantics that determine how Haskell programs must be evaluated on a machine. We will elaborate on this in Strict and Non-Strict Semantics[9].

56.1.2 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 an 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.

[8] Chapter 56.5.3 on page 380

[9] Chapter 56.4 on page 373

56.2 Bottom and Partial Functions

56.2.1 ⊥ Bottom

To define partial functions, we introduce a special value $\perp$, named **bottom**[10] and commonly written `_|_` in typewriter font. We say that $\perp$ is the completely **"undefined" value** or function. Every basic data type like `Integer` or `()` contains one $\perp$ besides their usual elements. So the possible values of type `Integer` are

$$\perp, 0, \pm 1, \pm 2, \pm 3, \ldots$$

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 `Integer`s. Arbitrary algebraic data types will be treated in section Algebraic Data Types[11] 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[12] that any value of the polymorphic type `forall a . a` must denote $\perp$.

56.2.2 Partial Functions and the Semantic Approximation Order

Now, $\perp$ (*bottom type*) gives us the possibility to denote partial functions:

$$f(n) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ -2 & \text{if } n \text{ is } 1 \\ \perp & \text{else} \end{cases}$$

[10] `http://en.wikibooks.org/wiki/%3Aw%3ABottom%20type`

[11] Chapter 56.5 on page 375

[12] Chapter 58 on page 405

Here, $f(n)$ yields well defined values for $n = 0$ and $n = 1$ but gives $\perp$ for all other n. Note that the type $\perp$ is universal, as $\perp$ has no value: the function $\perp$`:: Integer -> Integer` is given by

$$\perp(n) = \perp$$

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, since it does not acknowledge the special role of $\perp$. For example, the definition

$$g(n) = \begin{cases} 1 & \text{if } n \text{ is } \perp \\ \perp & \text{else} \end{cases}$$

looks counterintuitive, and, in fact, is wrong. 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). $\sqsubset$ 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 that $\perp \sqsubset x$ for all x, except the case when x happens to denote $\perp$ itself:

$$\forall x \neq \perp \quad \perp \sqsubset x$$

As no number is *more defined* than another, the mathematical relation $\sqsubset$ is false for any pair of numbers:

$$1 \sqsubset 1$$

does not hold.

neither

$$1 \sqsubset 2$$

nor

$$2 \sqsubset 1$$

hold.

This is contrasted to ordinary order predicate $\leq$, which can compare any two numbers. A quick way to remember this is the sentence: "1 and 2 are different in terms of *information content* but are equal in terms of *information quantity*". That's another reason why we use a different symbol: $\sqsubseteq$.

neither

$$1 \sqsubseteq 2$$

nor

$$2 \sqsubseteq 1$$

hold,

but

$$1 \sqsubseteq 1$$

holds.

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

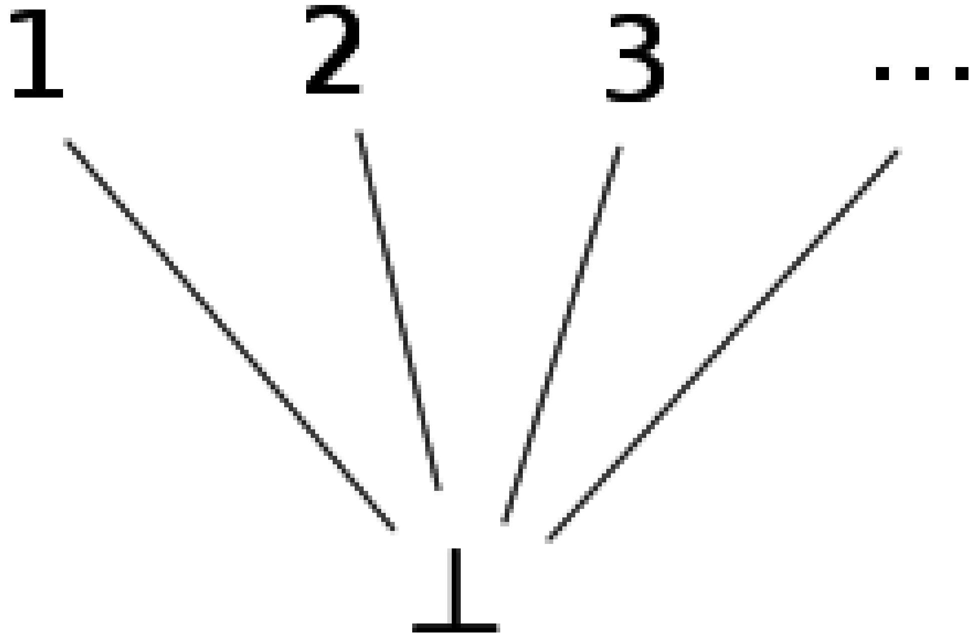


Figure 28

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 ⊥), one says that `Integer` is a *flat domain*. The picture also explains the name of ⊥: it's called *bottom* because it always sits at the bottom.

56.2.3 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:

$$x \sqsubseteq y \Longrightarrow f(x) \sqsubseteq f(y)$$

In particular, a function h with $h(\perp) = 1$ is constant: $h(n) = 1$ for all n. Note that here it is crucial that $1 \sqsubseteq 2$ etc. don't hold.

Translated to Haskell, monotonicity means that we cannot use ⊥ as a condition, i.e. we cannot pattern match on ⊥, or its equivalent `undefined`. Otherwise, the example g from above could be expressed as a Haskell program. As we shall see later, ⊥ also denotes non-terminating programs, so that the inability to observe ⊥ 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 $\bot(x) = \bot$ being the least element.

56.3 Recursive Definitions as Fixed Point Iterations

56.3.1 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 recursive function directly as a set description may lead to problems, we intuitively know that in order to calculate $f(n)$ for every given n we have to iterate 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 consists of the right hand side applied to the previous one, that is

$$f_{k+1}(n) = \text{ if } n == 0 \text{ then } 1 \text{ else } n \cdot f_k(n-1)$$

We start with the undefined function $f_0(n) = \bot$, and the resulting sequence of partial functions reads:

$$f_1(n) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ \bot & \text{else} \end{cases}, \; f_2(n) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ 1 & \text{if } n \text{ is } 1 \\ \bot & \text{else} \end{cases}, \; f_3(n) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ 1 & \text{if } n \text{ is } 1 \\ 2 & \text{if } n \text{ is } 2 \\ \bot & \text{else} \end{cases}$$

and so on. Clearly,

$$\bot = f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \dots$$

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

$$x_0 = \bot$$

and

$$g(x) = n \mapsto \text{ if } n == 0 \text{ then } 1 \text{ else } n * x(n-1)$$

If we start with $x_0 = \bot$, the iteration will yield increasingly defined approximations to the factorial function

$$\bot \sqsubseteq g(\bot) \sqsubseteq g(g(\bot)) \sqsubseteq g(g(g(\bot))) \sqsubseteq \dots$$

(Proof that the sequence increases: The first inequality $\bot \sqsubseteq g(\bot)$ follows from the fact that $\bot$ is less defined than anything else. The second inequality follows from the first one by applying g to both sides and noting that g is monotone. The third follows from the second in the same fashion and so on.)

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 `f0,f1,...` 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.

56.3.2 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 dcpos which have a least element $\perp$ which are called **complete partial order**s (**cpo**).

The `Integer`s 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.

56.3.3 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

$$f_0 = \perp, f_1 = \perp, \ldots$$

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.

56.3.4 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

$$\begin{array}{rcl} g(f) & = & g\left(\sup_{\sqsubseteq}\{x_0 \sqsubseteq g(x_0) \sqsubseteq g(g(x_0)) \sqsubseteq \dots\}\right) \\ & = & \sup_{\sqsubseteq}\{g(x_0) \sqsubseteq g(g(x_0)) \sqsubseteq \dots\} \\ & = & \sup_{\sqsubseteq}\{x_0 \sqsubseteq g(x_0) \sqsubseteq g(g(x_0)) \sqsubseteq \dots\} \\ & = & f \end{array}$$

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 $x_0 = \bot$ is also the least one, that it is smaller than any other fixed point. (Hint: $\bot$ 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 $\bot$. For continuity, we note that for an arbitrary type `a`, every simple function `a -> 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 `Integer -> Integer` must be continuous. For functionals like g`::(Integer -> Integer) -> (Integer -> Integer)`, the continuity then materializes due to currying, as the type is isomorphic to `::((Integer -> Integer), Integer) -> Integer` and we can take `a=((Integer -> Integer), 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 of Haskell programs on a firm ground. Yet it is very nice that we can explore these semantics in Haskell itself with the help of `undefined`.

56.4 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`.

56.4.1 Strict Functions

A function `f` with one argument is called **strict**, if and only if

`f ⊥ = ⊥`.

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 `⊥+ 1` is ⊥ or not. If it was not, then we should for example have `⊥+ 1 = 2` or more general `⊥+ 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

$$\texttt{succ } \perp = \perp + 1 = \perp$$

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.

56.4.2 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* can be no less 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)
1
```

which is not ⊥. This is reasonable as `one` completely ignores its argument. When interpreting ⊥ 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 ⊥. But one might as well say that every function must evaluate its arguments before computing the result which means that `one` ⊥ should be ⊥, too. That is, if the program computing the argument does not halt, `one` should not halt as well.[13] 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.

56.4.3 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 if

`f x ⊥ = ⊥`

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

[13] Strictness as premature evaluation of function arguments is elaborated in the chapter Graph Reduction ^{Chapter63 on page 433}.

```
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  x ⊥ = x
cond False x ⊥ = ⊥
```

and likewise for `x`. Apparently, `cond` is certainly ⊥ 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 ⊥ 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[14][15] for more on this subject.

56.5 Algebraic Data Types

After treating the motivation case of partial functions between `Integer`s, 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.

56.5.1 Constructors

Let's take the example types

[14] Chapter 64 on page 445

[15] The term *Laziness* comes from the fact that the prevalent implementation technique for non-strict languages is called *lazy evaluation*

```
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:

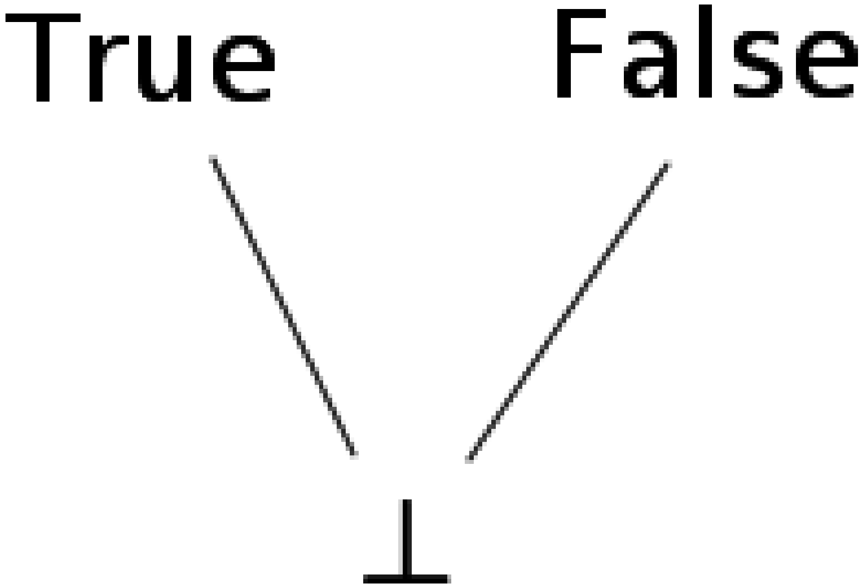


Figure 29

Remember that ⊥ is added as least element to the set of values `True` and `False`, we say that the type is **lifted**[16]. 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 ⊥ 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 ⊥. 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

[16] The term *lifted* is somewhat overloaded, see also Unboxed Types ^{Chapter56.1.2 on page 363}.

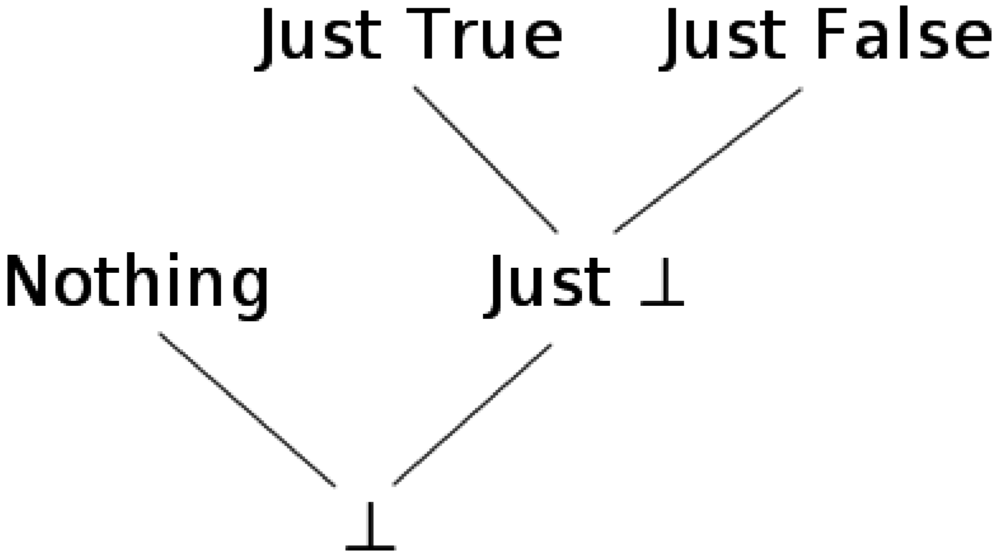


Figure 30

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

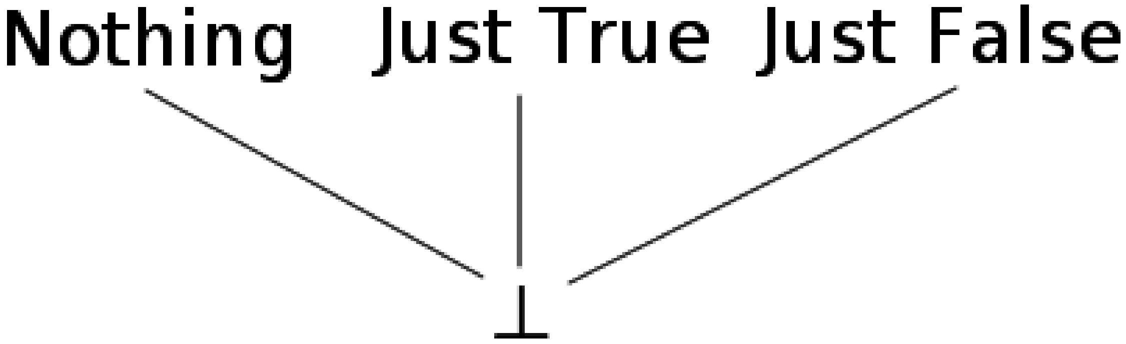


Figure 31

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[17], 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[18], but one prominent example are infinite lists treated in section Recursive Data Types and Infinite Lists[19].

56.5.2 Pattern Matching

In the section Strict Functions[20], 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 ⊥, too. However, the argument for case expressions may be more involved as in

[17] Chapter 63 on page 433

[18] Chapter 64 on page 445

[19] Chapter 56.5.3 on page 380

[20] Chapter 56.4.1 on page 373

```
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 ⊥ = 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 ⊥        = ⊥ + 1 = ⊥
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 ⊥.

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[a] gives the detailed semantics of pattern matching[b] and you should now be able to understand it. So go on and have a look!
2. Consider a function `or` of two `Bool`ean 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 ⊥ if both arguments are (at least when we restrict the arguments to `True` and ⊥). Can such a function be implemented in Haskell?

[a] http://www.haskell.org/onlinereport/

[b] http://www.haskell.org/onlinereport/exps.html#case-semantics

56.5.3 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 ⊥ 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.

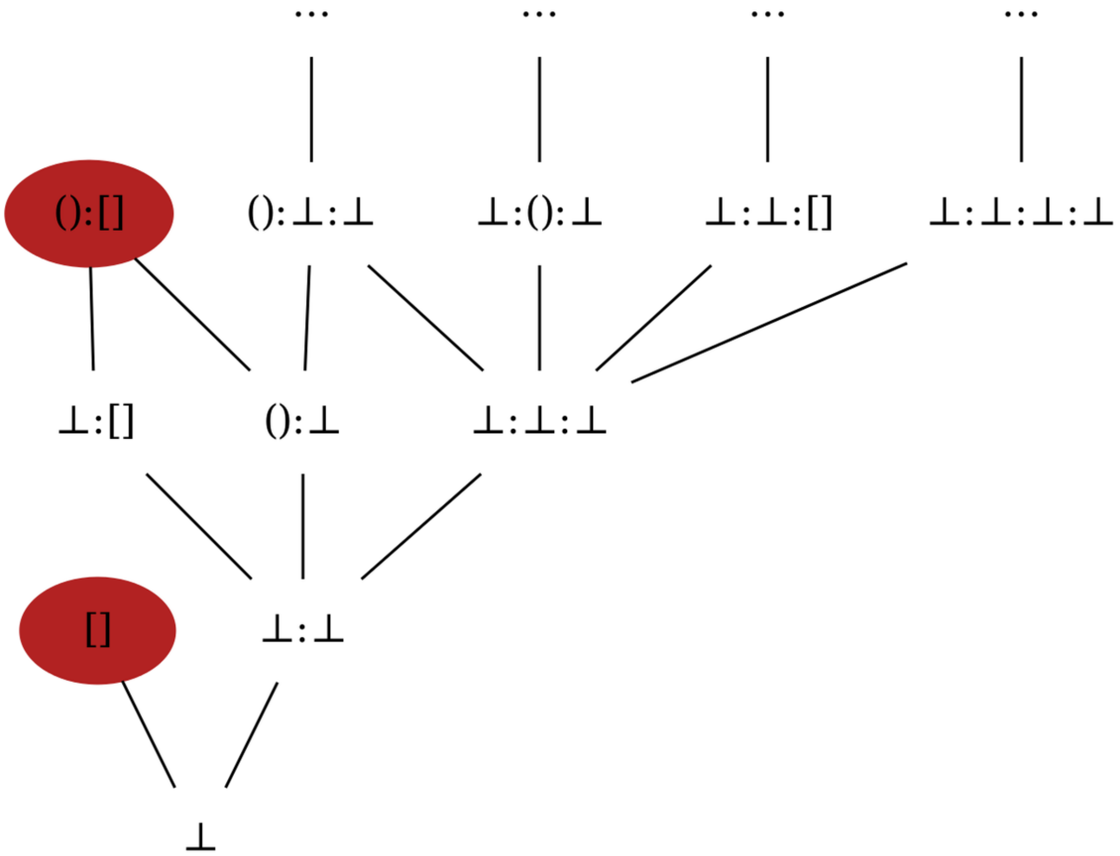


Figure 32

and so on. But now, there are also chains of infinite length like

$$\perp \sqsubseteq \texttt{():}\perp \sqsubseteq \texttt{():():}\perp \sqsubseteq \ldots$$

This causes us some trouble as we noted in section Convergence[21] 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[22]. 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]
```

[21] Chapter 56.3.2 on page 370
[22] Chapter 64 on page 445

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

$$\bot \sqsubseteq \texttt{1:}\bot \sqsubseteq \texttt{1:1:}\bot \sqsubseteq \texttt{1:1:1:}\bot \sqsubseteq \ldots,$$

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 be compactly depicted as

```
ones = 1 : 1 : 1 : 1 : ...
```

what simply means that

```
ones = (⊥ ⊑ 1:⊥ ⊑ 1:1:⊥ ⊑ ...)
```

Exercises:

1. Of course, there are more interesting infinite lists than `ones`. Can you write recursive definition in Haskell for
   a) the natural numbers `nats = 1:2:3:4:...`
   b) a cycle like `cycle123 = 1:2:3: 1:2:3 : ...`
2. Look at the Prelude functions `repeat` and `iterate` and try to solve the previous exercise with their help.
3. Use the example from the text to find the value the expression `drop 3 nats` denotes.
4. 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 3 (map (+1) nats) = take 3 (tail 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[23] for a formal construction.

[23] Chapter 56.7 on page 387

56.5.4 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 it should be strict in this argument. Hence we have `Just'` $\perp$ `=` $\perp$ in our example. Further information may be found in chapter Strictness[24].

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 $\perp$ and `Couldbe` $\perp$. 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'` $\perp$ will cause the pattern match on the constructor `Couldbe'` fail with the effect that `f'` $\perp$ `=` $\perp$. But for the newtype, the match on `Couldbe` will never fail, we get `f` $\perp$ `= 42`. In a sense, the difference can be stated as:

- for the strict case, `Couldbe'` $\perp$ is a synonym for $\perp$
- for the newtype, $\perp$ is a synonym for `Couldbe` $\perp$

with the agreement that a pattern match on $\perp$ fails and that a match on *`Constructor`*$\perp$ does not.

Newtypes may also be used to define recursive types. An example is the alternate definition of the list type `[a]`

[24] Chapter 65 on page 459

```
newtype List a = In (Maybe (a, List a))
```

Again, the point is that the constructor `In` does not introduce an additional lifting with ⊥.

56.6 Other Selected Topics

56.6.1 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 extent, the compiler performs **strictness analysis** just like we proved in some functions to be strict section Strict Functions[25]. 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[26].

56.6.2 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 ⊥ 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}
   \      /
    \    /
   ⊥ = {True, False}
```

The values `True` and `False` are encoded as the singleton sets `{True}` and `{False}` and ⊥ is the set of all possible values.

[25] Chapter 56.4.1 on page 373

[26] `http://haskell.org/haskellwiki/Research_papers/Compilation#Strictness`

Another example is `Maybe Bool`:

```
 {Just True}   {Just False}
        \     /
         \   /
{Nothing} {Just True, Just False}
    \      /
     \    /
 ⊥ = {Nothing, Just True, Just False}
```

We see that the semantic approximation order is equivalent to set inclusion, but with arguments switched:

$$x \sqsubseteq y \Longleftrightarrow x \supseteq y$$

This approach can be used to give a semantics to exceptions in Haskell[27].

56.6.3 Naïve Sets are unsuited for Recursive Data Types

In the section What to choose as Semantic Domain?[28], 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*[29].

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

`⊥, (⊥ -> Bool) -> Bool, (((⊥ -> Bool) -> Bool) -> Bool) -> Bool` and so on

where ⊥ denotes the domain with the single inhabitant ⊥. While the author of this text admittedly has no clue on what such a thing should mean, the constructor gives a perfectly

[27] 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.

[28] Chapter 56.1.2 on page 363

[29] John C. Reynolds. *Polymorphism is not set-theoretic.* INRIA Rapports de Recherche No. 296. May 1984.

well defined object for `U`. We see that the type `(U -> Bool) -> Bool` merely consists of shifted approximating sequences which means that it is isomorphic to `U`.

As a last note, Reynolds actually constructs an equivalent of `U` 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 `fix :: (a -> a) -> 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 -> Bool` should not have a bigger cardinality than `A`.

56.7 External Links

w:Denotational semantics[30]

Online books about Denotational Semantics

- Denotational Semantics. A Methodology for Language Development . Allyn and Bacon , , 1986

[30] `http://en.wikipedia.org/wiki/Denotational%20semantics`

57 Category theory

This article attempts to give an overview of category theory, in so far 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.

57.1 Introduction to categories

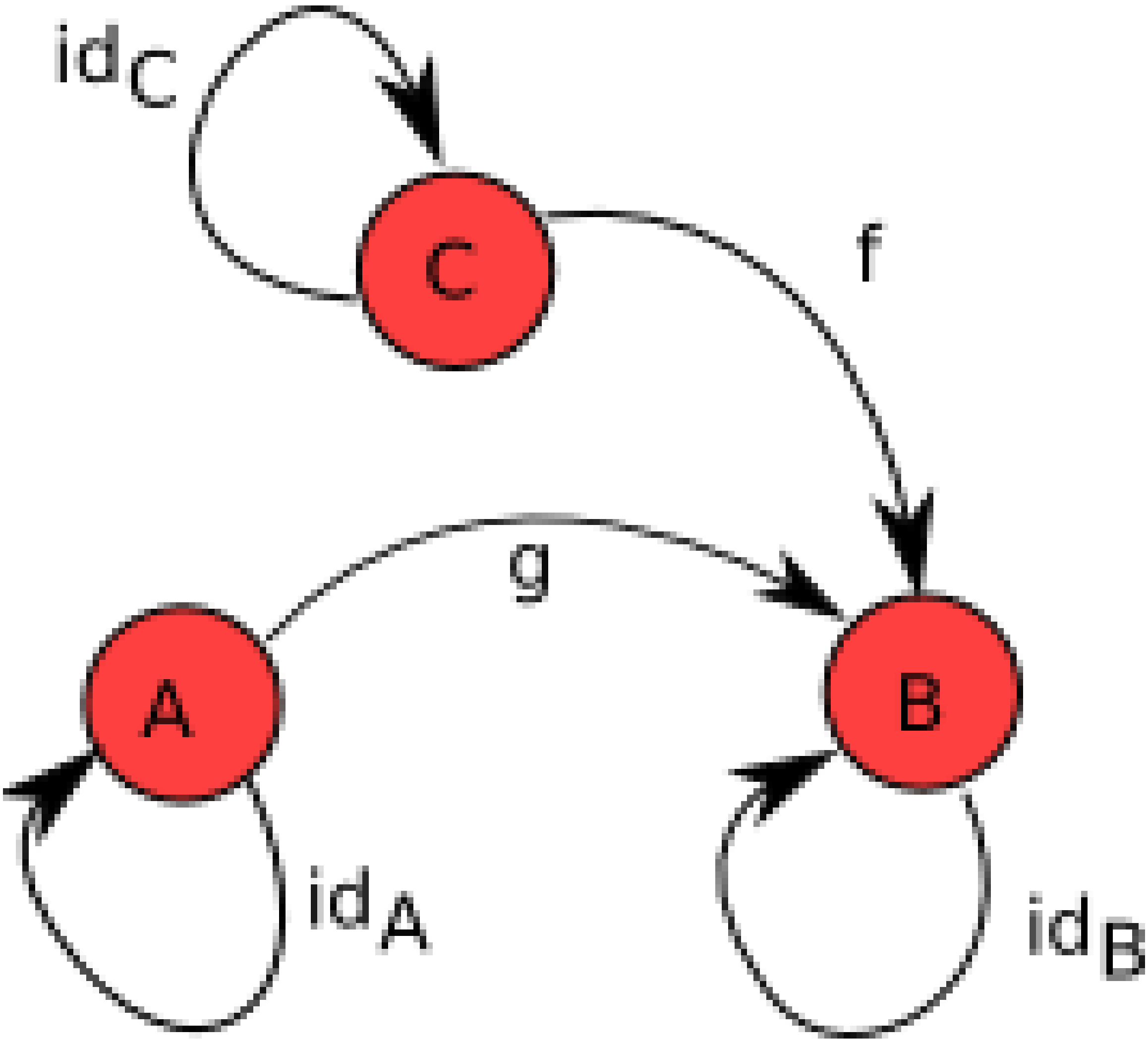


Figure 33 A simple category, with three objects A, B and C, three identity morphisms id_A, id_B and id_C, and two other morphisms $f : C \to B$ and $g : A \to B$. The third element (the specification of how to compose the morphisms) is not shown.

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 connotations in Haskell.) If f is a morphism with source object A and target object B, we write $f : A \to B$.
- A notion of **composition** of these morphisms. If $g : A \to B$ and $f : B \to C$ are two morphisms, they can be composed, resulting in a morphism $f \circ g : A \to C$.

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 $f : G \to H$ 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!

57.1.1 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 $f : B \to C$ and $g : A \to B$, then there must be some morphism $h : A \to C$ in the category such that $h = f \circ g$. We can see how this works using the following category:

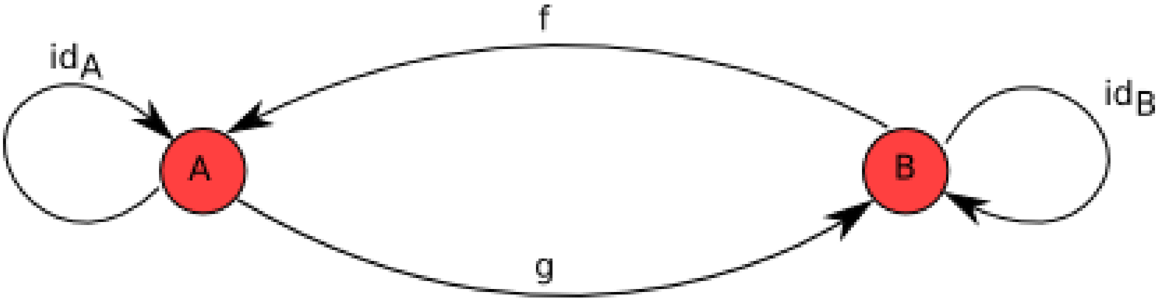


Figure 34

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*, $id_A : A \to A$ that is an identity of composition with other morphisms. Put precisely, for every morphism $g : A \to B$:

$$g \circ id_A = id_B \circ g = g$$

57.1.2 **Hask**, the Haskell category

The main category we'll be concerning ourselves with in this article is **Hask**, which treats Haskell types as objects and Haskell functions as morphisms and uses `(.)` for composition: a function `f :: A -> B` for types `A` and `B` is a morphism in **Hask**. We can check the first and second law 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
```

