Yet Another Haskell Tutorial

Hal Daumé III, et. al.

October 10, 2002

About This Report

The goal of the *Yet Another Haskell Tutorial* is to provide a complete intoduction to the Haskell programming language. It assumes no knowledge of the Haskell language or familiarity with functional programming in general. However, general familiarity with programming concepts (such as algorithms) will be helpful. This is not intended to be an introduction to programming in general; rather, to programming in Haskell. Sufficient familiarity with your operating system and a text editor is also necessary (this report only discusses installation on configuration on Windows and *Nix system; other operating systems may be supported – consult the documentation of your chosen compiler for more information on installing on other platforms).

What is Haskell?

Haskell is called a lazy, pure functional programming language. It is called *lazy* because expressions which are not needed to determine the answer to a problem are not evaluated. The opposize of lazy is *strict*, which is the evaluation strategry of most common programming languages (C, C++, Java, even ML). A strict language is one in which every expression is evaluated, whether the result of its computation is important or note. (This is probably not entirely true as optimizing compilers for strict languages often do what's called "dead code elmination" – this removes unused expressions from the program.) It is called *pure* because it does not allow side effects (A side effect is something that affects the "state" of the world. For instance, a function that prints something to the screen is said to be side-effecting, as is a function which affects the value of a global variable.) – of course, a programming language without side effects would be horribly useless; Haskell uses a system of *monads* to isolate all impure computations from the rest of the program and perform them in the safe way (see Section 9 for a discussion of monads proper or Section 5 for how to do input/output in a pure language).

Haskell is called a *functional* language because the evaluation of a program is equivalent to evaluating a function in the pure mathematical sense. This also differs from standard languages (like C and Java) which evaluate a sequence of statements, one after the other (this is termed an *imperative* langauge).

The History of Haskell

The history of Haskell is best described using the words of the authors. The following text is quoted from the published version of the Haskell 98 Report:

> In September of 1987 a meeting was held at the conference on Functional Programming Languages and Computer Architecture (FPCA '87) in Portland, Oregon, to discuss an unfortunate situation in the functional programming community: there had come into being more than a dozen non-strict, purely functional programming languages, all similar in expressive power and semantic underpinnings. There was a strong consensus at this meeting that more widespread use of this class of functional languages was being hampered by the lack of a common language. It was decided that a committee should be formed to design such a language, providing faster communication of new ideas, a stable foundation for real applications development, and a vehicle through which others would be encouraged to use functional languages. This document describes the result of that committee's efforts: a purely functional programming language called Haskell, named after the logician Haskell B. Curry whose work provides the logical basis for much of ours.
>
> The committee's primary goal was to design a language that satisfied these constraints:
>
> 1. It should be suitable for teaching, research, and applications, including building large systems.
> 2. It should be completely described via the publication of a formal syntax and semantics.
> 3. It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
> 4. It should be based on ideas that enjoy a wide consensus.
> 5. It should reduce unnecessary diversity in functional programming languages.
>
> The committee intended that Haskell would serve as a basis for future research in language design, and hoped that extensions or variants of the language would appear, incorporating experimental features.
>
> Haskell has indeed evolved continuously since its original publication. By the middle of 1997, there had been four iterations of the language design (the latest at that point being Haskell 1.4). At the 1997 Haskell Workshop in Amsterdam, it was decided that a stable variant of Haskell was needed; this stable language is the subject of this Report, and is called "Haskell 98".

> Haskell 98 was conceived as a relatively minor tidy-up of Haskell 1.4, making some simplifications, and removing some pitfalls for the unwary. It is intended to be a "stable" language in sense the *implementors are committed to supporting Haskell 98 exactly as specified, for the foreseeable future.*
>
> The original Haskell Report covered only the language, together with a standard library called the Prelude. By the time Haskell 98 was stabilised, it had become clear that many programs need access to a larger set of library functions (notably concerning input/output and simple interaction with the operating system). If these program were to be portable, a set of libraries would have to be standardised too. A separate effort was therefore begun by a distinct (but overlapping) committee to fix the Haskell 98 Libraries.

Why Use Haskell?

Clearly you're interested in Haskell since you're reading this tutorial. There are many motivations for using Haskell. My personal reason for using Haskell is that I have found that I write more bug-free code in less time using Haskell than any other language. I also find it very readable and extensible.

Perhaps most importantly, however, I have consistently found the Haskell community to be incredibly helpful. The language is constantly evolving (That's not to say it's instable; rather that there are numerous extensions that have been added to some compilers which I find very useful.) and user suggestions are often heeded when new extensions are to be implemented.

Why Not Use Haskell?

My two biggest complaints, and the complaints of most Haskellers I know, are: (1) the generated code tends to be slower than equivalent programs written in a language like C; and (2) it tends to be difficult to debug.

The second problem tends not be to a very big issue: most of the code I've written is not buggy, as most of the common sources of bugs in other languages simply don't exist in Haskell. The first issue certainly has come up a few times in my experience; however, CPU time is almost always cheaper than programmer time and if I have to wait a little longer for my results after having saved a few days programming and debugging.

Of course, this isn't the case of all applications. Some people may find that the speed hit taken for using Haskell is unbearable. However, Haskell has many *foreign-function interfaces* which allow you to link in code written in other languages, for when you need to get the most speed out of your code. If you don't find this sufficient, I would suggest taking a look at the language O'Caml, which often *out-performs* even C, yet also has many of the benefits of Haskell.

Target Audience

There have been many books and tutorials written about Haskell; for a (nearly) complete list, visit the `http://haskell.org/bookshelf` (Haskell Bookshelf) at the Haskell homepage. A brief survey of the tutorials available yields:

- *A Gentle Introduction to Haskell* is an introduction to Haskell, given that the reader is familiar with functional programming en large.
- *Haskell Companion* is a short reference of common concepts and definitions.
- *Online Haskell Course* is a short course (in German) for beginning with Haskell.
- *Two Dozen Short Lessons in Haskell* is the draft of an excellent textbook that emphasizes user involvement.
- *Haskell Tutorial* is based on a course given at the 3rd International Summer School on Advanced Functional Programming.
- *Haskell for Miranda Programmers* assumes knowledge of the language Miranda.
- *PLEAC-Haskell* is a tutorial in the style of the Perl Cookbook.

Though all of these tutorials is excellent, they are on their own incomplete: The "Gentle Introduction" is far too advanced for beginning Haskellers and the others tend to end too early, or not cover everything. Haskell is full of pitfalls for new programmers and experienced non-functional programmers alike, as can be witnessed by reading through the archives of the Haskell mailing list.

It became clear that there is a strong need for a tutorial which is introductory in the sense that it does not assume knowledge of functional programming, but which is advanced in the sense that it *does* assume some background in programming. This tutorial is not for beginning programmers; some experience and knowledge of programming and computers is assumed (though the appendix does contain some background information).

The Haskell language underwent a standardization process and the result is called Haskell 98. The majority of this book will cover the Haskell 98 standard. Any deviations from the standard will be noted (for instance, many compilers offer certain extensions to the standard which are useful; some of these may be discussed).

The goals of this tutorial are:

- to be *practical* above all else
- to provide a comprehensive, free introduction to the Haskell language
- to point out common pitfalls and their solutions

- to provide a good sense of how Haskell can be used in the real world

In order to emphasize real world applicability we center this tutorial around solving a very complicated problem. This problem was the subject of the ICFP (International Conference on Functional Programming) Contest in 2002. This contest runs for 72 hours and invites programmers to submit solutions to a problem using their programming language of choice. Unfortunately, the winners in 2002 did not use Haskell (they used C; the runners up used O'Caml).

We will use a slighly simplified version of the problem here, but the basic idea is the same:

You have a robot who begins at some position on a map. There are packages spread across the map (which has walls and rivers) and goal locations. The robot is supposed to pick up the packages and deliver them to a goal. Each package has a certain weight and the robot himself has a certain maximum carrying capacity.

When multiple robots are competing, it is possible for one to bump into another other. If a robot is pushed into a river, it dies. In order to control which robot gets precedence (i.e., which moves first), at each step a robot bids either a positive or negative value. The robot who bids the most goes first, and so on. You must bid a non-zero value for each move and once you run out of "money" you cannot move any more.

For each package delivery you make, you earn points equal to the weight of the packages you deliver. The robot with the most points at the end of a game wins.

We will get more specific about some of the rules as the tutorial moves on, but you will see this example popping up throughout the tutorial, especially in the exercises.

Acknowledgements

This tutorial is the combined effort of a whole host of people. I would like to especially thank ..., who served as chapter editors, ..., who wrote sections of this tutorial, ... who helped proofread and test the code in this tutorial, and finally ... for "betatesting" this document and providing useful feedback.

It would be inappropriate not to give credit also to the original designers of Haskell. Those are: Arvind, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Jon Fairbairn, Joseph Fasel, Andy Gordon, Maria Guzman, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Mark Jones, Dick Kieburtz, John Launchbury, Erik Meijer, Rishiyur Nikhil, John Peterson, Simon Peyton Jones, Mike Reeve, Alastair Reid, Colin Runciman, Philip Wadler, David Wise, Jonathan Young.

Finally, I would like to specifically thank Simon Peyton Jones, Simon Marlow, John Hughes, Alastair Reid, Koen Classen, Manuel Chakravarty, Sigbjorn Finne and Sven Panne, all of whom have made my life learning Haskell all the more enjoyable by always being supportive. There were doubtless others who helped and are not listed, but these are those who come to mind.

- Hal Daumé III

Contents

Chapter 1

Introduction

This tutorial contains a whole host of example code, all of which should have been included in its distribution. If not, please refer to the links off of the Haskell web site (`haskell.org`) to get it. This book is formatted to make example code stand out from the rest of the text.

```
Code will look like this.
```

Occasionally, we will refer to interaction betwen you and the operating system and/or the interactive shell (more on this in Section 2).

```
Interaction will look like this.
```

Strewn throughout the tutorial, we will often make additional notes to something written. These are often for making comparisons to other programming languages or adding helpful information.

> ■ NOTE ■ Notes will appear like this.

If we're covering a difficult or confusing topic and there is something you should watch out for, we will place a warning.

> ■ WARNING ■ Warnings will appear like this.

Finally, we will sometimes make reference to built-in functions (so-called Prelude-functions). This will look something like this:

$$map :: (a- > b)- > [a]- > [b]$$

Chapter 2

Getting Started

There are three well known Haskell system: Hugs, GHC and NHC. Hugs is exclusively an interpreter, meaning that you cannot compile stand-alone programs with it, but can test and debug programs in an interactive environment. GHC is both an interpreter (like Hugs) and a compiler which will produce stand-alone programs. NHC is exclusively a compiler. Which you use is entirely up to you. I've tried to make a list of some of the differences in the following list but of course this is far from exhaustive:

Hugs - very fast; implements almost all of Haskell 98 (the standard) and most extensions; built-in support for module browsing; cannot create stand-alones; written in C; works on almost every platform; build in graphics library.

GHC - interactive environment is slower than Hugs, but allows function definitions in the environment (in Hugs you have to put them in a file); implements all of Haskell 98 and extensions; good support for interfacing with other languages; in a sense the "de facto" standard.

NHC - less used and no interactive environment, but produces smaller and often faster executables than does GHC; supports Haskell 98 and some extensions.

I, personally, have all of them installed and use them for different purposes. I tend to use GHC to compile (primarily because I'm most familiar with it) and the Hugs interactive environment, since it is much faster. As such, this is what I would suggest. However, that is a fair amount to download an install, so if you had to go with just one, I'd get GHC, since it contains both a compiler and interactive environment.

Following is a descrition of how to download and install each of this as of the time this tutorial was written. It may have changed – see `http://haskell.org` (the Haskell website) for up-to-date information.

2.1 Hugs

Hugs supports almost all of the Haskell 98 standard (it lacks some of the libraries), as well as a number of advanced/experimental extensions, including: multi-parameter type classes, extensible records, rank-2 polymorphism, existentials, scoped type variables, and restricted type synonyms.

2.1.1 Where to get it

The official Hugs web page is at `http://haskell.org/hugs` (http://haskell.org/hugs). If you go there, there is a link titled "downloading" which will send you to the download page. From that page, you can download the appropriate version of Hugs for your computer.

2.1.2 Installation procedures

Once you've downloaded Hugs, installation differs depending on your platform, however, installation for Hugs is more of less identical to installation for any program on your platform.

For Windows when you click on the "msi" file to download, simply choose "Run This Program" and the installation will begin automatically. From there, just follow the on-screen instructions.

For RPMs use whatever RPM installation program you know best.

For source first gunzip the file, then untar it. Presumably if you're using a system which isn't otherwise supported, you know enough about your system to be able to run configure scripts and make things by hand.

2.1.3 How to run it

On Unix machines, the Hugs interpreter is usually started with a command line of the form: hugs [option — file] ...

On Windows , Hugs may be started by selecting it from the start menu or by double clicking on a file with the .hs or .lhs extension. (This manual assumes that Hugs has already been successfully installed on your system.)

Hugs uses options to set system parameters. These options are distinguished by a leading + or - and are used to customize the behaviour of the interpreter. When Hugs starts, the interpreter performs the following tasks:

- Options in the environment are processed. The variable HUGSFLAGS holds these options. On Windows 95/NT, the registry is also queried for Hugs option settings.
- Command line options are processed.

- Internal data structures are initialized. In particular, the heap is initialized, and its size is fixed at this point; if you want to run the interpreter with a heap size other than the default, then this must be specified using options on the command line, in the environment or in the registry.
- The prelude file is loaded. The interpreter will look for the prelude file on the path specified by the -P option. If the prelude, located in the file Prelude.hs, cannot be found in one of the path directories or in the current directory, then Hugs will terminate; Hugs will not run without the prelude file.
- Program files specified on the command line are loaded. The effect of a command hugs f1 ... fn is the same as starting up Hugs with the hugs command and then typing :load f1 ... fn. In particular, the interpreter will not terminate if a problem occurs while it is trying to load one of the specified files, but it will abort the attempted load command.

The environment variables and command line options used by Hugs are described in the following sections.

2.1.4 Program options

To list all of the options would take too much space. The most important option at this point is "+98" or "-98". When you start hugs with "+98" it is in Haskell 98 mode, which turns off all extensions. When you start in "-98", you are in Hugs mode and all extensions are turned on. If you've downloaded someone elses code and you're having trouble loading it, first make sure you have the "98" flag set properly.

Further information on the Hugs options is in the manual `http://cvs.haskell.org/Hugs/pages/hugsman/` (the Hugs getting-started manual).

2.1.5 How to get help

To get Hugs specific help, go to the Hugs web page. To get general Haskell help, go to the Haskell web page.

2.2 Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC) is a robust, fully-featured, optimising compiler and interactive environment for Haskell 98, GHC compiles Haskell to either native code or C. It implements numerous experimental language extensions to Haskell 98; for example: concurrency, a foreign language interface, multi-parameter type classes, scoped type variables, existential and universal quantification, unboxed types, exceptions, weak pointers, and so on. GHC comes with a generational garbage collector, and a space and time profiler.

2.2.1 Where to get it

Go to the official GHC web page `http://haskell.org/ghc` (GHC) to download the latest release. The current version as of the writing of this tutorial is 5.02.2 and can be downloaded off of the GHC download page (follow the "Download" link). From that page, you can download the appropriate version of GHC for your computer.

2.2.2 Installation procedures

Once you've downloaded GHC, installation differs depending on your platform; however, installation for GHC is more of less identical to installation for any program on your platform.

For Windows when you click on the "msi" file to download, simply choose "Run This Program" and the installation will begin automatically. From there, just follow the on-screen instructions.

For RPMs use whatever RPM installation program you know best.

For source first gunzip the file, then untar it. Presumably if you're using a system which isn't otherwise supported, you know enough about your system to be able to run configure scripts and make things by hand.

For a more detailed description of the installation procedure, look at the GHC users manual under "Installing GHC".

2.2.3 How to run the compiler

Running the compiler is fairly easy. Assuming that you have a program written with a `main` function in a file called `Main.hs`, you can compile it simply by writing:

```
% ghc --make Main.hs -o main
```

The "–make" option tells GHC that this is a program and not just a library and you want to build it and all modules it depends on. "Main.hs" stipulates the name of the file to compile; and the "-o main" means that you want to put the output in a file called "main".

> ■ NOTE ■ In Windows, you should say "-o main.exe" to tell Windows that this is an executable file.

You can then run the program by simply typing "main" at the prompt.

2.2.4 How to run the interpreter

GHCi is invoked with the command "ghci" or "ghc –interactive". One or more modules or filenames can also be specified on the command line; this instructs GHCi to load the specified modules or filenames (and all the modules they depend on), just as if you had said :load modules at the GHCi prompt.

2.2.5 Program options

To list all of the options would take too much space. The most important option at this point is "-fglasgow-exts". When you start GHCi without "-fglasgow-exts" it is in Haskell 98 mode, which turns off all extensions. When you start with "-fglasgow-exts", all extensions are turned on. If you've downloaded someone elses code and you're having trouble loading it, first make sure you have this flag set properly.

Further information on the GHC and GHCi options are in the manual off of the GHC web page.

2.2.6 How to get help

To get GHC(i) specific help, go to the GHC web page. To get general Haskell help, go to the Haskell web page.

2.3 NHC

About NHC ...

2.3.1 Where to get it

2.3.2 Installation procedures

2.3.3 How to run it

2.3.4 Program options

2.3.5 How to get help

2.4 Editors

With good text editor, programming is fun. Of course, you can get along with simplistic editor capable of just cut-n-paste, but good editor is capable of doing most of the chores for you, letting you concentrate on what you are writing. With respect to programming in Haskell, good text editor should have as much as possible of the following features:

- Syntax highlighting for source files

- Indentation of source files
- Interaction with Haskell interpreter (be it Hugs or GHCi)
- Computer-aided code navigation
- Code completion

At the time of writing, several options were available: Emacs/XEmacs support Haskell via *haskell-mode* and accompanying Elist code (available from `http://www.haskell.org/haskell-mode` ()), and . . . .

What's else available? . . .

(X)Emacs seem to do the best job, having all the features listed above. Indentation is aware about Haskell's 2-dimensional layout rules (see Section 7.8, very smart and have to be seen in action to be believed. You can quickly jump to the definition of chosen function with the help of "Definitions" menu, and name of the currently edited function is always displayed in the modeline.

Chapter 3

Language Basics

The purpose of this section is to get you familiar with using the interactive environments and show you have to compile a basic program, as well as get you familiar with the Haskell syntax, which will probably seem a bit alien at first if you're used to languages like C and Java. However, it should hopefully clear up enough that we can more quickly on to more advanced issues.