[1] 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 -> A)`. With this in mind, the above law would be rewritten as:

```
(id :: B -> B) . f = f . (id :: A -> 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.

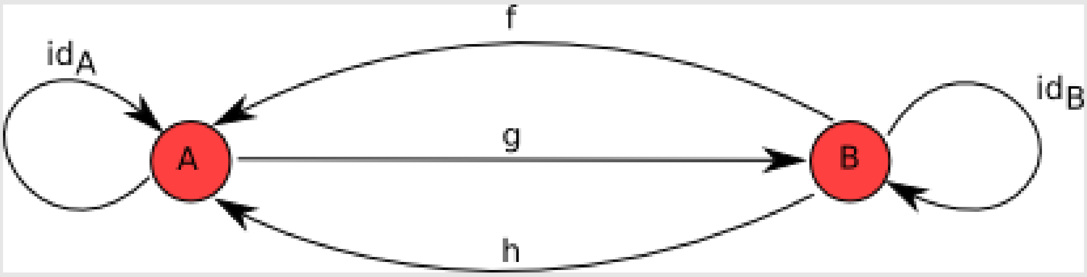


Figure 35

[1] 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 intents 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.

57.2 Functors

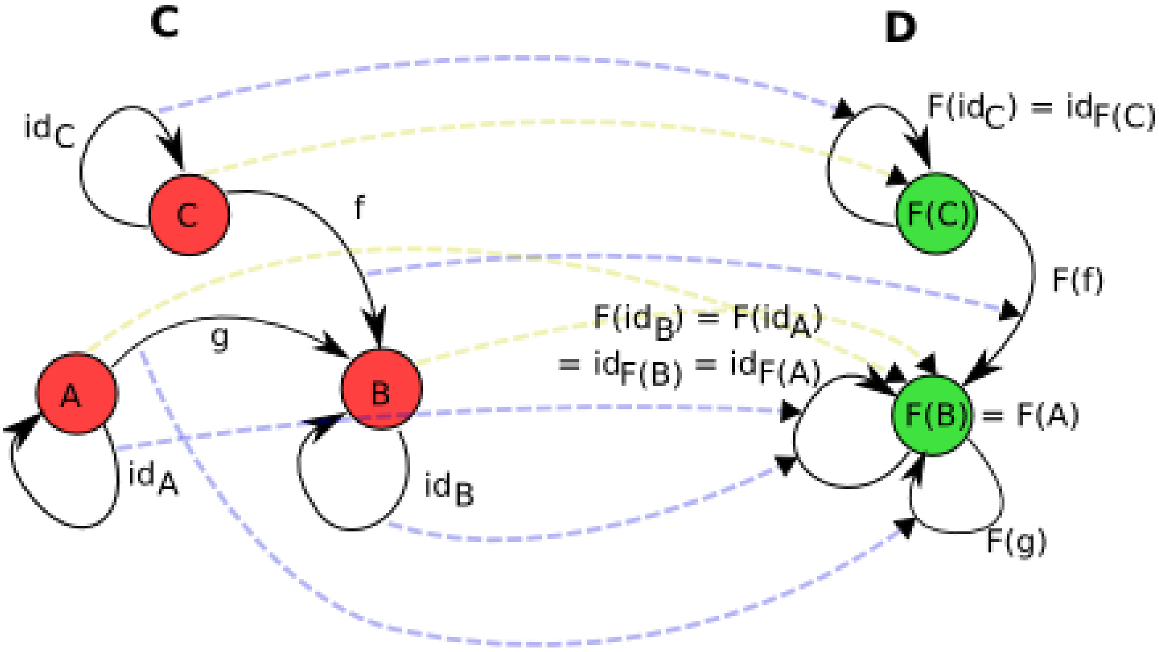

C
D
id_C
C
f
g
A
B
id_A
id_B
F(id_C) = id_{F(C)}
F(C)
F(f)
F(id_B) = F(id_A)
= id_{F(B)} = id_{F(A)}
F(B) = F(A)
F(g)

Figure 36 A functor between two categories, **C** and **D**. Of note is that the objects A and B both get mapped to the same object in **D**, and that therefore g becomes a morphism with the same source and target object (but isn't necessarily an identity), and id_A and id_B become the same morphism. The arrows showing the mapping of objects are shown in a dotted, pale olive. The arrows showing the mapping of morphisms are shown in a dotted, pale blue.

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 $F : C \to D$:

- Maps any object A in C to $F(A)$, in D.
- Maps morphisms $f : A \to B$ in C to $F(f) : F(A) \to F(B)$ in D.

One of the canonical examples of a functor is the forgetful functor **Grp** → **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** → **Set** which maps sets to their power sets and maps functions $f : X \to Y$ to functions $\mathcal{P}(X) \to \mathcal{P}(Y)$ which take inputs $U \subseteq X$ 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 $1_C : C \to C$, that just maps objects to themselves and morphisms to themselves. This will turn out to be useful in the monad laws[2] 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.:

[2] Chapter 57.4 on page 398

$$F(id_A) = id_{F(A)}$$

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.

57.2.1 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.

57.2.2 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.

57.3 Monads

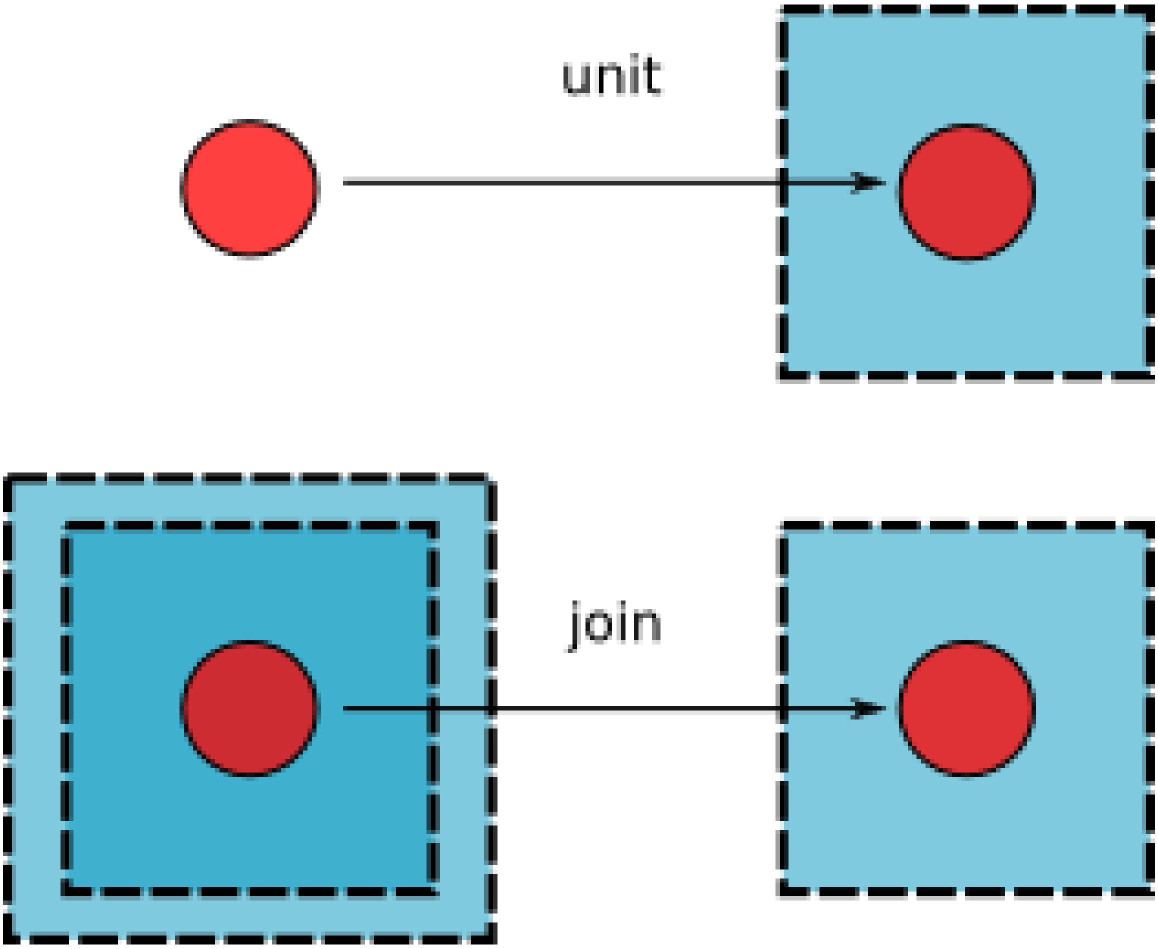


Figure 37 *unit* and *join*, the two morphisms that must exist for every object for a given monad.

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, from a category to that same category, that supports some additional structure. So, down to definitions. A monad is a functor $M : C \to C$, along with two morphisms[3] for every object X in C:

- $\mathit{unit}_X^M : X \to M(X)$
- $\mathit{join}_X^M : M(M(X)) \to M(X)$

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.

[3] 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 standard monad laws (laws 3 and 4) ^{Chapter57.4.3 on page 401}. 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.

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*; the other function, `(>>=)`, often called *bind*, bears no resemblance to *join*. There is however another monad function, `join :: Monad m => m (m a) -> m a`, that looks 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`, `fmap`, 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 `(>>=)`.[4] 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.

57.3.1 Example: the powerset functor is also a monad

The power set functor $P : \mathbf{Set} \to \mathbf{Set}$ described above forms a monad. For any set S you have a $\mathit{unit}_S(x) = \{x\}$, mapping elements to their singleton set. Note that each of these singleton sets are trivially a subset of S, so unit_S returns elements of the powerset of S, as is required. Also, you can define a function join_S 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,

$$\mathit{join}_S(L) = \bigcup L$$

[4] 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).

Hence P is a monad [5].

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	
Function type	**Definition**
Given a set S and a morphism $f : A \to B$:	
$P(f) : \mathcal{P}(A) \to \mathcal{P}(B)$	$(P(f))(S) = \{f(a) : a \in S\}$
$unit_S : S \to \mathcal{P}(S)$	$unit_S(x) = \{x\}$
$join_S : \mathcal{P}(\mathcal{P}(S)) \to \mathcal{P}(S)$	$join_S(L) = \bigcup L$
List monad from Haskell	
Function type	**Definition**
Given a type `T` and a function `f :: A -> B`	
`fmap f :: [A] -> [B]`	`fmap f xs = [ f a | a <- xs ]`
`return :: T -> [T]`	`return x = [x]`
`join :: [[T]] -> [T]`	`join xs = concat xs`

57.4 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 $M : C \to C$ and a morphism $f : A \to B$ for $A, B \in C$,

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. $join \circ M(M(f)) = M(f) \circ join$

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.

57.4.1 The first law

```
join . fmap join = join . join
```

[5] If you can prove that certain laws hold, which we'll explore in the next section.

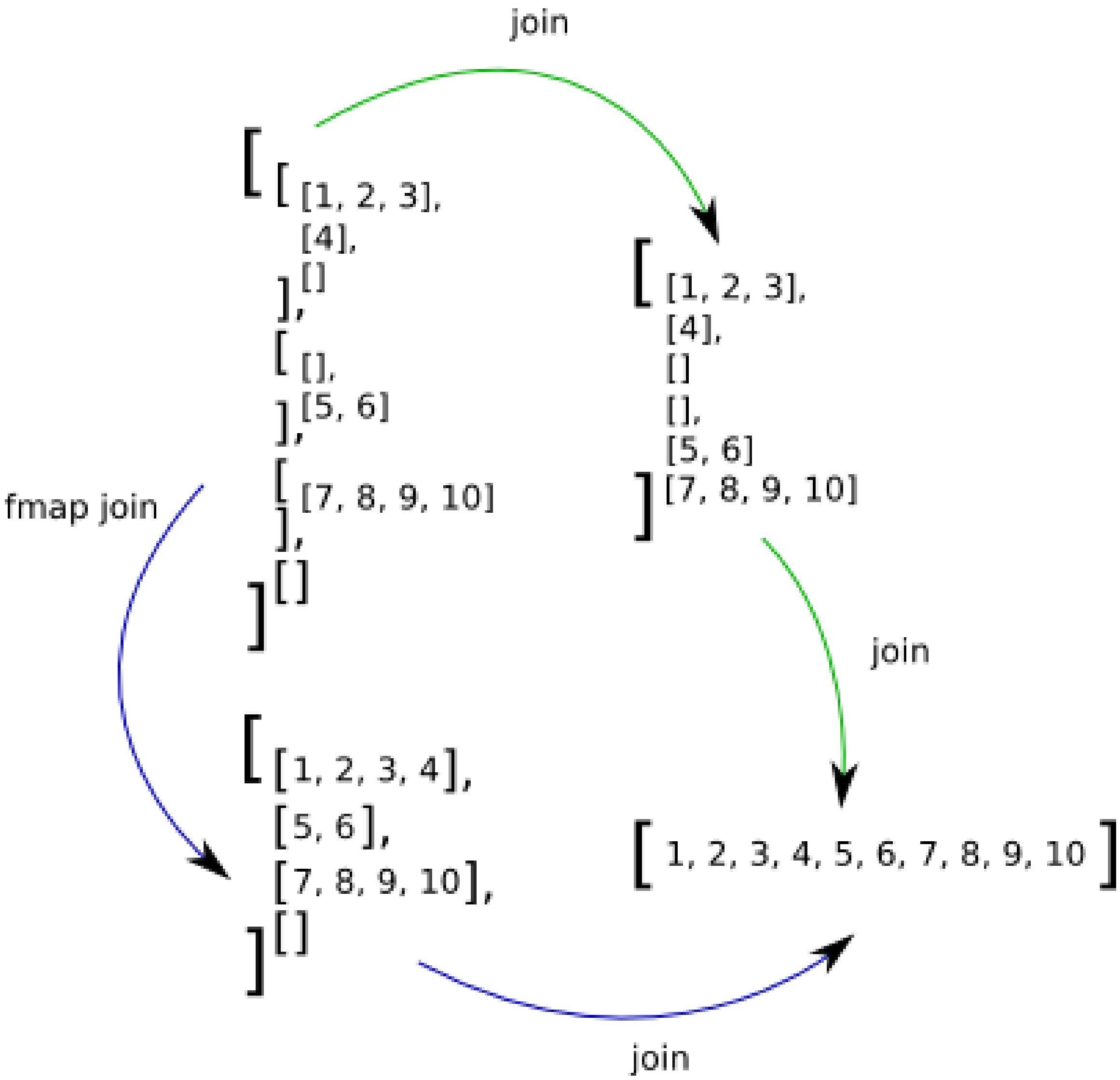


Figure 38 A demonstration of the first law for lists. Remember that `join` is `concat` and `fmap` is `map` in the list monad.

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`[6]`] -> [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.

[6] `http://en.wikibooks.org/wiki/a`

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`[7]`]` 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`.

`Maybe` is also a monad, with

```
return :: a -> Maybe a
return x = Just x

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.

57.4.2 The second law

`join . fmap return = join . return = id`

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 `join`ing the result should have the same effect whether you perform the `return` from inside the top layer or from outside it.

[7] http://en.wikibooks.org/wiki/a

Exercises:
Prove this second law for the Maybe monad.

57.4.3 The third and fourth laws

`return . f = fmap f . return`

`join . fmap (fmap f) = fmap f . join`

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:

1. `\x -> return (f x) = \x -> fmap f (return x)`
2. `\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.

57.4.4 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)) m             -- 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**
`return x >>= f = f x`	`do { v <- return x; f v } = do { f x }`
`m >>= return = m`	`do { v <- m; return v } = do { m }`

`(m >>= f) >>= g = m >>= (\x -> f x >>= g)`	`do { y <- do { x <- m; f x };`
	`g y }`
	`=`
	`do { x <- m;`
	`y <- f x;`
	`g y }`

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.

57.5 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.

58 The Curry-Howard isomorphism

The **Curry-Howard isomorphism** is a striking relationship connecting two seemingly unrelated areas of mathematics --- type theory and structural logic.

58.1 Introduction

The Curry-Howard isomorphism, hereafter referred to as simply C-H, tells us that in order to prove any mathematical theorem, all we have to do is construct a certain type which reflects the nature of that theorem, then find a value that has that type. This seems extremely weird at first: what do types have to do with theorems? However, as we shall see, the two are very closely related. A quick note before we begin: for these introductory paragraphs, we ignore the existence of expressions like `error` and `undefined` whose denotational semantics[1] are $\perp$. These have an extremely important role, but we will consider them separately in due time. We also ignore functions that bypass the type system like `unsafeCoerce#`.

We can build incredibly complicated types using Haskell's higher-order functions[2] feature. We might want to ask the question: given an arbitrary type, under what conditions does there exist a value with that type (we say the type is *inhabited*)? A first guess might be 'all the time', but this quickly breaks down under examples. For example, there is no function with type `a -> b`, because we have no way of turning something of type `a` into something of a completely different type `b` (unless we know in advance which types `a` and `b` are, in which case we're talking about a monomorphic function, such as `ord :: Char -> Int`).

Incredibly, it turns out that a type is only inhabited when it corresponds to a true theorem in mathematical logic. But what is the nature of this correspondence? What does a type like `a -> b` mean in the context of logic?

58.1.1 A crash course in formal logic

We need some background on formal logic before we can begin to explore its relationship to type theory. This is a *very* brief introduction; for a wider grounding we recommend you consult an introductory textbook on the subject matter.

In everyday language we use a lot of 'If... then...' sentences. For example, 'If the weather is nice today, then we'll walk into town'. These kinds of statements also crop up in mathematics; we can say things like 'If x is positive, then it has a (real) square root'. Formal logic is a way of translating these statements from loose, woolly, ambiguous English into

[1] Chapter 56 on page 361

[2] Chapter 19 on page 131

precise symbolism. We use the $\rightarrow$ sign (read as 'implies') to indicate that something is true if something else is true. For example, our earlier statement could be recast as 'The weather is nice today $\rightarrow$ we'll walk into town', which means that it is true that we'll walk into town if it is true that the weather is nice, which is just the same as the 'If... then...' version we used earlier. We'll often use letters to stand for entire statements, so for example if W is the statement 'the weather is nice', and T is the statement 'we'll walk into town', then our example becomes simply $W \rightarrow T$.

Notice the crafty way we phrased our definition of $\rightarrow$. $P \rightarrow Q$ means that *if* P is true, *then* Q is true. But if Q is some statement that is always true, no matter what the circumstances --- like 'the sun is hot' --- then it doesn't matter what P is. P could even be a false statement, Q would still be true if P were true, so $P \rightarrow Q$ would still hold. The fact that Q would be true if P isn't true is a whole different matter; we're not asking that question when we say $P \rightarrow Q$. So $\rightarrow$ doesn't really represent any kind of cause-effect relationship; things like 'the sky is pink $\rightarrow$ the sun is hot' are still valid statements. [3]

Other things that crop up lots in both everyday language and mathematics are things called conjunctions and disjunctions. The former represent statements involving an 'and', the latter statements involving an 'or'. We could represent the statement 'I will buy this magazine if it's in stock and I have enough money' by the symbolism $(M \wedge S) \rightarrow B$, where $M =$ 'I have enough money', $S =$ 'The magazine is in stock', $B =$ 'I will buy the magazine'. Essentially, one can just read the symbol $\wedge$ as 'and'. Similarly, one can read the symbol $\vee$ as 'or', so that the statement 'I will either walk or get the train to work' could be represented as $W \vee T$, where $W =$ 'I will walk to work', and $T =$ 'I will get the train to work'.

Using these symbols, and a few more which will be introduced as we go, we can produce arbitrarily complicated symbol strings. There are two classes of these symbol strings: those that represent true statements, often called the **theorems**; and those which represent false statements, called the **nontheorems**. Note that whether a symbol string is a theorem or nontheorem depends on what the letters stand for, so $P \vee Q$ is a theorem if, for example, P represents the statement 'It is daytime' and Q represents the statement 'It is night time' (ignoring exceptions like twilight), but it would be a nontheorem if P were 'Trees are blue' and Q were 'All birds can fly'. We'll often call a symbol string a **proposition** if we don't know whether it's a theorem or not.

There are *many* more subtleties to the subject of logic (including the fact that when we say 'If you eat your dinner you'll get dessert' we actually mean 'If and only if you eat your dinner will you get dessert'). If this is a subject that interests you, there are many textbooks around that comprehensively cover the subject.

[3] Another way of looking at this is that we're trying to *define* our logical operator $\rightarrow$ such that it captures our intuition of the "if... then" construct in natural language. So we want statements like for all all naturals x, "x is even" $\rightarrow$ "$x + 1$ is odd" to be true. I.e. that implication must hold when we substitute x for any natural including, say, 5. But "5 is even" and "6 is odd" are both false, so we must have that False $\rightarrow$ False is true. Similarly by considering the statement for all naturals $x > 3$, "x is prime" $\rightarrow$ "$x + 1$ is not prime", we must have that False $\rightarrow$ True is true. And obviously True $\rightarrow$ True must be true, and True $\rightarrow$ False is false. So we have that $x \rightarrow y$ unless x is true and y false.

58.1.2 Propositions are types

So, given a type `a -> b`, what does that mean in terms of symbolistic logic? Handily, it simply means that $a \to b$. Of course, this only makes sense if `a` and `b` are types which can further be interpreted in our symbolistic logic. This is the essence of C-H. Furthermore, as we mentioned before, $a \to b$ is a theorem if and only if `a -> b` is an inhabited type.

Let's see this using one of the simplest of Haskell functions. `const` has the type `a -> b -> a`. Translated into logic, we have that $a \to b \to a$. This must be a theorem, as the type `a -> b -> a` is inhabited by the value `const`. Now, another way of expressing $a \to b$ is that 'If we assume a is true, then b must be true.' So $a \to b \to a$ means that if we assume a is true, then if we further assume that b is true, then we can conclude a. This is of course a theorem; we assumed a, so a is true under our assumptions.

58.1.3 The problem with ⊥

We've mentioned that a type corresponds to a theorem if that type is inhabited. However, in Haskell, every type is inhabited by the value `undefined`. Indeed, more generally, anything with type `forall a. a`, a value with denotational semantics of ⊥, is a problem. ⊥ in type theory corresponds to inconsistency in logic; we can prove any theorem using Haskell types because every type is inhabited. Therefore, Haskell's type system actually corresponds to an inconsistent logic system. However, if we work with a limited subset of Haskell's type system, and in particular disallow polymorphic types, we have a consistent logic system we can do some cool stuff in. Hereafter it is assumed we are working in such a type system.

Now that we have the basics of C-H, we can begin to unpack a little more the relationship between types and propositions.

58.2 Logical operations and their equivalents

The essence of symbolic logic is a set of propositions, such as P and Q, and different ways of combining these propositions such as $Q \to P$ or $P \vee Q$. These ways of combining propositions can be thought of as operations on propositions. By C-H, propositions correspond to types, so we should have that the C-H equivalents of these proposition combinators are type operations, more normally known as type constructors. We've already seen an example of this: the implication operator → in logic corresponds to the type constructor `(->)`. The rest of this section proceeds to explore the rest of the proposition combinators and explain their correspondence.

58.2.1 Conjunction and Disjunction

In order for $A \wedge B$ to be a theorem, both A and B must be theorems. So a proof for $A \wedge B$ amounts to proving both A and B. Remember that to prove a proposition A we find a value of type `A`, where A and `A` are C-H correspondents. So in this instance we wish to find a value that contains two sub-values: the first whose type corresponds to A, and the second whose type corresponds to B. This sounds remarkably like a pair. Indeed, we represent the symbol string $A \wedge B$ by `(a, b)`, where `a` corresponds to A and `b` corresponds to B.

Disjunction is opposite to conjunction. In order for $A \vee B$ to be a theorem, either A or B must be a theorem. Again, we search for a value which contains either a value of type `A` or a value of type `B`. This is `Either`. `Either A B` is the type which corresponds to the proposition $A \vee B$.

58.2.2 Falsity

It is occasionally useful to represent a false statement in our logic system. By definition, a false statement is one that can't be proven. So we're looking for a type which isn't inhabited. Although none of these types exist in the default libraries (don't get confused with the `()` type, which has precisely one value), we can define one, if we turn on the `-XEmptyDataDecls` flag in GHC:

```
data Void
```

The effect of omitting the constructors means that `Void` is an uninhabited type. So the `Void` type corresponds to a nontheorem in our logic. There are a few handy corollaries here:

1. `(Void, A)` and `(A, Void)` are both uninhabited types for any type `A`, corresponding to the fact that $F \wedge A$ and $A \wedge F$ are both nontheorems if F is a nontheorem.
2. `Either Void A` and `Either A Void` are essentially the same as `A` for any type `A`, [4] corresponding to the fact that $F \vee A$ and $A \vee F$, where F is a nontheorem, are theorems only if A is a theorem.
3. Any type that corresponds to a nontheorem can be replaced with `Void`. This is because any nontheorem-type must be uninhabited, so replacing it with `Void` everywhere doesn't change anything. `Void` is really equivalent to any nontheorem type[5].
4. As we remarked in the first section, the implication $P \to Q$ is true if Q is true, regardless of the truth value of P. So we should be able to find a term with type `Void -> a`. In fact one does exist, but it's somewhat complicated to explain: the answer is the *empty function*. We can define a function `f :: A -> B` as a (probably infinite) set of pairs whose first element is an element of `A` (the *domain*) and second element is `f`'s output on this term, an element of `B` (the *range*). For example, the successor function on the naturals is represented as `{(0,1), (1,2), (2,3), ...}`. Note that in order to be a (total and well-defined) function, we must have precisely one pair `(a, f a)` for each term `a` with type `A`.
   The empty function, let's call it `empty` is represented in this way by the empty set. But as we must have a pair for each element of the domain, and there no pairs in our representation, the domain type must be empty, i.e. `Void`. What about the range type? `empty` never produces any output, so there are no restrictions placed on the range type. Thus, it is valid to assume that the range type has any type, so we can

[4] Technically, the types `Either Void A` and `A` are isomorphic. Seeing as you can't have a value of type `Void`, every value in `Either Void A` must be a `Right`-tagged value, so the transformation just strips the `Right` constructors.

[5] Again, the technical statement is that `Void` is isomorphic to any type which is a nontheorem.

say `empty :: forall a. Void -> a`. Unfortunately, it's not possible to write this function in Haskell; we'd ideally like to write something like:

```
    empty :: Void -> a
```

And stop there, but this is illegal Haskell. The closest we can come is the following:

```
    empty :: Void -> a
    empty _ = undefined
```

Alternatively:

```
    empty :: Void -> a
    empty = empty
```

Another reasonable way (also disallowed in Haskell) would be to write:

```
    empty x = case x of { }
```

The case statement is perfectly well formed since it handles every possible value of `x`. Note that this is perfectly safe, since the right-hand side of this function can never be reached (since we have nothing to pass it). So, the conclusion of all this is that `Void -> a` is an inhabited type, just as $P \to Q$ is true if P is false.

58.2.3 Negation

The $\neg$ operation in logic turns theorems into nontheorems and vice versa: if A is a theorem then $\neg A$ is a nontheorem; if A is a nontheorem then $\neg A$ is a theorem. How can we represent this in Haskell? The answer's a sneaky one. We define a type synonym:

```
type Not a = a -> Void
```

So for a type `A`, `Not A` is just `A -> Void`. How does this work? Well, if `A` was a theorem-type, then `A -> Void` must be uninhabited: there's no way any function could return any value, because the return type, `Void` has no values (The function has to provide values for all inhabitants of A)! On the other hand, if `A` was a nontheorem, then `A` can be replaced with `Void` as we explored in the last section. Then the function `id :: Void -> Void` is an inhabitant of `Not A`, so `Not A` is a theorem as required (The function doesn't have to provide any values, since there are no inhabitants in its domain. Nevertheless it's a function — with an empty graph).

58.3 Axiomatic logic and the combinatory calculus

So far we've only used some very basic features from Haskell's type system. Indeed, most of the features of logic we've mentioned can be explored using a very basic 'programming language', the combinator calculus. To fully appreciate how closely C-H ties together these two areas of mathematics, we need to *axiomatise* both our discussion of formal logic and our discussion of programming languages.

58.3.1 Axiomatic logic

We start with two axioms about how the → operation should behave (from now on, we assume that → is a right-associative function, i.e. $A \to B \to C$ means $A \to (B \to C)$):

1. $A \to B \to A$
2. $(A \to B \to C) \to (A \to B) \to A \to C$

The first axiom says that given any two propositions A and B, if we assume both A and B, we know that A is true. The second says that if A implies that B implies C (or equivalently, if C is true whenever A and B are true), and A itself implies B, then knowing A is true would be enough to conclude that C is true. This may seem complicated, but a bit of thought reveals it to be common sense. Imagine we have a collection of boxes of various colours, some with wheels, some with lids, such that all the red boxes with wheels also have lids, and all the red boxes have wheels. Pick one box. Let A = 'The box under consideration is red', B = 'The box under consideration has wheels', C = 'The box under consideration has a lid'. Then the second law tells us that, as $A \to B \to C$ (all red boxes with wheels also have lids), and $A \to B$ (all red boxes have wheels), then if A (if the box is red), then C must be true (the box has a lid).

We also allow one *inference law*, called *modus ponens*:

1. If $A \to B$, and A, then B.

This law allows us to create new theorems given old one. It should be fairly obvious; it is essentially the definition of what → means. This small basis provides a simple enough logic system which is expressive enough to cover most of our discussions. Here's a sample proof of the law $A \to A$ in our system:

Firstly, we know the two axioms to be theorems:

- $A \to B \to A$
- $(A \to B \to C) \to (A \to B) \to A \to C$

You'll notice that the left-hand side of the second axiom looks a bit like the first axiom. The second axiom guarantees that if we know that $A \to B \to C$, then we can conclude $(A \to B) \to A \to C$. In this case, if we let C be the same proposition as A, then we have that if $A \to B \to A$, then $(A \to B) \to A \to A$. But we already know $A \to B \to A$, that was the first axiom. Therefore, we have that $(A \to B) \to A \to A$ is a theorem. If we further let B be the proposition $C \to A$, for some other proposition C, then we have that if $A \to C \to A$, then $A \to A$. But, again, we know that $A \to C \to A$ (it's the first axiom again), so $A \to A$, as we wanted.