Before we get started there are a few things which deserve mention, so you are not led to confusion. Haskell is *lazy*, which means that no computation takes place unless it is forced to take place. This means, for instance, that you can create "infinitely" large data structures and use them, and your program will still not run indefinitely (of course it's equally possible that it will run indefinitely, but it's not guarenteed). For instance, using imperative-esque psuedo-code, we could create an infinite list containing only the number 1 in C/C++/Java by doing something like:

```
List makeList()
{
  List current = new List();
  current.value = 1;
  current.next = makeList();
  return current;
}
```

By looking at this code, we can see what it's trying to do: it creates a new list, sets its value to 1 and then recursively calls itself to make the rest of the list. Of course, if you actually wrote this code, the program would never terminate, because `makeList` would keep calling itself ad infinitum. The equivalent code in Haskell is:

```
makeList = 1 : makeList
```

Which basically reads: the value of `makeList` is an element of value 1 appended onto the beginning of `makeList` (which, in turn, is an element of value

1 appened onto the beginning of ...). Now, if you attempt to, say, write `makeList` to a file, or print it to the screen, or calculate the sum of its elements, the operation won't terminate because it would have to evaluate an infinitely long list. However, if you simply use a finite portion of the list, say, the first 10 elements, the fact that the list is infinitely long doesn't matter. If you only use the first 10 elements, only the first 10 elements are calculated.

Secondly, there is an issue about which I've heard some complaint (and an issue which actually bothered me when I first started with Haskell). Haskell distinguishes between *values* (things which have a value in the "real world", numbers $(1, 2, 3, \ldots)$, strings ("abc", "hello", ...), characters ('a', 'b', ' ', ...), even functions (for instance, the function which squares a value, or the square-root function)) and *types* (the categories to which values belong). This is not unusual. However, Haskell *requires* that the names given to values begin with a lower-case letter and the names given to types begin with an upper-case letter. For instance `f` or `mult` or `int` are *always* values and not types, while `A` and `Add` and `RED` are always types and not values (this is a slight simplification, but will do for now). The moral is, if your otherwise correct program won't compile, be sure you haven't violated this.

Being a functional language, Haskell eschews side effects. This means that if you're used to writing code in an imperative language (like C or Java), you're going to have to start thinking differently. Most importantly, if you have a value `x`, you must not think about `x` as a register or a memory location or anything like that. `x` is simply a name, just as "Hal" is a name. You cannot arbitrarily decide to store a different person in my name any more than you can arbitrarily decide to store a different value in `x`. This means that code which might look like the following C code is invalid (not only is it invalid, but it has no counterpart) in Haskell:

```
int x = 5;
x = x + 1;
```

Such a call `x = x + 1` is called *destructive update* because we are destroying whatever was in `x` before and replacing it with a new value. Destructive update does not happen in Haskell.

By not allowing destructive updates (or any other such side-effecting operations), Haskell code is very easy to reason about. That is, when we define a function `f` and call that function with a particular argument `a` in the beginning of a program and then, at the end of the program, again call `f` with the same argument `a`, we know we will get the same result out. This is because we know that `a` cannot have changed and because we know that `f` only depends on `a` (for instance, it didn't "increment a global counter" or anything of the sort). This property is called *referential transparency* and basically states that if two functions `f` and `g` produce the same values for the same arguments, then we may replace `f` with `g` (and vice-versa).

> ■ NOTE ■ The exact definition of referential transparency is not generally agreed upon. The one given above is the one I like best. They all carry the same interpretation; the differences lie in how they are formalized.

3.1 Arithmetic

Let's begin with arithmetic. Start up your favorite interactive shell (Hugs or GHCi; see section 2 for installation instructions). The shell will output to the screen a few lines talking about itself and what it's doing and then should finish with the cursor on a line reading:

```
Prelude>
```

From here, you can begin to evaluate *expressions*. An expression is basically something which has a value. For instance, the number 5 is an expression (its value is 5). Values can be built up from other values; for instance, $5 + 6$ is an expression (its value is 11). In fact, most simple arithmetic operations are supported by Haskell, including plus (**+**), minus (**-**), times (*****), divided-by (**/**), exponentiation (**^**) and square-root (**sqrt**). You can experiment with these by asking the interactive shell to evaluate expressions and give you their value. In this way, a Haskell shell can be used as a powerful calculator. Try some of the following:

```
Prelude> 5*4+3
23
Prelude> 5^5-2
3123
Prelude> sqrt 2
1.4142135623730951
Prelude> 5*(4+3)
35
```

We can see that in addition to the standard arithmetic operations, Haskell also allows grouping by parentheses, hence the difference between the values of **5*4+3** and **5*(4+3)**. The reason for this is that the "understood" grouping of the first expression is **(5*4)+3**, due to something called *operator precedence*. What this basically means is that the times operator takes precedence over the plus operator. Precedences are defined in order to reduce the number of paretheses needed to write an unambiguous expression.

> ■ WARNING ■ Even though parentheses are not always needed, due to precedence, sometimes it is better to leave them in anyway; afterall, other people will probably have to read your code and if extra parentheses make the intent of the code clearer, use them.

Now try entering **2^5000**. Does it work?

> ■ NOTE ■ If you're familiar with programming in other languages, you may find it odd that **sqrt 2** comes back with a decimal point (i.e., is a floating point number) even though the argument to the function seems to be an integer. This interchangability of numeric types is due to Haskell's system of *type classes* and will be discussed in grueling detail in section 4.2).

Exercises

Exercise 1 *We've seen that multiplication binds more tightly than division. Can you think of a way to determine whether function application binds more or less tightly than multiplication?*

3.2 Pairs, Triples and More

In addition to single values, we might want to talk about multiple values at the same time. For instance, in the robots game, we may want to refer to a position of a robot by its x/y coordinate, which would be a pair of integers. To make a pair of integers is simple: you enclose the pair in parenthesis and separate them with a comma. Try the following:

```
Prelude> (5,3)
(5,3)
```

Here, we have a pair of integers, 5 and 3. Pairs are allowed to be heterogeneous, meaning that you can store, for instance, an integer in the first position and a string in the second.

There are two predefined functions which allow you to extract the first and second elements of a pair. They are, respectively, **fst** and **snd**. You can see how they work as follows:

```
Prelude> fst (5, "hello")
5
Prelude> snd (5, "hello")
"hello"
```

In addition to pairs, you can define triples, quadruples and so on. To define a triple and a quadruple respectively, we write:

```
Prelude> (1,2,3)
(1,2,3)
Prelude> (1,2,3,4)
(1,2,3,4)
```

And so on. In general, pairs, triples, and so on are called *tuples* and can store fixed amounts of heterogeneous data.

> ■ NOTE ■ The functions **fst** and **snd** won't work on anything longer than a pair; if you try to use them you will get an error complaining that there was a type error. What this means will be explained in Section 4.

Exercises

Exercise 2 *Use a combination of* **fst** *and* **snd** *to extract the character out of the tuple ((1,'a'),"foo").*

3.3 Lists

The disadvantage to tuples is that they contain only a fixed number of elements. A data structure which can hold an arbitrary number of elements is a **List**. Lists are assembled in a very similar fashion to tuples, except they use square brackets instead of parentheses. We can define a list like:

```
Prelude> [1,2]
[1,2]
Prelude> [1,2,3]
[1,2,3]
```

Lists don't actually have to have any elements. The empty list is simply [].

Unlike tuples, we can very easily add an element on to the beginning of the list using the colon operator. This operator is often called the "cons" operator; the process of added an element is often called "consing". The etymology of this is that we're saying that we are "cons"tructing a new list from a new element and an old list. Be that as it may, we can add an element to the beginning of a list as follows:

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

We can actually build any list simply by using the cons operator (the colon) and the empty list:

```
Prelude> 5:1:2:3:4:[]
[5,1,2,3,4]
```

In fact, the **[5,1,2,3,4]** syntax is simply syntactic sugar for the cons expression. If we write something using the **[5,1,2,3,4]** notation, the compiler simply translates it to the other.

One further difference between lists and tuples is that while tuples are heterogeneous, lists must be homogenous. That means that you cannot have a list that holds both integers and strings. If you try to, you will get a type error.

Of course, lists don't have to just contain integers or strings; they can also contain tuples or even other lists. Tuples, similarly, can contain lists and other tuples. Try some of the following:

```
Prelude> [(1,1),(2,4),(3,9),(4,16)]
[(1,1),(2,4),(3,9),(4,16)]
Prelude> ([1,2,3,4],[5,6,7])
([1,2,3,4],[5,6,7])
```

There are two basic list functions: `head` and `tail`, which return the first element off of a (non-empty) list and all but the first element off of a (non-empty) list, respectively.

To get the length of a list, you use the `length` function:

```
Prelude> length [1,2,3,4,10]
5
Prelude> head [1,2,3,4,10]
1
Prelude> length (tail [1,2,3,4,10])
4
```

3.3.1 Strings

In Haskell, a `String` is simply a list of `Chars`. So we can create the string "Hello" as:

```
Prelude> 'H':'e':'l':'l':'o':[]
"Hello"
```

Strings can be concattenated using the `++` operator:

```
Prelude> "Hello " ++ "World"
"Hello World"
```

Additionally, non-string values can be converted to strings using the `show` function and strings can be converted to non-string values using the `read` function. Of course, if you try to read a value that's malformed, you'll get an error:

```
Prelude> "Five squared is " ++ show (square 5)
"Five squared is 25"
Prelude> read "5" + 3
8
Prelude> read "Hello" + 3
Program error: Prelude.read: no parse
```

(The exact error message is implementation dependent.)

3.3.2 Simple List Functions

Much of the computation in Haskell programs is done by processing lists. There are three primary list-processing functions: `map`, `filter` and `foldr` (also `foldl`). The `map` function takes as arguments a list of values and a function which should be applied to each of the values. For instance, there is a built-in function `toUpper` which takes as input a `Char` and produces a `Char` that is the upper-case version of the original argument. So, to covert an entire string (which is simply a list of characters) to upper case, we can map the `toUpper` function across the entire list:

```
Prelude> map toUpper "Hello World"
"HELLO WORLD"
```

When you map across a list, the length of the list never changes – only the individual values in the list change. To remove elements from the list, you can use the `filter` function. This function allows you to remove certain elements from a list. For instance, the function `isLower` tells you whether a given character is lower case. We can filter out all non-lowercase characters using this:

```
Prelude> filter isLower "Hello World"
"elloorld"
```

The function `foldr` takes a little more getting used to. `foldr` takes three arguments: a function, an initial value, and a list. The best way to think about `foldr` is that it replaces occurances of the list concatenator `:` with the function parameter and replaces the empty list constructor `[]` with the initial value. Thus, if we have a list:

```
3 : 8 : 12 : 5 : []
```

And we apply `foldr (+) 0` to it, we get:

```
3 + 8 + 12 + 5 + 0
```

Which sums the list. We can test this:

```
Prelude> foldr (+) 0 [3,8,12,5]
28
```

We can do the same sort of thing to calculate the product of all the elements on a list:

```
Prelude> foldr (*) 1 [4,8,5]
160
```

We said earlier that folding is like replacing `:` with a particular function ahd `[]` with an initial element. This raises a question as to what happens when the function isn't associative (a function $\cdot$ is associative if $a \cdot (b \cdot c) = (a \cdot b) \cdot c$). When we write $4 \cdot 8 \cdot 5 \cdot 1$, we need to specify where to put the parentheses. Namely, do we mean $((4 \cdot 8) \cdot 5) \cdot 1$ or $4 \cdot (8 \cdot ((5 \cdot 1))$? `foldr` assumes the function is right-associative (i.e., the correct bracketing is the latter). Thus, when we use it on a non-associtive function (like minus), we can see the effect:

```
Prelude> foldr (-) 1 [4,8,5]
0
```

The `foldl` function goes the other way and effectively produces the opposite bracketing. `foldl` looks the same when applied, so we could have done summing just as well with `foldl`:

```
Prelude> foldl (+) 0 [3,8,12,5]
28
```

However, we get different results when using the non-associative function minus:

```
Prelude> foldl (-) 1 [4,8,5]
-16
```

This is because `foldl` uses the opposite bracketing. The way it accomplishes this is essentially by going all the way down the list, taking the last element and combining it with the initial value via the provided function. It then takes the second-to-last element in the list and combines it to this new value. It does so until there is no more list left.

> ■ NOTE ■ `foldl` is often more efficient than `foldr` for reasons we will discuss in Section 7.6. However, `foldr` can work on infinite lists while `foldl` cannot. This is because before `foldl` does anything, it has to go to the end of the list. On the other hand, `foldr` starts producing output immediately. For instance, `foldr (:) [] [1,2,3,4,5]` simply returns the same list. Even if the list were infinite, it would produce output. A similar function using `foldl` would fail to produnce any output.

If this discussion of the folding functions doesn't entirely make sense, that's okay. We'll talk about them further in Section 7.6.

Exercises

Exercise 3 *Use* `map` *to convert a string into a list of booleans, each element in the new list representing whether the original element was a lower-case character or not. That is, it should take the string "aBCde" and return [True,False,False,True,True].*

Exercise 4 *Use the functions mentioned in this section (you will need two of them) to compute the number of lower-case letters in a string. For instance, on "aBCde" it should return 3.*

Exercise 5 *We've seen how to calculate sums and products using folding functions. Given that the function* `max` *returns the maximum of two numbers, write a function using a fold which will return the maximum value in a list (and zero if the list is empty). So, when applied to [5,10,2,8,1] it will return 10. Explain to yourself why it works.*

Exercise 6 *Write a function which takes a list of pairs of length at least 2 and returns the first component of the second element in the list. So, when provided with [(5,'b'),(1,'c'),(6,'a')], it will return 1.*

3.4 Source Code Files

Of course, as programmers, we don't want to simply evaluate small little expressions like these – we want to sit down an actually write code in our editor of choice, save it, and then use it. This couldn't be simpler. We already saw in Sections 2.2 and 2.3 how to write a Hello World program and compile it. Here, we show how to do something slightly different. To do this, create a file called `Test.hs` and enter the following code:

```
module Test
    where

x = 5
y = (6, "Hello")
z = x * fst y
```

This is a very simple "program" written in Haskell. It defines a module called "Test" (in general module names should match file names; see Section 6 for more on this). In this module, there are three definitions: x, y and z. Once you've written and saved this file, in the directory in which you saved it, load this up in your favorite interpreter, by executing either of the following:

```
% hugs Test.hs
% ghci Test.hs
```

This will start Hugs or GHCi, respectively, and load the file. Alternatively, if you already have one of them loaded, you can use the ":load" command (or just ":l") to load a module, as:

```
Prelude> :l Test.hs
...
Test>
```

Between the first and last line, some stuff will be written telling you what the interpreter is doing. If any errors crop up, you probably mistyped something; double check and then try again.

You'll notice that where it used to say "Prelude" it now says "Test". That means that Test is the current module. Which means, you're probably thinking, "Prelude" must also be a module. Exactly correct. The Prelude module (usually referred to simply as "the Prelude") is always loaded and contains the standard definitions (for instance, the **:** operator for lists, or **+** or *****, **fst**, **snd** and so on).

Now that we've loaded Test, we can use things which were defined in it. For example:

```
Test> x
5
Test> y
(6,"Hello")
Test> z
30
```

Perfect, just as we expected!

One final issue regards how to compile programs to stand-alone executables. In order for a program to be an executable, it *must* have the module name "Main" and must contain a function called "main". So, if you go in to Test.hs and rename it to "Main" (change the line that reads module Test to module Main), we simply need to add a main function. Try this:

```
main = putStrLn "Hello World"
```

Now, save the file and we can compile it (refer back to Section 2 for information on how to do this for your compiler. For example, in GHC, you would say:

```
% ghc --make Test.hs -o test
```

■ NOTE ■ For Windows, it would be "-o test.exe"

This will create a file calle "test" (or on Windows, "test.exe") which you can then run.

```
% ./test
Hello World
```

■ NOTE ■ Or, on Windows:

```
C:\> test.exe
Hello World
```

3.5 Functions

Now that we've seen how to write code in a file, we can start writing functions. As you might have expected, functions are central to Haskell, it being a *functional* language. This means that the evaluation of a program is simply the evaluation of a function. A function, in the pure mathematical, takes a value and produces a value. However, unlike programming languages where, for a given value a function will always produce the same result.

We can write a simple function to square a number and enter it into our Test.hs file. We might define this as follows:

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

In this function definition, we say that we're defining a function `square` who takes one argument (aka parameter) which we call `x`. We then say that the *value* of `square x` is equal to `x * x`.

Haskell also supports standard conditional expressions. For instance, we could define a function that returns -1 if its argument is less than 0, 0 if its argument *is* 0 and 1 if its argument is greater than 0 (this is called the signum function):

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

The if/then/else construct in Haskell is very similar to that of most other programming languages. It evaluates the condition (in this case $x < 0$ and, if this evaluates to **True**, it evaluates the *then* condition; if it evaluates to **False**, it evaluates the *else* condition.

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

Haskell, like many other languages, also supports **case** constructions. These are used when there are multiple values you want to check against (case expressions are actually quite a bit more powerful than this – see Section 7.2 for all the gory details). Suppose we wanted to define a function that had a value of 1 if its argument were 0, a value of 5 if its argument were 1, a value of 2 if its argument were 2 and a value of -1 in all other instances. Writing this function using **if** statements would be long and very unreadable; so we write it using a **case** statement as follows (we call this function `f`):

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

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

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

The general rule for layout is that an open-brace is inserted after the keywords **where**, **let**, **do** and **of**; the column position at which the next command appears is remembered. From then on, every new line which is indented the same amount automatically gets a semicolon. If a following line is indented less, a close-brace is inserted. This may sound complicated, but if you follow the general rule of indenting after each of those keywords, you'll never have to remember it (see Section 7.8 for a more complete discussion of layout).

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

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

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

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

Though why you would choose to structure your code that way is up in the air.

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

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

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

More complicated functions can be built up from simpler functions using *function composition*. Function composition is simply taking the result of the application of one function and using that as an argument to another. We've already seen this way back when we were doing arithmetic (Section 3.1), when we wrote `5*4+3`. In this, we were evaluating $5 * 4$ and then applying $+3$ to the result. We can do the same thing with our `square` and `f` functions:

```
Test> square (f 1)
25
Test> square (f 2)
4
Test> f (square 1)
5
Test> f (square 2)
-1
```

The result of each of these function applications is fairly straightforward. The parentheses around the inner function are necessary; otherwise in the first line, the interpreter would think you're trying to get the value of "square f" which has no meaning. Function application like this is fairly standard in most programming languages. There is another, more mathematically oriented, way to express function application, using the `.` (just a single period) function. This `.` function is supposed to look like the $\circ$ operator in mathematics.

> ■ NOTE ■ So, where in mathematics we might write $f \circ g$ to mean "f following g", in Haskell we write `f . g` also to mean "f following g."

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

```
Test> (square . f) 1
25
Test> (square . f) 2
4
Test> (f . square) 1
5
Test> (f . square) 2
-1
```

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

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

`sqrt`	the square root function
`id`	the identity function – `id x = x`
`fst`	extracts the first element from a pair
`snd`	extracts the second element from a pair
`null`	tells you whether or not a list is empty
`head`	returns the first element on a non-empty list
`tail`	returns everything but the first element of a non-empty list
`++`	concatenates two lists
`==`	checks to see if two things are equal
`/=`	checks to see if two things are unequal

Here, we show example usages of each of these functions:

```
Prelude> sqrt 2
1.41421
Prelude> id "hello"
"hello"
Prelude> id 5
5
Prelude> fst (5,2)
5
Prelude> snd (5,2)
2
Prelude> null []
```

```
True
Prelude> null [1,2,3,4]
False
Prelude> head [1,2,3,4]
1
Prelude> tail [1,2,3,4]
[2,3,4]
Prelude> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
Prelude> [1,2,3] == [1,2,3]
True
Prelude> 'a' /= 'b'
True
Prelude> head []

Program error: {head []}
```

We can see that applying `head` to an empty list gives an error (the exact error message depends on whether you're using GHCi or Hugs – the shown error message is from Hugs).

todo: write about infix

3.6 Comments

There are two types of comments in Haskell: line comments and block comments. Line comments begin with the token `--` and extend until the end of the line. Block comments begin with `{-` and extend to `-}`. Block comments can be nested.

> ■ NOTE ■ The `--` in Haskell corresponds to `//` in C++ or Java, and `{-` and `-}` correspond to `/*` and `*/`.

Comments are used to explain in English what is going on in your program and are completely ignored by compilers and interpreters. For example, if we have the following program:

```
module Test2
    where

main = putStrLn "Hello World"  -- write a string to the screen

{- f is a function which takes an integer and produces
another integer.  {- this is an embedded comment -} the original
comment extends to the matching end-comment token: -}
f x =
```

```
case x of
  0 -> 1    -- 0 maps to 1
  1 -> 5    -- 1 maps to 5
  2 -> 2    -- 2 maps to 2
  _ -> -1   -- everything else maps to -1
```

This example programs shows the use of both line comments and (embedded) block comments.

3.7 Interactivity

If you have read other tutorials, you might be wondering by now why you haven't seen many of the standard programs written in tutorials of other languages, like one which asks a user for his name and then says "Hi" to the user by name. The reason for this is simple. Being a pure functional language, it is not entirely clear how one should handle things like user input. After all, if the user types something one time and something else another, then you no longer have a function, since it would return two different values. The solution to this was found in the depths of category theory, a branch of formal mathematics: monads. We're not yet at all ready to talk about monads formally, but for now, think of them simply as a convenient way to express things like input/output. We'll discuss them in this context much more in Section 5 and then about monads for monads' sake in Section 9. For now, let's just see how you can write interactive programs. For this section, completely forget that you ever heard the word "monad."

Suppose we want to write a function that's interactive. The way to do this is to use the **do** keyword. This allows us to specify the order of operations (remember than normally in Haskell, since it's a lazy language, the order in which operations are evaluated is unspecified). So, to write a simple program which asks a user for his name and then address him directly, enter the following code into "Name.hs":

```
module Main
    where

import IO

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

You can then either load this code in your interpreter and execute "main" by simply typing "main" or you can compile it and run it from the command line. I'll show the results of the interactive approach:

```
Main> main
Please enter your name:
Hal
Hello, Hal, how are you?
Main>
```

And there's interactivity. Let's go back and look at the code a little, though. We name the module "Main" so we can compile it. We name the primary function "main" so that the compile knows that this is the function to run when the program is run. On the fourth line, we import the `IO` library so that we get access to IO functions. On the seventh line, we start with **do**, telling Haskell that we're executing a sequence of commands.

The first command is `hSetBuffering stdin LineBuffering` which you should probably ignore for now (incidentally, this is only required by GHC – in Hugs you can get by without it). The necessity for this is because when GHC reads input, it expects to read it in rather large blocks. A typical person's name is no where large enough to fill this block. Thus, when we try to read from `stdin`, it waits until it's gotten a whole block. We want to get rid of this, so we tell it to use `LineBuffering` instead of block buffering.

The next command is `putStrLn`, which prints a string to the screen. On the ninth line, we say "`name <- getLine`". This would normally be written "`name = getLine`" but using the arrow instead of the equality sign shows that `getLine` isn't a real function and can return different values. The last line constructs a string using what we read in on the previous line and then prints it to the screen.

Another example of a function which isn't really a function would be one that returns a random value. In this context, a function that does this is called `randomRIO`. Using this, we can write a "guess the number" program. Enter the following code into "Guess.hs":

```
module Main
    where

import IO
import Random

main =
    do hSetBuffering stdin LineBuffering
       num <- randomRIO (1::Int, 100)
       putStrLn "I'm thinking of a number between 1 and 100"
       doGuessing num

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!"
```

Let's examine this code. On the fifth line we write "import Random" to tell the compiler that we're going to be using some random functions (these aren't built into the Prelude). In the first line of `main`, we get a random number in the range $(1, 100)$. We need to write `::Int` to tell the compiler that we're using integers here, not floating point numbers or other numbers. We'll talk more about this in Section 4. On the next line, we tell the user what's going on and then on the last line of `main` we tell the compiler to execute the "function" `doGuessing`.

The `doGuessing` function takes as an argument the number the user is trying to guess. First, it asks the user to guess and then accepts their guess (which is a `String`) from the keyboard. The `if` statement checks first to see if their guess is too low. However, since `guess` is a string and `num` is an integer, we first need to convert `guess` to an integer by `read`ing it. If they guessed to low, we inform them and then start `doGuessing` over again. If they didn't guess too low, we check to see if they guessed too high. If they did, we tell them and start `doGuessing` again. Otherwise, they didn't guess too low and they didn't guess too high, so they must have gotten it correct, we we tell them that they won and exit.

You can either compile this code or load it into your interpreter and will get something like:

```
Main> main
I'm thinking of a number between 1 and 100
Enter your guess:
50
Too low!
Enter your guess:
75
Too low!
Enter your guess:
85
Too high!
Enter your guess:
80
Too high!
Enter your guess:
78
Too low!
Enter your guess:
```

```
79
You Win!
```

By now you should have a good understanding of how to write simple functions, compile them, test things in the interactive environment, and manipulate lists.

3.8 Sections and Infix Operators

We've already seen how to double the values of elements in a list using `map`:

```
Prelude> map (\x -> x*2) [1,2,3,4]
[2,4,6,8]
```

However, there is a more concise way to write this:

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

This type of thing can be done for any infix function:

```
Prelude> map (+5) [1,2,3,4]
[6,7,8,9]
Prelude> map (/2) [1,2,3,4]
[0.5,1.0,1.5,2.0]
Prelude> map (2/) [1,2,3,4]
[2.0,1.0,0.666667,0.5]
```

You might be tempted to try to subtract values from elements in a list by mapping `-2` across a list. This won't work, though, because while the `+` in `+2` is parsed as the standard plus operator (as there is no ambiguity), the `-` in `-2` is interpreted as the unary minus, not the binary minus. Thus `-2` here is the *number* -2, not the function $\lambda x.x - 2$.

In general, these are called sections. For binary infix operators (like `+`), we can cause the function to become prefix by enclosing it in paretheses. For example:

```
Prelude> (+) 5 3
8
Prelude> (-) 5 3
2
```

Additionally, we can provide either of its argument to make a section. For example:

```
Prelude> (+5) 3
8
Prelude> (/3) 6
2.0
Prelude> (3/) 6
0.5
```

Non-infix functions can be made infix by enclosing them in backquotes (“`”). For example:

```
Prelude> (+2) `map` [1..10]
[3,4,5,6,7,8,9,10,11,12]
```

Chapter 4

Type Basics

Haskell uses a system of *static type checking*. This means that every expression in Haskell is assigned a *type*. For instance 'a' would have type **Char**, for "character." Then, if you have a function which expects an argument of a certain type and you give it the wrong type, a compile-time error will be generated (that is, you will not be able to compile the program). This vastly reduces the number of bugs that can creep into your program.

Furthermore, Haskell uses a system of *type inference*. This means that you don't even need to specify the type of expressions. For comparison, in C, when you define a variable, you need to specify its type (for instance, int, char, etc.). In Haskell, you needn't do this – the type will be inferred from context.

> ■ NOTE ■ If you want, you certainly are allowed to explicitely specify the type of an expression; this often helps debugging. In fact, it is sometimes considered good style to explicitly specify the types of outermost functions.

Both Hugs and GHCi allow you to apply type inference to an expression to find its type. This is done by using the :t command. For instance, start up your favorite shell and try the following:

```
Prelude> :t 'c'
'c' :: Char
```

This tells us that the expression 'c' has type **Char** (the double colon :: is used throughout Haskell to specify types).

4.1 Simple Types

There are a slew of built-in types, including **Int** (for integers, both positive and negative), **Double** (for floating point numbers), **Char** (for single characters),

String (for strings), and others. We have already seen an expression of type **Char**; let's examime one of type **String**:

```
Prelude> :t "Hello"
"Hello" :: String
```

You can also enter more complicated expressions, for instance, a test of equality:

```
Prelude> :t 'a' == 'b'
'a' == 'b' :: Bool
```

You should note than even though this expression is false, it still has a type, namely the type **Bool**.

> ■ NOTE ■ **Bool** is short for Boolean (usually pronounced "boo-lee-uhn", though I've heard "boo-leen" once or twice) and has two possible values: **True** and **False**.

You can observe the process of type checking and type inference by trying to get the shell to give you the type of an ill-typed expression. For instance, the equality operator requires that the type of both of its arguments are of the same type. We can see that **Char** and **String** are of different types by trying to compare a character to a string:

```
Prelude> :t 'a' == "a"
ERROR - Type error in application
*** Expression     : 'a' == "a"
*** Term           : 'a'
*** Type           : Char
*** Does not match : [Char]
```

The first line of the error (the line containing "Expression") tells us the expression in which the type error occured. The second line tells us which part of this expression is ill-typed. The third line tells us the inferred type of this term and the fourth line tells us what it needs to have matched. In this case, it says that type type **Char** doesn't match the type **[Char]** (a list a characters – a string in Haskell is represented as a list of characters).

As mentioned before, you can explicitely specify the type of an expression using the **::** operator. For instance, instead of "a" in the previous example, we could have written ("a"::String). In this case, this has no effect since there's only one possible interpretation of "a". However, consider the case of numbers. You can try:

```
Prelude> :t 5 :: Int
5 :: Int
Prelude> :t 5 :: Double
5 :: Double
```

Here, we can see that the number 5 can be instantiated as either an **Int** our a **Double**. What if we don't specify the type?

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

Not quite what you expected? What this means, briefly, is that if some type a is an *instance* of the **Num** class, then type type of the expression 5 can be of type a. If that made no sense, that's okay for now. In Section 4.2 we talk extensively about type classes (which is what this is). The way to read this, though, is to say "a being an instance of Num implies a."

4.2 Type Classes

We saw last section some strange typing having to do with the number five. Before we delve too deeply into the subject of type classes, let's take a step back and see some of the motivation.

4.2.1 Motivation

In many languages (C++, Java, etc.), there exists a system of overloading. That is, a function can be written that takes parameters of differing types. For instance, the cannonical example is the equality function. If we want to compare two integers, we should use an integer comparison; if we want to compare two floating point numbers, we should use a floating point comparison; if we want to compare two characters, we should use a character comparison. In general, if we want to compare two things which have type α, we want to use an α-compare. We call α a *type variable* since it is a variable whose value is a type.

> ■ NOTE ■ In general, type variables will be written using the first part of the Greek alphabet: $\alpha, \beta, \gamma, \delta, \ldots$.

Unfortunately, this presents some problems for static type checking, since the type checker doesn't know which types a certain operation (for instance, equality testing) will be defined for. There are as many solutions to this problem as there are statically typed languages (perhaps a slight exageration, but not so much so). The one chosen in Haskell is the system of type classes. Whether this is the "correct" solution or the "best" solution of course depends on your application domain. It is, however, the one we have, so you should learn to love it.

4.2.2 Equality Testing

Returning to the issue of equality testing, what we want to be able to do is define a function == (the equality operator) which takes two parameters, each of the same type (call it α), and returns a boolean. But this function may not be defined for *every* type; just for some. Thus, we associate this function == with a type class, which we call **Eq**. If a specific type α belongs to a certain type class (that is, all functions associated with that class are implemented for α), we say that α is an *instance* of that class. For instance, **Int** is an instance of **Eq** since equality is defined over integers.

4.2.3 The Num Class

In addition to overloading operators like ==, Haskell has overloaded numeric constants (i.e., 1, 2, 3, etc.). This was done so that when you type in a number like 5, the compiler is free to say 5 is an integer or floating point number as it sees fit. It defines the **Num** class to contain all of these numbers and certain minimal operations over them (addition, for instance). The basic numeric types (Int, Double) are defined to be instances of **Num**.

We have only skimmed the surface of the power (and complexity) of type classes here. There will be much more discussion of them in Section 8.4, but we need some more background before we can get there. Before we do that, we need to talk a little more about functions.

4.2.4 The Show Class

Another of the standard classes in Haskell is the `Show` class. Types which are members of the `Show` class have functions which convert values of that type to a string. This function is called `show`. For instance `show` applied to the integer 5 is the string "5"; `show` applied to the character 'a' is the three-character string "'a"' (the first and last characters are apostrophes). `show` applied to a string simply puts quotes around it. You can test this in the interpreter:

```
Prelude> show 5
"5"
Prelude> show 'a'
"'a'"
Prelude> show "Hello World"
"\"Hello World\""
```

> ■ NOTE ■ The reason the backslashes appear in the last line is because the interior quotes are "escaped", meaning that they are part of the string, not part of the interpreter printing the value. The actual string doesn't contain the backslashes.

Some types are note instances of Show; functions for example. If you try to show a function (like `sqrt`), the compiler or interpreter will give you some cryptic error message, complaining about a missing instance declaration or an illegal class constraint.

4.3 Function Types

In Haskell, functions are *first class values*, meaning that just as `1` or `'c'` are values which have a type, so are functions like `square` or `++`. Before we talk too much about functions, we need to make a short diversion into very theoretical computer science (don't worry, it won't be too painful) and talk about the lambda calculus.

4.3.1 Lambda Calculus

The name "Lambda Calculus", while perhaps daunting, describes a fairly simple system for representing functions. The way we would write a squaring function in lambda calculus is: $\lambda x.x*x$, which means that we take a value, which we will call x (that's what "λx. means) and then multiply it by itself. The λ is called "lambda abstraction." In general, lambdas can only have one parameter. If we want to write a function that takes two numbers, doubles the first and adds it to the second, we would write: $\lambda x \lambda y.2*x+y$. When we apply a value to a lambda expression, we remove the outermost λ and replace every occurance of the lambda variable with the value. For instance, if we evaluate $(\lambda x.x*x)5$, we remove the lambda and replace every occurence of x with 5, yielding $(5*5)$ which is 25.

In fact, Haskell is largely based on an extension of the lambda calculus, and these two expressions can be written directly in Haskell (we simply replace the λ with a backslash and the . with an arrow; also we don't need to repeat the lambdas; and, of course, in Haskell we have to give them names if we're defining functions):

```
square = \x -> x*x
f = \x y -> 2*x + y
```

You can also evaluate lambda expressions in your interactive shell:

```
Prelude> (\x -> x*x) 5
25
Prelude> (\x y -> 2*x + y) 5 4
14
```

We can see in the second example that we need to give the lambda abstraction two arguments, one corresponding to `x` and the other corresponding to `y`.

4.3.2 Higher-Order Types

"Higher-Order Types" is the name given to functions. The type given to functions mimicks the lambda calculus representation of the functions. For instance, the definition of square gives $\lambda x.x*x$. To get the type of this, we first ask ourselves what the type of `x` is. Say we decide `x` is an `Int`. Then, we notice that the function `square` takes an `Int` and produces a value `x*x`. We know that when we multiply two `Int`s together, we get another `Int`, so the type of the results of `square` is also an `Int`. Thus, we say the type of `square` is `Int -> Int`.

We can apply a similar analysis to the function `f` above. The value of this function (remember, functions *are* values) is something which takes a value `x` and given that value, produces a *new* value, which takes a value `y` and produces `2*x+y`. For instance, if we take `f` and apply only one number to it, we get $(\lambda x \lambda y.2x + y)5$ which becomes our new value $\lambda y.2(5) + y$, where all occurances of x have been replaced with the applied value, 5.

So we know that `f` takes an `Int` and produces a value of some type, of which we're not sure. But we know the type of this value is the type of $\lambda y.2(5) + y$. We apply the above analysis and find out that this expression has type `Int -> Int`. Thus, `f` takes an `Int` and produces something which has type `Int -> Int`. So the type of `f` is `Int -> (Int -> Int)`.

> ■ NOTE ■ The parentheses are not necessary; in function types, if you have $\alpha \to \beta \to \gamma$ it is assume that $\beta \to \gamma$ is grouped. If you want the other way, with $\alpha \to \beta$ grouped, you need to put parentheses around them.

This isn't entirely accurate. As we saw before, numbers like `5` aren't really of type `Int`, they are of type `Num a => a`.

We can easily find the type of Prelude functions using ":t" as before:

```
Prelude> :t head
head :: [a] -> a
Prelude> :t tail
tail :: [a] -> [a]
Prelude> :t null
null :: [a] -> Bool
Prelude> :t fst
fst :: (a,b) -> a
Prelude> :t snd
snd :: (a,b) -> b
```

We read this as: "head" is a function that takes a list containing values of type "a" and gives back a value of type "a"; "tail" takes a list of "a"s and gives back another list of "a"s; "null" takes a list of "a"s and gives back a boolean; "fst" takes a pair of type "(a,b)" and gives back something of type "a", and so on.

> ■ NOTE ■ Saying that the type of `fst` is `(a,b) -> a` does not necessarily mean that it simply gives back the first element; it only means that it gives back something with the *same type* as the first element.

We can also get the type of operators like + and * and ++ and :; however, in order to do this we need to put them in parentheses. In general, any function which is used *infix* (meaning in the middle of two arguments rather than before them) must be put in parentheses when getting its type.

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

The types of + and * are the same, and mean that + is a function which, for some type `a` which is an instance of `Num`, takes a value of type `a` and produces another function which takes a value of type `a` and produces a value of type `a`. In short hand, we might say that + takes two values of type `a` and produces a value of type `a`, but this is less precise.

The type of ++ means, in shorthand, that, for a given type `a`, ++ takes two lists of `a`s and produces a new list of `a`s. Similarly, : takes a value of type `a` and another value of type `[a]` (list of `a`s) and produces another value of type `[a]`.

4.3.3 That Pesky IO Type

You might be tempted to try getting the type of a function like `putStrLn`:

```
Prelude> :t putStrLn
putStrLn :: String -> IO ()
Prelude> :t readFile
readFile :: FilePath -> IO String
```

What in the world is that `IO` thing? It's basically Haskell's way of representing that these functions aren't really functions. They're called "IO Actions" (hence the `IO`). The immediate question which arises is: okay, so how do I get rid of the `IO`. In brief, you can't directly remove it. That is, you cannot write a function with type `IO String -> String`. The only way to use things with an `IO` type is to combine them with other functions using (for example), the **do** notation.

For example, if you're reading a file using `readFile`, presumably you *want* to do something with the string it returns (otherwise, why would you read the file in the first place). Suppose you have a function `f` which takes a `String` and produces an `Int`. You can't directly apply `f` to the result of `readFile` since the input to `f` is `String` and the output of `readFile` is `IO String` and these don't match. However, you can combine these as:

```
main =
    do s <- readFile "somefile"
       let i = f s
       putStrLn (show i)
```

Here, we use the arrow convention to "get the string out of the IO action" and then apply `f` to the string (called `s`). We then, for example, print `i` to the screen. Note that the **let** here doesn't have a corresponding **in**. This is because we are in a **do** block. Also note that we don't write `i <- f s` because `f` is just a normal function, not an IO action.

4.3.4 Explicit Type Declarations

It is sometimes desirable to explicitly specify the types of some elements or functions, for one (or more) of the following reasons:

- Clarity
- Speed
- Debugging

Some people consider it good software engineering to specify the types of all top-level functions. If nothing else, if you're trying to compile a program and you get type errors that you cannot understand, if you declare the types of some of your functions explicitly, it may be easier to figure out where the error is.

Type declarations are written separatly from the function definition. For instance, we could explicitly type the function `square` as in the following code (an explicitly declared type is called a *type signature*):

```
square :: Num a => a -> a
square x = x*x
```

These two lines do not even have to be next to eachother. However, the type that you specify must match the inferred type of the function definition (or be more specific). In this definition, you could apply `square` to anything which is an instance of `Num`: `Int`, `Double`, etc. However, if you knew apriori that `square` were only going to be applied to value of type `Int`, you could *refine* its type as:

```
square :: Int -> Int
square x = x*x
```

Now, you could only apply `square` to values of type `Int`. Moreover, with this definition, the compiler doesn't have to generate the general code specified in the original function definition since it knows you will only apply `square` to `Int`s, so it may be able to generate faster code.

If you have extensions turned on ("-98" in Hugs or "-fglasgow-exts" in GHC(i)), you can also add a type signature to expressions and not just functions. For instance, you could write:

```
square (x :: Int) = x*x
```

which tells the compiler that `x` is an `Int`; however, it leaves the compiler alone to infer the type of the rest of the expression. What is the type of `square` in this example? Make your guess then you can check it either by entering this code into a file and loading it into your interpreter or by asking for the type of the expression:

```
Prelude> :t (\(x :: Int) -> x*x)
```

since this lambda abstraction is equivalent to the above function declaration.

4.3.5 Functional Arguments

In Section 3.3 we saw examples of functions taking other functions as arguments. For instance, `map` took a function to apply to each element in a list, `filter` took a function that told it which elements of a list to keep, and `foldl` took a function which told it how to combine list elements together. As with every other function in Haskell, these are well-typed.

Let's first think about the `map` function. It's job is to take a list of elements and produce another list of elements. These two lists don't necessarily have to have the same types of elements. So `map` will take a value of type `[a]` and produce a value of type `[b]`. How does it do this? It uses the user-supplied function to convert. In order to convert an `a` to a `b`, this function must have type `a -> b`. Thus, the type of `map` is `(a -> b) -> [a] -> [b]`, which you can verify in your interpreter with ":t".

We can apply the same sort of analysis to `filter` and discern that it has type `(a -> Bool) -> [a] -> [a]`. As we presented the `foldl` function, you might be tempted to give it type `(a -> a -> a) -> a -> [a] -> a`, meaning that you take a function which combines two `a`s into another one, an initial value of type `a`, a list of `a`s to produce a final value of type `a`. In fact, `foldl` has a more general type: `(a -> b -> a) -> a -> [b] -> a`. So it takes a function which turn an `a` and a `b` into an `a`, an initial value of type `a` and a list of `b`s. It produces an `a`.

To see this, we can write a function `count` which counts how many members of a list satisfy a given constraint. You can of course you `filter` and `length` to do this, but we will also do it using `foldr`:

```
module Count
    where

import Char

count1 p l = length (filter p l)
count2 p l = foldr (\x c -> if p x then c+1 else c) 0 l
```

The functioning of `count1` is simple. It filters the list `l` according to the predicate `p`, then takes the length of the resulting list. On the other hand, `count2` uses the intial value (which is an integer) to hold the current count. For each element in the list `l`, it applies the lambda expression shown. This takes two arguments, `c` which holds the current count and `x` which is the current element in the list that we're looking at. It checks to see if `p` holds about `x`. If it does, it returns the new value `c+1`, increasing the count of elements for which the predicate holds. If it doesn't, it just returns `c`, the old count.

4.4 Data Types

Tuples and lists are nice, common ways to define structured values. However, it is often desirable to be able to define our own data structures and functions over them. So-called "datatypes" are defined using the **data** keyword.

4.4.1 Pairs

For instance, a definition of a pair of elements (much like the standard, build-in pair type) could be:

```
data Pair a b = Pair a b
```

Let's walk through this code one word at a time. First we say "data" meaning that we're defining a datatype. We then give the name of the datatype, in this case, "Pair." The "a" and "b" that follow "Pair" are type parameters, just like the "a" is the type of the function `map`. So up until this point, we've said that we're going to define a data structure called "Pair" which is parameterized over two types, `a` and `b`.

After the equals sign, we specify the *constructors* of this data type. In this case, there is a single constructor, "Pair" (this doesn't necessarily have to have the same name as the type, but in this case it seems to make more sense). After this pair, we again write "a b", which means that in order to construct a `Pair` we need two values, one of type "a" and one of type "b".

This definition introduces a function, `Pair :: a -> b -> Pair a b` that you can use to construct `Pairs`. If you enter this code into a file and load it, you can see how these are constructed:

```
Datatypes> :t Pair
Pair :: a -> b -> Pair a b
Datatypes> :t Pair 'a'
Pair 'a' :: a -> Pair Char a
Datatypes> :t Pair 'a' "Hello"
:t Pair 'a' "Hello"
Pair 'a' "Hello" :: Pair Char [Char]
```

So, by giving `Pair` two values, we have completely constructed a value of type `Pair`. We can write functions involving pairs as:

```
pairFst (Pair x y) = x
pairSnd (Pair x y) = y
```

In this, we've used the *pattern matching* capabilities of Haskell to look at a pair an extract values from it. In the definition of `pairFst` we take an entire `Pair` and extract the first element; similarly for `pairSnd`. We'll discuss pattern matching in much more detail in Section 7.2.

4.4.2 Recursive Datatypes

We can also define *recursive datatypes.* These are datatypes whose definitions are based on themselves. For instance, we could define a list datatype as:

```
data List a = Nil
            | Cons a (List a)
```

In this definition, we have defined what it means to be of type `List a`. We say that a list is either empty (`Nil`) or it's the `Cons` of a value of type `a` and another value of type `List a`. This is almost identical to the actual definition of the list datatype in Haskell, except that uses special syntax where `[]` corresponds to `Nil` and `:` corresponds to `Cons`. We can write our own `length` function for our lists as:

```
listLength Nil = 0
listLength (Cons x xs) = 1 + listLength xs
```

This function is slightly more complicated and uses *recursion* to calculate the length of a `List`. The first line says that the length of an empty list (a `Nil`) is 0. This much is obvious. The second line tells us how to calculate the length of a non-empty list. A non-empty list must be of the form `Cons x xs` for some values of `x` and `xs`. We know that `xs` is another list and we know that whatever the length of the current list is, it's the length of its tail (the value of `xs`) plus one (to account for `x`). Thus, we apply the `listLength` function to `xs` and add one to the result. This gives us the length of the entire list.

4.4.3 Binary Trees

We can define datatypes that are more complicated that lists. Suppose we want to define a structure that looks like a binary tree. A binary tree is a structure that has a single root node; each node in the tree is either a "leaf" or a "branch." If it's a leaf, it holds a value; if it's a branch, it holds a value and a left child and a right child. Each of these children is another node. We can define such a data type as:

```
data BinaryTree a = Leaf a
                  | Branch (BinaryTree a) a (BinaryTree a)
```

In this datatype declaration we say that a `BinaryTree` of `a`s is either a `Leaf` which holds an `a`, or it's a branch with a left child (which is a `BinaryTree` of `a`s), a node value (which is an `a`), and a right child (which is also a `BinaryTree` of `a`s). It is simple to modify the `listLength` function so that instead of calculating the length of lists, it calculates the number of nodes in a `BinaryTree`. Can you figure out how? We can call this function `treeSize`. The solution is given below:

```
treeSize (Leaf x) = 1
treeSize (Branch left x right) = 1 + treeSize left + treeSize right
```

Here, we say that the size of a leaf is 1 and the size of a branch is the size of its left child, plus the size of its right child, plus one.

4.4.4 Enumerated Sets

You can also use datatypes to define things like enumerated sets, for instance, a type which can only have a constrained number of values. We could define a color type:

```
data Color = Red
           | Orange
           | Yellow
           | Green
           | Blue
           | Purple
           | White
           | Black
```

This would be sufficient to deal with simple colors. Suppose we were using this to write a drawing program, we could then write a function to convert between a `Color` and a RGB triple. We can write a `colorToRGB` function, as:

```
colorToRGB Red    = (255,0,0)
colorToRGB Orange = (255,128,0)
colorToRGB Yellow = (255,255,0)
colorToRGB Green  = (0,255,0)
colorToRGB Blue   = (0,0,255)
colorToRGB Purple = (255,0,255)
colorToRGB White  = (255,255,255)
colorToRGB Black  = (0,0,0)
```

If we wanted also to allow the user to define his own custom colors, we could change the `Color` datatype to something like:

```
data Color = Red
           | Orange
```

```
           | Yellow
           | Green
           | Blue
           | Purple
           | White
           | Black
           | Custom Int Int Int  -- R G B components, respectively
```

And add a final definition for `colorToRGB`:

```
colorToRGB (Custom r g b) = (r,g,b)
```

4.4.5 The Maybe type

Let us consider a simple function which searches through a list for an element satisfying a given predicate and then returns the first element satisfying that predicate. What should we do if none of the elements in the list satisfy the predicate? A few options are listed below:

- Raise an error
- Loop indefinately
- Write a check function
- Return the first element
- ...

Raising an error is certainly an option (see Section ?? to see how to do this). The problem is that it is difficult/impossible to recover from such errors. Looping indefinately is possible, but not terribly useful. We could write a sister function which checks to see if the list contains an element satisfying a predicate and leave it up to the user to always use this function first. We could return the first element, but this is very ad-hoc and difficult to remember.

The fact that there is no basic option to solve this problem simply means we have to think about it a little more. What are we trying to do? We're trying to write a function which might succeed and might not. Furthermore, if it does succeed, it returns some sort of value. Let's write a datatype:

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

In this, we represent failure by the value `Nothing` and success wiht value `a` by `Just a`. Our function might look something like this:

```
findElement :: (a -> Bool) -> [a] -> Maybe a
findElement p [] = Nothing
findElement p (x:xs) =
    if p x then Just x
    else findElement p xs
```

The first line here gives the type of the function. In this case, our first argument is the predicate (and takes an element of type `a` and returns `True` if and only if the element satisfies the predicate); the second argument is a list of `a`s. Our return value is *maybe* an `a`. That is, if the function succeeds, we will return `Just a` and if not, `Nothing`.

This `Maybe` datatype is so useful that it is part of the standard prelude; you can also get many helpful functions over `Maybe` from the `Maybe` module.

4.4.6 The Unit type

A final useful datatype defined in Haskell (from the Prelude) is the unit type. It's definition is:

```
data () = ()
```

The only true value of this type is `()`. This is essentially the same as a *void* type in a langauge like C or Java and will be useful when we talk about IO in Section 5.

We'll dwell much more on data types in Sections 7.2 and 8.1.

4.5 Continuation Passing Style

There is a style of functional programming called "Continuation Passing Style" (also simply "CPS"). The idea behind CPS is to pass around as a function argument what to do next. I will handwave through an example which is too complex to write out at this point and then give a real example, though one with less motivation.

Consider the problem of parsing. The idea here is that we have a sequence of tokens (words, letters, whatever) and we want to ascribe structure to them. The task of converting a string of Java tokens to a Java abstract syntax tree is an example of a parsing problem. So is the task of taking an English sentence and creating a parse tree (though the latter is quite a bit harder).

Suppose we're parsing something like C or Java where functions take arguments in parentheses. But for simplicity, assume they are not separated by commas. That is, a function call looks like `myFunction(x y z)`. We want to convert this into something like a pair containing first the string "myFunction" and then a list with three string elements: "x", "y" and "z".

The general approach to solving this would be to write a function which parses function calls like this one. First it would look for an identifier ("myFunction"), then for an open parenthesis, then for zero or more identifiers, then for a close parenthesis.

One way to do this would be to have two functions:

```
parseFunction   :: [Token] -> Maybe ((String, [String]), [Token])
parseIdentifier :: [Token] -> Maybe (String, [Token])
```

The idea would be that if we call `parseFunction`, if it doesn't return `Nothing`, then it returns the pair described earlier, together with whatever is left after parsing the function. Similarly, `parseIdentifier` will parse one of the arguments. If it returns `Nothing`, then it's not an argument; if it returns `Just` something, then that something is the argument paired with the rest of the tokens.

What the `parseFunction` function would do is to parse an identifier. If this fails, it fails itself. Otherwise, it continues and tries to parse a open parenthesis. If that succeeds, it repeatedly calls `parseIdentifier` until that fails. It then tries to parse a close parenthesis. If that succeeds, then it's done. Otherwise, it fails.

There is, however, another way to think about this problem. The advantage to this solution is that functions no longer need to return the remaining tokens (which tends to get ugly). Instead of the above, we write functions:

```
parseFunction   :: [Token] -> ((String, [String]) -> [Token] -> a) ->
                   ([Token] -> a) -> a
parseIdentifier :: [Token] -> (String -> [Token] -> a) ->
                   ([Token] -> a) -> a
```

Let's consider `parseIdentifier`. This takes three arguments: a list of tokens and two *continuations*. The first continuation is what to do when you succeed. The second continuation is what to do if you fail. What `parseIdentifier` does, then, is try to read an identifier. If this succeeds, it calls the first continuation with that identifier and the remaining tokens as arguments. If reading the identifier fails, it calls the second continuation with all the tokens.

Now consider `parseFunction`. Recall that it wants to read an identifier, an open parenthesis, zero or more identifiers and a close parenthesis. Thus, the first thing it does is call `parseIdentifier`. The first argument it gives is the list of tokens. The first continuation (which is what `parseIdentifier` should do if it succeeds) is in turn a function which will look for an open parenthesis, zero or more arguments and a close parethesis. The second argument (the failure argument) is just going to be the failure function given to `parseFunction`.

Now, we simply need to define this function which looks for an open parenthesis, zero or more arguments and a close parethesis. This is easy. We write a function which looks for the open parenthesis and then calls `parseIdentifier` with a success continuation that looks for more identifiers, and a "failure" continuation which looks for the close parenthesis (note that this failure doesn't really mean failure – it just means there are no more arguments left).

I realize this discussion has been quite abstract. I would willingly give code for all this parsing, but it is perhaps too complex at the moment. Instead, consider the problem of folding across a list. If we're willing to assume that the list is nonempty (actually, this code can be retrofitted to also work for empty lists), we can write a CPS-style fold as:

```
fold f z l = fold' f (\y -> f z y) l
```

```
fold' f g [x] = g x
fold' f g (x:xs) = g (fold' f (\y -> f x y) xs)
```

In this code, `fold` basically turns the accumulator argument `z` into a continuation. This continuation `g` says "give me a value and then I'll combine it with what I had before and give you the result). The function `fold'` first matches against a list with exactly one element: `x`. If this match succeeds, it says to `g`: "Here is a value, `x`. Give me the result." Of course, `g` obliges.

In the recursive case, `fold'` builds a new continuation that says: "give me a value and I'll combine it with what I see right now." It then recurses on itself and gives the result of this recursion to the original function `g` as a result.

You can test for yourself that this fold immitates one of `foldl`/`foldr`: try to figure out which one without using an interpreter. Also, try writing `map` or `filter` using CPS.

Chapter 5

Basic Input/Output

As we mentioned earlier, it is difficult to think of a good, clean way to integrate things like input/output into a pure functional language. Before we give the solution, let's take a step back and think about why such a thing is difficult.

Any IO library should provide a host of functions, containing (at a minimum) things 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 us consider the first two example first and think about what their types should be. Certainly the first item (I hesitate to call it a "function") should take a `String` argument and produce something, but what should it produce? It could produce a unit `()`, since there is essentially no return value from printing a string. The second item similarly should return a string, but how this happens is unclear.

We want both of these items to be functions. But they are by definition not functions. The item which reads a string from the keyboard cannot be a function, as it will not return the same thing every time. And if the first function simply returns `()` every time, there should be no problem with replacing it with a function `f _ = ()` because of referential transparency. But clearly this does not have the desired effect.

5.1 The RealWorld Solution

In a sense, the reason that these items are not functions is that they interact with the "real world." Their values depend directly on the real world. Supposing we had a type `RealWorld`, we might write these functions as having type:

```
printAString :: RealWorld -> String -> RealWorld
readAString  :: RealWorld -> (RealWorld, String)
```

That is, `printAString` takes a current state of the world and a string and modifies the state of the world in such a way that that string is now printed. Similarly, `readAString` takes a current state of the world and returns a *new* state of the world paired with the `String` that was typed.

This would be a possible way to do IO, though it is more than somewhat unweildy. In this style, our "Name.hs" program from Section ?? would look something like (assuming an initial `RealWorld` state were an argument to `main`):

```
main rW =
  let rW' = printAString rW "Please enter your name: "
      (rW'',name) = readAString rW'
  in  printAString rW'' ("Hello, " ++ name ++ ", how are you?")
```

This is not only hard to read, but prone to error if you accidentally use the wrong version of the real world. It also doesn't model the fact that the program below makes no sense:

```
main rW =
  let rW' = printAString rW "Please enter your name: "
      (rW'',name) = readAString rW'
  in  printAString rW' ("Hello, " ++ name ++ ", how are you?")
```

In this program the reference to `rW''` on the last line has been changed to a reference to `rW'`. It is completely unclear what this program should do. Clearly it must read a string in order to get a value for `name` to be printed. But that means that the real world has been updated. However, we then try to ignore this update by using an "old version" of the real world. There is clearly something very bad going on here.

5.2 The CPS Solution

A solution to this "old real world reference" problem is to use a continuation passing style. That is, the `printAString` function takes a `String` as well as something to do after this string has been printed. Thus, we never have access to the "old world." This results in types like:

```
printAString2 :: String -> (RealWorld -> a) -> RealWorld -> a
readAString2  :: (String -> RealWorld -> a) -> RealWorld -> a
```

These functions could then be used to build our class name program as:

```
main rw =
    printAString2 "Please enter your name: "
    (readAString2 (\name ->
        printAString2 ("Hello, " ++ name ++ ", how are you?")
                      (\_ -> ()))) rw
```

Here, first we print the query string then pass a function which reads a string and passes the read string to a function which prints the greating and then returns a constant `()`. The advantage to this style is that we now have no way of referencing the actual real world and so we don't have to worry about people accidentally (or maliciously) using old states of the world.

We can even define these functions in terms of the old ones (presumably a compiler would do this internally and simply not allow us access to the "basic" functions):

```
printAString2 s f rw = f (printAString rw s)

readAString2 f rw =
    let (rw', s) = readAString rw
    in  f s rw'
```

The disadvantage to this style is that it tends to be difficult to read. Even this simple program is difficult to understand at first glance (or perhaps even second glance).

Suffice it to say that doing IO operations in a pure lazy functional language is not trivial.

The breakthrough came when Phil Wadler decided 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 things 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).

5.3 Actions

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

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

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

> ■ NOTE ■ Actually, this type means that `putStrLn` is an action *within the IO monad*, but we will gloss over this for now.

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

```
getLine :: IO String
```

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

The question immediately arises: how do you "run" an action. This is something which is up to the compiler. You cannot actually run an action yourself; instead, a program is itself a single action which 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 which 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. In fact, we have already seen one way to do this using the **do** notation (how to "really" do this will be unveiled in Section 9). Let's consider the original name program, repeated here for convenience:

```
main =
    do hSetBuffering stdin LineBuffering
       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 four actions: setting buffering, a `putStrLn`, a `getLine` and another `putStrLn`. The `putStrLn` action has type `String -> IO ()` so we provide it a `String` so the fully applied action has type `IO ()`. This is something which we are allowed to execute.

The `getLine` action has type `IO String`, so it is okay to execute it directly. However, 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`."

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

```
    do ...
       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 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 us inspect 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 actually 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 will think you're trying to call `putStrLn` with four arguments; namely the string, the function `doGuessing` and the integer `num`. It will certainly complain (though the error may be somewhat difficult to comprehend at this point).

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

```
doGuessing num =
    do putStrLn "Enter your guess:"
       guess <- getLine
```

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

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

5.4 The IO Library

The IO Library (available by **import**ing the `IO` module) contains many definitions, the most common 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      :: 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 which, when run, produces the contents of that file. See Section 8.2 for more about type synonyms.

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 from a file, respectively. `hGetContents` reads in 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 an entire file without having to open it first.

The `bracket` function is used to perform actions safely. Consider a function which 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 ReadMode)
      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 closing all your handles.

5.5 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 (try reading a non-existant 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 IO

main =
    do hSetBuffering stdin LineBuffering
       doLoop

doLoop
    do putStrLn "Enter a command rFILENAME wFILENAME 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 characters are:"
                      putStrLn (take 100 contents))

doWrite filename =
    do putStrLn "Enter text to go into the file:"
       contents <- getLine
       bracket (openFile filename WriteMode) hClose
               (\h -> do 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. The `return` function is a function which 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', it checks to see if it was an 'r' followed by some string which 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`, gets 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 already exists, or if you specify some bad filename like `*#̂_@`. 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 withing 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 Section ??.

Chapter 6

Modules

In Haskell, program subcomponents are divided into modules. Each module sits in its own file and the name of the module should match the name of the file (without the ".hs" extension, of course), if you wish to ever use that module in a larger program.

For instance, suppose I am writing a game of poker. I may wish to have a separate module called "Cards" to handle the generation of cards, the shuffling and the dealing functions, and then use this "Cards" module in my "Poker" modules. That way, if I ever go back and want to write a blackjack program, I don't have to rewrite all the code for the cards; I can simply import the old "Cards" module.

6.1 Exports

Suppose as suggested we are writing a cards module. I have left out the implementation details, but suppose the skeleton of our module looks something like this:

```
module Cards
    where

data Card = ...
data Deck = ...

newDeck :: ... -> Deck
newDeck = ...

shuffle :: ... -> Deck -> Deck
shuffle = ...

-- 'deal deck n' deals 'n' cards from 'deck'
deal :: Deck -> Int -> [Card]
```

```
deal deck n = dealHelper deck n []

dealHelper = ...
```

In this code, the function `deal` calls a helper function `dealHelper`. The implementation of this helper function is very dependent on the exact data structures you used for `Card` and `Deck` so we don't want other people to be able to call this function. In order to do this, we create an *export list*, which we insert just after the module name declaration:

```
module Cards ( Card(),
               Deck(),
               newDeck,
               shuffle,
               deal
             )
    where

...
```

Here, we have specified exactly what functions the module exports, so people who use this module won't be able to access our `dealHelper` function. The `()` after `Card` and `Deck` specify that we are exporting the *type* but none of the constructors. For instance if our definition of `Card` were:

```
data Card = Card Suit Face
data Suit = Hearts
          | Spades
          | Diamonds
          | Clubs
data Face = Jack
          | Queen
          | King
          | Ace
          | Number Int
```

Then users of our module would be able to *use* things of type `Card`, but wouldn't be able to construct their own `Card`s and wouldn't be able to extract any of the suit/face information stored in them.

If we wanted users of our module to be able to access all of this information, we would have to specify it in the export list:

```
module Cards ( Card(Card),
               Suit(Hearts,Spades,Diamonds,Clubs),
               Face(Jack,Queen,King,Ace,Number),
               ...
             )
```

```
    where

...
```

This can get frustrating if you're exporting datatypes with many constructors, so if you want to export them all, you can simply write `(..)`, as in:

```
module Cards ( Card(..),
               Suit(..),
               Face(..),
               ...
             )
    where

...
```

And this will automatically export all the constructors.

6.2 Imports

There are a few idiosyncracies in the module import system, but as long as you stay away from the corner cases, you should be fine. Suppose, as before, you wrote a module called "Cards" which you saved in the file "Cards.hs". You are now writing your poker module and you want to *import* all the definitions from the "Cards" module. To do this, all you need to do is write:

```
module Poker
    where

import Cards
```

This will enable to you use any of the functions, types and constructors exported by the module "Cards". You may refer to them simply by their name in the "Cards" module (as, for instance, `newDeck`), or you may refer to them explicitely as imported from "Cards" (as, for instance, `Cards.newDeck`). It may be the case that two module export functions or types of the same name. In these cases, you can import one of the modules **qualified** which means that you would no longer be able to simply use the `newDeck` format but must use the longer `Cards.newDeck` format, to remove ambiguity. If you wanted to import "Cards" in this qualified form, you would write:

```
import qualified Cards
```

Another way to avoid problems with overlapping function definitions is to import only certain functions from modules. Suppose we knew the only function from "Cards" that we wanted was `newDeck`, we could import only this function by writing:

```
import Cards (newDeck)
```

On the other hand, suppose we knew that that the `deal` function overlapped with another module, but that we didn't need the "Cards" version of that function. We could hide the definition of `deal` and import everything else by writing:

```
import Cards hiding (deal)
```

Finally, suppose we want to import "Cards" as a qualified module, but don't want to have to type `Cards.` out all the time and would rather just type, for instance, `C.` – we could do this using the **as** keyword:

```
import qualified Cards as C
```

These options can be mixed and matched – you can give explicit import lists on qualified/as imports, for instance.

6.3 Hierarchical Imports

Though technically not part of the Haskell 98 standard, most Haskell compilers support hierarchical imports. This was designed to get rid of clutter in the directories in which modules are stored. Hierarchical imports allow you to specify (to a certain degree) where in the directory structure a module exists. For instance, if you have a "haskell" directory on your computer and this directory is in your compiler's path (see your compiler notes for how to set this; in GHC it's "-i", in Hugs it's "-P"), then you can specify module locations in subdirectories to that directory.

Suppose instead of saving the "Cards" module in your general haskell directory, you created a directory specifically for it called "Cards". The full path of the `Cards.hs` file is then `haskell/Cards/Cards.hs` (or, for Windows `haskell\Cards\Cards.hs`). If you then change the name of the Cards module to "Cards.Cards", as in:

```
module Cards.Cards(...)
    where

...
```

You could them import it in any module, regardless of this module's directory, as:

```
import Cards.Cards
```

If you start importing these module qualified, I highly recommend using the **as** keyword to shorten the names, so you can write:

```
import qualified Cards.Cards as Cards

... Cards.newDeck ...
```

instead of:

```
import qualified Cards.Cards

... Cards.Cards.newDeck ...
```

which tends to get ugly.

6.4 Literate Versus Non-Literate

The idea of literate programming is a relatively simple one, but took quite a while to become popularized. When we think about programming, we think about the *code* being the default mode of entry and *comments* being secondary. That is, we write code without any special annotation, but comments are annotated with either `--` or `{- ... -}`. Literate programming swaps these preconceptions.

There are two types of literate programs in Haskell; the first uses so-called Bird-scripts and the second uses LaTeX-style markup. Each will be discussed individually. No matter which you use, literate scripts must have the extension lhs instead of hs to tell the compiler that the program is written in a literate style.

6.4.1 Bird-scripts

In a Bird-style literate program, comments are default and code is introduced with a leading greater-than sign (">"). Everything else remains the same. For example, our Hello World program would be written in Bird-style as:

```
This is a simple (literate!) Hello World program.

> module Main
>     where

All our main function does is print a string:

> main = putStrLn "Hello World"
```

Note that the spaces between the lines of code and the "comments" are necessary (your compiler will probably complain if you are missing them). When compiled or loaded in an interpreter, this program will have the exact same properties as the non-literate version from Section 3.4.

6.4.2 LaTeX-scripts

LaTeX is a text-markup language very popular in the academic community for publishing. If you are unfamiliar with LaTeX, you may not find this section terribly usefaul.

Again, a literate Hello World program written in LaTeX-style would look like:

```
This is another simple (literate!) Hello World program.

\begin{code}
module Main
    where
\end{code}

All our main function does is print a string:

\begin{code}
main = putStrLn "Hello World"
\end{code}
```

In LaTeX-style scripts, the blank lines are *not* necessary.

Chapter 7

Advanced Features

Discussion

7.1 Common Sub-Expressions

There are many computations which require using the result of the same computation in multiple places in a function. For instance, if you remember back to your grade school mathematics courses, there is the following equation used to find the roots (zeros) of a polynomial of the form $ax^2 + bx + c = 0$: $x = (-b \pm \sqrt{b^2 - 4ac})/2a$. We could write the following function to compute the two values of x:

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

The compiler would probably be smart enough to realize that most of the computations here are identical and wouldn't repeat them. This doesn't make much difference for the additions and multiplications, but for the square roots, which are relatively slow operations, it could make a difference. Besides, as a reader, at first glance, it's difficult to tell exactly what the difference is between the first and second element of the pair is.

To remedy this problem, Haskell allows for local bindings. That is, we can create values inside of a function which only that function can see. For instance, we could create a local binding for `sqrt(b*b-4*a*c)` and call it, say, `sbb4ac` and then use that in both places where `sqrt(b*b-4*a*c)` occurred. We do this using a **where** clause. `where` clauses come immediately after function definitions and introduce a new level of layout (see Section 7.8). We write this as:

```
roots a b c =
    ((-b + sbb4ac) / (2*a), (-b - sbb4ac) / (2*a))
    where sbb4ac = sqrt(b*b-4*a*c)
```

Any values defined in a **where** clause *shadow* any other values with the same name. For instance, if we had the following code block:

```
sbb4ac = "Hello World"

roots a b c =
    ((-b + sbb4ac) / (2*a), (-b - sbb4ac) / (2*a))
    where sbb4ac = sqrt(b*b-4*a*c)

f _ = sbb4ac
```

The value of `roots` doesn't notice the top-level declaration of `sbb4ac`, since it is shadowed by the local definition (the fact that the types don't match doesn't matter either). Furthermore, since `f` cannot "see inside" of `roots`, the only thing it knows about `sbb4ac` is what is available at the top level, which is the string "Hello World." Thus, `f` is a function which takes any argument to that string.

Where clauses can contain any number of subexpressions, but they must be aligned for layout. For instance, we could also pull out the `2*a` computation and get the following code:

```
roots a b c =
    ((-b + sbb4ac) / (a2), (-b - sbb4ac) / (a2))
    where sbb4ac = sqrt(b*b-4*a*c)
          a2 = 2*a
```

Sub-expressions in **where** clauses must come after function definitions. Sometimes it is more convenient to put the local definitions before the actual expression of the function. This can be done by using **let/in** clauses. These are virtually identical to their **where** clause cousins except for their placement. The same `roots` function can be written:

```
roots a b c =
    let sbb4ac = sqrt(b*b-4*a*c)
        a2 = 2*a
    in  ((-b + sbb4ac) / (a2), (-b - sbb4ac) / (a2))
```

These two types of clauses can be mixed (i.e., you can write a function which has both a **let** cause and a **where** clause). This is strongly advised *against*, as it tends to make code difficult to read. However, if you choose to do it, values in the **let** clause shadow those in the **where** clause. So if you define the function:

```
f x =
    let y = x+1
    in  y
    where y = x+2
```

The value of `f 5` is `6`, not `7`. Of course, I plead with you to never ever write code that looks like this. No one should have to remember this rule and by shadowing **where**-defined values in a **let** clause only makes your code difficult to understand.

In general, whether you should use **let** clauses or **where** clauses is largely a matter of personal preference. Usually, the names you give to the subexpressions should be sufficiently expressive that without reading their definitions any reader of your code should be able to figure out what they do. In this case, **where** clauses are probably more desirable because they allow the reader to see immediately what a function does. However, in real life, values are often given cryptic names. In which case **let** clauses may be better. Either is probably okay, though I think **where** clauses are more common.

7.2 Pattern Matching

Pattern matching is one of the most powerful features of Haskell (and most functional programming languages). It is most commonly used in conjunction with `case` expressions, which we have already seen in Section 3.5. Let's return to our `Color` example from Section 4.4. I'll repeat the definition we already had for the datatype:

```
data Color = Red
           | Orange
           | Yellow
           | Green
           | Blue
           | Purple
           | White
           | Black
           | Custom Int Int Int  -- R G B components, respectively
           deriving (Show,Eq)
```

We then want to write a function that will convert between something of type `Color` and a triple of `Int`s, which correspond to the RGB values, respectively. Specifically, if we see a `Color` which is `Red`, we want to return `(255,0,0)`, since this is the RGB value for red. So we write that (remember that piecewise function definitions are just **case** statements):

```
colorToRGB Red = (255,0,0)
```

If we see a `Color` which is `Orange`, we want to return `(255,128,0)`; and if we see `Yellow`, we want to return `(255,255,0)`, and so on. Finally, if we see a custom color, which is comprised of three components, we want to make a triple out of these, so we write:

```
colorToRGB Orange = (255,128,0)
colorToRGB Yellow = (255,255,0)
colorToRGB Green  = (0,255,0)
colorToRGB Blue   = (0,0,255)
colorToRGB Purple = (255,0,255)
colorToRGB White  = (255,255,255)
colorToRGB Black  = (0,0,0)
colorToRGB (Custom r g b) = (r,g,b)
```

Then, in our interpreter, if we type:

```
Color> colorToRGB Yellow
(255,255,0)
```

What is happening is this: we create a value, call it x, which has value `Red`. We then apply this to `colorToRGB`. We check to see if we can "match" x against `Red`. This match fails because according to the definition of `Eq Color`, `Red` is not equal to `Yellow`. We continue down the definitions of `colorToRGB` and try to match `Yellow` against `Orange`. This fails, too. We the try to match `Yellow` against `Yellow`, which succeeds, so we use this function definition, which simply returns the value `(255,255,0)`, as expected.

Suppose instead, we used a custom color:

```
Color> colorToRGB (Custom 50 200 100)
(50,200,100)
```

We apply the same matching process, failing on all values from `Red` to `Black`. We then get to try to match `Custom 50 200 100` against `Custom r g b`. We can see that the `Custom` part matches, so then we go see if the subelements match. In the matching, the variables `r`, `g` and `b` are essentially wild cards, so there is no trouble matching `r` with 50, `g` with 200 and `b` with 100. As a "side-effect" of this matching, `r` gets the value 50, `g` gets the value 200 and `b` gets the value 100. So the entire match succeeded and we look at the definition of this part of the function and bundle up the triple using the matched values of `r`, `g` and `b`.

We can also write a function to check to see if a `Color` is a custom color or not:

```
isCustomColor (Custom _ _ _) = True
isCustomColor _ = False
```

When we apply a value to `isCustomColor` it tries to match that value against `Custom _ _ _`. This match will succeed if the value is `Custom x y z` for any `x`, `y` and `z`. The `_` (underscore) character is a "wildcard" and will match anything, but will not do the binding that would happen if you put a variable name there. If this match succeeds, the function returns `True`; however, if this match fails, it goes on to the next line, which will match anything and then return `False`.

For some reason we might want to define a function which tells us whether a given color is "bright" or not, where my definition of "bright" is that one of its RGB components is equal to 255 (admittedly and arbitrary definition, but it's simply an example). We could define this function as:

```
isBright = isBright' . colorToRGB
    where isBright' (255,_,_) = True
          isBright' (_,255,_) = True
          isBright' (_,_,255) = True
          isBright' _         = False
```

Let's dwell on this definition for a second. The `isBright` function is the composition of our previously defined function `colorToRGB` and a helper function `isBright'`, which tells us if a given RGB value is bright or not. We could replace the first line here with `isBright c = isBright' (colorToRGB c)` but there is no need to explicitly write the parameter here, so we don't. Again, this function composition style of programming takes some getting used to, so I will try to use it frequently in this tutorial.

The `isBright'` helper function takes the RGB triple produced by `colorToRGB`. It first tries to match it against `(255,_,_)` which succeeds if the value has 255 in its first position. If this match succeeds, `isBright'` returns `True` and so does `isBright`. The second and third line of definition check for 255 in the second and third position in the triple, respectively. The fourth line, the *fallthrough*, matches everything else and reports it as not bright.

We might want to also write a function to convert between RGB triples and `Colors`. We could simple stick everything in a `Custom` constructor, but this would defeat the purpose; we want to use the `Custom` slot only for values which don't match the predefined colors. However, we don't want to allow the user to construct custom colors like (600,-40,99) since these are invalid RGB values. We could throw an error if such a value is given, but this can be difficult to deal with. Instead, we use the `Maybe` datatype. This is defined (in the Prelude) as:

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

The way we use this is as follows: our `rgbToColor` function returns a value of type `Maybe Color`. If the RGB value passed to our function is *invalid*, we return `Nothing`, which corresponds to a failure. If, on the other hand, the RGB value is valid, we create the appropriate `Color` value and return `Just` that. The code to do this is:

```
rgbToColor 255   0   0 = Just Red
rgbToColor 255 128   0 = Just Orange
rgbToColor 255 255   0 = Just Yellow
rgbToColor   0 255   0 = Just Green
rgbToColor   0   0 255 = Just Blue
rgbToColor 255   0 255 = Just Purple
```

```
rgbToColor 255 255 255 = Just White
rgbToColor   0   0   0 = Just Black
rgbToColor   r   g   b =
    if 0 <= r && r <= 255 &&
       0 <= g && g <= 255 &&
       0 <= b && b <= 255
      then Just (Custom r g b)
      else Nothing   -- invalid RGB value
```

The first eight lines match the RGB arguments against the predefined values and, if they match, `rgbToColor` returns `Just` the appropriate color. If none of these matches, the last definition of `rgbToColor` matches the first argument against `r`, the second against `g` and the third against `b` (which causes the side-effect of binding these values). It then checks to see if these values are valid (each is greater than or equal to zero and less than or equal to 255). If so, it returns `Just (Custom r g b)`; if not, it returns `Nothing` corresponding to an invalid color.

Using this, we can write a function that checks to see if a right RGB value is valid:

```
rgbIsValid = rgbIsValid' . rgbToColor
    where rgbIsValid' (Just _) = True
          rgbIsValid' _        = False
```

Here, we compose the helper function `rgbIsValid'` with our function `rgbToColor`. The helper function checks to see if the value returned by `rgbToColor` is `Just` anything (the wildcard). If so, it returns `True`. If not, it matches anything and returns `False`.

Pattern matching isn't magic, though. You can only match against datatypes; you cannot match against functions. For instance, the following is invalid:

```
f x = x + 1

g (f x) = x
```

Even though the intended meaning of `g` is clear (i.e., `g x = x - 1`), the compiler doesn't know in general that `f` has an inverse function, so it can't perform matches like this.

7.3 Guards

Guards can be thought of as an extension to the pattern matching facility. They enable you to allow piecewise function definitions to be taken according to arbitrary boolean expressions. Guards appear after all arguments to a function but before the equals sign, and are begun with a vertical bar `|`. We could use guards to write a simple function which returns a string telling you the result of comparing two elements:

```
comparison x y | x < y = "The first is less"
               | x > y = "The second is less"
               | otherwise = "They are equal"
```

You can read the vertical bar as "such that." So we say that the value of `comparison x y` "such that" x is less than y is "The first is less." The value such that x is greater than y is "The second is less" and the value **otherwise** is "They are equal". The keyword `otherwise` is simply defined to be equal to `True` and thus matches anything that falls through that far. So, we can see that this works:

```
Guards> comparison 5 10
"The first is less"
Guards> comparison 10 5
"The second is less"
Guards> comparison 7 7
"They are equal"
```

One thing to note about guards is that they are tested *after* pattern matching, not in conjunction with pattern matching. This means that once a pattern matches, if none of the guards succeed, further pattern matches will not be attempted. So, if we had instead defined:

```
comparison2 x y | x < y = "The first is less"
                | x > y = "The second is less"
comparison2 _ _ = "They are equal"
```

The intention would be that if both of the guards failed, it would "fall through" to the final match and say that they were equal. This is not what happens, though.

```
Guards> comparison2 7 7
*** Exception: Guards.hs:8: Non-exhaustive patterns in function comparison2
```

If we think about what is happening in the compiler this makes sense. When we apply two sevens to `comparison2`, they are matched against `x` and `y`, which succeeds and the values are bound. Pattern matching then stops completely, since it has succeeded. The guards are then activated and `x` and `y` are compared. Neither of the guards succeeds, so an error is raised.

One nicety about guards is that **where** clauses are common to all guards. So another possible definition for our `isBright` function from the previous section would be:

```
isBright2 c | r == 255 = True
            | g == 255 = True
            | b == 255 = True
            | otherwise = False
    where (r,g,b) = colorToRGB c
```

The function is equivalent to the previous version, but performs its calculation slightly differently. It takes a color, `c`, and applies `colorToRGB` to it, yielding an RGB triple which is matched (using pattern matching!) against `(r,g,b)`. This match succeeds and the values `r`, `g` and `b` are bound to their respective values. The first guard checks to see if `r` is 255 and, if so, returns true. The second and third guard check `g` and `b` against 255, respectively and return true if they match. The last guard fires as a last resort and returns `False`

7.4 Instance Declarations

In order to declare a type to be an instance of a class, you need to provide an instance declaration for it. Most classes provide what's called a "minimal complete definition." This means the functions which must be implemented for this class in order for its definition to be satisfied. Once you've written these functions for your type, you can declare it an instance of the class.

7.4.1 The Eq Class

The `Eq` class has two members (i.e., two functions):

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

The first of these type signatures reads that the function `==` is a function which takes two `a`s which are members of `Eq` and produces a `Bool`. The type signature of `/=` (not equal) is identical. A minimal complete definition for the `Eq` class requires that either one of these functions be defined (if you define `==`, then `/=` is defined automatically by negating the result of `==`, and vice versa). These declarations must be provided inside the instance declaration.

This is best demonstrated by example. Suppose we have our color example, repeded here for convenience:

```
data Color = Red
           | Orange
           | Yellow
           | Green
           | Blue
           | Purple
           | White
           | Black
           | Custom Int Int Int  -- R G B components, respectively
```

We can define `Color` to be an instance of `Eq` by the following declaration:

```
instance Eq Color where
    Red == Red = True
    Orange == Orange = True
    Yellow == Yellow = True
    Green == Green = True
    Blue == Blue = True
    Purple == Purple = True
    White == White = True
    Black == Black = True
    (Custom r g b) == (Custom r' g' b') =
        r == r' && g == g' && b == b'
    _ == _ = False
```

The first line here begins with the keyword **instance** telling the compiler that we're making an instance declaration. It then specifies the class, `Eq`, and the type, `Color` which is going to be an instance of this class. Following that, there's the `where` keyword. Finally there's the method declaration.

The first eight lines of the method declaration are basically identical. The first one, for instance, says that the value of the expression `Red == Red` is equal to `True`. Lines two through eight are identical. The declaration for custom colors is a bit different. We pattern match `Custom` on both sides of `==`. On the left hand side, we bind `r`, `g` and `b` to the components, respectively. On the right hand side, we bind `r'`, `g'` and `b'` to the components. We then say that these two custom colors are equal precisely when `r == r'`, `g == g'` and `b == b'` are all equal. The fallthrough says that any pair we haven't previously declared as equal are unequal.

7.4.2 The `Show` Class

The `Show` class is used to display arbitrary values as strings. This class has three methods:

```
show :: Show a => a -> String
showsPrec :: Show a => Int -> a -> String -> String
showList :: Show a => [a] -> String -> String
```

A minimal complete definition is either `show` or `showsPrec` (we will talk about `showsPrec` later – it's in there for efficiency reasons). We can define our `Color` datatype to be an instance of `Show` with the following instance declaration:

```
instance Show Color where
    show Red = "Red"
    show Orange = "Orange"
    show Yellow = "Yellow"
    show Green = "Green"
    show Blue = "Blue"
    show Purple = "Purple"
```

```
    show White = "White"
    show Black = "Black"
    show (Custom r g b) =
        "Custom " ++ show r ++ " " ++ show g ++ " " ++ show b
```

This declaration specifies exactly how to convert values of type `Color` to `Strings`. Again, the first eight lines are identical and simply take a `Color` and produce a string. The last line for handling custom colors matches out the RGB components and creates a string by concattenating the result of `show`ing the components individually (with spaces in between and "Custom" at the beginning).

7.4.3 Other Important Classes

There are a few other important classes which I will mention briefly because either they are commonly used or because we will be using them shortly. I won't provide example instance declarations; how you can do this should be clear by now.

The `Ord` Class

The ordering class, the functions are:

```
compare :: Ord a => a -> a -> Ordering
(<=) :: Ord a => a -> a -> Bool
(>) :: Ord a => a -> a -> Bool
(>=) :: Ord a => a -> a -> Bool
(<) :: Ord a => a -> a -> Bool
min :: Ord a => a -> a -> a
max :: Ord a => a -> a -> a
```

The almost any of the functions alone is a minimal complete definition; it is recommended that you implement `compare` if you implement only one, though. This function returns a value of type `Ordering` which is defined as:

```
data Ordering = LT | EQ | GT
```

So, for instance, we get:

```
Prelude> compare 5 7
LT
Prelude> compare 6 6
EQ
Prelude> compare 7 5
GT
```

In order to declare a type to be an instance of `Ord` you must already have declared it an instance of `Eq` (in other words, `Ord` is a *subclass* of `Eq` – more about this in Section 8.4).

The `Enum` Class

The `Enum` class is for enumerated types; that is, for types where each element has a successor and a predecessor. It's methods are:

```
pred :: Enum a => a -> a
succ :: Enum a => a -> a
toEnum :: Enum a => Int -> a
fromEnum :: Enum a => a -> Int
enumFrom :: Enum a => a -> [a]
enumFromThen :: Enum a => a -> a -> [a]
enumFromTo :: Enum a => a -> a -> [a]
enumFromThenTo :: Enum a => a -> a -> a -> [a]
```

The minimal complete definition contains both `toEnum` and `fromEnum`, which converts from and to `Int`s. The `pred` and `succ` functions give the predecessor and successor, respectively. The `enum` functions enumerate lists of elements. For instance, `enumFrom x` lists all elements after `x`; `enumFromThen x step` lists all elements starting at `x` in steps of size `step`. The `To` functions end the enumeration at the given element.

The `Num` Class

The `Num` class provides the standard arithmetic operations:

```
(-) :: Num a => a -> a -> a
(*) :: Num a => a -> a -> a
(+) :: Num a => a -> a -> a
negate :: Num a => a -> a
signum :: Num a => a -> a
abs :: Num a => a -> a
fromInteger :: Num a => Integer -> a
```

All of these are obvious except for perhaps `negate` which is the unary minus. That is, `negate x` means $-x$.

The `Read` Class

The `Read` class is the opposite of the `Show` class. It is a way to take a string and read in from it a value of arbitrary type. The methods for `Read` are:

```
readsPrec :: Read a => Int -> String -> [(a, String)]
readList :: String -> [([a], String)]
```

The minimal complete definition is `readsPrec`. The most important function related to this is `read`, which uses `readsPrec` as:

```
read s = fst (head (readsPrec 0 s))
```

This will fail if parsing the string fails. You could define a `maybeRead` function as:

```
maybeRead s =
    case readsPrec 0 s of
      [(a,_)] -> Just a
      _ -> Nothing
```

How to write and use `readsPrec` directly will be discussed further in the examples.

7.4.4 Class Contexts

Suppose we are definition the `Maybe` datatype from scratch. The definition would be something like:

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

Now, when we go to write the instance declarations, for, say, `Eq`, we need to know that `a` is an instance of `Eq` otherwise we can't write a declaration. We express this as:

```
instance Eq a => Eq (Maybe a) where
    Nothing == Nothing = True
    (Maybe x) == (Maybe x') == x == x'
```

This first line can be read "That `a` is an instance of `Eq` *implies* (`=>`) that `Maybe a` is an instance of `Eq`."

7.4.5 Deriving Classes

Writing obvious `Eq`, `Ord`, `Read` and `Show` classes like these is tedious and should be automated. Luckily for us, it is. If you write a datatype that's "simple enough" (almost any datatype you'll write unless you start writing fixed point types), the compiler can automatically *derive* some of the most basic classes. To do this, you simply add a **deriving** clause to after the datatype declaration, as in:

```
data Color = Red
           | Orange
           | Yellow
           | Green
           | Blue
           | Purple
           | White
           | Black
           | Custom Int Int Int  -- R G B components, respectively
           deriving (Eq, Ord, Show, Read)
```

This will automatically create instances of the `Color` datatype of the named classes. Similarly, the declaration:

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

derives these classes just when `a` is appropriate.

All in all, you are allowed to derive instances of `Eq`, `Ord`, `Enum`, `Bounded`, `Show` and `Read`. There is considerable work in the area of "polytypic programming" or "generic programming" which, among other things, would allow for instance declarations for *any* class to be derived. This is much beyond the scope of this tutorial; instead, I refer you to the literature.

7.5 Datatypes Revisited

I know by this point you're probably terribly tired of hearing about datatypes. They are, however, incredibly important, otherwise I wouldn't devote so much time to them. Datatypes offer a sort of notational convenience if you have, for instance, a datatype that holds many many values. These are called named fields.

7.5.1 Named Fields

Consider a datatype whose purpose is to hold configuration settings. Usually when you extract members from this type, you really only care about one or possibly two of the many settings. Moreover, if many of the settings have the same type, you might often find yourself wondering "wait, was this the fourth or *fifth* element?" One thing you could do would be to write accessor functions. Consider the following made-up configuration type for a terminal program:

```
data Configuration =
    Configuration String                -- user name
                  String                -- local host
                  String                -- remote host
                  Bool                  -- is guest?
                  Bool                  -- is super user?
                  String                -- current directory
                  String                -- home directory
                  Integer               -- time connected
              deriving (Eq, Show)
```

You could then write accessor functions, like (I've only listed a few):

```
getUserName (Configuration un _ _ _ _ _ _ _) = un
getLocalHost (Configuration _ lh _ _ _ _ _ _) = lh
```

```
getRemoteHost (Configuration _ _ rh _ _ _ _ _) = rh
getIsGuest (Configuration _ _ _ ig _ _ _ _) = ig
...
```

You could also write update functions to update a single element. Of course, now if you add an element to the configuration, or remove one, all of these functions now have to take a different number of arguments. This is highly annoying and is an easy place for bugs to slip in. However, there's a solution. We simply give names to the fields in the datatype declaration, as follows:

```
data Configuration =
    Configuration { username      :: String,
                    localhost     :: String,
                    remotehost    :: String,
                    isguest       :: Bool,
                    issuperuser   :: Bool,
                    currentdir    :: String,
                    homedir       :: String,
                    timeconnected :: Integer
                  }
```

This will automatically generate the following accessor functions for us:

```
username :: Configuration -> String
localhost :: Configuration -> String
...
```

Moreover, it gives us very convenient update methods. Here is a short example for a "post working directory" and "change directory" like functions that work on `Configurations`:

```
changeDir :: Configuration -> String -> Configuration
changeDir cfg newDir =
    if directoryExists newDir          -- make sure the directory exists
      then cfg{currentdir = newDir}    -- change our current directory
      else error "directory does not exist"

postWorkingDir :: Configuration -> String
postWorkingDir cfg = currentdir cfg    -- retrieve our current directory
```

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}`.

You can of course continue to pattern match against `Configurations` 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 contstring 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 stype, as shown in the second definition below:

```
initCFG =
    Configuration "nobody" "nowhere" "nowhere" False False "/" "/" 0
initCFG' =
    Configuration
       { username="nobody",
         localhost="nowhere",
         remotehost="nowhere",
         isguest=False,
         issuperuser=False,
         currentdir="/",
         homedir="/",
         timeconnected=0 }
```

Though the second is probably much more understandable unless you litter your code with comments.

7.6 More Lists

todo: put something here

7.6.1 Standard List Functions

Recall that the definition of the built-in Haskell list datatype is equivalent to:

```
data List a = Nil
            | Cons a (List a)
```

With the exception that `Nil` is called `[]` and `Cons x xs` is called `x:xs`. This is simply to make pattern matching easier and code smaller. Let's investigate how some of the standard list functions may be written. Consider `map`. A definition is given below:

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

Here, the first line says that when you `map` across an empty list, no matter what the function is, you get an empty list back. The second line says that when you `map` across a list with `x` as the head and `xs` as the tail, the result is `f` applied to `x` consed onto the result of mapping `f` on `xs`.

The `filter` can be defined similarly:

```
filter _ [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

How this works should be clear. For an empty list, we return an empty list. For a non empty list, we return the filter of the tail, perhaps with the head on the front, depending on whether it satisfies the predicate `p` or not.

We can define `foldr` as:

```
foldr _ z [] = z
foldr f z (x:xs) = f x (foldlr f z xs)
```

Here, the best interpretation is that we are replacing the empty list (`[]`) with a particular value and the list constructor (`:`) with some function. On the first line, we can see the replacement of `[]` for `z`. Using backquotes to make `f` infix, we can write the second line as:

```
foldr f z (x:xs) = x `f` (foldlr f z xs)
```

From this, we can directly see how `:` is being replaced by `f`.