This example demonstrates that given some simple axioms and a simple way to make new theorems from old, we can derive more complicated theorems. It may take a while to get there --- here we had several lines of reasoning to prove just that the obvious statement $A \to A$ is a theorem! --- but we get there in the end. This kind of formalisation is attractive because we have essentially defined a very simple system, and it is very easy to study how that system works.

58.3.2 Combinator calculus

The lambda calculus[6] is a way of defining a simple programming language from a very simple basis. If you haven't already read the chapter that was just linked to, we recommend you read at least the introductory sections on the untyped version of the calculus. Here's a refresher in case you're feeling dusty. A lambda term is one of three things:

- A *value*, v.
- A *lambda abstraction* $\lambda x.t$, where t is another lambda term.
- An *application* $(t_1 t_2)$, where t_1 and t_2 are lambda terms.

There is one reduction law, too, called *beta-reduction*:

- $((\lambda x.t_1)t_2) \to t_1[x := t_2]$, where $t_1[x := t_2]$ means t_1 with all the free occurrences of x replaced with t_2.

As mentioned in the lambda calculus[7] article, the difficulty comes when trying to pin down the notion of a free occurrence of an identifier. The combinator calculus was invented by the American mathematician Haskell Curry (after whom a certain programming language is named) because of these difficulties. There are many variants on the basic combinator calculus, but we consider one of the simplest here. We start with two so-called **combinators**:

- **K** takes two values and returns the first. In the lambda calculus, $\mathbf{K} = \lambda xy.\ x$.
- **S** takes a binary function, a unary function and a value, and applies that value and the value passed into the unary function to the binary function. again, in the lambda calculus: $\mathbf{S} = \lambda xyz.\ xz(yz)$.

The first function you should recognise as `const`. The second is more complicated, it is the monadic function `ap` in the `((->) e)` monad (which is essentially Reader). These two combinators form a complete basis for the entire lambda calculus. Every lambda calculus program can be written using just these two functions.

58.4 Sample proofs

58.5 Intuitionistic vs classical logic

So far, all of the results we have proved are theorems of intuitionistic logic. Let's see what happens when we try to prove the basic theorem of classical logic, `Not Not A -> A`. Recall that this translates as `((A -> Void) -> Void) -> A`. So, given a function of type `(A -> Void) -> Void` we need a function of type A. Now a function of type `(A -> Void) -> Void` exists precisely if type `A -> Void` is uninhabited, or in other words if type A is inhabited.

[6] http://en.wikibooks.org/wiki/Haskell%2FLambda%20calculus

[7] http://en.wikibooks.org/wiki/Haskell%2FLambda%20calculus

So we need a function which takes any inhabited type, and returns an element of that type. Although it is simple enough to do this on a computer - we need only find the "simplest" or "first" inhabitant of each type - there is no way to do this using standard lambda-calculus or combinator techniques. So we see that this result cannot be proved using these two techniques, and hence that the underlying logic is intuitionistic rather than classical.

Instead, consider a traditional error handling function which calls `throw` when an error occurs, transferring computation to `catch`. The `throw` function cancels any return value from the original function, so it has type `A -> Void`, where `A` is the type of its arguments. The `catch` function then takes the `throw` function as its argument, and, if the `throw` triggers (i.e. returns a `Void`) will return the argument of the `throw` function. So the type of `catch` is `((A -> Void) -> Void) -> A`. [8]

[8] This argument is taken from Dan Piponi . Adventures in Classical Land Adventures in Classical Land . *The Monad Reader* ,

59 fix and recursion

The fix function is a particularly weird-looking function when you first see it. However, it is useful for one main theoretical reason: introducing it into the (typed) lambda calculus as a primitive allows you to define recursive functions.

59.1 Introducing fix

Let's have the definition of fix before we go any further:

```
fix :: (a -> a) -> a
fix f = let x = f x in x
```

This immediately seems quite magical. Surely fix f will yield an infinite application stream of fs: f (f (f (...)))? The resolution to this is our good friend, *lazy evaluation.* Essentially, this sequence of applications of f will converge to a value if (and only if) f is a lazy function. Let's see some examples:

Example:
fix examples

```
Prelude> :m Control.Monad.Fix
Prelude Control.Monad.Fix> fix (2+)
*** Exception: stack overflow
Prelude Control.Monad.Fix> fix (const "hello")
"hello"
Prelude Control.Monad.Fix> fix (1:)
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...
```

We first import the Control.Monad.Fix module to bring fix into scope (this is also available in the Data.Function). Then we try some examples. Since the definition of fix is so simple, let's expand our examples to explain what happens:

```
  fix (2+)
= 2 + (fix (2+))
= 2 + (2 + fix (2+))
= 4 + (fix (2+))
= 4 + (2 + fix (2+))
= 6 + fix (2+)
= ...
```

It's clear that this will never converge to any value. Let's expand the next example:

```
  fix (const "hello")
= const "hello" (fix (const "hello"))
= "hello"
```

This is quite different: we can see after one expansion of the definition of `fix` that because `const` ignores its second argument, the evaluation concludes. The evaluation for the last example is a little different, but we can proceed similarly:

```
  fix (1:)
= 1 : fix (1:)
= 1 : (1 : fix (1:))
= 1 : (1 : (1 : fix (1:)))
```

Although this similarly looks like it'll never converge to a value, keep in mind that when you type `fix (1:)` into GHCi, what it's really doing is applying `show` to that. So we should look at how `show (fix (1:))` evaluates (for simplicity, we'll pretend `show` on lists doesn't put commas between items):

```
  show (fix (1:))
= "[" ++ map show (fix (1:)) ++ "]"
= "[" ++ map show (1 : fix (1:)) ++ "]"
= "[" ++ "1" ++ map show (fix (1:)) ++ "]"
= "[" ++ "1" ++ "1" ++ map show (fix (1:)) ++ "]"
```

So although the `map show (fix (1:))` will never terminate, it does produce output: GHCi can print the beginning of the string, `"[" ++ "1" ++ "1"`, and continue to print more as `map show (fix (1:))` produces more. This is lazy evaluation at work: the printing function doesn't need to consume its entire input string before beginning to print, it does so as soon as it can start.

Exercises:
What, if anything, will the following expressions converge to?

- `fix ("hello"++)`
- `fix (\x -> cycle (1:x))`
- `fix reverse`
- `fix id`
- `fix (\x -> take 2 $ cycle (1:x))`

59.2 `fix` and fixed points

A *fixed point* of a function `f` is a value `a` such that `f a == a`. For example, `0` is a fixed point of the function `(* 3)` since `0 * 3 == 0`. This is where the name of `fix` comes from: it finds the *least-defined fixed point* of a function. (We'll come to what "least defined" means in a minute.) Notice that for both of our examples above that converge, this is readily seen:

```
const "hello" "hello" -> "hello"
(1:) [1,1,..]         -> [1,1,...]
```

And since there's no number `x` such that `2+x == x`, it also makes sense that `fix (2+)` diverges.

Exercises:
For each of the functions `f` in the above exercises for which you decided that `fix f` converges, verify that `fix f` finds a fixed point.

In fact, it's obvious from the definition of `fix` that it finds a fixed point. All we need to do is write the equation for `fix` the other way around:

```
f (fix f) = fix f
```

Which is precisely the definition of a fixed point! So it seems that `fix` should always find a fixed point. But sometimes `fix` seems to fail at this, as sometimes it diverges. We can repair this property, however, if we bring in some denotational semantics[1]. Every Haskell type actually include a special value called bottom, written ⊥. So the values with type, for example, `Int` include, in fact, ⊥ as well as `1, 2, 3` etc.. Divergent computations are denoted by a value of ⊥, i.e., we have that `fix (2+) = ⊥`.

The special value `undefined` is also denoted by this ⊥. Now we can understand how `fix` finds fixed points of functions like `(2+)`:

Example:
Fixed points of `(2+)`

```
Prelude> (2+) undefined
*** Exception: Prelude.undefined
```

So feeding `undefined` (i.e., ⊥) to `(2+)` gives us `undefined` back. So ⊥ is a fixed point of `(2+)`!

In the case of `(2+)`, it is the only fixed point. However, there are other functions `f` with several fixed points for which `fix f` still diverges: `fix (*3)` diverges, but we remarked above that `0` is a fixed point of that function. This is where the "least-defined" clause comes in. Types in Haskell have a partial order[2] on them called *definedness*. In any type, ⊥ is the least-defined value (hence the name "bottom"). For simple types like `Int`, the only pairs in the partial order are `⊥≤1`, `⊥≤2` and so on. We do not have `m ≤n` for any non-bottom `Ints` `m`, `n`. Similar comments apply to other simple types like `Bool` and `()`. For "layered" values such as lists or `Maybe`, the picture is more complicated, and we refer to the chapter on denotational semantics[3].

[1] Chapter 56 on page 361

[2] http://en.wikipedia.org/wiki/Partial_order

[3] Chapter 56 on page 361

So since ⊥ is the least-defined value for all types and `fix` finds the least-defined fixed point, if `f ⊥= ⊥`, we will have `fix f = ⊥` (and the converse is also true). If you've read the denotational semantics article, you will recognise this as the criterion for a *strict function*: `fix f` diverges if and only if `f` is strict.

59.3 Recursion

If you've come across examples of `fix` on the internet, or on the #haskell IRC channel[4], the chances are that you've seen examples involving `fix` and recursion. Here's a classic example:

Example:
Encoding recursion with `fix`

```
Prelude> let fact n = if n == 0 then 1 else n * fact (n-1) in fact 5
120
Prelude> fix (\rec n -> if n == 0 then 1 else n * rec (n-1)) 5
120
```

Here we have used `fix` to "encode" the factorial function: note that (if we regard `fix` as a language primitive) our second definition of `fact` doesn't involve recursion at all. In a language like the typed lambda calculus that doesn't feature recursion, we can introduce `fix` in to write recursive functions in this way. Here are some more examples:

Example:
More `fix` examples

```
Prelude> fix (\rec f l -> if null l then [] else f (head l) : rec f (tail l))
 (+1) [1..3]
[2,3,4]
Prelude> map (fix (\rec n -> if n == 1 || n == 2 then 1 else rec (n-1) + rec
 (n-2))) [1..10]
[1,1,2,3,5,8,13,21,34,55]
```

So how does this work? Let's first approach it from a denotational point of view with our `fact` function. For brevity let's define:

```
fact' rec n = if n == 0 then 1 else n * rec (n-1)
```

So that we're computing `fix fact' 5`. `fix` will find a fixed point of `fact'`, i.e. the *function* `f` such that `f == fact' f`. But let's expand what this means:

```
f = fact' f
  = \n -> if n == 0 then 1 else n * f (n-1)
```

[4] http://www.haskell.org/haskellwiki/IRC_channel

All we did was substitute `rec` for `f` in the definition of `fact'`. But this looks exactly like a *recursive* definition of a factorial function! `fix` feeds `fact'` *itself* as its first parameter in order to create a recursive function out of a higher-order function.

We can also consider things from a more operational point of view. Let's actually expand the definition of `fix fact'`:

```
  fix fact'
= fact' (fix fact')
= (\rec n -> if n == 0 then 1 else n * rec (n-1)) (fix fact')
= \n -> if n == 0 then 1 else n * fix fact' (n-1)
= \n -> if n == 0 then 1 else n * fact' (fix fact') (n-1)
= \n -> if n == 0 then 1
        else n * (\rec n' -> if n' == 0 then 1 else n' * rec (n'-1)) (fix
fact') (n-1)
= \n -> if n == 0 then 1
        else n * (if n-1 == 0 then 1 else (n-1) * fix fact' (n-2))
= \n -> if n == 0 then 1
        else n * (if n-1 == 0 then 1
                  else (n-1) * (if n-2 == 0 then 1
                                else (n-2) * fix fact' (n-3)))
= ...
```

Notice that the use of `fix` allows us to keep "unravelling" the definition of `fact'`: every time we hit the `else` clause, we product another copy of `fact'` via the evaluation rule `fix fact' = fact' (fix fact')`, which functions as the next call in the recursion chain. Eventually we hit the `then` clause and bottom out of this chain.

Exercises:

1. Expand the other two examples we gave above in this sense. You may need a lot of paper for the Fibonacci example!
2. Write non-recursive versions of `filter` and `foldr`.

59.4 The typed lambda calculus

In this section we'll expand upon a point mentioned a few times in the previous section: how `fix` allows us to encode recursion in the typed lambda calculus. It presumes you've already met the typed lambda calculus. Recall that in the lambda calculus, there is no `let` clause or top-level bindings. Every program is a simple tree of lambda abstractions, applications and literals. Let's say we want to write a `fact` function. Assuming we have a type called `Nat` for the natural numbers, we'd start out something like the following:

```
λn:Nat. if iszero n then 1 else n * <blank> (n-1)
```

The problem is, how do we fill in the `<blank>`? We don't have a name for our function, so we can't call it recursively. The only way to bind names to terms is to use a lambda abstraction, so let's give that a go:

```
(λf:Nat→Nat. λn:Nat. if iszero n then 1 else n * f (n-1))
  (λm:Nat. if iszero m then 1 else m * <blank> (m-1))
```

This expands to:

```
λn:Nat. if iszero n then 1
        else n * (if iszero n-1 then 1 else (n-1) * <blank> (n-2))
```

We still have a `<blank>`. We could try to add one more layer in:

```
(λf:Nat→Nat. λn:Nat. if iszero n then 1 else n * f (n-1)
  ((λg:Nat→Nat. λm:Nat. if iszero n' then 1 else n' * g (m-1))
    (λp:Nat. if iszero p then 1 else p * <blank> (p-1))))

->

λn:Nat. if iszero n then 1
        else n * (if iszero n-1 then 1
                  else (n-1) * (if iszero n-2 then 1 else (n-2) * <blank>
(n-3)))
```

It's pretty clear we're never going to be able to get rid of this `<blank>`, no matter how many levels of naming we add in. Never, that is, unless we use `fix`, which, in essence, provides an object from which we can always unravel one more layer of recursion and still have what we started off:

```
fix (λf:Nat→Nat. λn:Nat. if iszero n then 1 else n * f (n-1))
```

This is a perfect factorial function in the typed lambda calculus plus `fix`.

`fix` is actually slightly more interesting than that in the context of the typed lambda calculus: if we introduce it into the language, then every type becomes inhabited, because given some concrete type `T`, the following expression has type `T`:

```
fix (λx:T. x)
```

This, in Haskell-speak, is `fix id`, which is denotationally ⊥. So we see that as soon as we introduce `fix` to the typed lambda calculus, the property that every well-typed term reduces to a value is lost.

59.5 Fix as a data type

It is also possible to make a fix data type in Haskell.

There are three ways of defining it.

```
newtype Fix f=Fix (f (Fix f))
```

or using the RankNTypes extension

```
newtype Mu f=Mu (forall a.(f a->a)->a)
data Nu f=forall a.Nu a (a->f a)
```

Mu and Nu help generalize folds, unfolds and refolds.

```
fold :: (f a -> a) -> Mu f -> a
fold g (Mu f)=f g
unfold :: (a -> f a) -> a -> Nu f
unfold f x=Nu x f
refold :: (a -> f a) -> (g a-> a) -> Mu f -> Nu g
refold f g=unfold g . fold f
```

Mu and Nu are restricted versions of Fix. Mu is used for making inductive noninfinite data and Nu is used for making coinductive infinite data. Eg)

```
newpoint Stream a=Stream (Nu ((,) a)) -- forsome b. (b,b->(a,b))
newpoint Void a=Void (Mu ((,) a)) -- forall b.((a,b)->b)->b
```

Unlike the fix point function the fix point types do not lead to bottom. In the following code Bot is perfectly defined. It is equivalent to the unit type ().

```
newtype Id a=Id a
newtype Bot=Bot (Fix Id) -- equals          newtype Bot=Bot Bot
-- There is only one allowable term. Bot $ Bot $ Bot $ Bot ..,
```

The Fix data type cannot model all forms of recursion. Take for instance this nonregular data type.

```
data Node a=Two a a|Three a a a
data FingerTree a=U a|Up (FingerTree (Node a))
```

It is not easy to implement this using Fix.

60 Haskell Performance

61 Introduction

61.1 Execution Model

61.1.1 Introducing Lazy Evaluation

Programming is not only about writing programs that work but also about programs that require little memory and time to execute on a computer. While both time and memory use are relatively straightforward to predict in an imperative programming language or in *strict* functional languages like LISP or ML, things are different here. In Haskell, expressions are evaluated on demand. For instance, the expression

```
head (map (2 *) [1 .. 10])
```

will be evaluated as follows

```
⇒ head (map (2 *) (1 : [2 .. 10]))      ([1 .. 10])
⇒ head (2 * 1 : map (2 *) [2 .. 10])    (map)
⇒ 2 * 1                                 (head)
⇒ 2                                     (*)
```

The function head demands only the first element of the list and consequently, the remaining part map (2 *) [2 .. 10] of the list is never evaluated. Since this strategy only performs as much evaluation as necessary, it's called **lazy evaluation**. The chapter ../Graph reduction/[1] will present it in detail.

While lazy evaluation is the commonly employed implementation technique for Haskell, the language standard[2] only specifies that Haskell has **non-strict** denotational semantics[3] without fixing a particular execution model. A function f with one argument is said to be *strict* if it doesn't terminate or yields an error whenever the evaluation of its argument will loop forever or is otherwise undefined. Writing ⊥ for the "result" of an infinite loop, the definition for strictness is

A function f is called **strict** if f ⊥ = ⊥ .

For example, trying to add 1 to a number that loops forever will still loop forever, so ⊥+1 = ⊥ and the addition function (+1) is strict. The function head is also strict since the first element of a list won't be available if the whole list is undefined. But it may well be that

[1] Chapter 63 on page 433

[2] http://www.haskell.org/onlinereport/

[3] Chapter 56 on page 361

the first element is well-defined while the remaining list is not. In this case, we have

```
head (x : ⊥) = x
```

This equation means that `head` does not evaluate the remaining list, otherwise it would loop as well. Thus, such purely algebraic strictness properties are a great help for reasoning about execution time and memory. In strict languages like LISP or ML, we would always have `head (x:⊥) = ⊥`, whereas Haskell being "non-strict" means that we can write functions which are not strict, like the above property of `head` or the simplest example `(const 1) ⊥ = 1`. With the help of the constant `undefined` from the Prelude[4], we can even explore strictness interactively

```
> head (1 : undefined)
1
> head undefined
*** Exception: Prelude.undefined
```

Strictness and ⊥ will be used through these chapters. ../Graph reduction/[5] presents further introductory examples and the denotational point of view is elaborated in ../Denotational semantics/[6].

61.1.2 Example: fold

The best way to get a first feeling for lazy evaluation is to study an example. Therefore, we will consider some prototypical use case of `foldr` and its variants. The needed basics for studying the time and space complexity of programs are recapped in chapter ../Algorithm complexity/[7].

Time

Consider the following function `isPrime` that examines whether a number is a prime number or not

```
isPrime n     = not $ any (`divides` n) [2..n-1]
d `divides` n = n `mod` d == 0
any p         = or . map p
or            = foldr (||) False
```

The helper function `any` from the Haskell Prelude[8] checks whether there is at least one element in the list that satisfies some property `p`. For `isPrime` , the property is "being a divisor of `n`".

[4] `http://hackage.haskell.org/packages/archive/base/4.1.0.0/doc/html/Prelude.html`

[5] Chapter 63 on page 433

[6] Chapter 56 on page 361

[7] Chapter 66 on page 461

[8] `http://hackage.haskell.org/packages/archive/base/4.1.0.0/doc/html/Prelude.html`

The amount of time it takes to evaluate an expression is of course measured by the number of reduction steps. If n is a prime number, the above algorithm will examine the full list of numbers from 2 to n-1 and thus has a worst-case running time of $O(n)$ reductions. However, if n is not a prime number, we do not need to loop through every one of these numbers, we can stop as soon as we found one divisor and report n as being composite. The joy of lazy evaluation is that this behavior is already *built-in* into the logical disjunction ||!

```
isPrime 42
 ⇒ not $ any (`divides` 42) [2..41]
 ⇒ not ( or (map (`divides` 42) [2..41]                         ) )
 ⇒ not ( or ((42 `mod` 2 == 0) :      map (`divides` 42) [3..41]) )
 ⇒ not (     (42 `mod` 2 == 0) || or (map (`divides` 42) [3..41]) )
 ⇒ not (                 True  || or (map (`divides` 42) [3..41]) )
 ⇒ not True
 ⇒ False
```

The function returns False after seeing that 42 is even, || does not look at its second argument when the first one determines the result to be True. In other words, we have the following strictness property

```
True || ⊥ = True
```

Of course, the above algorithm can be implemented with a custom loop. But the crux of lazy evaluation is that we could formulate the algorithm in a transparent way by reusing the standard foldr and still get early bail-out. Put differently, lazy evaluation is about formulating fast algorithms in a modular way. An extreme example is to use *infinite* data structures to efficiently modularize generate & prune - algorithms. This and many other neat techniques with lazy evaluation will be detailed in the chapter ../Laziness/[9].

It's neither feasible nor necessary to perform a detailed graph reduction to analyze execution time. Shortcuts are available, like non-strictness or the fact that lazy evaluation will always take fewer reduction steps than eager evaluation. They will be presented in ../Graph reduction/[10].

Space

While execution time is modeled by the number of reduction steps, memory usage is modeled by the size of an expression during evaluation. Unfortunately, it's harder to predict and deviating from the normal course lazy evaluation by more strictness can ameliorate it. More details in ../Graph reduction/[11]. Here, we will present the prototypical example for unexpected space behavior.

Naively summing a huge number of integers

[9] Chapter 64 on page 445

[10] Chapter 63 on page 433

[11] Chapter 63 on page 433

```
> foldr (+) 0 [1..1000000]
*** Exception: stack overflow
```

produces a stack overflow. What happens? The evaluation proceeds as follows

```
foldr (+) 0 [1..1000000]
 ⇒ 1+(foldr (+) 0 [2..1000000])
 ⇒ 1+(2+(foldr (+) 0 [3..1000000]))
 ⇒ 1+(2+(3+(foldr (+) 0 [4..1000000]))
```

and so on. We see that the expression grows larger and larger, needing more and more memory. In this case, the memory is allocated on the stack for performing the pending additions after the recursive calls to `foldr` return. At some point, the memory needed exceeds the maximum possible stack size raising the "stack overflow" error. Don't worry whether it's stack or heap, the thing to keep in mind is that the size of the expression corresponds to the memory used and we see that in general, evaluating `foldr (+) 0 [1..n]` needs $O(n)$ memory.

But it should be possible to do it in $O(1)$ space by keeping an accumulated sum of numbers seen so far, exploiting that `+` is associative. (Some readers may notice that this means to make the function tail recursive.) This is what foldl does:

```
foldl (+) 0 [1..n]
 ⇒ foldl (+) (0+1) [2..n]
 ⇒ foldl (+) ((0+1)+2) [3..n]
 ⇒ foldl (+) (((0+1)+2)+3) [4..n]
```

But much to our horror, the accumulated sum will not be reduced any further! It will grow on the heap until the end of the list is reached

```
 ⇒ ... ⇒ foldl (+) ((((0+1)+2)+3)+...) []
 ⇒ ((((0+1)+2)+3)+...)
```

and subsequent reduction of this huge unevaluated sum will fail with a stack overflow, too. (So, just introducing an accumulating parameter doesn't make it tail recursive.)

The problem is that the unevaluated sum is an overly large representation for a single integer and it's cheaper to evaluate it eagerly. This is what `foldl'` does:

```
foldl' f z []     = z
foldl' f z (x:xs) = z `seq` foldl' f (f z x) xs
```

Here, evaluating ``a `seq` b`` will reduce `a` to weak head normal form[12] before proceeding with the reduction of `b`. With this, the sum proceeds as

[12] Chapter 63.3.7 on page 439

```
foldl' (+) 0 [1..n]
 ⇒ foldl' (+) (0+1) [2..n]
 ⇒ foldl' (+) (1+2) [3..n]
 ⇒ foldl' (+) (3+3) [4..n]
 ⇒ foldl' (+) (6+4) [5..n]
 ⇒ ...
```

in constant space.

For more details and examples, read the chapter ../Graph reduction/[13].

61.1.3 Low-level Overhead

Compared to eager evaluation, lazy evaluation adds a considerable overhead, even integers or characters have to be stored as pointers since they might be ⊥. An array of 1-byte characters is several times more compact than a `String = [Char]` of similar length. With strictness annotations, unboxed types and automatic strictness analysis, the overhead can be reduced. The Haskell wiki is a good resource `http://www.haskell.org/haskellwiki/Performance` concerning these low-level details, the wikibook currently doesn't cover them.

61.2 Algorithms & Data Structures

61.2.1 Algorithms

While this wikibook is not a general book on algorithms, there are many techniques of writing efficient programs unique to functional programming. The chapter ../Algorithm complexity/[14] recaps the big-O notation and presents a few examples from practice.

In ../Laziness/[15], the focus is on exploiting lazy evaluation to write efficient algorithms in a modular way.

A common theme of ../Program derivation/[16] and ../Equational reasoning/[17] is to derive an efficient program from a specification by applying and proving equations like

```
map f . reverse  = reverse . map f
filter p . map f = map f . filter (p . f)
```

This quest has given rise to a gemstone, namely a purely algebraic approach to dynamic programming which will be introduced in *some chapter with a good name.*

Another equational technique known as **fusion** or **deforestation** aims to remove intermediate data structures in function compositions. For instance, the composition on the left

[13] Chapter 63 on page 433

[14] Chapter 66 on page 461

[15] Chapter 64 on page 445

[16] `http://en.wikibooks.org/wiki/..%2FProgram%20derivation%2F`

[17] `http://en.wikibooks.org/wiki/..%2FEquational%20reasoning%2F`

hand side of

```
map f . map g = map (f . g)
```

constructs and deconstructs an intermediate list whereas the right hand side is a single pass over the list. The general theme here is to fuse constructor-deconstructor pairs like

```
case (Just y) of { Just x -> f x; Nothing -> ...; } = f y
```

This will be detailed in *some chapter with a good name.*

61.2.2 Data structures

Choosing the right data structure is key to success. While lists are common and can lead surprisingly far, they are more a kind of materialized loops than a classical data structure with fast access and updates. Many languages have special support for particular data structures of that kind like arrays in C, hash tables in Perl or lists in LISP. Natively, Haskell favors any kind of tree. But thanks to parametric polymorphism and type classes, any abstract type like balanced binary trees is easy to use and reuse! The chapter ../Data structures/[18] details the natural choices of data structures for common problems.

Because Haskell is purely functional, data structures share the common trait of being **persistent**. This means that older version of a data structure are still available like in

```
foo set = (newset, set)
  where newset = insert "bar" set
```

Compared to that, the default in imperative languages is **ephemeral**, i.e. data is updated in place and old versions are overridden. For example, arrays are ephemeral. That's why they are either immutable or require monads to use in Haskell. Fortunately, many ephemeral structures like queues or heaps have persistent counterparts and the chapter ../Data structures/[19] will also gently introduce some design principles in that direction, for instance concerning amortization.

61.3 Parallelism

The goal of parallelism is to run an algorithm on multiple cores / computers in parallel for faster results. Because Haskell doesn't impose an execution order thanks to its purity, it is well-suited for formulating parallel algorithms.

[18] http://en.wikibooks.org/wiki/..%2FData%20structures%2F

[19] http://en.wikibooks.org/wiki/..%2FData%20structures%2F

Currently, parallelism in Haskell is still a research topic and subject to experimentation. There are combinators for controlling the execution order Control.Parallel.Strategies[20], but they are rather fine-grained. Research on a less ambitious but more practical alternative Data Parallel Haskell[21] is ongoing.

The chapter ../Parallelism/[22] is not yet written but is intended to be a gentle introduction to parallel algorithms and current possibilities in Haskell.

[20] http://hackage.haskell.org/packages/archive/parallel/1.0.0.0/doc/html/Control-Parallel-Strategies.html

[21] http://haskell.org/haskellwiki/GHC/Data_Parallel_Haskell

[22] http://en.wikibooks.org/wiki/..%2FParallelism%2F

62 Step by Step Examples

Goal: Explain optimizations step by step with examples that actually happened.

62.1 Tight loop

dons: Write Haskell as fast as C: exploiting strictness, laziness and recursion.[1] << DEAD LINK

dons: Write Haskell as fast as C: exploiting strictness, laziness and recursion.[2]

62.2 CSV Parsing

haskell-cafe: another Newbie performance question[3] I hope he doesn't mind if I post his code here, I still have to ask him. -- apfeλmus[4] 08:46, 18 May 2008 (UTC)

```
type CSV = [[String]]

main = do
                  args <- getArgs
                  file <- readFile (head args)
                  writeFile (head args ++ "2") (processFile (args !! 1) file)

processFile s       = writeCSV . doInteraction s . readCSV
doInteraction line csv = insertLine (show line) (length csv - 1) csv
writeCSV            = (\x -> x ++ "\n") . concat . intersperse "\n" . (map (concat
 . intersperse "," . (map show)))
insertLine line pos csv = (take pos csv) ++ [readCSVLine line] ++ drop pos csv
readCSVLine         = read . (\x -> "["++x++"]")
readCSV             = map readCSVLine . lines
```

I think there was another cvs parsing thread on the mailing list which I deemed appropriate, but I can't remember.

[1] http://cgi.cse.unsw.edu.au/~dons/blog/2008/05/16#fast

[2] http://donsbot.wordpress.com/2008/05/06/write-haskell-as-fast-as-c-exploiting-strictness-laziness-and-re
#fast

[3] http://thread.gmane.org/gmane.comp.lang.haskell.cafe/40114

[4] http://en.wikibooks.org/wiki/User%3AApfelmus

62.3 Space Leak

jkff asked about some code in #haskell which was analyzing a logfile. Basically, it was building a histogram

```
foldl' (\m (x,y) -> insertWith' x (\[y] ys -> y:ys) [y] m) M.empty
  [(ByteString.copy foo, ByteString.copy bar) | (foo,bar) <- map (match regex)
 lines]
```

The input was a 1GB logfile and the program blew the available memory mainly because the `ByteString.copy` weren't forced and the whole file lingered around in memory.

63 Graph reduction

63.1 Notes and TODOs

- *TODO: Pour lazy evaluation explanation from ../Laziness/[1] into this mold.*
- *TODO: better section names.*
- *TODO: ponder the graphical representation of graphs.*
  - *No grapical 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*

63.2 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.

[1] Chapter 64 on page 445

63.3 Evaluating Expressions by Lazy Evaluation

63.3.1 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 **reduction**s 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.

63.3.2 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 redex**es that are not inside another redex. Here, the outermost reduction uses fewer 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.

63.3.3 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.

63.3.4 Graph Reduction (Reduction + Sharing)

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,

```
 ----------
|   |     ↓
◊ * ◊     (1+2)
```

represents the expression (1+2)*(1+2). Now, the **outermost graph reduction** of square (1+2) proceeds as follows

```
square (1+2)
 ⇒  __________          (square)
   |   |     ↓
   ◊ * ◊     (1+2)
 ⇒  __________          (+)
   |   |     ↓
   ◊ * ◊      3

 ⇒ 9                    (*)
```

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[2].

Sharing of expressions is also introduced with let and where constructs. For instance, consider Heron's formula[3] for the area of a triangle with sides a,b and c:

```
area a b c = let s = (a+b+c)/2 in
     sqrt (s*(s-a)*(s-b)*(s-c))
```

Instantiating this to an equilateral triangle will reduce as

```
area 1 1 1
 ⇒       _____________________            (area)
        |  |    |     |      ↓
   sqrt (◊*(◊-a)*(◊-b)*(◊-c))  ((1+1+1)/2)
 ⇒       _____________________            (+),(+),(/)
        |  |    |     |      ↓
   sqrt (◊*(◊-a)*(◊-b)*(◊-c))  1.5
 ⇒
   ...
 ⇒
   0.433012702
```

[2] Chapter 63.5 on page 442

[3] http://en.wikipedia.org/wiki/Heron%27s%20formula

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.[4]

Any implementation of Haskell is in some form based on outermost graph reduction which thus provides a good model for reasoning about the asympotic 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?

63.3.5 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

[4] John Maraist, Martin Odersky, and Philip Wadler . The call-by-need lambda calculus The call-by-need lambda calculus ^{homepages.inf.ed.ac.uk/wadler/topics/call-{}by-{}need.html#need-{}journal} . *Journal of Functional Programming* , **8** : 257-317 May 1998

```
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. 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]`

63.3.6 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 $\mathit{expression}_1 \mathit{expression}_2$, call-by-need first reduce $\mathit{expression}_1$ until this becomes a function whose definition can be unfolded with the argument $\mathit{expression}_2$. 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`****.*

63.3.7 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

```
f x = expression
f   = \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 `(\variable->expression`$_1$`) expression`$_2$
- or a case-expression `case expressionof { ... }`

lazy evaluation.

Weak Head Normal Form

An expression is in weak head normal form, iff it is either

- a constructor (possibly applied to arguments) like `True`, `Just (square 42)` or `(:) 1`
- a built-in function applied to too few arguments (perhaps none) like `(+) 2` or `sqrt`.
- or a lambda abstraction `\x -> 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.

63.3.8 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 (`f x = loop`doesn't need its argument, for example).

63.4 Controlling Space

"Space" here may be better visualized as traversal of a graph. Either a data structure, or an induced dependencies graph. For instance : Fibonacci(N) depends on : Nothing if N = 0 or N = 1 ; Fibonacci(N-1) and Fibonacci(N-2) else. As Fibonacci(N-1) depends on Fibonacci(N-2), the induced graph is not a tree. Therefore, there is a correspondence between implementation technique and data structure traversal :

Corresponding Implementation technique	**Data Structure Traversal**
Memoization	Depth First Search (keep every intermediary result in memory)
Parallel evaluation	Breadth First Search (keep every intermediary result in memory, too)
Sharing	Directed acyclic graph traversal (Maintain only a "frontier" in memory.)
Usual recursion	Tree traversal (Fill a stack)
Tail recursion	List traversal / Greedy Search (Constant space)

The classical :

```
fibo 0 = 1
fibo 1 = 1
fibo n = fibo (n-1) + fibo (n-2)
```

Is a tree traversal applied to a directed acyclic graph for the worse. The optimized version :

```
fibo n =
 let f a b m =
    if m = 0 then a
    if m = 1 then b
    f b (a+b) (m-1)
 in f 1 1 n
```

Uses a DAG traversal. Luckily, the frontier size is constant, so it's a tail recursive algorithm.

NOTE: The chapter ../Strictness[5] is intended to elaborate on the stuff here.

[5] Chapter 65 on page 459

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).

63.4.1 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 []      = [6]
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?).

63.4.2 Tail recursion

NOTE: Does this belong to the space section? I think so, it's about stack space.

Tail recursion in Haskell looks different.

63.5 Reasoning about Time

Note: introducing strictness before the upper time bound saves some hassle with explanation?

63.5.1 Lazy eval < Eager eval

When reasoning about execution time, naively performing graph reduction by hand to get a clue on what's going on 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.

63.5.2 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 anything = ⊥ 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 ⊥.*
- *Can* `head . mergesort`*be analyzed only with ⊥? In any case, this example is too involed and belongs to ../Laziness[7].*

[7] Chapter 64 on page 445

63.5.3 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.

63.6 Implementation of Graph reduction

Small talk 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.

63.7 References

- Introduction to Functional Programming using Haskell . Prentice Hall , , 1998
- The Implementation of Functional Programming Languages . Prentice Hall , , 1987

64 Laziness

Quote:
Hard work pays off later. Laziness pays off now! – Steven Wright

64.1 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 the reasons and implications of 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. In practice, laziness often introduces an overhead that leads programmers to hunt for places where they can make their code more strict. 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.

64.1.1 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 use 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.

64.2 Thunks and Weak head normal form

There are two principles you need to understand to get how programs execute in Haskell. First, we have the property of nonstrictness: we evaluate as little as possible for as long as possible. Second, 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 `(*thunk*, *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 ...
```

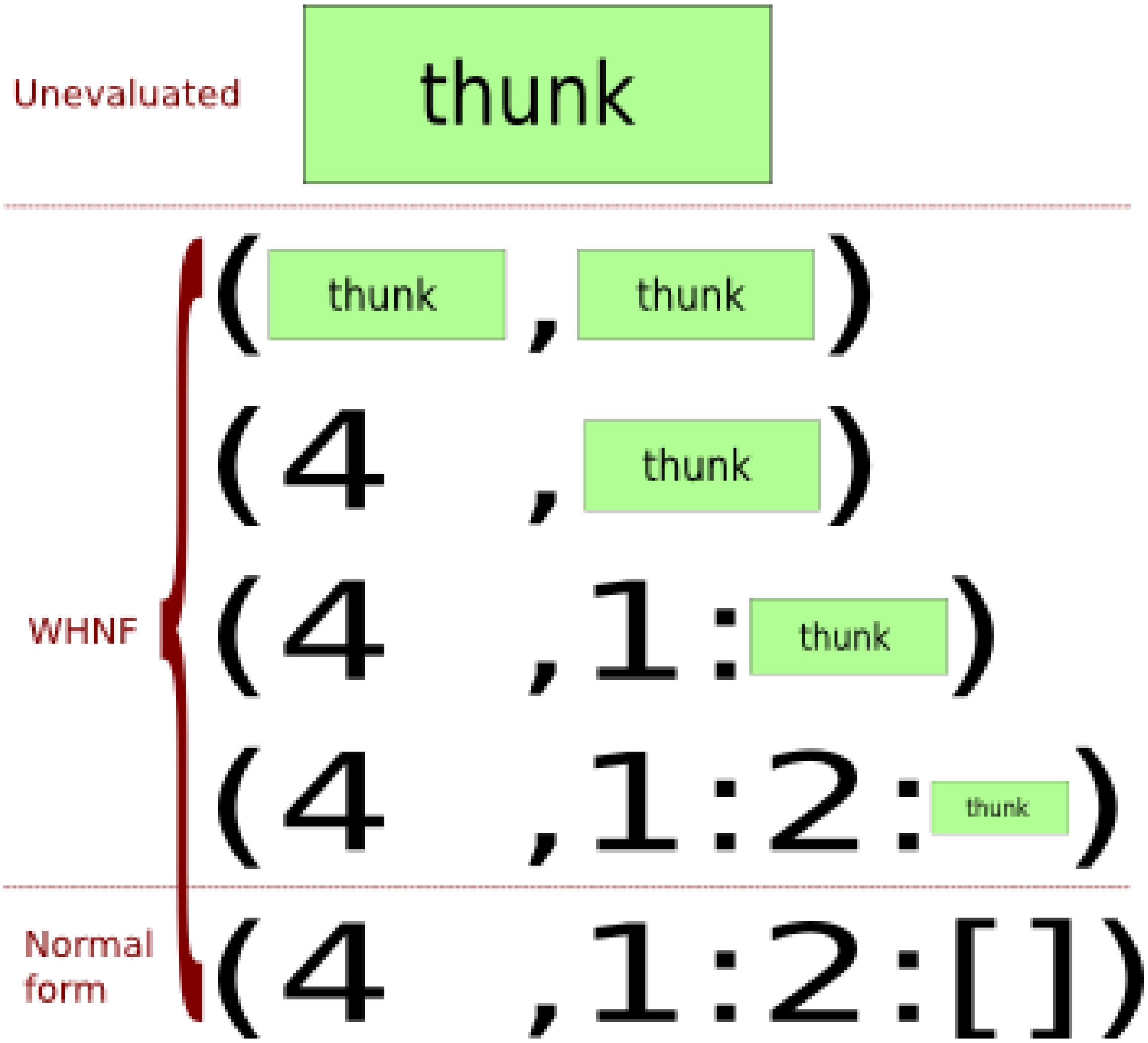


Figure 39 Evaluating the value `(4, [1, 2])` step by step. The first stage is completely unevaluated; all subsequent forms are in WHNF, and the last one is also in normal form.

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 = *thunk* : *thunk*`. (If `s` had been an empty list we would encounter an pattern match failure error at this point.)
- Evaluates the first thunk it just revealed to make sure it's `'h':ss = 'h' : *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: `(*thunk*, *thunk*)`. If we have a list, the minimum amount of evaluation takes us either to a cons cell `*thunk* : *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, 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 result. 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 it is 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.

64.3 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 `(length . show)`. If you had `length $ 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 = length $ 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.

64.3.1 Black-box strictness analysis

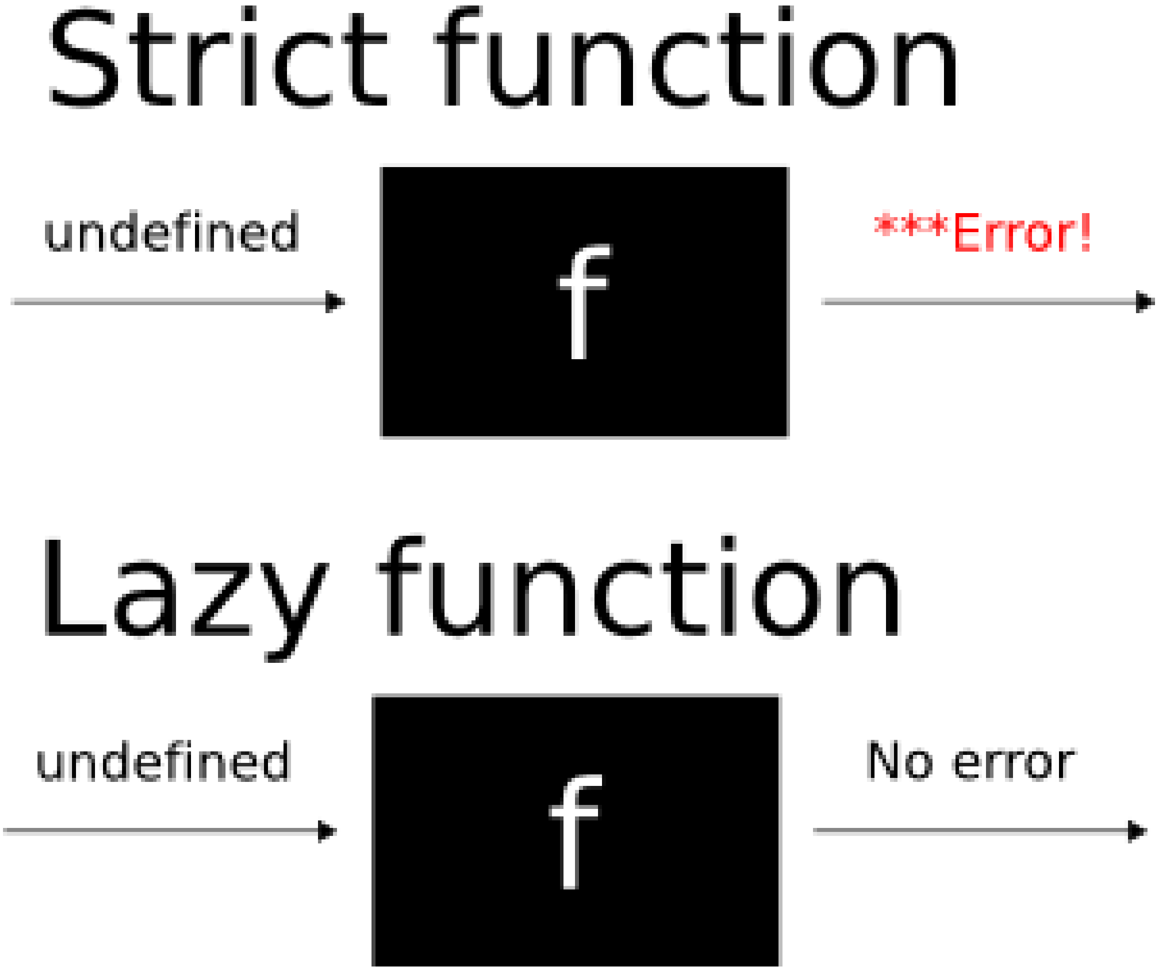


Figure 40 If `f` returns an error when passed undefined, it must be strict. Otherwise, it's lazy.

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, 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.

64.3.2 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 its 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 gets 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. 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.
```

64.3.3 The denotational view on things

If you're familiar with denotational semantics (perhaps you've read the wikibook chapter[1] on it?), then the strictness of a function can be summed up very succinctly:

$$f \perp = \perp \Leftrightarrow f \text{ is strict}$$

Assuming that you say that everything with type `forall a. a`, including `undefined`, `error "any string"`, `throw` and so on, has denotation $\perp$.

64.4 Lazy pattern matching

You might have seen pattern matches like the following in Haskell sources.

Example:
A lazy pattern match

```
-- From Control.Arrow
(***) 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

[1] Chapter 56 on page 361

won't be evaluated: just the 'top level'. Try `let f (Just x) = 1 in f (Just undefined)` to see 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
Prelude> f undefined
*** Exception: Prelude.undefined

Prelude> let f ~(x,y) = 1
Prelude> 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.

64.4.1 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?
- If the first equation is changed to use an ordinary refutable pattern, will the behavior of `head'` still be different from that of `head`? If so, how?
- More to come...

64.5 Benefits of nonstrict semantics

We've been explaining lazy evaluation so far, how nonstrict semantics are actually implemented in terms of thunks. But why is Haskell a nonstrict language in the first place? What advantages are there? In this section, we give some of the benefits of nonstrictness.

64.5.1 Separation of concerns without time penality

Lazy evaluation encourages a kind of "what you see is what you get" mentality when it comes to coding. For example, let's say you wanted to find the lowest three numbers in a long list. In Haskell this is achieved extremely naturally: `take 3 (sort xs)`. However a naïve translation of that code in a strict language would be a very bad idea! It would involve computing the entire sorted list, then chopping off all but the first three elements. However, with lazy evaluation we stop once we've sorted the list up to the third element, so this natural definition turns out to be efficient, too (depending on the implementation of sort).

To give a second example, the function `isInfixOf`[2] from Data.List allows you to see if one string is a substring of another string. It's easily definable as:

```
isInfixOf :: Eq a => [a] -> [a] -> Bool
isInfixOf x y = any (isPrefixOf x) (tails y)
```

Again, this would be suicide in a strict language as to compute all the tails of a list would be very time-consuming. However here we only compute as many as is necessary: once we've found one that `x` is a prefix of we can stop and return `True` once away, "shortcutting" the rest of the work. In fact there is a further advantage: we can use an infinite list for `y`. So, for example, finding out whether a given list is a sequence of consecutive natural numbers is easy: `isInfixOf xs [1..]`. FIXME: this doesn't work because if xs is NOT a sequence of consecutive natural numbers, the function isInfixOf will run endlessly. ENDFIXME

[2] `http://haskell.org/ghc/docs/latest/html/libraries/base/Data-List.html#v%3AisInfixOf`

There are many more examples along these lines.[3] So we can write code that looks and feels natural but doesn't actually incur any time penalty. As one of the highest aims of programming is to write code that is maintainable and clear, this is a big bonus!

However, as always in Computer Science (and in life), a tradeoff exists (in particular, a space-time tradeoff). Using a thunk instead of a plain `Int`, for a trivial calculation like `2+2`, can be a waste. For more examples, see the page on ../Strictness[4].

64.5.2 Improved code reuse

TODO: integrate in.

Maybe the above TODO talks about the above section, doesn't it?

Often code reuse is far better.

To show an example, we take again (but in more detail) `isInfixOf`[5] from Data.List. Let's look at the full definition

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
isInfixOf :: Eq a => [a] -> [a] -> Bool
isInfixOf 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 subsequence of its second, `y`; when applied on `String`'s (i.e. `[Char]`), it checks if `x` is a substring of `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.

[3] In general expressions like `prune . generate`, where `generate` produces a list of items and `prune` cuts them down, will be much more efficient in a nonstrict language.

[4] Chapter 65 on page 459

[5] `http://haskell.org/ghc/docs/latest/html/libraries/base/Data-List.html#v%3AisInfixOf`

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 is 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[6] — largely focuses on examples where laziness is crucial, and provides a strong argument for lazy evaluation being the default.

64.5.3 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

64.6 Common nonstrict idioms

64.6.1 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 haskell-wiki, 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 a}
```

[6] http://www.md.chalmers.se/~rjmh/Papers/whyfp.html

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.

64.6.2 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 -> ....`

64.7 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.

64.8 References

- Laziness on the Haskell wiki[7]
- Lazy evaluation tutorial on the Haskell wiki[8]

[7] http://www.haskell.org/haskellwiki/Performance/Laziness

[8] http://www.haskell.org/haskellwiki/Haskell/Lazy_Evaluation

65 Strictness

65.1 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[1] 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.

65.2 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.

[1] Chapter 64 on page 445

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 main reason why laziness can be problematic.

Additionally, if the resulting value *is* used, no computation is saved; instead, a slight overhead (of a constant factor) for building the thunk is paid. However, this overhead is not something the programmer should deal with most of the times; more important factors must be considered and may give a much bigger improvements; additionally, optimizing Haskell compilers like GHC can perform 'strictness analysis' and remove that slight overhead.

65.3 Strictness annotations

65.4 seq

65.4.1 DeepSeq

65.5 References

- Strictness on the Haskell wiki[2]

[2] `http://www.haskell.org/haskellwiki/Performance/Strictness`

66 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 $f(n)$ stepts 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 an 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.

66.1 Optimising

66.1.1 Profiling

67 Libraries Reference

68 The 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 defacto 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[1]. For an explanation of the Cabal system for packaging Haskell software see ../Packaging[2].

68.1 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.

[1] Chapter 22 on page 143

[2] Chapter 79 on page 519

One thing worth noting with Haddock is that types and classes are cross-referenced by instance. So for example in the `Data.Maybe`[3] 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 definiton of the `Ord` class and its (very long) list of instances. The instance for `Maybe` will be down there as well.

[3] http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Maybe.html

69 Lists

The **List** datatype (see Data.List[1]) 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.

69.1 Theory

A singly-linked list is a set of values in a defined order. The list can only be traversed in one direction (i.e., 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).

69.2 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

[1] http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-List.html

```
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]
```

69.3 Basic list usage

69.3.1 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
```

69.3.2 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
```

69.4 List utilities

FIXME: is this not covered in the chapter on list manipulation[2]?

69.4.1 Maps

69.4.2 Folds, unfolds and scans

69.4.3 Length, head, tail etc.

[2] Chapter 14 on page 97

70 Arrays

Haskell'98 supports just one array constructor type, namely Array[1], which gives you immutable boxed arrays. "Immutable" means that these arrays, like any other pure functional data structure, have contents fixed at construction time. You can't modify them, only query. There are "modification" operations, but they just return new arrays and don't modify the original one. This makes it possible to use Arrays in pure functional code along with lists. "Boxed" means that array elements are just ordinary Haskell (lazy) values, which are evaluated on demand, and can even contain bottom (undefined) value. You can learn how to use these arrays at `http://haskell.org/tutorial/arrays.html` and I'd recommend that you read this before proceeding to the rest of this page.

Nowadays the main Haskell compilers, GHC and Hugs, ship with the same set of Hierarchical Libraries[2], and these libraries contain a new implementation of arrays which is backward compatible with the Haskell'98 one, but which has far more features. Suffice it to say that these libraries support 9 types of array constructors: Array, UArray, IOArray, IOUArray, STArray, STUArray, DiffArray, DiffUArray and StorableArray. It is no wonder that the array libraries are a source of so much confusion for new Haskellers. However, they are actually very simple - each provides just one of two interfaces, and one of these you already know.

70.1 Quick reference

	Immutable `instance IArray a e`	IO monad `instance MArray a e IO`	ST monad `instance MArray a e ST`
Standard	`Array` `DiffArray`	`IOArray`	`STArray`
Unboxed	`UArray` `DiffUArray`	`IOUArray` `StorableArray`	`STUArray`

70.2 Immutable arrays

The first interface provided by the new array library, is defined by the typeclass IArray (which stands for "immutable array" and defined in the module Data.Array.IArray[3]) and

[1] `http://haskell.org/onlinereport/array.html`

[2] `http://www.haskell.org/ghc/docs/latest/html/libraries/index.html`

[3] `http://hackage.haskell.org/packages/archive/array/latest/doc/html/Data-Array-IArray.html`

defines the same operations that were defined for Array in Haskell '98. Here's a simple example of its use that prints (37,64):

```
import Data.Array
buildPair :: (Int, Int)
buildPair = let arr  = listArray (1,10) (repeat 37) :: Array Int Int
                arr' = arr // [(1, 64)]
            in (arr ! 1, arr' ! 1)

main = print buildPair
```

The big difference is that it is now a typeclass and there are 4 array type constructors, each of which implements this interface: Array, UArray, DiffArray, and DiffUArray. We will later describe the differences between them and the cases when these other types are preferable to use instead of the good old Array. Also note that to use Array type constructor together with other new array types, you need to import Data.Array.IArray module instead of Data.Array.

70.3 Mutable IO arrays

The second interface is defined by the type class MArray (which stands for "mutable array" and is defined in the module Data.Array.MArray[4]) and contains operations to update array elements in-place. Mutable arrays are very similar to IORefs, only they contain multiple values. Type constructors for mutable arrays are IOArray and IOUArray (in Data.Array.IO[5]) and operations which create, update and query these arrays all belong to the IO monad:

```
import Data.Array.IO
main = do arr <- newArray (1,10) 37 :: IO (IOArray Int Int)
          a <- readArray arr 1
          writeArray arr 1 64
          b <- readArray arr 1
          print (a,b)
```

This program creates an array of 10 elements with all values initially set to 37. Then it reads the first element of the array. After that, the program modifies the first element of the array and then reads it again. The type declaration in the second line is necessary because our little program doesn't provide enough context to allow the compiler to determine the concrete type of arr.

70.4 Mutable arrays in ST monad

In the same way that IORef has its more general cousin STRef, IOArray has a more general version STArray (and similarly, IOUArray is parodied by STUArray; both defined

[4] http://hackage.haskell.org/packages/archive/array/latest/doc/html/Data-Array-MArray.html

[5] http://hackage.haskell.org/packages/archive/array/latest/doc/html/Data-Array-IO.html

in Data.Array.ST[6]). These array types allow one to work with mutable arrays in the ST monad:

```
import Control.Monad.ST
import Data.Array.ST

buildPair = do arr <- newArray (1,10) 37 :: ST s (STArray s Int Int)
               a <- readArray arr 1
               writeArray arr 1 64
               b <- readArray arr 1
               return (a,b)

main = print $ runST buildPair
```

Believe it or not, now you know all that is needed to *use* any array type. Unless you are interested in speed issues, just use Array, IOArray and STArray where appropriate. The following topics are almost exclusively about selecting the proper array type to make programs run faster.

70.5 Freezing and thawing

Haskell allows conversion between immutable and mutable arrays with the freeze and thaw functions:

```
freeze :: (Ix i, MArray a e m, IArray b e) => a i e -> m (b i e)
thaw   :: (Ix i, IArray a e, MArray b e m) => a i e -> m (b i e)
```

For instance, the following converts an Array into an STArray, alters it, and then converts it back:

```
import Data.Array
import Control.Monad.ST
import Data.Array.ST

buildPair :: (Int, Int)
buildPair = let arr  = listArray (1,10) (repeat 37) :: Array Int Int
                arr' = modifyAsST arr
            in (arr ! 1, arr' ! 1)

modifyAsST :: Array Int Int -> Array Int Int
modifyAsST arr = runST $ do starr <- thaw arr
                            compute starr
                            newarr <- freeze starr
                            return newarr

compute :: STArray s Int Int -> ST s ()
compute arr = do writeArray arr 1 64

main = print buildPair
```

[6] http://hackage.haskell.org/packages/archive/array/latest/doc/html/Data-Array-ST.html

Freezing and thawing both copy the entire array. If you want to use the same memory locations before and after freezing or thawing but can allow some access restrictions, see the section Unsafe operations and running over array elements[7].

70.6 DiffArray

As we already stated, the update operation on immutable arrays (IArray) just creates a new copy of the array, which is very inefficient, but it is a pure operation which can be used in pure functions. On the other hand, updates on mutable arrays (MArray) are efficient but can be done only in monadic code. DiffArray (defined in Data.Array.Diff[8]) combines the best of both worlds - it supports the IArray interface and therefore can be used in a purely functional way, but internally it uses the efficient update of MArrays.

How does this trick work? DiffArray has a pure external interface, but internally it is represented as a reference to an IOArray.

When the '//' operator is applied to a diff array, its contents are physically updated in place. The old array silently changes its representation without changing the visible behavior: it stores a link to the new current array along with the difference to be applied to get the old contents.

So if a diff array is used in a single-threaded style, that is, after '//' application the old version is no longer used, then `a ! i` takes O(1) time and `a // d` takes O(n) time. Accessing elements of older versions gradually becomes slower.

Updating an array which is not current makes a physical copy. The resulting array is unlinked from the old family. So you can obtain a version which is guaranteed to be current and thus has fast element access by performing the "identity update", `old // []`.

The library provides two "differential" array constructors - DiffArray, made internally from IOArray, and DiffUArray, based on IOUArray. If you really need to, you can construct new "differential" array types from any 'MArray' types living in the 'IO' monad. See the module documentation[9] for further details.

Usage of DiffArray doesn't differ from that of Array, the only difference is memory consumption and speed:

```
import Data.Array.Diff

main = do
       let arr = listArray (1,1000) [1..1000] :: DiffArray Int Int
           a = arr ! 1
           arr2 = arr // [(1,37)]
           b = arr2 ! 1
       print (a,b)
```

You can use 'seq' to force evaluation of array elements prior to updating an array:

[7] Chapter 70.11 on page 478

[8] http://hackage.haskell.org/packages/archive/array/latest/doc/html/Data-Array-Diff.html

[9] http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Array-Diff.html

```
import Data.Array.Diff

main = do
      let arr = listArray (1,1000) [1..1000] :: DiffArray Int Int
          a = arr ! 1
          b = arr ! 2
          arr2 = a `seq` b `seq` (arr // [(1,37),(2,64)])
          c = arr2 ! 1
      print (a,b,c)
```

70.7 Unboxed arrays

In most implementations of lazy evaluation, values are represented at runtime as pointers to either their value, or code for computing their value. This extra level of indirection, together with any extra tags needed by the runtime, is known as a box. The default "boxed" arrays consist of many of these boxes, each of which may compute its value separately. This allows for many neat tricks, like recursively defining an array's elements in terms of one another, or only computing the specific elements of the array which are ever needed. However, for large arrays, it costs a lot in terms of overhead, and if the entire array is always needed, it can be a waste.

Unboxed arrays (defined in Data.Array.Unboxed[10]) are more like arrays in C - they contain just the plain values without this extra level of indirection, so that, for example, an array of 1024 values of type Int32 will use only 4 kb of memory. Moreover, indexing of such arrays can be significantly faster.

Of course, unboxed arrays have their own disadvantages. First, unboxed arrays can be made only of plain values having a fixed size -- Int, Word, Char, Bool, Ptr, Double and others (listed in the UArray class definition[11]). You can even implement unboxed arrays yourself for other simple types, including enumerations. But Integer, String and any other types defined with variable size cannot be elements of unboxed arrays. Second, without that extra level of indirection, all of the elements in an unboxed array must be evaluated when the array is evaluated, so you lose the benefits of lazy evaluation. Indexing the array to read just one element will construct the entire array. This is not much of a loss if you will eventually need the whole array, but it does prevent recursively defining the array elements in terms of each other, and may be too expensive if you only ever need specific values. Nevertheless, unboxed arrays are a very useful optimization instrument, and I recommend using them as much as possible.