Finally, `foldl`:

```
foldl _ z [] =  z
foldl f z (x:xs) = foldl f (f z x) xs
```

This is slightly more complicated. Remember, `z` can be thought of as the current state. So if we're folding across a list which is empty, we simply return the current state. On the other hand, if the list is not empty, it's of the form `x:xs`. In this case, we get a new state by appling `f` to the current state `z` and the current list element `x` and then recursively call `foldl` on `xs` with this new state.

There is another class of functions: the `zip` and `unzip` functions, which respectively take multiple lists and make one or take one lists and split them apart. For instance, `zip` does the following:

```
Prelude> zip "hello" [1,2,3,4,5]
[('h',1),('e',2),('l',3),('l',4),('o',5)]
```

Basically, it pairs the first elements of both lists and makes that the first element of the new list. It then pairs the second elements of both lists and makes that the second element, etc. What if the lists have unequal length? It simply stops when the shorter one stops. A reasonable definition for `zip` is:

```
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

The `unzip` function does the opposite. It takes a zipped list and returns the two "original" lists:

```
Prelude> unzip [('f',1),('o',2),('o',3)]
("foo",[1,2,3])
```

There are a whole slew of `zip` and `unzip` functions, named `zip3`, `unzip3`, `zip4`, `unzip4` and so on; the `...3` functions use triples instead of pairs; the `...4` functions use 4-tuples, etc.

Finally, the function `take` takes an integer n and a list and returns the first n elements off the list. Correspondingly, `drop` takes an integer n and a list and returns the result of throwing away the first n elements off the list. Neither of these functions produces an error; if n is too large, they both will just return shorter lists.

7.6.2 List Comprehensions

There is some syntactic sugar for dealing with lists whose elements are members of the `Enum` class (see Section 7.4), such as `Int` or `Char`. If we want to create a list of all the elements from 1 to 10, we can simply write:

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

We can also introduce an amount to step by:

```
Prelude> [1,3..10]
[1,3,5,7,9]
Prelude> [1,4..10]
[1,4,7,10]
```

These expressions are short hand for `enumFromTo` and `enumFromThenTo`, respectively. Of course, you don't need to specify an upper bound. Try the following (but be ready to hit Control+C to stop the computation!):

```
Prelude> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12{Interrupted!}
```

Probably yours printed a few thousand more elements than this. As we said before, Haskell is lazy. That means that a list of all numbers from 1 on is perfectly well formed and that's exactly what this list is. Of course, if you attempt to print the list (which we're implicitly doing by typing it in the interpreter), it won't halt. But if we only evaluate an initial segment of this list, we're fine:

```
Prelude> take 3 [1..]
[1,2,3]
Prelude> take 3 (drop 5 [1..])
[6,7,8]
```

This comes in useful if, say, we want to assign an ID to each element in a list. Without laziness we'd have to write something like this:

```
assignID :: [a] -> [(a,Int)]
assignID l = zip l [1..length l]
```

Which means that the list will be traversed twice. However, because of laziness, we can simply write:

```
assignID l = zip l [1..]
```

And we'll get exactly what we want. We can see that this works:

```
Prelude> assignID "hello"
[('h',1),('e',2),('l',3),('l',4),('o',5)]
```

Finally, there is some useful syntactic sugar for `map` and `filter`, based on standard set-notation in mathematics. In math, we would write something like $\{f(x)|x \in s \wedge p(x)\}$ to mean the set of all values of f when applied to elements of s which satisfy p. This is equivalent to the Haskell statement `map f (filter p s)`. However, we can also use more math-like notation and write `[f x | x <- s, p x]`. While in math the ordering of the statements on the side after the pipe is free, it is not so in Haskell. We could not have put `p x` before `x <- s` otherwise the compiler wouldn't know yet what `x` was. We can use this to do simple string processing. Suppose we want to take a string, remove all the lower-case letters and convert the rest of the letters to upper case. We could do this in either of the following two equivalent ways:

```
Prelude> map toLower (filter isUpper "Hello World")
"hw"
Prelude> [toLower x | x <- "Hello World", isUpper x]
"hw"
```

These two are equivalent, and, depending on the exact functions you're using, one might be more readable than the other. There's more you can do here, though. Suppose you want to create a list of pairs, one for each point between

(0,0) and (5,7) below the diagonal. Doing this manually with lists and maps would be cumbersome and possibly difficult to read. It couldn't be easier with list comprehensions:

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

If you reverse the order of the `x <-` and `y <-` clauses, the order in which the space is traversed will be reversed (of course, in that case, `y` could no longer depend on `x` and you would need to make `x` depend on `y` but this is trivial).

7.7 Arrays

Lists are nice for many things. It is easy to add elements to the beginning of them and to manipulate them in various ways that change the length of the list. However, they are bad for random access, having average complexity $\mathcal{O}(n)$ to access an arbitrary element (if you don't know what $\mathcal{O}(\dots)$ means, you can either ignore it or take a quick detour and read Appendix A, a two-page introduction to complexity theory). So, if you're willing to give up fast insertion and deletion because you need random access, you should use arrays instead of lists.

In order to use arrays you must import the `Array` module. There are a few methods for creating arrays, the `array` function, the `listArray` function, and the `accumArray` function. The `array` function takes a pair which is the bounds of the array, and an association list which specifies the initial values of the array. The `listArray` function takes bounds and then simply a list of values. Finally, the `accumArray` function takes an accumulation function, an initial value and an association list and accumulates pairs from the list into the array. Here are some examples of arrays being created:

```
Arrays> array (1,5) [(i,2*i) | i <- [1..5]]
array (1,5) [(1,2),(2,4),(3,6),(4,8),(5,10)]
Arrays> listArray (1,5) [3,7,5,1,10]
array (1,5) [(1,3),(2,7),(3,5),(4,1),(5,10)]
Arrays> accumArray (+) 2 (1,5) [(i,i) | i <- [1..5]]
array (1,5) [(1,3),(2,4),(3,5),(4,6),(5,7)]
```

When arrays are printed out (via the show function), they are printed with an association list. For instance, in the first example, the association list says that the value of the array at 1 is 2, the value of the array at 2 is 4, and so on.

You can extract an element of an array using the `!` function, which takes an array and an index, as in:

```
Arrays> (listArray (1,5) [3,7,5,1,10]) ! 3
5
```

Moreover, you can update elements in the array using the // function. This takes an array and an association list and updates the positions specified in the list:

```
Arrays> (listArray (1,5) [3,7,5,1,10]) // [(2,99),(3,-99)]
array (1,5) [(1,3),(2,99),(3,-99),(4,1),(5,10)]
```

There are a few other functions which are of interest:

`bounds`	returns the bounds of an array
`indices`	returns a list of all indices of the array
`elems`	returns a list of all the values in the array in order
`assocs`	returns an association list for the array

If we define `arr` to be `listArray (1,5) [3,7,5,1,10]`, the result of these functions applied to `arr` are:

```
Arrays> bounds arr
(1,5)
Arrays> indices arr
[1,2,3,4,5]
Arrays> elems arr
[3,7,5,1,10]
Arrays> assocs arr
[(1,3),(2,7),(3,5),(4,1),(5,10)]
```

7.8 Layout

Chapter 8

Advanced Types

As you've probably ascertained by this point, the type system is integral to Haskell. While this chapter is called "Advanced Types", you will probably find it to be more general than that and it must not be skipped simply because you're not interested in the type system.

8.1 Datatypes

We've already seen datatypes used in a variety of contexts. This section concludes some of the discussion and introduces some of the common datatypes in Haskell.

8.2 Type Synonyms

Type synonyms exist in Haskell simply for convenience: their removal would not make Haskell any less powerful.

Consider the case when you are constantly dealing with lists of three-dimensional points. For instance, you might have a function with type $[(Double, Double, Double)] \rightarrow Double \rightarrow [(Double, Double, Double)]$. Since you are a good software engineer, you want to place type signatures on all your top-level functions. However, typing $[(Double, Double, Double)]$ all the time gets very tedious. To get around this, you can define a type synonym:

```
type List3D = [(Double,Double,Double)]
```

Now, the type signature for your functions may be written $List3D \rightarrow Double \rightarrow List3D$.

We should note that type synonyms cannot be self-referential. That is, you cannot have:

```
type BadType = Int -> BadType
```

This is because this is an "infinite type." Since Haskell removes type synonyms very early on, any instance of *BadType* will be replaced by $Int \rightarrow BadType$, which will result in an infinite loop.

Type synonyms can also be parameterized. For instance, you might want to be able to change the types of the points in the list of 3D points. For this, you could define:

```
type List3D a = [(a,a,a)]
```

Then your references to $[(Double, Double, Double)]$ would become $List3D\ Double$.

8.3 Newtypes

Consider the problem in which you need to have a type which is very much like `Int`, but its ordering is defined differently. Perhaps you wish to order `Int`s first by even numbers then by odd numbers (that is, all odd numbers are greater than any even number and within the odd/even subsets, ordering is standard).

Unfortunately, you cannot define a new instance of `Ord` for `Int` because then Haskell won't know which one to use. What you want is to define a type which is *isomorphic* to `Int`.

> ■ NOTE ■ "Isomorphic" is a common term in mathematics which basically means "structurally identical." For instance, in graph theory, if you have two graphs which are identical except they have different labels on the nodes, they are isomorphic. In our context, two types are isomorphic if they have the same underlying structure.

One way to do this would be to define a new datatype:

```
data MyInt = MyInt Int
```

We could then write appropriate code for this datatype. The problem (and this is very subtle) is that this type is not truly isomorphic to `Int`: it has one more value. When we think of the type `Int`, we usually think that it takes all values of integers, but it really has one more value: $\perp$ (pronounced "bottom"), which is used to represent erroneous or undefined computations. Thus, `MyInt` has not only values $MyInt 0$, $MyInt 1$ and so on, but also $MyInt \perp$. However, since datatypes can themselves be undefined, it has an additional value: $\perp$ which differs from $MyInt \perp$ and this makes the types non-isomorphic. (See Section 11.1 for more information on bottom.)

Disregarding that subtlety, there may be efficiency issues with this representation: now, instead of simply storing an integer, we have to store a pointer to an integer and have to follow that pointer whenever we need the value of a `MyInt`.

To get around these problems, Haskell has a **newtype** construction. A **newtype** is a cross between a datatype and a type synonym: it has a constructor like a datatype, but it can have only one constructor and this constructor can have only one argument. For instance, we can define:

```
newtype MyInt = MyInt Int
```

But we cannot define any of:

```
newtype Bad1 = Bad1a Int | Bad1b Double
newtype Bad2 = Bad2 Int Double
```

Of course, the fact that we cannot define `Bad2` as above is not a big issue: we can simply define the following by pairing the types:

```
newtype Good2 = Good2 (Int,Double)
```

Now, suppose we've defined `MyInt` as a **newtype**. This enables use to write our desired instance of `Ord` as:

```
instance Ord MyInt where
  MyInt i < MyInt j
      | odd i  && odd  j = i < j
      | even i && even j = i < j
      | even i           = True
      | otherwise        = False
      where odd x = (x `mod` 2) == 0
            even  = not . odd
```

Like datatype, we can still derive classes like `Show` and `Eq` over newtypes (in fact, I'm implicitly assuming we have derived `Eq` over `MyInt` – where is my assumption in the above code?).

Moreover, in recent versions of GHC (see Section 2.2), on newtypes, you are allowed to derive *any* class of which the base type (in this case, `Int`) is an instance. For example, we could derive `Num` on `MyInt` to provide arithmetic functions over it.

Pattern matching over newtypes is exactly as in datatypes. We can write constructor and destructor functions for `MyInt` as follows:

```
mkMyInt i = MyInt i
unMyInt (MyInt i) = i
```

8.4 Classes

We have already encountered type classes a few times, but only in the context of previously existing type classes. This section is about how to define your own. We will begin the discussion by talking about Pong and then move on to a useful generalization of computations.

8.4.1 Pong

The discussion here will be motivated by the construction of the game Pong (see Appendix ?? for the full code). In Pong, there are three things drawn on the screen: the two paddles and the ball. While the paddles and the ball are different in a few respects, they share many commonalities, such as position, velocity, acceleration, color, shape, and so on. We can express these commonalities by defining a class for Pong entities, which we call `Entity`. We make such a definition as follows:

```
class Entity a where
    getPosition :: a -> (Int,Int)
    getVelocity :: a -> (Int,Int)
    getAcceleration :: a -> (Int,Int)
    getColor :: a -> Color
    getShape :: a -> Shape
```

This code defines a typeclass `Entity`. This class has five methods: `getPosition`, `getVelocity`, `getAcceleration`, `getColor` and `getShape` with the corresponding types.

The first line here uses the keyword **class** to introduce a new typeclass. We can read this typeclass definition as "There is a typeclass 'Entity'; a type 'a' is an instance of Entity if it provides the following five functions: . . . ". To see how we can write an instance of this class, let us define a player (paddle) datatype:

```
data Paddle =
   Paddle { paddlePosX, paddlePosY,
            paddleVelX, paddleVelY,
            paddleAccX, paddleAccY :: Int,
            paddleColor :: Color,
            paddleHeight :: Int,
            playerNumber :: Int }
```

Given this data declaration, we can define `Paddle` to be an instance of `Entity`:

```
instance Entity Paddle where
  getPosition p = (paddlePosX p, paddlePosY p)
  getVelocity p = (paddleVelX p, paddleVelY p)
  getAcceleration p = (paddleAccX p, paddleAccY p)
  getColor = paddleColor
  getShape = Rectangle 5 . paddleHeight
```

The actual Haskell types of the class functions all have included the context `Entity a =>`. For example, `getPosition` has type `Entity a => a -> (Int,Int)`. However, it will turn out that many of our routines will need entities to also be instances of `Eq`. We can therefore choose to make `Entity` a subclass of Eq: namely, you can only be an instance of Entity if you are already an instance of Eq. To do this, we change the first line of the class declaration to:

```
class Eq a => Entity a where
```

Now, in order to define Paddles to be instances of Entity we will first need them to be instances of Eq – we can do this by deriving the class.

8.4.2 Computations

Let's think back to our original motivation for defining the Maybe datatype from Section 4.4.5. We wanted to be able to express that functions (i.e., computations) can fail.

Let us consider the case of performing search on a graph. Allow us to take a small aside to set up a small graph library:

```
data Graph v e = Graph [(Int,v)] [(Int,Int,e)]
```

The Graph datatype takes two type arguments which correspond to vertex and edge labels. The first argument to the Graph constructor is a list (set) of vertices; the second is the list (set) of edges. We will assume these lists are always sorted and that each vertex has a unique id and that there is at most one edge between any two vertices.

Suppose we want to search for a path between two vertices. Perhaps there is no path between those vertices. To represent this, we will use the Maybe datatype. If it succeeds, it will return the list of vertices traversed. Our search function could be written (naively) as follows:

```
search :: Graph v e -> Int -> Int -> Maybe [Int]
search g@(Graph vl el) src dst
    | src == dst = Just [src]
    | otherwise  = search' el
    where search' [] = Nothing
          search' ((u,v,_):es)
              | src == u  =
                case search g v dst of
                  Just p  -> Just (u:p)
                  Nothing -> search' es
              | otherwise = search' es
```

This algorithm works as follows (try to read along): to search in a graph g from src to dst, first we check to see if these are equal. If they are, we have found our way and just return the trivial solution. Otherwise, we want to traverse the edge-list. If we're traversing the edge-list and it is empty, we've failed, so we return Nothing. Otherwise, we're looking at an edge from u to v. If u is our source, then we consider this step and recursively search the graph from `v` to `dst`. If this fails, we try the rest of the edges; if this succeeds, we put our current position before the path found and return. If `u` is not our source, this edge is useless and we continue traversing the edge-list.

This algorithm is terrible: namely, if the graph contains cycles, it can loop indefinately. Nevertheless, it is sufficent for now. Be sure you understand it well: things only get more complicated.

Now, there are cases where the `Maybe` datatype is not sufficient: perhaps we wish to include an error message together with the failure. We could define a datatype to express this as:

```
data Failable a = Success a | Fail String
```

Now, failures come with a failure string to express what went wrong. We can rewrite our search function to use this datatype:

```
search2 :: Graph v e -> Int -> Int -> Failable [Int]
search2 g@(Graph vl el) src dst
    | src == dst = Success [src]
    | otherwise  = search' el
    where search' [] = Fail "No path"
          search' ((u,v,_):es)
              | src == u  =
                case search2 g v dst of
                  Success p -> Success (u:p)
                  _         -> search' es
              | otherwise = search' es
```

This code is a straightforward translation of the above.

There is another option for this computation: perhaps we want not just one path, but all possible paths. We can express this as a function which returns a list of lists of vertices. The basic idea is the same:

```
search3 :: Graph v e -> Int -> Int -> [[Int]]
search3 g@(Graph vl el) src dst
    | src == dst = [[src]]
    | otherwise  = search' el
    where search' [] = []
          search' ((u,v,_):es)
              | src == u  = (map (u:) (search3 g v dst)) ++ search' es
              | otherwise = search' es
```

The code here has gotten a little shorter, thanks to the standard prelude `map` function, though it is essentially the same.

We may ask ourselves what all of these have in common and try to gobble up those commonalities in a class. In essense, we need some way of representing success and some way of representing failure. Furthermore, we need a way to combine two successes (in the first two cases, the first success is chosen; in the third, they are strung together). Finally, we need to be able to augment a previous success (if there was one) with some new value. We can fit this all into a class as follows:

```
class Computation c where
    success :: a -> c a
    failure :: String -> c a
    augment :: c a -> (a -> c b) -> c b
    combine :: c a -> c a -> c a
```

In this class declaration, we're saying that `c` is an instance of the class `Computation` if it provides four functions: `success`, `failure`, `augment` and `combine`. The `success` function takes a value of type `a` and returns it wrapped up in `c`, representing a successful computation. The `failure` function takes a `String` and returns a computation representing a failure. The `combine` function takes two previous computation and produces a new one which is the combination of both. The `augment` function is a bit more complex.

The `augment` function takes some previously given computation (namely, `c a`) and a function which takes the value of that computation (the `a`) and returns a `b` and produces a `b` inside of that computation. Note that in our current situation, giving `augment` the type `c a -> (a -> a) -> c a` would have been sufficient, since `a` is always `[Int]`, but we make it this more general time just for generality.

How `augment` works is probably best shown by example. We can define `Maybe`, `Failable` and `[]` to be instances of `Computation` as:

```
instance Computation Maybe where
    success = Just
    failure = const Nothing
    augment (Just x) f = f x
    augment Nothing  _ = Nothing
    combine Nothing y = y
    combine x _ = x
```

Here, success is represented with `Just` and `failure` ignores its argument and returns `Nothing`. The `combine` function takes the first success we found and ignores the rest. The function `augment` checks to see if we succeeded before (and thus had a `Just` something) and, if we did, applies `f` to it. If we failed before (and thus had a `Nothing`), we ignore the function and return `Nothing`.