All main array types in the library have unboxed counterparts:

```
Array - UArray          (module Data.Array.Unboxed)
IOArray - IOUArray      (module Data.Array.IO)
```

[10] http://hackage.haskell.org/packages/archive/array/latest/doc/html/Data-Array-Unboxed.html

[11] http://www.haskell.org/ghc/docs/latest/html/libraries/array/Data-Array-Unboxed.html#t%3AUArray

```
STArray - STUArray      (module Data.Array.ST)
DiffArray - DiffUArray  (module Data.Array.Diff)
```

So, basically replacing boxed arrays in your program with unboxed ones is very simple - just add 'U' to the type signatures, and you are done! Of course, if you change Array to UArray, you also need to add "Data.Array.Unboxed" to your imports list.

70.8 StorableArray

A storable array (Data.Array.Storable[12]) is an IO-mutable array which stores its contents in a contiguous memory block living in the C heap. Elements are stored according to the class 'Storable'. You can obtain the pointer to the array contents to manipulate elements from languages like C.

It is similar to 'IOUArray' (in particular, it implements the same MArray interface) but slower. The advantage is that it's compatible with C through the foreign function interface. The memory addresses of storable arrays are fixed, so you can pass them to C routines.

The pointer to the array contents is obtained by 'withStorableArray'. The idea is similar to 'ForeignPtr' (used internally here). The pointer should be used only during execution of the 'IO' action returned by the function passed as argument to 'withStorableArray'.

```
{-# OPTIONS_GHC -fglasgow-exts #-}
import Data.Array.Storable
import Foreign.Ptr
import Foreign.C.Types

main = do arr <- newArray (1,10) 37 :: IO (StorableArray Int Int)
          a <- readArray arr 1
          withStorableArray arr
              (\ptr -> memset ptr 0 40)
          b <- readArray arr 1
          print (a,b)

foreign import ccall unsafe "string.h"
    memset  :: Ptr a -> CInt -> CSize -> IO ()
```

If you want to use this pointer afterwards, ensure that you call 'touchStorableArray' AFTER the last use of the pointer, so that the array will be not freed too early.

Additional comments: GHC 6.6 should make access to a 'StorableArray' as fast as to any other unboxed array. The only difference between 'StorableArray' and 'UArray' will be that UArray lies in relocatable part of GHC heap while 'StorableArray' lies in non-relocatable part and therefore keep the fixed address, what allow to pass this address to the C routines and save it in the C data structures.

GHC 6.6 also adds 'unsafeForeignPtrToStorableArray' operation that allows to use any Ptr as address of 'StorableArray' and in particular work with arrays returned by C routines.

[12] http://hackage.haskell.org/packages/archive/array/latest/doc/html/Data-Array-Storable.html

Example of using this operation:

```
import Data.Array.Storable
import Foreign.Marshal.Alloc
import Foreign.Marshal.Array
import Foreign.ForeignPtr

main = do ptr <- mallocArray 10
          fptr <- newForeignPtr_ ptr
          arr <- unsafeForeignPtrToStorableArray (1,10) fptr :: IO
(StorableArray Int Int)
          writeArray arr 1 64
          a <- readArray arr 1
          print a
          free ptr
```

This example allocs memory for 10 Ints (which emulates array returned by some C function), then converts returned 'Ptr Int' to 'ForeignPtr Int' and 'ForeignPtr Int' to 'StorableArray Int Int'. It then writes and reads first element of array. At the end, memory used by array is deallocated by 'free' which again emulates deallocation by C routines. We can also enable automatic freeing of allocated block by replacing "newForeignPtr_ ptr" with "newForeignPtr finalizerFree ptr". In this case memory will be automatically freed after last array usage, as for any other Haskell objects.

70.9 The Haskell Array Preprocessor (STPP)

Using mutable arrays in Haskell (IO and ST ones) is not very handy. But there is one tool which adds syntax sugar and makes using of such arrays very close to that in imperative languages. It is written by Hal Daume III and you can get it as `http://hal3.name/STPP/stpp.tar.gz`

Using this tool, you can index array elements in arbitrary complex expressions with just "arr[|i|]" notation and this preprocessor will automatically convert such syntax forms to appropriate calls to 'readArray' and 'writeArray'. Multi-dimensional arrays are also supported, with indexing in the form "arr[|i|][|j|]". See further descriptions at `http://hal3.name/STPP/`

70.10 ArrayRef library

The ArrayRef library[13] reimplements array libraries with the following extensions:

- dynamic (resizable) arrays
- polymorphic unboxed arrays

It also adds syntax sugar[14] what simplifies arrays usage. Although not so elegant as with STPP, it's on other hand implemented entirely inside Haskell language, without any preprocessors.

[13] `http://haskell.org/haskellwiki/Library/ArrayRef#Reimplemented_Arrays_library`

[14] `http://haskell.org/haskellwiki/Library/ArrayRef#Syntax_sugar_for_mutable_types`

70.11 Unsafe operations and running over array elements

As mentioned above, there are operations that converts between mutable and immutable arrays of the same type, namely 'freeze' (mutable → immutable) and 'thaw' (immutable → mutable). These copy the entire array. If you are sure that a mutable array will not be modified or that an immutable array will not be used after the conversion, you can use unsafeFreeze/unsafeThaw instead - these operations convert array in-place if input and resulting arrays have the same memory representation (i.e. the same type and boxing). Please note that "unsafe*" operations modifies memory - they sets/clears flag in array header which tells about array mutability. So these operations can't be used together with multi-threaded access to array (using threads or some form of coroutines).

There are also operations that converts unboxed arrays to another element type, namely castIOUArray and castSTUArray. These operations rely on actual type representation in memory and therefore there is no any guarantees on their results. In particular, these operations can be used to convert any unboxable value to the sequence of bytes and vice versa, f.e. it's used in AltBinary library to serialize floating-point values. Please note that these operations don't recompute array bounds according to changed size of elements. You should do it yourself using 'sizeOf' operation!

While arrays can have any type of indexes, internally any index after bounds checking is converted to plain Int value (position) and then one of low-level operations, unsafeAt, unsafeRead, unsafeWrite, is used. You can use these operations yourself in order to outpass bounds checking and make your program faster. These operations are especially useful if you need to walk through entire array:

```
-- | Returns a list of all the elements of an array, in the same order
-- as their indices.
elems arr = [ unsafeAt arr i | i <- [0 .. rangeSize (bounds arr)-1] ]
```

"unsafe*" operations in such loops are really safe because 'i' loops only through positions of existing array elements.

70.12 GHC-specific topics

70.12.1 Parallel arrays (module GHC.PArr)

As we already mentioned, array library supports two array varieties - lazy boxed arrays and strict unboxed ones. Parallel array implements something intervening - it's a strict boxed immutable array. This keeps flexibility of using any data type as array element while makes both creation and access to such arrays much faster. Array creation implemented as one imperative loop that fills all array elements, while access to array elements don't need to check "box". It should be obvious that parallel arrays are not efficient in cases where calculation of array elements are rather complex and you will not use most of array elements. One more drawback of practical usage is that parallel arrays don't support IArray interface which means that you can't write generic algorithms which work both with Array and parallel array constructor.

Like many GHC extensions, this is described in a paper: An Approach to Fast Arrays in Haskell[15], by Manuel M. T. Chakravarty and Gabriele Keller.

You can also look at sources of GHC.PArr[16] module, which contains a lot of comments.

Special syntax for parallel arrays is enabled by "ghc -fparr" or "ghci -fparr" which is undocumented in user manual of GHC 6.4.1.

70.12.2 Welcome to the machine

Array#, MutableArray#, ByteArray#, MutableByteArray#, pinned and moveable byte arrays

GHC heap contains two kinds of objects - some are just byte sequences, other contains pointers to other objects (so called "boxes"). These segregation allows to find chains of references when performing garbage collection and update these pointers when memory used by heap is compacted and objects are moved to new places. Internal (raw) GHC's type Array# represents a sequence of object pointers (boxes). There is low-level operation in ST monad which allocates array of specified size in heap, its type is something like (Int -> ST s Array#). The Array# type is used inside Array type which represents boxed immutable arrays.

There is a different type for **mutable** boxed arrays (IOArray/STArray), namely MutableArray#. Separate type for mutable arrays required because of 2-stage garbage collection mechanism. Internal representation of Array# and MutableArray# are the same excluding for some flags in header and this make possible to on-place convert MutableArray# to Array# and vice versa (this is that unsafeFreeze and unsafeThaw operations do).

Unboxed arrays are represented by the ByteArray# type. It's just a plain memory area in the heap, like the C's array. There are two primitive operations that creates a ByteArray# of specified size - one allocates memory in normal heap and so this byte array can be moved each time when garbage collection occurs. This prevents converting of ByteArray# to plain memory pointer that can be used in C procedures (although it's still possible to pass current ByteArray# pointer to "unsafe foreign" procedure if it don't try to store this pointer somewhere). The second primitive allocates ByteArray# of specified size in "pinned" heap area, which contains objects with fixed place. Such byte array will never be moved by GC so it's address can be used as plain Ptr and shared with C world. First way to create ByteArray# used inside implementation of all UArray types, second way used in StorableArray (although StorableArray can also point to data, allocated by C malloc).

There is also MutableByteArray# type what don't had much difference with ByteArray#, but GHC's primitives support only monadic read/write operations for MutableByteArray#, and only pure reads for ByteArray#, plus unsafeFreeze/unsafeThaw operations that changes appropriate fields in headers of this arrays. This differentiation don't make much sense except for additional safety checks.

[15] `http://www.cse.unsw.edu.au/~chak/papers/CK03.html`

[16] `http://darcs.haskell.org/packages/base/GHC/PArr.hs`

So, pinned MutableByteArray# or C malloced memory used inside StorableArray, unpinned MutableByteArray# used inside IOUArray and STUArray, and unpinned ByteArray# are used inside UArray.

Both boxed and unboxed arrays API are almost the same:

```
marr <- alloc n            - allocates mutable array with given size
arr  <- unsafeFreeze marr  - converts mutable array to immutable one
marr <- unsafeThaw arr     - converts immutable array to mutable one
x    <- unsafeRead marr i  - monadic reading of value with given index from
mutable array
unsafeWrite marr i x       - monadic writing of value with given index from
mutable array
let x = unsafeAt arr i     - pure reading of value with given index from
immutable array
(all indexes are counted from 0)
```

Based on these primitive operations, the arrays library implements indexing with any type and with any lower bound, bounds checking and all other high-level operations. Operations that creates/updates immutable arrays just creates them as mutable arrays in ST monad, make all required updates on this array and then use unsafeFreeze before returning array from runST. Operations on IO arrays are implemented via operations on ST arrays using stToIO operation.

70.12.3 Mutable arrays and GC

GHC implements 2-stage GC which is very fast - minor GC occurs after each 256 kb allocated and scans only this area (plus recent stack frames) when searching for a "live" data. This solution uses the fact that usual Haskell data are immutable and therefore any data structures that was created before previous minor GC can't point to the data structures created after this GC (due to immutability data can contain only "backward" references).

But this simplicity breaks when we add to the language mutable boxed references (IORef/STRef) and arrays (IOArray/STArray). On each GC, including minor ones, each element in mutable data structures should be scanned because it may be updated since last GC and now point to the data allocated since last GC.

For programs that contains a lot of data in mutable boxed arrays/references GC times may easily outweigh useful computation time. Ironically, one of such programs is GHC itself. Solution for such programs is to add to cmdline option like "+RTS -A10m" which increases size of minor GC chunks from 256 kb to 10 mb, i.e. makes minor GC 40 times less frequent. You can see effect of this change using "+RTS -sstderr" option - "%GC time" should significantly decrease.

There is a way to include this option into your executable so it will be used automatically on each execution - you should just add to your project C source file that contains the following line:

```
char *ghc_rts_opts = "-A10m";
```

Of course, you can increase or decrease this value according to your needs.

Increasing "-A" value don't comes for free - aside from obvious increasing of memory usage, execution times (of useful code) will also grow. Default "-A" value tuned to be close to modern CPU cache sizes that means that most of memory references are fall inside the cache. When 10 mb of memory are allocated before doing GC, this data locality is no more holds. So, increasing "-A" can either increase or decrease program speed, depending on its nature. Try various settings between 64 kb and 16 mb while running program with "typical" parameters and try to select a best setting for your concrete program and cpu combination.

There is also another way to avoid increasing GC times - use either unboxed or immutable arrays. Also note that immutable arrays are build as mutable ones and then "freezed", so during the construction time GC will also scan their contents.

Hopefully, GHC 6.6 fixed the problem - it remembers what references/arrays was updated since last GC and scans only them. You can suffer from the old problems only in the case when you use very large arrays.

Further information:

- RTS options to control the garbage collector[17]
- Problem description by Simon Marlow and report about GHC 6.6 improvements in this area[18]
- Notes about GHC garbage collector[19]
- Papers about GHC garbage collector[20]

[17] http://www.haskell.org/ghc/docs/latest/html/users_guide/runtime-control.html

[18] http://hackage.haskell.org/trac/ghc/ticket/650

[19] http://hackage.haskell.org/trac/ghc/wiki/GarbageCollectorNotes

[20] http://research.microsoft.com/~simonpj/Papers/papers.html#gc

71 Maybe

The **Maybe** data type is a means of being explicit that you are not sure that a function will be successful when it is executed.

71.1 Motivation

Many languages require you to guess what they will do when a calculation did not finish properly. For example, an array lookup signature may look like this in pseudocode:

```
getPosition(Array a, Value v) returns Integer
```

But what happens if it *doesn't* find the item? It could return a `null` value, or the integer '-1' which would also be an obvious sign that something went wrong. But there's no way of knowing what will happen without examining the code for this procedure to see what the programmer chose to do. In a library without available code this might not even be possible.

The alternative is to explicitly state what the function *should* return (in this case, Integer), but also that it might not work as intended --- Maybe Integer. This is the intention of the Maybe datatype. So in Haskell, we could write the above signature as:

```
getPosition :: Array -> Value -> Maybe Integer
```

If the function is successful you want to return the result; otherwise, you want to return an explicit failure. This could be simulated as a tuple of type `(Bool, a)` where *a* is the "actual" return type of the function. But what would you put in the *a* slot if it failed? There's no obvious answer. Besides which, the Maybe type is easier to use and has a selection of library functions for dealing with values which may fail to return an explicit answer.

71.2 Definition

The Standard Prelude defines the Maybe type as follows, and more utility functions exist in the Data.Maybe library.

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

The type *a* is polymorphic and can contain complex types or even other monads (such as IO () types).

71.3 Library functions

The module `Data.Maybe`, in the standard hierarchical libraries[1], contains a wealth of functions for working with Maybe values.

71.3.1 Querying

There are two obvious functions to give you information about a Maybe value.

`isJust`

This returns True if the argument is in the form `Just _`.

```
isJust :: Maybe a -> Bool
isJust (Just _) = True
isJust Nothing  = False
```

`isNothing`

The dual of `isJust`: returns True if its argument is `Nothing`.

```
isNothing :: Maybe a -> Bool
isNothing (Just _) = False
isNothing Nothing  = True
```

71.3.2 Getting out

There are a handful of functions for converting Maybe values to non-Maybe values.

`maybe`

`maybe` is a function that takes a default value to use if its argument is `Nothing`, a function to apply if its argument is in the form `Just _`, and a Maybe value.

[1] Chapter 68 on page 465

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe _ f (Just x) = f x
maybe z _ Nothing  = z
```

fromMaybe

A frequent pattern is to use the `maybe` function, but not want to change the value if it was a `Just`. That is, call `maybe` with the second parameter being `id`. This is precisely `fromMaybe`.

```
fromMaybe :: a -> Maybe a -> a
fromMaybe z = maybe z id
```

fromJust

There are certain occasions when you *know* a function that ends in a Maybe value will produce a `Just`. In these cases, you can use the `fromJust` function, which just strips off a `Just` constructor.

```
fromJust :: Maybe a -> a
fromJust (Just x) = x
fromJust Nothing  = error "fromJust: Nothing"
```

71.3.3 Lists and Maybe

Lists are, in some ways, similar to the Maybe datatype (indeed, this relationship will be further explored when you learn about monads). As such, there are a couple of functions for converting between one and the other.

listToMaybe

This function, and the following one, makes a lot of sense when you think about Maybe and list values in terms of computations (which will be more fully explained in the section on Advanced monads[2]).

With lists, `[]` represents a failed computation. With Maybe, `Nothing` does. `listToMaybe` converts between the list and Maybe monad. When the parameter (in the list monad) indicated a successful computation, only the first solution is taken to place in the Maybe value.

[2] http://en.wikibooks.org/wiki/..%2F..%2FAdvanced%20monads

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

maybeToList

The obvious opposite of `listToMaybe`.

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

71.3.4 Lists manipulation

Finally, there are a couple of functions which are analogues of the normal Prelude list manipulation functions, but specialised to Maybe values.

Continue on some failures (like 'or')

catMaybes

Given a list of Maybe values, `catMaybes` extracts all the values in the form `Just _`, and strips off the `Just` constructors. This is easily defined with a list comprehension, as we showed in the pattern matching chapter[3]:

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

mapMaybe

`mapMaybe` applies a function to a list, and collects the successes. It can be understood as a composition of functions you already know:

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe f xs = catMaybes (map f xs)
```

But the actual definition may be more efficient and traverse the list once:

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe _ []     = []
mapMaybe f (x:xs) =
  case f x of
```

[3] Chapter 16 on page 113

```
      Just y  -> y : mapMaybe f xs
      Nothing -> mapMaybe f xs
```

Stop on failure

`sequence`
Sometimes you want to collect the values if and only if all succeeded:

```
sequence :: [Maybe a] -> Maybe [a]
sequence []           = Just []
sequence (Nothing:xs) = Nothing
sequence (Just x:xs)  = case sequence xs of
  Just xs' -> Just (x:xs')
  _        -> Nothing
```

72 Maps

The module Data.Map provides the `Map` datatype, which allows you to store *values* attached to specific *keys*. This is called a lookup table, dictionary or associative array in other languages.

72.1 Motivation

Very often it would be useful to have some kind of data structure that relates a value or list of values to a specific key. This is often called a dictionary after the real-world example: a real-life dictionary associates a definition (the value) to each word (the key); we say the dictionary is a *map from words to definitions*. A filesystem driver might keep a map from filenames to file information. A phonebook application might keep a map from contact names to phone numbers. Maps are a very versatile and useful datatype.

72.1.1 Why not just `[(a, b)]`?

You may have seen in other chapters that a list of pairs (or 'lookup table') is often used as a kind of map, along with the function `lookup :: [(a, b)] -> a -> Maybe b`. So why not just use a lookup table all the time? Here are a few reasons:

- Working with maps gives you access to a whole load more useful functions for working with lookup tables.
- Maps are implemented far more efficiently than a lookup table would be, both in terms of speed and the amount of memory it takes up.

72.2 Definition

Internally, maps are defined using a complicated datatype called a *balanced tree* for efficiency reasons, so we don't give the details here.

72.3 Library functions

The module Data.Map provides an absolute wealth of functions for dealing with Maps, including setlike operations like unions and intersections. There are so many we don't include a list here; instead the reader is deferred to the hierarchical libraries documentation[1] for Data.Map.

72.4 Example

The following example implements a password database. The user is assumed to be trusted, so is not authenticated and has access to view or change passwords.

```
{- A quick note for the over-eager refactorers out there: This is (somewhat)
   intentionally ugly. It doesn't use the State monad to hold the DB because it
   hasn't been introduced yet. Perhaps we could use this as an example of How
   Monads Improve Things? -}

module PassDB where

import qualified Data.Map as M
import System.Exit

type UserName = String
type Password = String
type PassDB   = M.Map UserName Password
  -- PassBD is a map from usernames to passwords

-- | Ask the user for a username and new password, and return the new PassDB
changePass :: PassDB -> IO PassDB
changePass db = do
  putStrLn "Enter a username and new password to change."
  putStr "Username: "
  un <- getLine
  putStrLn "New password: "
  pw <- getLine
  if un `M.member` db               -- if un is one of the keys of the map
    then return $ M.insert un pw db -- then update the value with the new
password
    else do putStrLn $ "Can't find username '" ++ un ++ "' in the database."
            return db

-- | Ask the user for a username, whose password will be displayed.
viewPass :: PassDB -> IO ()
viewPass db = do
  putStrLn "Enter a username, whose password will be displayed."
  putStr "Username: "
  un <- getLine
  putStrLn $ case M.lookup un db of
               Nothing -> "Can't find username '" ++ un ++ "' in the database."
               Just pw -> pw

-- | The main loop for interacting with the user.
mainLoop :: PassDB -> IO PassDB
mainLoop db = do
  putStr "Command [cvq]: "
  c <- getChar
  putStr "\n"
  -- See what they want us to do. If they chose a command other than 'q', then
  -- recurse (i.e. ask the user for the next command). We use the Maybe
datatype
  -- to indicate whether to recurse or not: 'Just db' means do recurse, and in
  -- running the command, the old datbase changed to db. 'Nothing' means don't
  -- recurse.
  db' <- case c of
```

[1] http://haskell.org/ghc/docs/latest/html/libraries/containers/Data-Map.html

```
           'c' -> fmap Just $ changePass db
           'v' -> do viewPass db; return (Just db)
           'q' -> return Nothing
           _   -> do putStrLn $ "Not a recognised command, '" ++ [c] ++ "'."
                     return (Just db)
  maybe (return db) mainLoop db'

-- | Parse the file we've just read in, by converting it to a list of lines,
--   then folding down this list, starting with an empty map and adding the
--   username and password for each line at each stage.
parseMap :: String -> PassDB
parseMap = foldr parseLine M.empty . lines
    where parseLine ln map =
            let [un, pw] = words ln
            in M.insert un pw map

-- | Convert our database to the format we store in the file by first
converting
--   it to a list of pairs, then mapping over this list to put a space between
--   the username and password
showMap :: PassDB -> String
showMap = unlines . map (\(un, pw) -> un ++ " " ++ pw) . M.toAscList

main :: IO ()
main = do
  putStrLn $ "Welcome to PassDB. Enter a command: (c)hange a password, " ++
             "(v)iew a password or (q)uit."
  dbFile <- readFile "/etc/passdb"
  db'    <- mainLoop (parseMap dbFile)
  writeFile "/etc/passdb" (showMap db')
```

Exercises:
FIXME: write some exercises

73 IO

73.1 The IO Library

The IO Library (available by `import`ing the `System.IO` module) contains many definitions, the most commonly used of which are listed below:

```
data IOMode  = ReadMode   | WriteMode
             | AppendMode | ReadWriteMode

openFile     :: FilePath -> IOMode -> IO Handle
hClose       :: Handle -> IO ()

hIsEOF       :: Handle -> IO Bool

hGetChar     :: Handle -> IO Char
hGetLine     :: Handle -> IO String
hGetContents :: Handle -> IO String

getChar      :: IO Char
getLine      :: IO String
getContents  :: IO String

hPutChar     :: Handle -> Char -> IO ()
hPutStr      :: Handle -> String -> IO ()
hPutStrLn    :: Handle -> String -> IO ()

putChar      :: Char -> IO ()
putStr       :: String -> IO ()
putStrLn     :: String -> IO ()

readFile     :: FilePath -> IO String
writeFile    :: FilePath -> String -> IO ()

{- bracket must be imported from Control.Exception -}
bracket      ::
    IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

Note:
The type `FilePath` is a *type synonym* for `String`. That is, there is no difference between `FilePath` and `String`. So, for instance, the `readFile` function takes a `String` (the file to read) and returns an action that, when run, produces the contents of that file. See the section ../../Type declarations/[a] for more about type synonyms.

[a] Chapter 15 on page 107

Most of these functions are self-explanatory. The `openFile` and `hClose` functions open and close a file, respectively, using the `IOMode` argument as the mode for opening the file. `hIsEOF` tests for end-of file. `hGetChar` and `hGetLine` read a character or line (respectively) from a file. `hGetContents` reads the entire file. The `getChar`, `getLine` and `getContents` variants read from standard input. `hPutChar` prints a character to a file; `hPutStr` prints a string; and `hPutStrLn` prints a string with a newline character at the end. The variants without the `h` prefix work on standard output. The `readFile` and `writeFile` functions read and write an entire file without having to open it first.

The `bracket` function is used to perform actions safely. Consider a function that opens a file, writes a character to it, and then closes the file. When writing such a function, one needs to be careful to ensure that, if there were an error at some point, the file is still successfully closed. The `bracket` function makes this easy. It takes three arguments: The first is the action to perform at the beginning. The second is the action to perform at the end, regardless of whether there's an error or not. The third is the action to perform in the middle, which might result in an error. For instance, our character-writing function might look like:

```
writeChar :: FilePath -> Char -> IO ()
writeChar fp c =
    bracket
      (openFile fp WriteMode)
      hClose
      (\h -> hPutChar h c)
```

This will open the file, write the character and then close the file. However, if writing the character fails, `hClose` will still be executed, and the exception will be reraised afterwards. That way, you don't need to worry too much about catching the exceptions and about closing all of your handles.

73.2 A File Reading Program

We can write a simple program that allows a user to read and write files. The interface is admittedly poor, and it does not catch all errors (such as reading a non-existent file). Nevertheless, it should give a fairly complete example of how to use IO. Enter the following code into "FileRead.hs," and compile/run:

```
module Main
    where

import System.IO
import Control.Exception

main = doLoop

doLoop = do
  putStrLn "Enter a command rFN wFN or q to quit:"
  command <- getLine
  case command of
    'q':_ -> return ()
    'r':filename -> do putStrLn ("Reading " ++ filename)
                       doRead filename
                       doLoop
    'w':filename -> do putStrLn ("Writing " ++ filename)
                       doWrite filename
                       doLoop
    _            -> doLoop

doRead filename =
    bracket (openFile filename ReadMode) hClose
            (\h -> do contents <- hGetContents h
                      putStrLn "The first 100 chars:"
                      putStrLn (take 100 contents))

doWrite filename = do
  putStrLn "Enter text to go into the file:"
  contents <- getLine
  bracket (openFile filename WriteMode) hClose
          (\h -> hPutStrLn h contents)
```

What does this program do? First, it issues a short string of instructions and reads a command. It then performs a **case** switch on the command and checks first to see if the first character is a `q.' If it is, it returns a value of unit type.

Note:
The `return` function is a function that takes a value of type `a` and returns an action of type `IO a`. Thus, the type of `return ()` is `IO ()`.

If the first character of the command wasn't a `q,' the program checks to see if it was an 'r' followed by some string that is bound to the variable `filename`. It then tells you that it's reading the file, does the read and runs `doLoop` again. The check for `w' is nearly identical. Otherwise, it matches `_`, the wildcard character, and loops to `doLoop`.

The `doRead` function uses the `bracket` function to make sure there are no problems reading the file. It opens a file in `ReadMode`, reads its contents and prints the first 100 characters (the `take` function takes an integer *n* and a list and returns the first *n* elements of the list).

The `doWrite` function asks for some text, reads it from the keyboard, and then writes it to the file specified.

Note:
Both `doRead` and `doWrite` could have been made simpler by using `readFile` and `writeFile`, but they were written in the extended fashion to show how the more complex functions are used.

The only major problem with this program is that it will die if you try to read a file that doesn't already exists or if you specify some bad filename like `*\bs^#_@`. You may think that the calls to `bracket` in `doRead` and `doWrite` should take care of this, but they don't. They only catch exceptions within the main body, not within the startup or shutdown functions (`openFile` and `hClose`, in these cases). We would need to catch exceptions raised by `openFile`, in order to make this complete. We will do this when we talk about exceptions in more detail in the section on Exceptions[1].

[1] http://en.wikibooks.org/wiki/..%2FIo%20advanced%23Exceptions

Exercises:
Write a program that first asks whether the user wants to read from a file, write to a file or quit. If the user responds quit, the program should exit. If he responds read, the program should ask him for a file name and print that file to the screen (if the file doesn't exist, the program may crash). If he responds write, it should ask him for a file name and then ask him for text to write to the file, with "." signaling completion. All but the "." should be written to the file.
For example, running this program might produce:

```
Do you want to [read] a file, [write] a file or [quit]?
read
Enter a file name to read:
foo
...contents of foo...
Do you want to [read] a file, [write] a file or [quit]?
write
Enter a file name to write:
foo
Enter text (dot on a line by itself to end):
this is some
text for
foo
.
Do you want to [read] a file, [write] a file or [quit]?
read
Enter a file name to read:
foo
this is some
text for
foo
Do you want to [read] a file, [write] a file or [quit]?
blech
I don't understand the command blech.
Do you want to [read] a file, [write] a file or [quit]?
quit
Goodbye!
```

73.3 Another way of reading files

Sometimes (for example in parsers) we need to open a file from command line, by typing "program.exe input.in" . Of course, we could just use "< " to redirect standard input, but we can do it in more elegant way. We could do this like that:

```
-- $ runghc program input
-- Input interpreted
{-# LANGUAGE ScopedTypeVariables #-}
module Main where

import System.IO
import System.Environment (getArgs)
import System.Exit
import Data.List (isSuffixOf)
import qualified Control.Exception as Err

parse fname str = str

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

```
    arguments <- getArgs
    if length arguments == 0
    then putStrLn "No input file given.\n Proper way of running program is: \n
 program input.in"
    else do
        let suffix = if isSuffixOf ".in" (head arguments) then "" else ".in"
-- Using this trick will allow users to type "program input" as well
        handle <- Err.catch (openFile (head arguments ++ suffix) ReadMode)
            (\(e::Err.IOException) -> error $ show e)

        readable <- hIsReadable handle
        if not readable
        then error "File is being used by another user or program"
        else do
            unparsedString <- hGetContents handle
            case parse (head arguments) unparsedString of
{-This is how it would look like, when our parser was based on Parsec
                Left err -> error $ show err
                Right program -> do
                    outcome <- interprete program
                    case outcome of
                        Abort -> do
                            putStrLn "Program aborted"
                            exitWith $ ExitFailure 1
                        _ -> do
                            putStrLn "Input interpreted"
                            exitWith ExitSuccess
But to make this example less complicated I replaced these lines with:-}
                        unparsedString -> do
                            putStrLn "Input interpreted"
                            exitWith ExitSuccess
```

74 Random Numbers

74.1 Random examples

Here are a handful of uses of random numbers by example

Example:
Ten random integer numbers

```
import System.Random

main = do
   gen <- newStdGen
   let ns = randoms gen :: [Int]
   print $ take 10 ns
```

The IO action newStdGen, as its name implies, creates a new StdGen pseudorandom number generator state. This StdGen can be passed to functions which need to generate pseudorandom numbers.

(There also 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. This is perhaps a library wart, however, as all that you really ever need is newStdGen.)

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:

```
[0.110407025,0.8453985,0.307
7821,0.78138804,0.5242582,0.5196911,0.20084688,0.4794773,0.3240164,6.1566383e-2]
```

Example:
Unsorting a list (imperfectly)

```
import Data.List ( sortBy )
import Data.Ord ( comparing )
import System.Random ( Random, RandomGen, randoms, newStdGen )

main :: IO ()
main =
 do gen <- newStdGen
    interact $ unlines . unsort gen . lines

unsort :: (RandomGen g) => 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 single random number along with a new StdGen to be used for the next one, as well as randomR and randomRs which take a parameter for specifying the range. See below for more ideas.

74.2 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
```

This basically introduces StdGen, the standard random number generator "object". It's an instance of the RandomGen class which specifies the operations other pseudorandom number generator libraries need to implement to be used with the System.Random library.

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.

74.3 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[1] on HaskellWiki 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 "unGen" function executes an action and returns the random result. The type is:

[1] http://www.haskell.org/haskellwiki/Introduction_to_QuickCheck

```
unGen :: Gen a -> StdGen -> Int -> a
```

The three arguments are:

1. The generator action.
2. A random number generator.
3. 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.

So for example:

```
let
   triple = unGen randomTriple (mkStdGen 1) 1
```

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 declarative 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
```

75 General Practices

76 Building a standalone application

1. REDIRECT Haskell/Standalone programs[1]

[1] http://en.wikibooks.org/wiki/Haskell%2FStandalone%20programs

77 Debugging

77.1 Debug prints with Debug.Trace

One common way to debug programs in general is to use debug prints. In imperative languages we can just sprinkle the code with print statements that output debug information (e.g. value of a particular variable, or some human-readable message) to standard output or some as log file. In Haskell, however, it is not possible to output any information if we are not in IO monad; and therefore this trivial debug print technique can't be used with pure functions without rewriting them to use the IO monad.

To deal with this problem, the standard library provides the Debug.Trace[1]. One of the functions this module exports is called `trace`, which provides a convenient way to attach debug print statements anywhere in a program. For instance, this program prints every argument passed to `fib` that is not equal to 0 or 1:

```
module Main where
import Debug.Trace

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = trace ("n: " ++ show n) $ fib (n - 1) + fib (n - 2)

main = putStrLn $ "fib 4: " ++ show (fib 4)
```

Below is the resulting output:

```
n: 4
n: 3
n: 2
n: 2
fib 4: 3
```

Also `trace` makes possible to trace execution steps of program; that is, which function is called first, second, etc. To do so, we annotate parts of functions we are interested in, like this:

```
module Main where
import Debug.Trace

factorial :: Int -> Int
factorial n | n == 0    = trace ("branch 1") 1
            | otherwise = trace ("branch 2") $ n * (factorial $ n - 1)
```

[1] http://hackage.haskell.org/packages/archive/base/latest/doc/html/Debug-Trace.html

```
main = do
    putStrLn $ "factorial 6: " ++ show (factorial 6)
```

When a program annotated in such way is run, it will print the debug strings in the same order the annotated statements were executed. That output might help to locate errors in case of missing statements or similar things.

77.1.1 Some extra advice

As demonstrated above, `trace` can be used outside of the IO monad; and indeed its type signature...

```
trace :: String -> a -> a
```

...indicates that it is a pure function. That, however, looks mystifying - surely `trace` *is* doing IO while printing useful messages; how can it be pure then? In fact, `trace` uses a dirty trick of sorts to circumvent the separation between the IO monad and pure code. That is reflected in the following disclaimer, found in the documentation for `trace`[2]:

> The trace function should *only* be used for debugging, or for monitoring execution. The function is not referentially transparent: its type indicates that it is a pure function but it has the side effect of outputting the trace message.

Naturally, there is no reason to worry provided you follow the advice above and only use `trace` for debugging, and not for regular functionality.

Also, a common mistake in using `trace` is, while trying to fit the debug traces into an existing function, accidentally including the value being evaluated in the message to be printed by `trace`; i.e. don't do anything like this:

```
let foo = trace ("foo = " ++ show foo) $ bar
in  baz
```

This leads to infinite recursion, as trace message will be evaluated before bar expression which will lead to evaluation of foo in terms of trace message and bar again and trace message will be evaluated before bar and so forth to infinity. Instead of `show foo`, the trace message rather have `show bar`:

```
let foo = trace ("foo = " ++ show bar) $ bar
in  baz
```

77.1.2 Useful idioms

A helper function that incorporates `show` can be convenient:

```
traceThis :: (Show a) => a -> a
traceThis x = trace (show x) x
```

[2] http://hackage.haskell.org/packages/archive/base/latest/doc/html/Debug-Trace#v:trace.html

In a similar vein, `Debug.Trace` defines a `traceShow` function, that "prints" its first argument and evaluates to the second one:

```
traceShow :: (Show a) => a -> b -> b
traceShow = trace . show
```

Finally, a function `debug` like this one may prove handy as well:

```
debug = flip trace
```

This will allow you to write code like...

```
main = (1 + 2) `debug` "adding"
```

... making it easier to comment/uncomment debugging statements.

77.2 Incremental development with GHCi

77.3 Debugging with Hat

77.4 General tips

78 Testing

78.1 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.

78.1.1 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'])
```

78.1.2 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!

78.1.3 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: $\forall s. length(take5s) = 5$

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 no more than 5 characters long. So let's weaken the property a bit: $\forall s. length(take5s) \leq 5$

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!

78.1.4 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: $\forall s.\forall e.(e \in take5 s) \Rightarrow (e \in \{a,b,c,d,e\})$

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.

78.1.5 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]
```

78.1.6 More information on QuickCheck

- http://haskell.org/haskellwiki/Introduction_to_QuickCheck
- http://haskell.org/haskellwiki/QuickCheck_as_a_test_set_generator

78.2 HUnit

Sometimes it is easier to give an example for a test than 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[1].

[1] http://hunit.sourceforge.net/HUnit-1.0/Guide.html

79 Packaging your software (Cabal)

A guide to the best practice for creating a new Haskell project or program.

79.1 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.

79.1.1 Revision control

Use darcs[1], unless you have a specific reason not to, in which case use git[2]. If you don't like git, go back and look at darcs. It's written in Haskell, and it's used by many Haskell developers. See the wikibook Understanding darcs[3] to get started.

79.1.2 Build system

Use Cabal[4]. You should read at least the start of section 2 of the Cabal User's Guide[5].

79.1.3 Documentation

For libraries, use Haddock[6]. We recommend using recent versions of haddock (2.8 or above, as of December 2010).

79.1.4 Testing

Pure code can be tested using QuickCheck[7] or SmallCheck[8], impure code with HUnit[9].

[1] http://darcs.net
[2] http://git-scm.com
[3] http://en.wikibooks.org/wiki/Understanding%20darcs
[4] http://haskell.org/cabal
[5] http://haskell.org/cabal/users-guide/index.html
[6] http://haskell.org/haddock
[7] http://www.md.chalmers.se/~rjmh/QuickCheck/
[8] http://www.mail-archive.com/haskell@haskell.org/msg19215.html
[9] http://hunit.sourceforge.net/

To get started, try Haskell/Testing[10]. For a slightly more advanced introduction, Simple Unit Testing in Haskell[11] is a blog article about creating a testing framework for QuickCheck using some Template Haskell.

79.2 Structure of a simple project

The basic structure of a new Haskell project can be adopted from HNop[12], 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 or .git -- 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.

79.2.1 Create a directory

Create somewhere for the source:

```
$ mkdir haq
$ cd haq
```

79.2.2 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
```

[10] Chapter 78 on page 515

[11] http://blog.codersbase.com/2006/09/simple-unit-testing-in-haskell.html

[12] http://semantic.org/hnop/

```
-- 'main' runs the main program
main :: IO ()
main = getArgs >>= print . haqify . head

haqify s = "Haq! " ++ s
```

79.2.3 Stick it in darcs

Place the source under revision control:

```
$ darcs init
$ darcs add Haq.hs
$ darcs record
addfile ./Haq.hs
Shall I record this change? (1/?)  [ynWsfqadjkc], or ? for help: y
hunk ./Haq.hs 1
+--
+-- 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
Shall I record this change? (2/?)  [ynWsfqadjkc], or ? for help: y
What is the patch name? Import haq source
Do you want to add a long comment? [yn]n
Finished recording patch 'Import haq source'
```

And we can see that darcs is now running the show:

```
$ ls
Haq.hs _darcs
```

For git:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
$ git init
$ git add *
$ git commit -m 'Import haq source'
$ ls -A
.git Haq.hs
```

79.2.4 Add a build system

Create a .cabal file describing how to build your project:

```
$ cat > haq.cabal
Name:                haq
Version:             0.0
Synopsis:            Super cool mega lambdas
```

```
Description:        My super cool, indeed, even mega lambdas
                    will demonstrate a basic project. You will marvel.
License:            GPL
License-file:       LICENSE
Author:             Don Stewart
Maintainer:         Don Stewart <dons@cse.unsw.edu.au>
Build-Depends:      base

Executable:         haq
Main-is:            Haq.hs
```

(If your package uses other packages, e.g. `array`, you'll need to add them to the `Build-Depends:` field.) Add a `Setup.lhs` that will actually do the building:

```
$ cat > Setup.lhs
#! /usr/bin/env runhaskell

> import Distribution.Simple
> main = defaultMain
```

Cabal allows either `Setup.hs` or `Setup.lhs`; as long as the format is appropriate, it doesn't matter which one you choose. But it's a good idea to always include the `#! /usr/bin/env runhaskell` line; because it follows the shebang[13] convention, you could execute the Setup.hs directly in a Unix shell instead of always manually calling `runhaskell` (assuming the Setup file is marked executable, of course).

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'
```

Git:

```
$ git add haq.cabal Setup.lhs
$ git commit -m 'Add a build system'
```

79.2.5 Build your project

Now build it!

```
$ runhaskell Setup.lhs configure --prefix=$HOME --user
$ runhaskell Setup.lhs build
$ runhaskell Setup.lhs install
```

[13] http://en.wikipedia.org/wiki/Shebang%20%28Unix%29

79.2.6 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"
```

79.2.7 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.

79.2.8 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!

79.2.9 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 e.g.: `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.

79.2.10 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'
```

Advanced Darcs functionality: lazy get

As your repositories accumulate patches, new users can become annoyed at how long it takes to accomplish the initial `darcs get`. (Some projects, like yi[14] or GHC, can have thousands of patches.) Darcs is quick enough, but downloading thousands of individual patches can still take a while. Isn't there some way to make things more efficient?

Darcs provides the `--lazy` option to `darcs get`. This enables to download only the latest version of the repository. Patches are later downloaded on demand if needed.

Distribution

When distributing your Haskell program, you have roughly three options:

1. distributing via a Darcs repository
2. distributing a tarball
   a) a Darcs tarball
   b) a Cabal tarball

With a Darcs repository, if it is public, than you are done. However: perhaps you don't have a server with Darcs, or perhaps your computer isn't set up for people to `darcs pull` from it. In which case you'll need to distribute the source via tarball.

Tarballs via darcs

Darcs provides a command where it will make a compressed tarball, and it will place a copy of all the files it manages into it. (Note that nothing in _darcs will be included - it'll just be your source files, no revision history.)

```
$ darcs dist -d haq-0.0
```

[14] http://en.wikipedia.org/wiki/Yi%20%28editor%29

```
Created dist as haq-0.0.tar.gz
```

And you're all set up!

Tarballs via Cabal

Since our 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 advantages and disadvantages compared to a Darcs-produced tarball. The primary *advantage* is that Cabal will do more checking of our repository, and more importantly, it'll ensure that the tarball has the structure needed by HackageDB and cabal-install.

However, it does have a disadvantage: it packages up only the files needed to build the project. It will deliberately fail to include other files in the repository, even if they turn out to be necessary at some point[15]. To include other files (such as `Test.hs` in the above example), we need to add lines to the cabal file like:

```
extra-source-files: Tests.hs
```

If we had them, we could make sure files like AUTHORS or the README get included as well:

```
data-files: AUTHORS, README
```

79.2.11 Summary

The following files were created:

```
    $ ls
    Haq.hs          Tests.hs        dist            haq.cabal
    Setup.lhs       _darcs          haq-0.0.tar.gz
```

79.3 Libraries

The process for creating a Haskell library is almost identical. The differences are as follows, for the hypothetical "ltree" library:

[15] This is actually a good thing, since it allows us to do things like create an elaborate test suite which doesn't get included in the tarball, so users aren't bothered by it. It also can reveal hidden assumptions and omissions in our code - perhaps your code was only building and running because of a file accidentally generated.

79.3.1 Hierarchical source

The source should live under a directory path that fits into the existing module layout guide[16]. 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

79.3.2 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
```

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:

[16] http://www.haskell.org/˜simonmar/lib-hierarchy.html

```
$ ghci -package ltree
Prelude> :m + Data.LTree
Prelude Data.LTree>
```

The new library is in scope, and ready to go.

79.3.3 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[17].

Internal modules

If your library uses internal modules that are not exposed, do not forget to list them in the *other-modules* field:

```
other-modules: My.Own.Module
```

Failing to do so (as of GHC 6.8.3) may lead to your library deceptively building without errors but actually being unusable from applications, which would fail at build time with a linker error.

79.4 Automation

79.4.1 cabal init

A package management tool for Haskell called cabal-install provides a command line tool to help developers create a simple cabal project. Just run and answer all the questions. Default values are provided for each.

```
$ cabal init
Package name [default "test"]?
Package version [default "0.1"]?
Please choose a license:
...
```

[17] http://www.haskell.org/ghc/docs/latest/html/Cabal/index.html

79.4.2 mkcabal

mkcabal is a tool that existed before cabal init, which also automatically populates a new cabal project :

```
    darcs get http://code.haskell.org/˜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
```

79.5 Licenses

Code for the common base library package must be BSD licensed or something more Free/Open. Otherwise, it is entirely up to you as the author.

Choose a licence (inspired by this[18]). 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 or public domain, and the GPL/LGPLers (this split roughly mirrors the copyleft/noncopyleft divide in Free software

[18] http://www.dina.dk/˜abraham/rants/license.html

communities). 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.

79.6 Releases

It's important to release your code as stable, tagged tarballs. Don't just rely on darcs for distribution[19].

- **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 get --lazy --tag 0.8`, to get just the tagged version (and not the entire history).

79.7 Hosting

You can host public and private Darcs repositories on `http://patch-tag.com/` for free. Otherwise, a Darcs repository can be published simply by making it available from a web page. Another option is to host on the Haskell Community Server at `http://code.`

[19] http://web.archive.org/web/20070627103346/http://awayrepl.blogspot.com/2006/11/we-dont-do-releases.html

haskell.org/. You can request an account via http://community.haskell.org/admin/. You can also use https://github.com/ for Git hosting.

79.8 Example

A complete example[20] of writing, packaging and releasing a new Haskell library under this process has been documented.

[20] http://www.cse.unsw.edu.au/~dons/blog/2006/12/11#release-a-library-today

80 Using the Foreign Function Interface (FFI)

Using Haskell is fine, but in the real world there are a large number of useful libraries in other languages, especially C. To use these libraries, and let C code use Haskell functions, there is the Foreign Function Interface (FFI).

80.1 Calling C from Haskell

80.1.1 Marshalling (Type Conversion)

When using C functions, it is necessary to convert Haskell types to the appropriate C types. These are available in the `Foreign.C.Types` module; some examples are given in the following table.

Haskell	**Foreign.C.Types**	**C**
Double	CDouble	double
Char	CUChar	unsigned char
Int	CLong	long int

The operation of converting Haskell types into C types is called **marshalling** (and the opposite, predictably, *unmarshalling*). For basic types this is quite straightforward: for floating-point one uses `realToFrac` (either way, as e.g. both `Double` and `CDouble` are instances of classes `Real` and `Fractional`), for integers `fromIntegral`, and so on.

⚠ **Warning**

If you are using GHC previous to 6.12.x, note that the `CLDouble` type does not really represent a `long double`, but is just a synonym for `CDouble`: *never use it*, since it will lead to silent type errors if the C compiler does not also consider `long double` a synonym for `double`. Since 6.12.x `CLDouble` has been removed[1], pending proper implementation[2].

80.1.2 Calling a pure C function

A pure function implemented in C does not present significant trouble in Haskell. The `sin` function of the C standard library is a fine example:

```
{-# LANGUAGE ForeignFunctionInterface #-}

import Foreign
import Foreign.C.Types

foreign import ccall unsafe "math.h sin"
     c_sin :: CDouble -> CDouble
```

First, we specify a GHC extension for the FFI in the first line. We then import the `Foreign` and `Foreign.C.Types` modules, the latter of which contains information about `CDouble`, the representation of double-precision floating-point numbers in C.

We then specify that we are *import*ing a *foreign* function, with a call to C. A "safety level" has to be specified with the keyword `safe` (the default) or `unsafe`. In general, `unsafe` is more efficient, and `safe` is required only for C code that could call back a Haskell function. Since that is a very particular case, it is actually quite safe to use the `unsafe` keyword in most cases. Finally, we need to specify header and function name, separated by a space.

The Haskell function name is then given, in our case we use a standard `c_sin`, but it could have been anything. Note that the function signature must be correct—GHC will not check the C header to confirm that the function actually takes a `CDouble` and returns another, and writing a wrong one could have unpredictable results.

It is then possible to generate a wrapper around the function using `CDouble` so that it looks exactly like any Haskell function.

```
haskellSin :: Double -> Double
haskellSin = realToFrac . c_sin . realToFrac
```

Importing C's `sin` is simple because it is a pure function that takes a plain `double` as input and returns another as output: things will complicate with impure functions and pointers, which are ubiquitous in more complicated C libraries.

80.1.3 Impure C Functions

A classic impure C function is `rand`, for the generation of pseudo-random numbers. Suppose you do not want to use Haskell's `System.Random.randomIO`, for example because you want to replicate exactly the series of pseudo-random numbers output by some C routine. Then, you could import it just like `sin` before:

```
{-# LANGUAGE ForeignFunctionInterface #-}

import Foreign
import Foreign.C.Types

foreign import ccall unsafe "stdlib.h rand"
     c_rand :: CUInt -- Oops!
```

If you try this naïve implementation in GHCI, you will notice that `c_rand` is returning always the same value:

```
> c_rand
1714636915
> c_rand
1714636915
```

indeed, we have told GHC that it is a pure function, and GHC sees no point in calculating twice the result of a pure function. Note that GHC did not give any error or warning message.

In order to make GHC understand this is no pure function, we have to use the IO monad[3]:

```
{-# LANGUAGE ForeignFunctionInterface #-}

import Foreign
import Foreign.C.Types

foreign import ccall unsafe "stdlib.h rand"
     c_rand :: IO CUInt

foreign import ccall "stdlib.h srand"
     c_srand :: CUInt -> IO ()
```

Here, we also imported the **srand** function, to be able to seed the C pseudo-random generator.

```
> c_rand
1957747793
> c_rand
424238335
> c_srand 0
> c_rand
1804289383
> c_srand 0
> c_rand
1804289383
```

80.1.4 Working with C Pointers

The most useful C functions are often those that do complicated calculations with several parameters, and with increasing complexity the need of returning control codes arises. This means that a typical paradigm of C libraries is to give pointers of allocated memory as "targets" in which the results may be written, while the function itself returns an integer value (typically, if 0, computation was successful, otherwise there was a problem specified by the number). Another possibility is that the function will return a pointer to a structure (possibly defined in the implementation, and therefore unavailable to us).

As a pedagogical example, we consider the **gsl_frexp** function[4] of the GNU Scientific Library[5], a freely available library for scientific computation. It is a simple C function with prototype:

[3] Chapter 33 on page 205

[4] http://www.gnu.org/software/gsl/manual/html_node/Elementary-Functions.html

[5] http://en.wikipedia.org/wiki/GNU_Scientific_Library

```
double gsl_frexp (double x, int * e)
```

The function takes a `double` x, and it returns its normalised fraction f and integer exponent e so that:

$$x = f \times 2^e \qquad e \in \mathbb{Z}, \quad 0.5 \leq f < 1$$

We interface this C function into Haskell with the following code:

```
{-# LANGUAGE ForeignFunctionInterface #-}

import Foreign
import Foreign.Ptr
import Foreign.C.Types

foreign import ccall unsafe "gsl/gsl_math.h gsl_frexp"
     gsl_frexp :: CDouble -> Ptr CInt -> IO CDouble
```

The new part is `Ptr`, which can be used with any instance of the `Storable` class, among which all C types, but also several Haskell types.

Notice how the result of the `gsl_frexp` function is in the `IO` monad. This is typical when working with pointers, be they used for input or output (as in this case); we will see shortly what would happen had we used a simple `CDouble` for the function.

The `frexp` function is implemented in pure Haskell code as follows:

```
frexp :: Double -> (Double, Int)
frexp x = unsafePerformIO $
    alloca $ \expptr -> do
        f <- gsl_frexp (realToFrac x) expptr
        e <- peek expptr
        return (realToFrac f, fromIntegral e)
```

We know that, memory management details aside, the function is pure: that's why the signature returns a tuple with f and e outside of the `IO` monad. Yet, f is provided *inside* of it: to extract it, we use the function *unsafePerformIO*, which extracts values from the `IO` monad: obviously, it is legitimate to use it only when we *know* the function is pure, and we can allow GHC to optimise accordingly.

To allocate pointers, we use the `alloca` function, which also takes responsibility for freeing memory. As an argument, `alloca` takes a function of type `Ptr a -> IO b`, and returns the `IO b`. In practice, this translates to the following usage pattern with λ functions:

```
... alloca $ \pointer -> do
        c_function( argument, pointer )
        result <- peek pointer
        return result
```

The pattern can easily be nested if several pointers are required:

```
... alloca $ \firstPointer ->
        alloca $ \secondPointer -> do
            c_function( argument, firstPointer, secondPointer )
            first  <- peek firstPointer
            second <- peek secondPointer
            return (first, second)
```

Back to our `frexp` function: in the λ function that is the argument to `alloca`, the function is evaluated and the pointer is read immediately afterwards with `peek`. Here we can understand why we wanted the imported C function `gsl_frexp` to return a value in the `IO` monad: if GHC could decide when to calculate the quantity *f*, it would likely decide not to do it until it is necessary: that is at the last line when `return` uses it, and *after e* has been read from an allocated, but yet uninitialised memory address, which will contain random data. In short, we want `gsl_frexp` to return a monadic value because we want to determine the sequence of computations ourselves.

If some other function had required a pointer to *provide input* instead of storing output, one would have used the similar `poke` function to set the pointed value, obviously *before* evaluating the function:

```
... alloca $ \inputPointer ->
        alloca $ \outputPointer -> do
            poke inputPointer value
            c_function( argument, inputPointer, outputPointer )
            result <- peek outputPointer
            return result
```

In the final line, the results are arranged in a tuple and returned, after having been converted from C types.

To test the function, remember to link GHC to the GSL; in GHCI, do:

```
$ ghci frexp.hs -lgsl
```

(Note that most systems do not come with the GSL preinstalled, and you may have to download and install its development packages.)

80.1.5 Working with C Structures

Very often data are returned by C functions in form of `structs` or pointers to these. In some rare cases, these structures are returned directly, but more often they are returned as pointers; the return value is most often an `int` that indicates the correctness of execution.

We will consider another GSL function, `gsl_sf_bessel_Jn_e`[6]. This function provides the regular cylindrical Bessel function for a given order *n*, and returns the result as a `gsl_sf_result` structure pointer. The structure contains two `double`s, one for the result and one for the error. The integer error code returned by the function can be transformed in a C string by the function `gsl_strerror`. The signature of the Haskell function we are looking for is therefore:

```
BesselJn :: Int -> Double -> Either String (Double, Double)
```

[6] http://www.gnu.org/software/gsl/manual/html_node/Regular-Cylindrical-Bessel-Functions.html

where the first argument is the order of the cylindrical Bessel function, the second is the function's argument, and the returned value is either an error message or a tuple with result and margin of error.

Making a New Instance of the `Storable` class

In order to allocate and read a pointer to a `gsl_sf_result` structure, it is necessary to make it an instance of the `Storable` class.

In order to do that, it is useful to use the `hsc2hs` program: we create first a `Bessel.hsc` file, with a mixed syntax of Haskell and C macros, which is later expanded into Haskell by the command:

```
$ hsc2hs Bessel.hsc
```

After that, we simply load the `Bessel.hs` file in GHC.

This is the first part of file `Bessel.hsc`:

```
{-# LANGUAGE ForeignFunctionInterface #-}

module Bessel (besselJn) where

import Foreign
import Foreign.Ptr
import Foreign.C.String
import Foreign.C.Types

#include <gsl/gsl_sf_result.h>

data GslSfResult = GslSfResult { gsl_value :: CDouble, gsl_error :: CDouble }

instance Storable GslSfResult where
    sizeOf    _ = (#size gsl_sf_result)
    alignment _ = alignment (undefined :: CDouble)
    peek ptr = do
        value <- (#peek gsl_sf_result, val) ptr
        error <- (#peek gsl_sf_result, err) ptr
        return  GslSfResult { gsl_value = value, gsl_error = error }
    poke ptr (GslSfResult value error) = do
        (#poke gsl_sf_result, val) ptr value
        (#poke gsl_sf_result, err) ptr error
```

We use the `#include` directive to make sure `hsc2hs` knows where to find information about `gsl_sf_result`. We then define a Haskell data structure mirroring the GSL's, with two `CDouble`s: this is the class we make an instance of `Storable`. Strictly, we need only `sizeOf`, `alignment` and `peek` for this example; `poke` is added for completeness.

- `sizeOf` is obviously fundamental to the allocation process, and is calculated by `hsc2hs` with the `#size` macro.
- `alignment` is the size in bytes of the data structure alignment[7]. In general, it should be the largest `alignment` of the elements of the structure; in our case, since the two elements

[7] http://en.wikipedia.org/wiki/Data_structure_alignment

are the same, we simply use CDouble's. The value of the argument to alignment is inconsequential, what is important is the type of the argument.

- peek is implemented using a do-block and the #peek macros, as shown. val and err are the names used for the structure fields in the GSL source code.
- Similarly, poke is implemented with the #poke macro.

Importing the C Functions

```
foreign import ccall unsafe "gsl/gsl_bessel.h gsl_sf_bessel_Jn_e"
     c_besselJn :: CInt -> CDouble -> Ptr GslSfResult -> IO CInt

foreign import ccall unsafe "gsl/gsl_errno.h gsl_set_error_handler_off"
     c_deactivate_gsl_error_handler :: IO ()

foreign import ccall unsafe "gsl/gsl_errno.h gsl_strerror"
     c_error_string :: CInt -> IO CString
```

We import several functions from the GSL libraries: first, the Bessel function itself, which will do the actual work. Then, we need a particular function, gsl_set_error_handler_off, because the default GSL error handler will simply crash the program, even if called by Haskell: we, instead, plan to deal with errors ourselves. The last function is the GSL-wide interpreter that translates error codes in human-readable C strings.

Implementing the Bessel Function

Finally, we can implement the Haskell version of the GSL cylindrical Bessel function of order *n*.

```
besselJn :: Int -> Double -> Either String (Double, Double)
besselJn n x = unsafePerformIO $
    alloca $ \gslSfPtr -> do
        c_deactivate_gsl_error_handler
        status <- c_besselJn (fromIntegral n) (realToFrac x) gslSfPtr
        if status == 0
            then do
                GslSfResult val err <- peek gslSfPtr
                return $ Right (realToFrac val, realToFrac err)
            else do
                error <- c_error_string status
                error_message <- peekCString error
                return $ Left ("GSL error: "++error_message)
```

Again, we use unsafePerformIO because the function is pure, even though its nuts-and-bolts implementation is not. After allocating a pointer to a GSL result structure, we deactivate the GSL error handler to avoid crashes in case something goes wrong, and finally we can call the GSL function. At this point, if the status returned by the function is 0, we unmarshal the result and return it as a tuple. Otherwise, we call the GSL error-string function, and pass the error as a Left result instead.

Examples

Once we are finished writing the `Bessel.hsc` function, we have to convert it to proper Haskell and load the produced file:

```
$ hsc2hs Bessel.hsc
$ ghci Bessel.hs -lgsl
```

We can then call the Bessel function with several values:

```
> besselJn 0 10
Right (-0.2459357644513483,1.8116861737200453e-16)
> besselJn 1 0
Right (0.0,0.0)
> besselJn 1000 2
Left "GSL error: underflow"
```

80.1.6 Advanced Topics

This section contains an advanced example with some more complex features of the FFI. We will import into Haskell one of the more complicated functions of the GSL, the one used to calculate the integral of a function between two given points with an adaptive Gauss-Kronrod algorithm[8]. The GSL function is `gsl_integration_qag`.

This example will illustrate function pointers, export of Haskell functions to C routines, enumerations, and handling pointers of unknown structures.

Available C Functions and Structures

The GSL has three functions which are necessary to integrate a given function with the considered method:

```
gsl_integration_workspace * gsl_integration_workspace_alloc (size_t n);
void gsl_integration_workspace_free (gsl_integration_workspace * w);
int gsl_integration_qag (const gsl_function * f, double a, double b,
                         double epsabs, double epsrel, size_t limit,
                         int key, gsl_integration_workspace * workspace,
                         double * result, double * abserr);
```

The first two deal with allocation and deallocation of a "workspace" structure of which we know nothing (we just pass a pointer around). The actual work is done by the last function, which requires a pointer to a workspace.

To provide functions, the GSL specifies an appropriate structure for C:

```
struct gsl_function
{
  double (* function) (double x, void * params);
```

[8] `http://www.gnu.org/software/gsl/manual/html_node/QAG-adaptive-integration.html`

```
  void * params;
};
```

The reason for the `void` pointer is that it is not possible to define λ functions in C: parameters are therefore passed along with a parameter of unknown type. In Haskell, we do not need the `params` element, and will consistently ignore it.

Imports and Inclusions

We start our `qag.hsc` file with the following:

```
{-# LANGUAGE ForeignFunctionInterface, EmptyDataDecls #-}

module Qag ( qag,
             gauss15,
             gauss21,
             gauss31,
             gauss41,
             gauss51,
             gauss61 ) where

import Foreign
import Foreign.Ptr
import Foreign.C.Types
import Foreign.C.String

#include <gsl/gsl_math.h>
#include <gsl/gsl_integration.h>

foreign import ccall unsafe "gsl/gsl_errno.h gsl_strerror"
     c_error_string :: CInt -> IO CString

foreign import ccall unsafe "gsl/gsl_errno.h gsl_set_error_handler_off"
    c_deactivate_gsl_error_handler :: IO ()
```

We declare the `EmptyDataDecls` pragma, which we will use later for the `Workspace` data type. Since this file will have a good number of functions that should not be available to the outside world, we also declare it a module and export only the final function `qag` and the `gauss-` flags. We also include the relevant C headers of the GSL. The import of C functions for error messages and deactivation of the error handler was described before.

Enumerations

One of the arguments of `gsl_integration_qag` is `key`, an integer value that can have values from 1 to 6 and indicates the integration rule. GSL defines a macro for each value, but in Haskell it is more appropriate to define a type, which we call `IntegrationRule`. Also, to have its values automatically defined by `hsc2hs`, we can use the `enum` macro:

```
newtype IntegrationRule = IntegrationRule { rule :: CInt }
#{enum IntegrationRule, IntegrationRule,
    gauss15 = GSL_INTEG_GAUSS15,
    gauss21 = GSL_INTEG_GAUSS21,
    gauss31 = GSL_INTEG_GAUSS31,
    gauss41 = GSL_INTEG_GAUSS41,
    gauss51 = GSL_INTEG_GAUSS51,
    gauss61 = GSL_INTEG_GAUSS61
  }
```

hsc2hs will search the headers for the macros and give our variables the correct values. The variables cannot be modified and are essentially constant flags. Since we did not export the IntegrationRule constructor in the module declaration, but only the gauss flags, it is impossible for a user to even construct an invalid value. One thing less to worry about!

Haskell Function Target

We can now write down the signature of the function we desire:

```haskell
qag :: IntegrationRule                  -- Algorithm type
    -> Int                              -- Step limit
    -> Double                           -- Absolute tolerance
    -> Double                           -- Relative tolerance
    -> (Double -> Double)               -- Function to integrate
    -> Double                           -- Integration interval start
    -> Double                           -- Integration interval end
    -> Either String (Double, Double)   -- Result and (absolute) error estimate
```

Note how the order of arguments is different from the C version: indeed, since C does not have the possibility of partial application, the ordering criteria are different than in Haskell.

As in the previous example, we indicate errors with a Either String (Double, Double) result.

Passing Haskell Functions to the C Algorithm

```haskell
type CFunction = CDouble -> Ptr () -> CDouble

data GslFunction = GslFunction (FunPtr CFunction) (Ptr ())
instance Storable GslFunction where
    sizeOf    _ = (#size gsl_function)
    alignment _ = alignment (undefined :: Ptr ())
    peek ptr = do
        function <- (#peek gsl_function, function) ptr
        return $ GslFunction function nullPtr
    poke ptr (GslFunction fun nullPtr) = do
        (#poke gsl_function, function) ptr fun

makeCfunction :: (Double -> Double) -> (CDouble -> Ptr () -> CDouble)
makeCfunction f = \x voidpointer -> realToFrac $ f (realToFrac x)

foreign import ccall "wrapper"
    makeFunPtr :: CFunction -> IO (FunPtr CFunction)
```

We define a shorthand type, CFunction, for readability. Note that the void pointer has been translated to a Ptr (), since we have no intention of using it. Then it is the turn of the gsl_function structure: no surprises here. Note that the void pointer is always assumed to be null, both in peek and in poke, and is never really read nor written.

To make a Haskell Double -> Double function available to the C algorithm, we make two steps: first, we re-organise the arguments using a λ function in makeCfunction; then, in makeFunPtr, we take the function with reordered arguments and produce a function pointer that we can pass on to poke, so we can construct the GslFunction data structure.

Handling Unknown Structures

```
data Workspace
foreign import ccall unsafe "gsl/gsl_integration.h
 gsl_integration_workspace_alloc"
    c_qag_alloc :: CSize -> IO (Ptr Workspace)
foreign import ccall unsafe "gsl/gsl_integration.h
 gsl_integration_workspace_free"
    c_qag_free  :: Ptr Workspace -> IO ()

foreign import ccall safe "gsl/gsl_integration.h gsl_integration_qag"
    c_qag :: Ptr GslFunction -- Allocated GSL function structure
          -> CDouble -- Start interval
          -> CDouble -- End interval
          -> CDouble -- Absolute tolerance
          -> CDouble -- Relative tolerance
          -> CSize   -- Maximum number of subintervals
          -> CInt    -- Type of Gauss-Kronrod rule
          -> Ptr Workspace -- GSL integration workspace
          -> Ptr CDouble -- Result
          -> Ptr CDouble -- Computation error
          -> IO CInt -- Exit code
```

The reason we imported the `EmptyDataDecls` pragma is this: we are declaring the data structure `Workspace` without providing any constructor. This is a way to make sure it will always be handled as a pointer, and never actually instantiated.

Otherwise, we normally import the allocating and deallocating routines. We can now import the integration function, since we have all the required pieces (`GslFunction` and `Workspace`).

The Complete Function

It is now possible to implement a function with the same functionality as the GSL's QAG algorithm.

```
qag gauss steps abstol reltol f a b = unsafePerformIO $ do
    c_deactivate_gsl_error_handler
    workspacePtr <- c_qag_alloc (fromIntegral steps)
    if workspacePtr == nullPtr
        then
            return $ Left "GSL could not allocate workspace"
        else do
            fPtr <- makeFunPtr $ makeCfunction f
            alloca $ \gsl_f -> do
                poke gsl_f (GslFunction fPtr nullPtr)
                alloca $ \resultPtr -> do
                    alloca $ \errorPtr -> do
                        status <- c_qag gsl_f
                                        (realToFrac a)
                                        (realToFrac b)
                                        (realToFrac abstol)
                                        (realToFrac reltol)
                                        (fromIntegral steps)
                                        (rule gauss)
                                        workspacePtr
                                        resultPtr
                                        errorPtr
                        c_qag_free workspacePtr
                        freeHaskellFunPtr fPtr
                        if status /= 0
```

```
                           then do
                               c_errormsg <- c_error_string status
                               errormsg   <- peekCString c_errormsg
                               return $ Left errormsg
                           else do
                               c_result <- peek resultPtr
                               c_error  <- peek  errorPtr
                               let result = realToFrac c_result
                               let error  = realToFrac c_error
                               return $ Right (result, error)
```

First and foremost, we deactivate the GSL error handler, that would crash the program instead of letting us report the error.

We then proceed to allocate the workspace; notice that, if the returned pointer is null, there was an error (typically, too large size) that has to be reported.

If the workspace was allocated correctly, we convert the given function to a function pointer and allocate the `GslFunction` struct, in which we place the function pointer. Allocating memory for the result and its error margin is the last thing before calling the main routine.

After calling, we have to do some housekeeping and free the memory allocated by the workspace and the function pointer. Note that it would be possible to skip the bookkeeping using `ForeignPtr`, but the work required to get it to work is more than the effort to remember one line of cleanup.

We then proceed to check the return value and return the result, as was done for the Bessel function.

80.1.7 Self-Deallocating Pointers

In the previous example, we manually handled the deallocation of the GSL integration workspace, a data structure we know nothing about, by calling its C deallocation function. It happens that the same workspace is used in several integration routines, which we may want to import in Haskell.

Instead of replicating the same allocation/deallocation code each time, which could lead to memory leaks when someone forgets the deallocation part, we can provide a sort of "smart pointer", which will deallocate the memory when it is not needed any more. This is called `ForeignPtr` (do not confuse with `Foreign.Ptr`: this one's qualified name is actually `Foreign.ForeignPtr`!). The function handling the deallocation is called the *finalizer*.

In this section we will write a simple module to allocate GSL workspaces and provide them as appropriately configured `ForeignPtr`s, so that users do not have to worry about deallocation.

The module, written in file `GSLWorkspace.hs`, is as follows:

```
{-# LANGUAGE ForeignFunctionInterface, EmptyDataDecls #-}

module GSLWorkSpace (Workspace, createWorkspace) where

import Foreign.C.Types
import Foreign.Ptr
import Foreign.ForeignPtr
```

```haskell
data Workspace
foreign import ccall unsafe "gsl/gsl_integration.h
 gsl_integration_workspace_alloc"
    c_ws_alloc :: CSize -> IO (Ptr Workspace)
foreign import ccall unsafe "gsl/gsl_integration.h
 &gsl_integration_workspace_free"
    c_ws_free  :: FunPtr( Ptr Workspace -> IO () )

createWorkspace :: CSize -> IO (Maybe (ForeignPtr Workspace) )
createWorkspace size = do
    ptr <- c_ws_alloc size
    if ptr /= nullPtr
        then do
           foreignPtr <- newForeignPtr c_ws_free ptr
           return $ Just foreignPtr
        else
           return Nothing
```

We first declare our empty data structure `Workspace`, just like we did in the previous section.

The `gsl_integration_workspace_alloc` and `gsl_integration_workspace_free` functions will no longer be needed in any other file: here, note that the deallocation function is called with an ampersand ("&"), because we do not actually want the function, but rather a *pointer* to it to set as a finalizer.

The workspace creation function returns a IO (Maybe) value, because there is still the possibility that allocation is unsuccessful and the null pointer is returned. The GSL does not specify what happens if the deallocation function is called on the null pointer, so for safety we do not set a finalizer in that case and return `IO Nothing`; the user code will then have to check for "`Just`-ness" of the returned value.

If the pointer produced by the allocation function is non-null, we build a foreign pointer with the deallocation function, inject into the `Maybe` and then the `IO` monad. That's it, the foreign pointer is ready for use!

⚠ Warning

This function requires object code to be compiled, so if you load this module with GHCI (which is an interpreter) you must indicate it:

```
$ ghci GSLWorkSpace.hs -fobject-code
```

Or, from within GHCI:

```
> :set -fobject-code
> :load GSLWorkSpace.hs
```

The `qag.hsc` file must now be modified to use the new module; the parts that change are:

```haskell
{-# LANGUAGE ForeignFunctionInterface #-}

-- [...]

import GSLWorkSpace

import Data.Maybe(isNothing, fromJust)
```

```
-- [...]

qag gauss steps abstol reltol f a b = unsafePerformIO $ do
    c_deactivate_gsl_error_handler
    ws <- createWorkspace (fromIntegral steps)
    if isNothing ws
        then
            return $ Left "GSL could not allocate workspace"
        else do
            withForeignPtr (fromJust ws) $ \workspacePtr -> do

-- [...]
```

Obviously, we do not need the `EmptyDataDecls` extension here any more; instead we import the `GSLWorkSpace` module, and also a couple of nice-to-have functions from `Data.Maybe`. We also remove the foreign declarations of the workspace allocation and deallocation functions.

The most important difference is in the main function, where we (try to) allocate a workspace `ws`, test for its `Just`ness, and if everything is fine we use the `withForeignPtr` function to extract the workspace pointer. Everything else is the same.

80.2 Calling Haskell from C

Sometimes it is also convenient to call Haskell from C, in order to take advantage of some of Haskell's features which are tedious to implement in C, such as lazy evaluation.

We will consider a typical Haskell example, Fibonacci numbers. These are produced in an elegant, haskellian one-liner as:

```
fibonacci = 0 : 1 : zipWith (+) fibonacci (tail fibonacci)
```

Our task is to export the ability to calculate Fibonacci numbers from Haskell to C. However, in Haskell, we typically use the `Integer` type, which is unbounded: this cannot be exported to C, since there is no such corresponding type. To provide a larger range of outputs, we specify that the C function shall output, whenever the result is beyond the bounds of its integer type, an approximation in floating-point. If the result is also beyond the range of floating-point, the computation will fail. The status of the result (whether it can be represented as a C integer, a floating-point type or not at all) is signalled by the status integer returned by the function. Its desired signature is therefore:

```
int fib( int index, unsigned long long* result, double* approx )
```

80.2.1 Haskell Source

The Haskell source code for file `fibonacci.hs` is:

```
{-# LANGUAGE ForeignFunctionInterface #-}

module Fibonacci where

import Foreign
import Foreign.C.Types
```

```
fibonacci :: (Integral a) => [a]
fibonacci = 0 : 1 : zipWith (+) fibonacci (tail fibonacci)

foreign export ccall fibonacci_c :: CInt -> Ptr CULLong -> Ptr CDouble -> IO
 CInt
fibonacci_c :: CInt -> Ptr CULLong -> Ptr CDouble -> IO CInt
fibonacci_c n intPtr dblPtr
    | badInt && badDouble = return 2
    | badInt              = do
        poke dblPtr dbl_result
        return 1
    | otherwise           = do
        poke intPtr (fromIntegral result)
        poke dblPtr dbl_result
        return 0
    where
    result     = fibonacci !! (fromIntegral n)
    dbl_result = realToFrac result
    badInt     = result > toInteger (maxBound :: CULLong)
    badDouble  = isInfinite dbl_result
```

When exporting, we need to wrap our functions in a module (it is a good habit anyway). We have already seen the Fibonacci infinite list, so let's focus on the exported function: it takes an argument, two pointers to the target `unsigned long long` and `double`, and returns the status in the `IO` monad (since writing on pointers is a side effect).

The function is implemented with input guards, defined in the `where` clause at the bottom. A successful computation will return 0, a partially successful 1 (in which we still can use the floating-point value as an approximation), and a completely unsuccessful one will return 2.

Note that the function does not call `alloca`, since the pointers are assumed to have been already allocated by the calling C function.

The Haskell code can then be compiled with GHC:

```
ghc -c fibonacci.hs
```

80.2.2 C Source

The compilation of `fibonacci.hs` has spawned several files, among which `fibonacci_stub.h`, which we include in our C code in file `fib.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include "fibonacci_stub.h"

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <number>\n", argv[0]);
        return 2;
    }

    hs_init(&argc, &argv);

    const int arg = atoi(argv[1]);
    unsigned long long res;
    double approx;
    const int status = fibonacci_c(arg, &res, &approx);
    hs_exit();
    switch (status) {
    case 0:
        printf("F_%d: %llu\n", arg, res);
        break;
    case 1:
        printf("Error: result is out of bounds\n");
        printf("Floating-point approximation: %e\n", approx);
        break;
    case 2:
        printf("Error: result is out of bounds\n");
        printf("Floating-point approximation is infinite\n");
        break;
    default:
        printf("Unknown error: %d\n", status);
    }

    return status;
}
```

The notable thing is that we need to initialise the Haskell environment with `hs_init`, which we call passing it the command-line arguments of main; we also have to shut Haskell down with `hs_exit()` when we are done. The rest is fairly standard C code for allocation and error handling.

Note that you have to compile the C code *with GHC*, not your C compiler!

```
ghc -no-hs-main fib.c fibonacci.o fibonacci_stub.o -o fib
```

You can then proceed to test the algorithm:

```
./fib 42
F_42: 267914296
$ ./fib 666
Error: result is out of bounds
Floating-point approximation: 6.859357e+138
$ ./fib 1492
Error: result is out of bounds
Floating-point approximation is infinite
./fib -1
fib: Prelude.(!!): negative index
```

81 Generic Programming : Scrap your boilerplate

The "Scrap your boilerplate" approach, "described" in http://www.cs.vu.nl/boilerplate/, is a way to allow your data structures to be traversed by so-called "generic" functions: that is, functions that abstract over the specific data constructors being created or modified, while allowing for the addition of cases for specific types.

For instance if you want to serialize all the structures in your code, but you want to write only one serialization function that operates over any instance of the Data.Data.Data class (which can be derived with -XDeriveDataTypeable).

81.1 Serialization Example

The goal is to convert all our data into a format below:

```
data Tag = Con String | Val String
```

81.2 Comparing Haskell ASTs

The haskell-src-exts package[1] parses Haskell into a quite complicated syntax tree. Let's say we want to check if two source files that are nearly identical.

To start:

```
import System.Environment
import Language.Haskell.Exts
```

```
main = do
   -- parse the filenames given by the first two command line arguments,
   -- proper error handling is left as an exercise
   ParseOk moduleA: ParseOk moduleB:_ <- mapM parseFile . take 2 =<< getArgs

   putStrLn $ if moduleA == moduleB
        then "Your modules are equal"
        else "Your modules differ"
```

[1] http://hackage.haskell.org/package/haskell-src-exts

From a bit of testing, it will be apparent that identical files with different names will not be equal to (==). However, to correct the fact, without resorting to lots of boilerplate, we can use generic programming:

81.3 TODO

describe using Data.Generics.Twins.gzip*? to write a function to find where there are differences?

Or use it to write a variant of geq that ignores the specific cases that are unimportant (the SrcLoc elements) (i.e. syb doesn't allow generic extension... contrast it with other libraries?).

Or just explain this hack (which worked well enough) to run before (==), or geq::

```
everyWhere (mkT $ \ _ -> SrcLoc "" 0 0) :: Data a => a -> a
```

Or can we develop this into writing something better than sim_mira (for hs code), found here: `http://dickgrune.com/Programs/similarity_tester/`

82 Specialised Tasks

83 Graphical user interfaces (GUI)

Haskell has at least four toolkits for programming a graphical interface:

- wxHaskell[1] - provides a Haskell interface to the wxWidgets toolkit
- Gtk2Hs[2] - provides a Haskell interface to the GTK+ library
- hoc[3] (documentation at sourceforge[4]) - provides a Haskell to Objective-C binding which allows users to access to the Cocoa library on MacOS X
- qtHaskell[5] - provides a set of Haskell bindings for the Qt Widget Library from Nokia

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.

83.1 Getting and running wxHaskell

To install wxHaskell, look for your version of instructions at: Linux[6] Mac[7] Windows[8]

or the wxHaskell download page[9] and follow the installation instructions provided on the wxHaskell download page. Don't forget to register wxHaskell with GHC, or else it won't run (automatically registered with Cabal). 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

[1] http://en.wikibooks.org/wiki/%3Aw%3AWxHaskell
[2] http://www.haskell.org/haskellwiki/Gtk2Hs
[3] http://code.google.com/p/hoc/
[4] http://hoc.sourceforge.net/
[5] http://qthaskell.berlios.de/
[6] http://www.haskell.org/haskellwiki/WxHaskell/Linux
[7] http://www.haskell.org/haskellwiki/WxHaskell/Mac
[8] http://www.haskell.org/haskellwiki/WxHaskell/Windows
[9] http://wxhaskell.sourceforge.net/download.html

(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.

83.2 Hello World

Here's the basic Haskell "Hello World" program:

```
module Main where

main :: IO ()
main = putStr "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)

83.3 Controls

> **Information**
>
> From here on, its good practice to keep a browser window or tab open with the wxHaskell documentation[10]. It's also available in $wxHaskellDir/doc/index.html.

83.3.1 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:

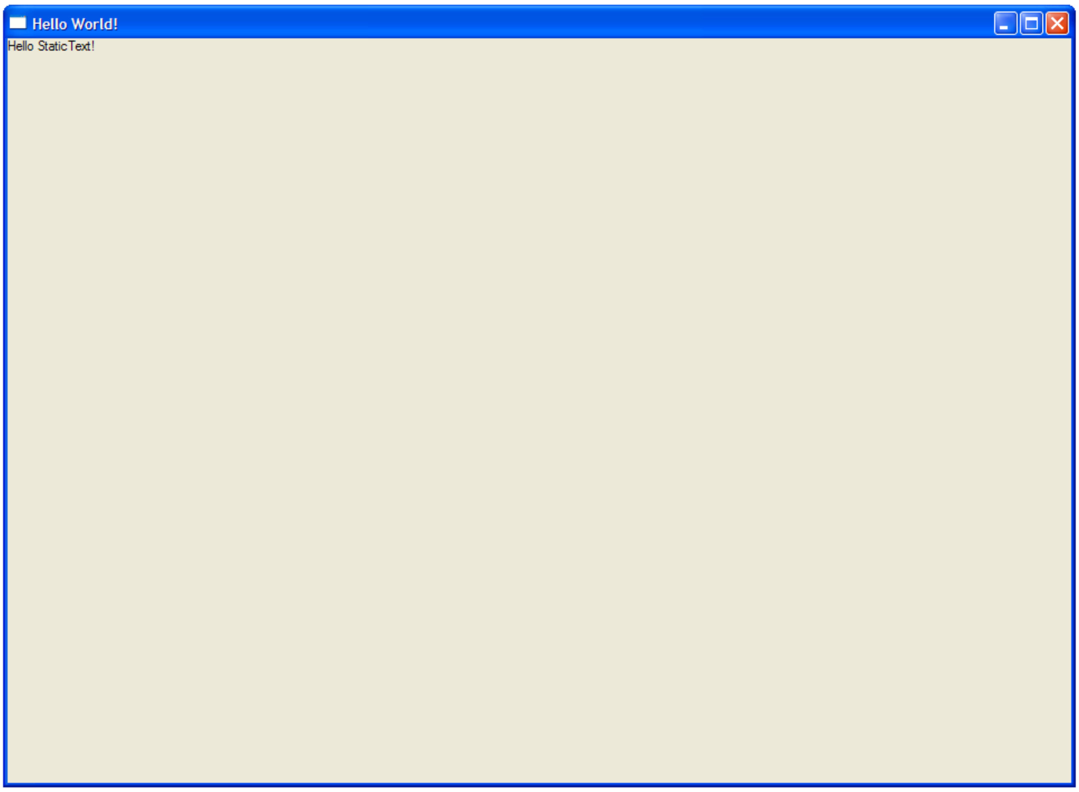


Figure 41 Hello StaticText! (winXP)

```
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!

83.3.2 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:

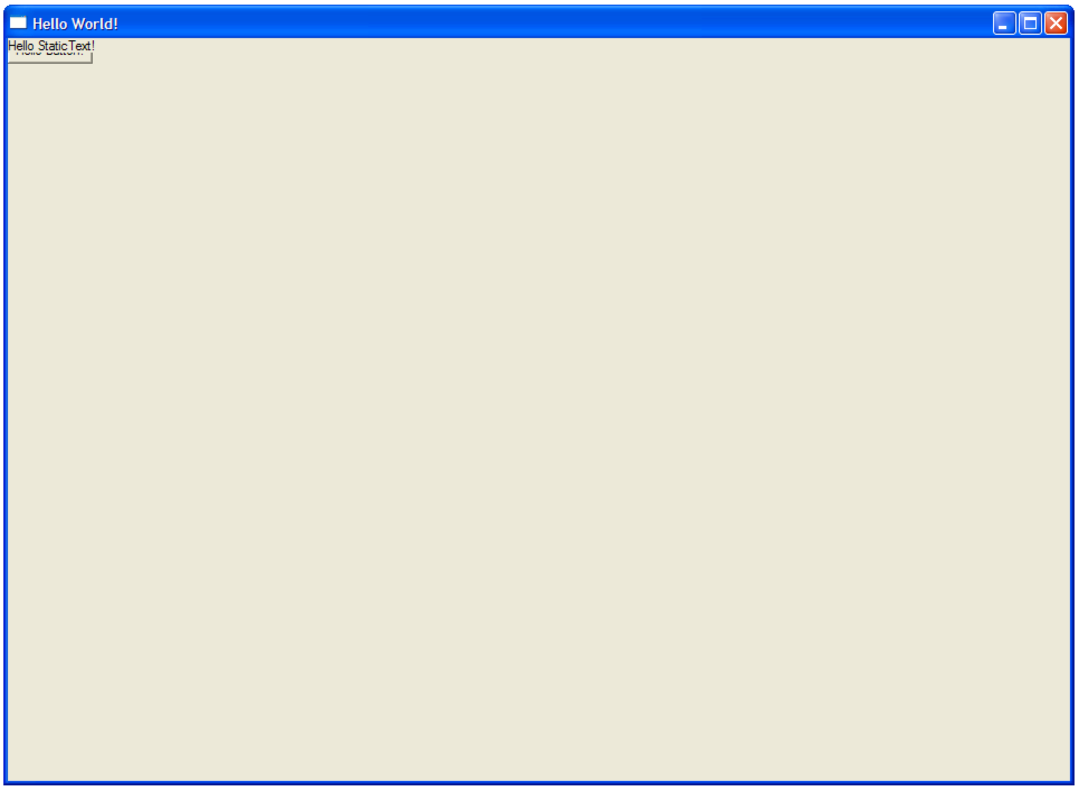


Figure 42 Overlapping button and StaticText (winXP)

```
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.

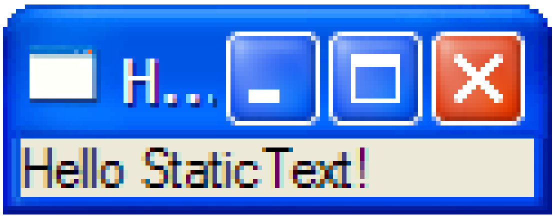
83.4 Layout

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 occurrence 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:



Hello StaticText!

Figure 43 StaticText with layout (winXP)

```
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:

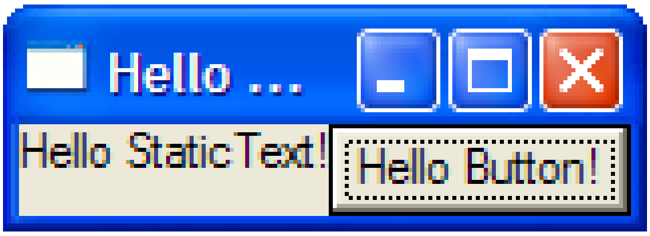


Figure 44 A row layout (winXP)

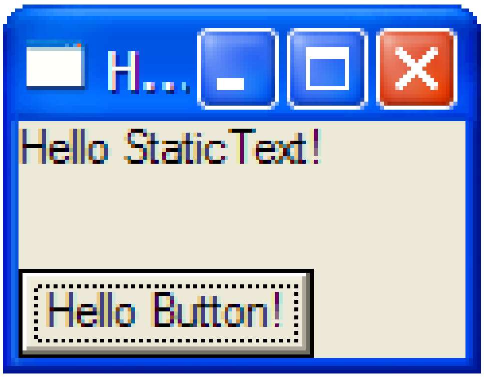


Figure 45 Column layout with a spacing of 25 (winXP)

```
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 a bug in wxhaskell.*)

After having completed the exercises, the end result should look like this:

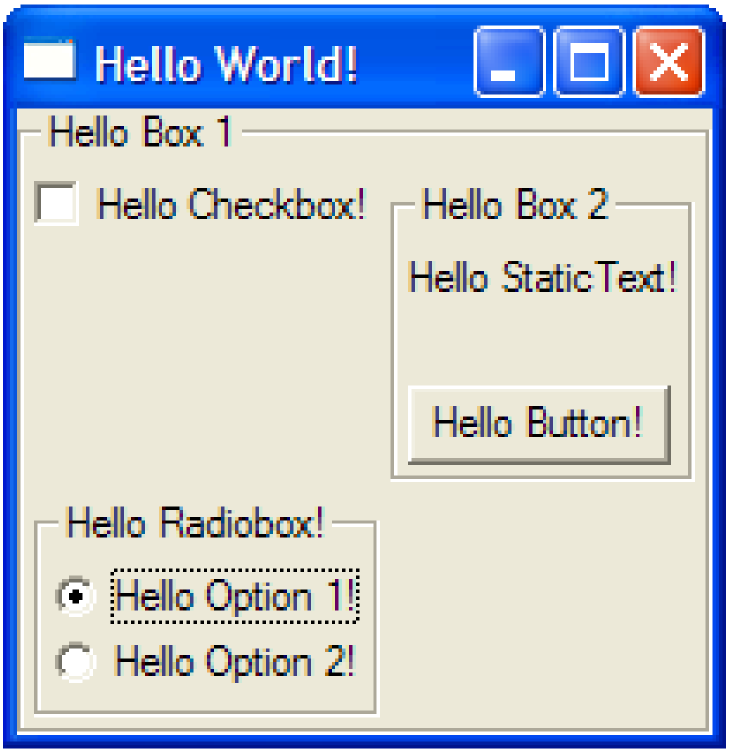


Figure 46 Answer to exercises

You could have used different spacing for `row` and `column`, or the options of the radiobox are displayed horizontally.

83.5 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.

83.5.1 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 a 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.

83.5.2 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`[11], and click the link that says "Button"[12]. 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 ../Classes and types/[13] 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.

83.6 Events

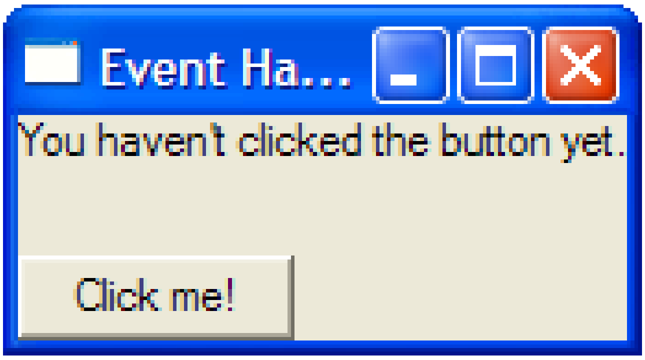
There are a few classes that deserve special attention. They are the `Reactive` class and the `Commanding` class. As you can see in the documentation of these classes, they don't add attributes (of the form `Attr w a`), but *events*. The `Commanding` class adds the `command` event. We'll use a button to demonstrate event handling.

Here's a simple GUI with a button and a staticText:

[11] `http://hackage.haskell.org/packages/archive/wx/0.10.2/doc/html/Graphics-UI-WX-Controls.html`

[12] `http://hackage.haskell.org/packages/archive/wx/0.10.2/doc/html/Graphics-UI-WX-Controls.html#4`

[13] Chapter 26 on page 169



You haven't clicked the button yet.

Figure 47 Before (winXP)

```
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

84 Databases

84.1 Introduction

Haskell's most popular database module is HDBC[1]. HDBC provides an abstraction layer between Haskell programs and SQL relational databases. This lets you write database code once, in Haskell, and have it work with a number of backend SQL databases.

HDBC is modeled loosely on Perl's DBI interface[2], though it has also been influenced by Python's DB-API v2, JDBC in Java, and HSQL in Haskell. As DBI requires DBD in Perl, HDBC requires a driver module beneath it to work. These HDBC backend drivers exist: PostgreSQL, SQLite, and ODBC (for Windows and Unix/Linux/Mac). MySQL is the most popular open-sourced database, and there are two drivers for MySQL: HDBC-mysql[3] (native) and HDBC-odbc[4] (ODBC). MySQL users can use the ODBC driver on any MySQL-supported platform, including Linux. An advantage of using ODBC is that the syntax of the SQL statement is insulated from the different kinds of database engines. This increases the portability of the application should you have to move from one database to another. The same argument for preferring ODBC applies for other commercial databases, such as Oracle and DB2.

84.2 Installation

84.2.1 SQLite

See here[5] for more information.

84.2.2 PostgreSQL

See here[6] for more information.

84.2.3 Native MySQL

The native ODBC-mysql library requires the C MySQL client library to be present.

[1] https://github.com/hdbc/hdbc/wiki
[2] http://search.cpan.org/~timb/DBI/DBI.pm
[3] http://hackage.haskell.org/package/HDBC-mysql
[4] http://hackage.haskell.org/package/HDBC-odbc
[5] https://github.com/hdbc/hdbc/wiki/FrequentlyAskedQuestions
[6] https://github.com/hdbc/hdbc/wiki/FrequentlyAskedQuestions

You may need to wrap your database accesses[7] to prevent runtime errors.

84.2.4 ODBC/MySQL

Instruction how to install ODBC/MySQL. It is somewhat involved to make HDBC work with MySQL via ODBC, especially if you do not have root privilege and have to install everything in a non-root account.

- If your platform doesn't already provide an ODBC library (and most do), install Unix-ODBC. See here[8] for more information.
- Install MySQL-ODBC Connector. See here[9] for more information.
- Install Database.HDBC module
- Install Database.HDBC.ODBC module
- Add the mysql driver to odbcinst.ini file (under $ODBC_HOME/etc/) and your data source in $HOME/.odbc.ini.
- Create a test program

Since the ODBC driver is installed using shared library by default, you will need the following env:

```
export LD_LIBRARY_PATH=$ODBC_HOME/lib
```

If you do not like adding an additional env variables, you should try to compile ODBC with static library option enabled.

The next task is to write a simple test program that connects to the database and print the names of all your tables, as shown below.

You may need to wrap your database accesses[10] in order to prevent runtime errors.

```
module Main where
import Database.HDBC.ODBC
import Database.HDBC
main =
  do c  <- connectODBC "DSN=PSPDSN"
     xs <- getTables c
     putStr $ "tables "++(foldr jn "." xs)++"\n"
  where jn a b = a++" "++b
```

[7] http://www.serpentine.com/blog/2010/09/04/dealing-with-fragile-c-libraries-e-g-mysql-from-haskell/

[8] http://sourceforge.net/projects/unixodbc/

[9] http://dev.mysql.com/downloads/connector/odbc/

[10] http://www.serpentine.com/blog/2010/09/04/dealing-with-fragile-c-libraries-e-g-mysql-from-haskell/

84.3 General Workflow

84.3.1 Connect and Disconnect

The first step of any database operation is to connect to the target database. This is done via the driver-specific connect API, which has the type of:

```
String -> IO Connection
```

Given a connect string, the connect API will return `Connection` and put you in the IO monad.

Although most program will garbage collect your connections when they are out of scope or when the program ends, it is a good practice to disconnect from the database explicitly.

```
 conn->Disconnect
```

84.3.2 Running Queries

Running a query generally involves the following steps:

- Prepare a statement
- Execute a statement with bind variables
- Fetch the result set (if any)
- Finish the statement

HDBC provides two ways for bind variables and returning result set: `[ SqlValue ]` and `[ Maybe String ]`. You need to use the functions with `s` prefix when using `[ Maybe String ]`, instead of `[ SqlValue ]`. `[ SqlValue ]` allows you to use strongly typed data if type safety is very important in your application; otherwise, `[ Maybe String ]` is more handy when dealing with lots of database queries. When you use `[ Maybe String ]`, you assume the database driver will perform automatic data conversion. Be aware there is a performance price for this convenience.

Sometimes, when the query is simple, there are simplified APIs that wrap multiple steps into one. For example, **Run** and **sRun** are wrappers of "prepare and execute". **quickQuery** is a wrapper of "prepare, execute, and fetch all rows".

84.4 Running SQL Statements

84.4.1 Select

84.4.2 Insert

84.4.3 Update

84.4.4 Delete

84.5 Transaction

Database transaction is controlled by `commit` and `rollback`. However, be aware some databases (such as mysql) do not support transaction. Therefore, every query is in its atomic transaction.

HDBC provides `withTransaction` to allow you automate the transaction control over a group of queries.

84.6 Calling Procedure

85 Web programming

An example web application, using the HAppS framework, is hpaste[1], 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[2] exist, and might be useful.

[1] http://hpaste.org

[2] http://homepages.paradise.net.nz/warrickg/haskell/http/

86 Working with 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[1] chapter.

86.0.1 Libraries for parsing XML

- The Haskell XML Toolbox (hxt)[2] is a collection of tools for parsing XML, aiming at a more general approach than the other tools.
- HaXml[3] is a collection of utilities for parsing, filtering, transforming, and generating XML documents using Haskell.
- HXML[4] is a non-validating, lazy, space efficient parser that can work as a drop-in replacement for HaXml.

86.0.2 Libraries for generating XML

- HSXML represents XML documents as statically typesafe s-expressions.

86.0.3 Other options

- tagsoup[5] is a library for parsing unstructured HTML, i.e. it does not assume validity or even well-formedness of the data.

86.1 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[6], including GHCi, and you should have downloaded and installed HXT according to the instructions[7].

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:

[1] Chapter 85 on page 569
[2] `http://www.fh-wedel.de/~si/HXmlToolbox/`
[3] `http://projects.haskell.org/HaXml/`
[4] `http://www.flightlab.com/~joe/hxml/`
[5] `http://www.cs.york.ac.uk/fp/darcs/tagsoup/tagsoup.htm`
[6] Chapter 2 on page 5
[7] `http://www.fh-wedel.de/~si/HXmlToolbox/#install`

```
 Prelude> :m + Text.XML.HXT.Parser.XmlParsec
 Prelude Text.XML.HXT.Parser.XmlParsec> 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.XmlParsec Text.XML.HXT.DOM> :i NTree
 data NTree a = NTree a (NTrees a)
                          -- Defined in Data.Tree.NTree.TypeDefs
Prelude Text.XML.HXT.Parser.XmlParsec 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.XmlParsec> :m + Text.XML.HXT.DOM
  Prelude Text.XML.HXT.Parser.XmlParsec 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.XmlParsec> 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[8]:

```
  Prelude> :set prompt "> "
  > :m + Text.XML.HXT.Parser.XmlParsec Text.XML.HXT.XPath.XPathEval
  > let xml = "<foo>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
```

[8] http://en.wikipedia.org/wiki/XPath

87 Using Regular Expressions

Good tutorials where to start

- serpentine.com[1]
- Regular Expressions (Haskell Wiki)[2]

Also see Haskell/Pattern matching[3] (absolutely not the same as regular expressions in Haskell, but instead of trivial regexps, it may be more elegant).

[1] http://www.serpentine.com/blog/2007/02/27/a-haskell-regular-expression-tutorial/

[2] http://www.haskell.org/haskellwiki/Regular_expressions

[3] Chapter 16 on page 113

88 Parsing Mathematical Expressions

This chapter discusses how to turn strings of text such as "3*sin x + y" into an abstract syntactic representation like Plus (Times (Number 3) (Apply "sin" (Variable "x"))) (Variable "y").

We are going to use Text.ParserCombinators.ReadP[1] throughout, so you will need to have the reference open to refer to.

88.1 First Warmup

```
import Text.ParserCombinators.ReadP
```

For a warmup, to get started on the problem, we first try an easier problem. A language where the symbols are just the letter "o", a single operator "&" and brackets. First define a data type for these trees:

```
data Tree = Branch Tree Tree | Leaf deriving Show
```

Now a parser for leaves is defined using the ReadP library:

```
leaf = do char 'o'
          return Leaf
```

now to define a parser for the branches, made up by "&" operator we need to choose an associativity. That is, whether o&o&o should be the same as (o&o)&o or o&(o&o) - let us pick the latter.

For a first approximation we can forget about brackets, adding them in after the first "milestone":

```
branch = do a <- leaf
            char '&'
            b <- tree
            return (Branch a b)

tree = leaf +++ branch
```

[1] http://hackage.haskell.org/packages/archive/base/latest/doc/html/Text-ParserCombinators-ReadP.html

It's now possible to test this out and see if it acts properly on a few inputs:

```
*Main> readP_to_S tree "o"
[(Leaf,"")]
*Main> readP_to_S tree "o&o"
[(Leaf,"&o"),(Branch Leaf Leaf,"")]
*Main> readP_to_S tree "o&o&o"
[(Leaf,"&o&o"),(Branch Leaf Leaf,"&o"),(Branch Leaf (Branch Leaf Leaf),"")]
```

Since that worked fine we can proceed to add support for parenthesis. Brackets are defined generally, so that we can reuse it later on

```
brackets p = do char '('
                r <- p
                char ')'
                return r
```

We can now update the branch and tree parsers to support brackets:

```
branch = do a <- leaf +++ brackets tree
            char '&'
            b <- tree
            return (Branch a b)

tree = leaf +++ branch +++ brackets tree
```

A bit of testing shows that it seems to work

```
   *Main> readP_to_S tree "((o&((o&o)))&o&((o&o)&o)&o)"
   [(Branch (Branch Leaf (Branch Leaf Leaf)) (Branch Leaf (Branch (Branch
(Branch Leaf Leaf) Leaf) Leaf)),"")]
```

88.2 Adaptation

This gives a good starting point for adaptation. The first modification towards the ultimate goal, which is quite easy to do, is changing the leaves from just "o" to any string. To do this we have change to `Leaf` to `Leaf String` in the data type and update the leaf function:

```
data Tree = Branch Tree Tree | Leaf String deriving Show

leaf = do s <- many1 (choice (map char ['a'..'z']))
          return (Leaf s)
```

For the next adaptation we try and add a new operation "|" which binders weaker than "&". I.e. "foo&bar|baz" should parse as "(foo&bar)|baz". First we need to update the data type representing syntax:

```
    data Operator = And | Or deriving Show

    data Tree = Branch Operator Tree Tree | Leaf String deriving Show
```

The obvious thing to do is duplicate the `branch` function and call it `andBranch` and `orBranch`, and give or precedence using the left choice operator `<++`:

```
    andBranch = do a <- leaf +++ brackets tree
                   char '&'
                   b <- tree
                   return (Branch And a b)

    orBranch = do a <- leaf +++ brackets tree
                  char '|'
                  b <- tree
                  return (Branch Or a b)

    tree = leaf +++ (orBranch <++ andBranch) +++ brackets tree
```

This modification does not work though, if we think of an expression such as "a&b&c&d|e&f&g&h|i&j&k|l&m&n&o|p&q&r|s" as a tree "X|Y|Z|W|P|Q" (which we already know how to parse!) except that the leaves are a more complicated form (but again, one we already know how to parse) then we can compose a working parser:

```
    andBranch = do a <- leaf +++ brackets tree
                   char '&'
                   b <- andTree
                   return (Branch And a b)

    andTree = leaf +++ brackets tree +++ andBranch

    orBranch = do a <- andTree +++ brackets tree
                  char '|'
                  b <- orTree
                  return (Branch Or a b)

    orTree = andTree +++ brackets tree +++ orBranch

    tree = orTree
```

While this approach does work, for example:

```
    *Main> readP_to_S tree "(foo&bar|baz)"
    [(Leaf "","(foo&bar|baz)"),(Branch Or (Branch And (Leaf "foo") (Leaf "bar"))
  (Leaf "baz"),""),(Branch Or (Branch And (Leaf "foo") (Leaf "bar")) (Leaf
  "baz"),"")]
    *Main> readP_to_S tree "(foo|bar&baz)"
    [(Leaf "","(foo|bar&baz)"),(Branch Or (Leaf "foo") (Branch And (Leaf "bar")
  (Leaf "baz")),""),(Branch Or (Leaf "foo") (Branch And (Leaf "bar") (Leaf
  "baz")),"")]
```

it parses ambiguously, which is undesirable for efficiency reasons as well as hinting that we may have done something unnatural. Both `andTree` and `orTree` functions have `brackets tree` in them, since `orTree` contains `andTree` this is where the ambiguity creeps in. To solve it we simply delete from `orTree`.

```
    orTree = andTree +++ orBranch
```

88.3 Structure Emerges

All the previous fiddling and playing has actually caused a significant portion of the structure of our final program to make its-self clear. Looking back at what was written we could quite easily extend it to add another operator, and another after that (Exercise for the reader: if it is not clear exactly how this would be done, figure it out and do it). A moments meditation suggests that we might complete this pattern and abstract it out, given an arbitrarily long list of operators

```
    operators = [(Or,"|"),(And,"+")]
```

or perhaps

```
    data Operator = Add | Mul | Exp deriving Show

    operators = [(Add,"+"),(Mul,"*"),(Exp,"^")]
```

the parser should be computed from it, nesting it (as we did manually in the past) so that parses happen correctly without ambiguity.

The seasoned haskell programmer will have already seen, in her minds eye, the following:

```
    tree = foldr (\(op,name) p ->
                   let this = p +++ do a <- p +++ brackets tree
                                       char name
                                       b <- this
                                       return (Branch op a b)
                    in this)
                 (leaf +++ brackets tree)
                 operators
```

which is then tested.

```
    *Main> readP_to_S tree "(x^e*y+w^e*z^e)"
    [(Leaf "","(x^e*y+w^e*z^e)"),(Branch Add (Branch Mul (Branch Exp (Leaf "x")
 (Leaf "e")) (Leaf "y")) (Branch Mul (Branch Exp (Leaf "w") (Leaf "e")) (Branch
 Exp (Leaf "z") (Leaf "e"))),"")]
```

This is a good checkpoint to pause, in summary we have distilled the embryonic parser down to the following script:

```
import Text.ParserCombinators.ReadP

brackets p = do char '('
                r <- p
                char ')'
                return r

data Operator = Add | Mul | Exp deriving Show
operators = [(Add,'+'),(Mul,'*'),(Exp,'^')]

data Tree = Branch Operator Tree Tree | Leaf String deriving Show

leaf = do s <- many1 (choice (map char ['a'..'z']))
          return (Leaf s)

tree = foldr (\(op,name) p ->
               let this = p +++ do a <- p +++ brackets tree
                                   char name
                                   b <- this
                                   return (Branch op a b)
                in this)
             (leaf +++ brackets tree)
             operators
```

88.4 Whitespace and applicative notation

Since both the functional/applicative notation and ignoring whitespace depend on some of the same characters (space characters) it is a useful question to ask which should be implemented first, or whether it is not important which should be programmed first.

Considering the expression "f x", suggests that we should find how to parse whitespace before handling applicative notation, since once it has been dealt with function application should just correspond to simple juxtaposition (as intended).

There is a technical difficultly making our current parser ignore whitespace: if we were to make a `skipWhitespace` parser, and put it everywhere that whitespace could occur we would be inundated with ambiguous parses. Hence it is necessary to skip whitespace only in certain crucial places, for example we could pick the convention that whitespace is always skipped *before* reading a token. Then " a + b * c " would be seen by the parser chunked in the following way "[a][+][b][*][c][ ]". Which convention we choose is arbitrary, but ignoring whitespace before seems slightly neater, since it handles " a" without any complaints.

We define the following:

```
skipWhitespace = do many (choice (map char [' ','\n']))
                    return ()
```

and update all the parses written before, so that they follow the new convention

```
brackets p = do skipWhitespace
                char '('
                r <- p
                skipWhitespace
                char ')'
                return r

leaf = do skipWhitespace
          s <- many1 (choice (map char ['a'..'z']))
          return (Leaf s)

tree = foldr (\(op,name) p ->
               let this = p +++ do a <- p +++ brackets tree
                                   skipWhitespace
                                   char name
                                   b <- this
                                   return (Branch op a b)
                in this)
             (leaf +++ brackets tree)
             operators
```

In order to add applicative support clearly the syntax needs to allow for it:

```
   data Tree = Apply Tree Tree | Branch Operator Tree Tree | Leaf String
deriving Show
```

This syntax tree will allow for sentences such as "(x + y) foo", while this not correct other sentences like "(f . g) x" are commonplace in haskell - it should be the job of the type-checker to decide which is meaningful and which is not: This separation of concerns lets our problem (parsing) remain simple and homogeneous.

Our parser is essentially just two functions `leaf` and `tree` (`skipWhitespace` and `brackets` being considered "library" or helper functions). The function `tree` eats up all the operators it can, attaching leaves onto them as it can. While the `leaf` function could be thought of as reading in anything which doesn't have operators in it. Given this view of the program it is clear that to support applicative notation one needs to replace leaf with something that parses a chain of functional applications.

The obvious thing to try is then,

```
   leaf = chainl1 (do skipWhitespace
                      s <- many1 (choice (map char ['a'..'z']))
                      return (Leaf s))
                  (return Apply)
```

and it is easily extended to support the "commonplace" compound sentences discussed earlier:

```
   leaf = chainl1 (brackets tree
                   +++ do skipWhitespace
                          s <- many1 (choice (map char ['a'..'z']))
                          return (Leaf s))
                  (return Apply)
```

This is the problem completely solved! Our original goal is completed, one only needs to specify the operators they would like to have (in order) and write a traversal function converts the `Tree` into say mathematical expressions -- giving errors if unknown functions were used etc.

88.4.1 Making it Modular

The algorithms written are general enough to be useful in different circumstances, and even if they only had a single use -- if we were planning on using them in a larger program it is essential that we isolate the internals from the extenals (its interface).

```
module Parser
 ( Tree(..), parseExpression
 ) where

import Data.Maybe
import Text.ParserCombinators.ReadP

skipWhitespace = do many (choice (map char [' ','\n']))
                    return ()

brackets p = do skipWhitespace
                char '('
                r <- p
                skipWhitespace
                char ')'
                return r

data Tree op = Apply (Tree op) (Tree op) | Branch op (Tree op) (Tree op) | Leaf
String deriving Show

parseExpression operators = listToMaybe . map fst . filter (null .snd) .
readP_to_S tree where
 leaf = chainl1 (brackets tree
                 +++ do skipWhitespace
                        s <- many1 (choice (map char ['a'..'z']))
                        return (Leaf s))
                (return Apply)
 tree = foldr (\(op,name) p ->
                let this = p +++ do a <- p +++ brackets tree
                                    skipWhitespace
                                    char name
                                    b <- this
                                    return (Branch op a b)
                 in this)
             (leaf +++ brackets tree)
              operators
```

89 Contributors

Edits	User
1	Abdelazer[1]
6	Acangiano[2]
1	AdRiley[3]
11	Adrignola[4]
3	Albmont[5]
5	AllenZh[6]
20	Amire80[7]
195	Apfelmus[8]
1	AugPi[9]
22	Avicennasis[10]
10	BCW[11]
2	Bartosz[12]
14	BiT[13]
1	Billinghurst[14]
14	Byorgey[15]
11	Catamorphism[16]
1	CommonsDelinker[17]
255	DavidHouse[18]
58	Digichoron[19]
16	Dirk Hünniger[20]
760	Duplode[21]

[1] http://en.wikibooks.org/wiki/User:Abdelazer
[2] http://en.wikibooks.org/wiki/User:Acangiano
[3] http://en.wikibooks.org/wiki/User:AdRiley
[4] http://en.wikibooks.org/wiki/User:Adrignola
[5] http://en.wikibooks.org/wiki/User:Albmont
[6] http://en.wikibooks.org/wiki/User:AllenZh
[7] http://en.wikibooks.org/wiki/User:Amire80
[8] http://en.wikibooks.org/wiki/User:Apfelmus
[9] http://en.wikibooks.org/wiki/User:AugPi
[10] http://en.wikibooks.org/wiki/User:Avicennasis
[11] http://en.wikibooks.org/wiki/User:BCW
[12] http://en.wikibooks.org/wiki/User:Bartosz
[13] http://en.wikibooks.org/wiki/User:BiT
[14] http://en.wikibooks.org/wiki/User:Billinghurst
[15] http://en.wikibooks.org/wiki/User:Byorgey
[16] http://en.wikibooks.org/wiki/User:Catamorphism
[17] http://en.wikibooks.org/wiki/User:CommonsDelinker
[18] http://en.wikibooks.org/wiki/User:DavidHouse
[19] http://en.wikibooks.org/wiki/User:Digichoron
[20] http://en.wikibooks.org/wiki/User:Dirk_H%25C3%25BCnniger
[21] http://en.wikibooks.org/wiki/User:Duplode

2	EvanCarroll[22]
3	Goldenburg111[23]
1	Gracenotes[24]
52	Gwern[25]
2	Hairy Dude[26]
1	HethrirBot[27]
2	HowardBGolden[28]
12	Ihope127[29]
4	Jdgilbey[30]
38	Jguk[31]
3	Ketil[32]
934	Kowey[33]
1	LokiClock[34]
3	Marky1991[35]
17	Marudubshinki[36]
9	Msouth[37]
8	Nattfodd[38]
18	Panic2k4[39]
4	Physis[40]
5	Pi zero[41]
1	Pingveno[42]
5	QuiteUnusual[43]
2	Qwertyus[44]
1	Ravichandar84[45]
1	Recent Runes[46]

[22] `http://en.wikibooks.org/wiki/User:EvanCarroll`
[23] `http://en.wikibooks.org/wiki/User:Goldenburg111`
[24] `http://en.wikibooks.org/wiki/User:Gracenotes`
[25] `http://en.wikibooks.org/wiki/User:Gwern`
[26] `http://en.wikibooks.org/wiki/User:Hairy_Dude`
[27] `http://en.wikibooks.org/wiki/User:HethrirBot`
[28] `http://en.wikibooks.org/wiki/User:HowardBGolden`
[29] `http://en.wikibooks.org/wiki/User:Ihope127`
[30] `http://en.wikibooks.org/wiki/User:Jdgilbey`
[31] `http://en.wikibooks.org/wiki/User:Jguk`
[32] `http://en.wikibooks.org/wiki/User:Ketil`
[33] `http://en.wikibooks.org/wiki/User:Kowey`
[34] `http://en.wikibooks.org/wiki/User:LokiClock`
[35] `http://en.wikibooks.org/wiki/User:Marky1991`
[36] `http://en.wikibooks.org/wiki/User:Marudubshinki`
[37] `http://en.wikibooks.org/wiki/User:Msouth`
[38] `http://en.wikibooks.org/wiki/User:Nattfodd`
[39] `http://en.wikibooks.org/wiki/User:Panic2k4`
[40] `http://en.wikibooks.org/wiki/User:Physis`
[41] `http://en.wikibooks.org/wiki/User:Pi_zero`
[42] `http://en.wikibooks.org/wiki/User:Pingveno`
[43] `http://en.wikibooks.org/wiki/User:QuiteUnusual`
[44] `http://en.wikibooks.org/wiki/User:Qwertyus`
[45] `http://en.wikibooks.org/wiki/User:Ravichandar84`
[46] `http://en.wikibooks.org/wiki/User:Recent_Runes`

1	Robert Matthews[47]
1	Ruud Koot[48]
1	Sebastian Goll[49]
5	Snarius[50]
1	Snowolf[51]
1	Stereotype441[52]
4	Stuhacking[53]
7	Stw[54]
1	Swift[55]
62	Tchakkazulu[56]
10	Toby Bartels[57]
4	Van der Hoorn[58]
1	Whym[59]
1	Will48[60]
16	WillNess[61]
1	Withinfocus[62]

[47] http://en.wikibooks.org/wiki/User:Robert_Matthews

[48] http://en.wikibooks.org/wiki/User:Ruud_Koot

[49] http://en.wikibooks.org/wiki/User:Sebastian_Goll

[50] http://en.wikibooks.org/wiki/User:Snarius

[51] http://en.wikibooks.org/wiki/User:Snowolf

[52] http://en.wikibooks.org/wiki/User:Stereotype441

[53] http://en.wikibooks.org/wiki/User:Stuhacking

[54] http://en.wikibooks.org/wiki/User:Stw

[55] http://en.wikibooks.org/wiki/User:Swift

[56] http://en.wikibooks.org/wiki/User:Tchakkazulu

[57] http://en.wikibooks.org/wiki/User:Toby_Bartels

[58] http://en.wikibooks.org/wiki/User:Van_der_Hoorn

[59] http://en.wikibooks.org/wiki/User:Whym

[60] http://en.wikibooks.org/wiki/User:Will48

[61] http://en.wikibooks.org/wiki/User:WillNess

[62] http://en.wikibooks.org/wiki/User:Withinfocus

List of Figures

- GFDL: Gnu Free Documentation License. http://www.gnu.org/licenses/fdl.html
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- GPL: GNU General Public License. http://www.gnu.org/licenses/gpl-2.0.txt
- LGPL: GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. http://artlibre.org/licence/lal/de
- CFR: Copyright free use.

- EPL: Eclipse Public License. `http://www.eclipse.org/org/documents/epl-v10.php`

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses[63]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

[63] Chapter 90 on page 593

1	Dirk Hünniger	
2	Randall Britten[64], Randall Britten[65]	CC-BY-SA-3.0
3	Gaz[66] at en.wikipedia[67]	CC-BY-SA-3.0
4	Cyberpunk, Drilnoth, Emijrpbot, Hazard-Bot, Juiced lemon	
5	Emijrpbot, Hazard-Bot, Kowey	
6	Emijrpbot, Hazard-Bot, Kowey	
7	Emijrpbot, Hazard-Bot, Kowey	
8	Emijrpbot, Hazard-Bot, Kowey	
9	Emijrpbot, Hazard-Bot, Kowey	
10	Emijrpbot, Hazard-Bot, Kowey	
11	Emijrpbot, Hazard-Bot, Kowey	
12	Emijrpbot, Hazard-Bot, Kowey	
13	Emijrpbot, Hazard-Bot, Kowey	
14	Emijrpbot, Hazard-Bot, Kowey	
15	Emijrpbot, Hazard-Bot, Kowey	
16	Emijrpbot, Hazard-Bot, Kowey	
17	Emijrpbot, Hazard-Bot, Kowey	
18	Adrignola, Apfelmus	
19	Adrignola, Apfelmus	
20	Adrignola, Apfelmus	
21	Adrignola, Apfelmus	
22	Adrignola, Apfelmus	
23	Adrignola, Apfelmus	
24	Adrignola, Apfelmus	
25	en:Apfelmus[68]	CC-BY-SA-2.5
26	Adrignola, Apfelmus	
27	Adrignola, Apfelmus	
28	DavidHouse	
29	DavidHouse	
30	DavidHouse	
31	DavidHouse	
32	Svick[69], Svick[70]	PD
33	DavidHouse	
34	DavidHouse	
35	DavidHouse	
36	DavidHouse	
37	DavidHouse	
38	DavidHouse	
39	DavidHouse[71], DavidHouse[72]	GFDL

[64] http:////commons.wikimedia.org/w/index.php?title=User:Randallbritten&amp;action=edit&amp;redlink=1

[65] http:///w/index.php?title=User:Randallbritten&amp;action=edit&amp;redlink=1

[66] http:////en.wikipedia.org/wiki/User:Gaz

[67] http://en.wikipedia.org

[68] http:////en.wikibooks.org/wiki/en:Apfelmus

[69] http:////commons.wikimedia.org/wiki/User:Svick

[70] http:///wiki/User:Svick

[71] http:////commons.wikimedia.org/w/index.php?title=User:DavidHouse&amp;action=edit&amp;redlink=1

[72] http:///w/index.php?title=User:DavidHouse&amp;action=edit&amp;redlink=1

40	DavidHouse[73], DavidHouse[74]	GFDL
41	Tchakkazulu	
42	Tchakkazulu	
43	Tchakkazulu	
44	Tchakkazulu	
45	Tchakkazulu	
46	Tchakkazulu	
47	Tchakkazulu	

[73] `http:////commons.wikimedia.org/w/index.php?title=User:DavidHouse&amp;action=edit&amp;redlink=1`

[74] `http:///w/index.php?title=User:DavidHouse&amp;action=edit&amp;redlink=1`

90 Licenses

90.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to “keep intact all notices”. * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's “contributor version”.

A contributor's “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'. This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

90.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

90.3 GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.