```
instance Computation Failable where
    success = Success
    failure = Fail
    augment (Success x) f = f x
    augment (Fail s) _ = Fail s
    combine (Fail _) y = y
    combine x _ = x
```

These definitions are obvious. Finally:

```
instance Computation [] where
    success a = [a]
    failure = const []
    augment l f = concat (map f l)
    combine = (++)
```

Here, the value of a successful computation is a singleton list containing that value. Failure is represented with the empty list and to combine previous successes we simply catentate them. Finally, augmenting a computation amounts to mapping the function across the list of previous computations and concatentate them. we apply the function to each element in the list and then concatenate the results.

Using these computations, we can express all of the above versions of search as:

```
searchAll g@(Graph vl el) src dst
    | src == dst = success [src]
    | otherwise  = search' el
    where search' [] = failure "no path"
          search' ((u,v,_):es)
              | src == u  = (searchAll g v dst `augment` (success . (u:)))
                            `combine` search' es
              | otherwise = search' es
```

In this, we see the uses of all the functions from the class `Computation`.

If you've understood this discussion of computations, you are in a very good position as you have understood the concept of *monads*, probably the most difficult concept in Haskell. In fact, the `Computation` class is almost exactly the `Monad` class, except that `success` is called `return`, `failure` is called `fail` and `augment` is called `>>=` (read "bind"). The `combine` function isn't actually required by monads, but is found in the `MonadPlus` class for reasons which will become obvious later.

If you didn't understand everything here, read through it again and then wait for the proper discussion of monads in Section 9.

8.5 Instances

We have already seen how to declare instances of some simple classes; allow us to consider some more advanced classes here. There is a `Functor` class defined in the `Functor` module.

> ■ NOTE ■ The name "functor", like "monad" comes from category theory. There, a functor is like a function, but instead of mapping elements to elements, it maps structures to structures.

The definition of the functor class is:

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

The type definition for `fmap` (not to mention its name) is very similar to the function `map` over lists. In fact, `fmap` is essentially a generalization of `map` to arbitrary structures (and, of course, lists are already instances of `Functor`). However, we can also define other structures to be instances of functors. Consider the following datatype for binary trees:

```
data BinTree a = Leaf a
               | Branch (BinTree a) (BinTree a)
```

We can immediately identify that the `BinTree` type essentially "raises" a type `a` into trees of that type. There is a naturally associated functor which goes along with this raising. We can write the instance:

```
instance Functor BinTree where
    fmap f (Leaf a) = Leaf (f a)
    fmap f (Branch left right) =
        Branch (fmap f left) (fmap f right)
```

Now, we've seen how to make something like `BinTree` an instance of `Eq` by using the **deriving** keyword, but here we will do it by hand. We want to make `BinTree` as instances of `Eq` but obviously we cannot do this unless `a` is itself an instance of `Eq`. We can specify this dependence in the instance declaration:

```
instance Eq a => Eq (BinTree a) where
    Leaf a == Leaf b = a == b
    Branch l r == Branch l' r' = l == l' && r == r'
    _ == _ = False
```

The first line of this can be read "if `a` is an instance of `Eq`, then `BinTree a` is also an instance of `Eq`". We then provide the definitions. If we did not include the "`Eq a =>`" part, the compiler would complain because we're trying to use the `==` function on `a`s in the second line.

The "`Eq a =>`" part of the definition is called the "context." We should note that there are some restrictions on what can appear in the context and what can appear in the declaration. For instance, we're not allowed to have instance declarations that don't contain type constructors on the right hand side. To see why, consider the following declarations:

```
class MyEq a where
    myeq :: a -> a -> Bool

instance Eq a => MyEq a where
    myeq = (==)
```

As it stands, there doesn't seem to be anything wrong with this definition. However, if elsewhere in a program we had the definition:

```
instance MyEq a => Eq a where
    (==) = myeq
```

In this case, if we're trying to establish if some type is an instance of `Eq`, we could reduce it to trying to find out if that type is an instance of `MyEq`, which we could in turn reduce to trying to find out if that type is an instance of `Eq`, and so on. The compiler protects itself against this by refusing the first instance declaration.

This is commonly known as the *closed-world assumption*. That is, we're assuming, when we write a definition like the first one, that there won't be any declarations like the second. However, this assumption is invalid because there's nothing to prevent the second declaration (or some equally evil declaration). The closed world assumption can also bite you in cases like:

```
class OnlyInts a where
    foo :: a -> a -> Bool

instance OnlyInts Int where
    foo == (==)

bar :: OnlyInts a => a -> Bool
bar = foo 5
```

We've again made the closed-world assumption: we've assumed that the only instance of `OnlyInts` is `Int`, but there's no reason another instance couldn't be defined elsewhere, ruining our defintion of `bar`.

8.6 Kinds

Let us take a moment and think about what types are available in Haskell. We have simple types, like `Int`, `Char`, `Double` and so on. We then have type constructors like `Maybe` which take a type (like `Char`) and produce a new type, `Maybe Char`. Similarly, the type constructor `[]` (lists) takes a type (like `Int`) and produces `[Int]`. We have more complex things like `->` (function arrow) which takes *two* types (say `Int` and `Bool`) and produces a new type `Int -> Bool`.

In a sense, these types themselves have type. Types like `Int` have some sort of basic type. Types like `Maybe` have a type which takes something of basic type and returns something of basic type. And so forth.

Talking about the types of types becomes unwieldy and highly ambiguous, so we call the types of types "kinds." What we have been calling "basic types" have kind "`*`". Something of kind `*` is something which can have an actual value. There is also a single kind constructor, `->` with which we can build more complex kinds.

Consider `Maybe`. This takes something of kind `*` and produces something of kind `*`. Thus, the kind of `Maybe` is `* -> *`. Recall the definition of `Pair` from Section 4.4.1:

```
data Pair a b = Pair a b
```

Here, `Pair` is a type constructor which takes two arguments, each of kind `*` and produces a type of kind `*`. Thus, the kind of `Pair` is `* -> (* -> *)`. However, we again assume associativity so we just write `* -> * -> *`.

Let us make a slightly strange datatype definition:

```
data Strange c a b = MkStrange (c a) (c b)
```

Before we analyze the kind of `Strange`, let's think about what it does. It is essentially a pairing constructor, though it doesn't pair actual elements, but elements within another constructor. For instance, think of `c` as `Maybe`. Then `MkStrange` pairs `Maybe`s of the two types `a` and `b`. However, `c` need not be `Maybe` but could instead by `[]`, or many other things.

What do we know about `c`, though? We know that it must have kind `* -> *`. This is because we have `c a` on the right hand side. The type variables `a` and `b` each have kind `*` as before. Thus, the kind of `Strange` is `(* -> *) -> * -> * -> *`. That is, it takes a constructor (`c`) of kind `* -> *` together with two types of kind `*` and produces something of kind `*`.

A question may arise regarding how we know `a` has kind `*` and not some other kind `k`. In fact, the inferred kind for `Strange` is `(k -> *) -> k -> k -> *`. However, this requires polymorphism on the kind level, which is too complex, so we make a default assumption that `k = *`.

> ■ NOTE ■ There are extensions to GHC which allow you to specify the kind of constructors directly. For instance, if you wanted a different kind, you could write this explicitly:
>
> ```
> data Strange (c :: (* -> *) -> *) a b = MkStrange (c a) (c b)
> ```
>
> to give a different kind to `Strange`.

The notation of kinds suggests that we can perform partial application, as we can for functions. And, in fact, we can. For instance, we could have:

```
type MaybePair = Strange Maybe
```

The kind of `MaybePair` is, not surprisingly, `* -> * -> *`.

We should note here that all of the following definitions are acceptable:

```
type MaybePair1     = Strange Maybe
type MaybePair2 a   = Strange Maybe a
type MaybePair3 a b = Strange Maybe a b
```

These all appear to be the same, but they are in fact not identical as far as Haskell's type system is concerned. The following are all valid type definitions using the above:

```
type MaybePair1a = MaybePair1
type MaybePair1b = MaybePair1 Int
type MaybePair1c = MaybePair1 Int Double

type MaybePair2b = MaybePair2 Int
type MaybePair2c = MaybePair2 Int Double

type MaybePair3c = MaybePair3 Int Double
```

But the following are *not* valid:

```
type MaybePair2a = MaybePair2

type MaybePair3a = MaybePair3
type MaybePair3b = MaybePair3 Int
```

This is because while it is possible to partially apply type constructors on datatypes, it is not possible on type synonyms. For instance, the reason `MaybePair2a` is invalid is because `MaybePair2` is defined as a type synonym with one argument and we have given it none. The same applies for the invalid `MaybePair3` definitions.

8.7 Class Hierarchies

8.8 Default

what is it?

Chapter 9

Monads

Learning about monads is the hardest thing you will have to do when learning Haskell. I will distinguish two different subcomponents of learning about monads: (1) learning how to use already-existing monads and (2) learning how to write your own. If you want to use Haskell, you must be able to use already-existing monads. Only if you want to be a super Haskell guru must you learn how to write your own. However, if you do get the hang of writing your own monads, your life will become a lot less painful.

So far we've seen two uses of monads. The first was IO actions. We've seen that by using monads, we can abstract away from the problems plaguing both the RealWorld and CPS solutions presented in Section 5. The second was using monads to represent different types of computations in Section 8.4.2. Overall, we needed a way to represent sequencing operations and we saw that a sufficient definition (at least for computations) was:

```
class Computation c where
    success :: a -> c a
    failure :: String -> c a
    augment :: c a -> (a -> c b) -> c b
    combine :: c a -> c a -> c a
```

Let us see if this will enable us to also do IO. Essentially, we need a way to represent taking a value out of an action and performing some new operation on it (as in the example from Section 4.3.3, rephrased slightly):

```
main =
    do s <- readFile "somefile"
       putStrLn (show (f s))
```

But this is exactly what `augment` does. In fact, using `augment`, we can write the above code as:

```
main =
```

```
readFile "somefile" `augment` \s ->
putStrLn (show (f s))
```

This certainly seems to be sufficient. And, in fact, it turns out to be more than sufficient.

9.1 Definition

The definition of a monad is a slightly trimmed down version of our `Computation` class. It contains four methods, but one of them can be defined using another:

```
class Monad m where
    fail    :: String -> m a
    return  :: a -> m a
    (>>=)   :: m a -> (a -> m b) -> m b
    (>>)    :: m a -> m b -> m b
```

In this definition, `fail` is equivalent to our `failure`; `return` is equivalent to our `success`; and `>>=` (read: "bind") is equivalent to our `augment`. The `>>` (read: "then") is simply a version of `>>=` which ignores the `a`. This will turn out to be useful, although it can be defined in terms of `>>=`.

There are three rules which all monads must obey (and it is up to *you* to ensure your monads obey these rules) called the "Monad Laws":

1. `return a >>= f` ≡ `f a`
2. `f >>= return` ≡ `f`
3. `f >>= (\x -> g x >>= h)` ≡ `(f >>= g) >>= h`

Let's look at each of these individually.

Law 1

This states that `return a >>= f` ≡ `f a`. Suppose we think about monads as computations. This means that if we create a trivial computation which simply returns the value `a` irregardless of anything else going on (this is the `return a` part) and then bind it together with some other computation `f`, then this is equivalent to simply performing the computation `f` on `a` directly.

For example, suppose `f` is the function `putStrLn` and `a` is the string "Hello World". This states binding a computation whose result is "Hello World" to `putStrLn` is the same as simply printing it to the screen. This seems to make sense.

Law 2

This states that `f >>= return` $\equiv$ `f`. That is, `f` is some computation and the law states that if we perform the computation `f` and then pass the result on to the trivial `return` function, then this is the same as simply performing the computation.

This should be obvious. Think of `f` as `getLine` (reads a string from the keyboard). This states that reading a string and then returing the value read is exactly the same as just reading the string.

Law 3

This states that `f >>= (\x -> g x >>= h)` $\equiv$ `(f >>= g) >>= h`. This is not as easy to grasp at first glance as the other two. It is essentially a commutative law for monads.

> ■ NOTE ■ Outside the world of monads, a function $\cdot$ is commutative if $(f \cdot g) \cdot h = f \cdot (g \cdot h)$. For instance, `+` and `*` are commutative, since bracketing on these functions doesn't make a difference. On the other hand, `-` and `/` are not commutative since, for example, $5 - (3 - 1) \neq (5 - 3) - 1$.

If we throw away the messiness with the lambdas, we get that this law states: `f >>= (g >>= h)` $\equiv$ `(f >>= g) >> h`. The intuition behind this law is that when we're string together actions, it doesn't matter how we group them.

For a concrete example, take `f` to be `getLine`. Take `g` to be an action which takes a value as input, prints it to the screen and reads another string via `getLine` and then returns that string. Take `h` to be `putStrLn`.

Let's consider what `(\x -> g x >>= h)` does. It takes a value called `x`, and runs `g` on it, feeding the results into `h`. In this instance, this means that it's going to take a value, print it, read another value and then print that. Thus, the entire left hand side of the law reads a string and then does what we've just described.

On the other hand, consider `(f >>= g)`. This action reads a string from the keyboard, prints it, and then reads another string, returning that newly read string as a result. When we bind this with `h` as on the right hand side of the law, we get an action which does this, and then prints the results.

Clearly, these two actions are the same.

While this explanation is quite complicated, and the text of the law is also quite complicated, the actual meaning is simple: if we have three actions and we compose them in the same order, it doesn't matter where we put the parentheses. The rest is just notation.

9.2 Do Notation

We have hinted that there is a connection between monads and the **do** notation. Here, we make that relationship concrete. As it turns out, there is nothing special about the **do** notation. It is simply syntactic sugar for monadic operations.

As we mentioned earlier, using our `Computation` class, we could define our above program as:

```
main =
    readFile "somefile" `augment` \s ->
    putStrLn (show (f s))
```

But we now know that `augment` is called `>>=` in the monadic world. Thus, this program really reads (and this is completely valid Haskell at this point: if you defined a function `f :: Show a => String -> a`, you could compile and run this program):

```
main =
    readFile "somefile" >>= \s ->
    putStrLn (show (f s))
```

This suggests that we can translate:

```
  do x <- f
     g x
```

into `f >>= \x -> g x`. This is exactly what the compiler does. Talking about **do** becomes easier if we abstract away from layout (see Section ?? for how to do this). There are four translation rules:

1. `do {e}` → `e`
2. `do {e; es}` → `e >> do {es}`
3. `do {let decls; es}` → `let decls in do {es}`
4. `do {p <- e; es}` → `let ok p = do {es} ; ok _ = fail "..." in e >>= ok`

Again, we will take these one at a time.

Translation Rule 1

The first translation rule, `do {e}` → `e`, states (as we have stated before) that when performing a single action, having a **do** or not is irrelevant. This is essentially the base case for an inductive definition of **do**. The base case has one action (namely `e` here); the other three translation rules handle the cases where there is more than one action.

Translation Rule 2

This states that `do {e; es}` $\rightarrow$ `e >> do {es}`. This tells us what to do if we have an action (`e`) followed by a list of actions (`es`). Here, we make use of the `>>` function defined in monads. This is exactly the same function as `>>=` except that it ignores the `a` inside the first action. All this rule says is that to **do** `e; es`, first we perform the action `e`, throw away the result and then **do** `es`.

For instance, if `e` is `putStrLn s` for some string `s`, then the translation of `do {e; es}` is to perform `e` (i.e., print the string) and then **do** `es`. This is clearly what we want.

Translation Rule 3

This states that hsdo {let decls; es} $\rightarrow$ `let decls in do {es}`. This basically tells us how to deal with **let**s inside of a **do** statement. All we do is lift the declarations within the **let** out and **do** whatever comes after the declarations.

Translation Rule 4

This states that `do {p <- e; es}` $\rightarrow$ `let ok p = do {es} ; ok _ = fail "..." in e >>= ok`. Again, it is not exactly obvious what is going on here. However, an alternative formulation of this rule which is roughly equivalent is: `do {p <- e; es}` $\rightarrow$ `e >>= \p -> es`. Here, it is clear what is happening. We run the action `e`, and then send the results into `es`, but first give the result the name `p`.

The reason for the complex definition is that `p` doesn't need to simply be a variable; it could be some complex pattern. For instance, the following is valid code:

```
foo = do ('a':'b':'c':x:xs) <- getLine
         putStrLn (x:xs)
```

In this, we're assuming that the results of the action `getLine` will begin with the string "abc" and will have at least one more character. The question becomes what should happen if this pattern match fails. The compiler could simply throw an error like usual for failed pattern matches, but since we're within a monad, we have access to a special `fail` function, and we'd prefer to fail using that function, rather than the "catch all" `error` function. Thus, the translation as defined allows the compiler to fill in the `...` with an appropriate error message about the pattern matching having failed. Other than that, though, the two definitions are equivalent.

9.3 A Simple State Monad

9.4 MonadPlus

Chapter 10

Input/Output

Chapter 11

Theory

11.1 Bottom

Chapter 12

For Further Information

Appendix A

Brief Complexity Theory

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 that simply $\mathcal{O}(n^2)$ and there are ones that run more quickly than $\mathcal{O}(n)$.

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.

Appendix B

Search Algorithms

Appendix C

The Minimax Algorithm

Appendix D

“Real World” Examples

In this section we have attempted to include the code for some real world applications written in Haskell. We have attempted to comment the code sufficiently that it can be understood with only the knowledge of Haskell which comes from this tutorial. The code for all of these programs is available off the tutorial website `http://www.isi.edu/ hdaume/htut/` (.)

Appendix E

Solutions To Exercises

Solution 1 It binds more tightly; actually, function application binds more tightly than anything else. To see this, we can do something like:

```
Prelude> sqrt 3 * 3
5.19615
```

If multiplication bound more tightly, the result would have been 9.

Solution 2 Solution: snd (fst ((1,'a'),"foo")). This is because first we want to take the first half the the tuple: (1,'a') and then out of this we want to take the second half, yielding just 'a'.

If you tried fst (snd ((1,'a'),"foo")) you will have gotten a type error. This is because the application of snd will leave you with fst "foo". However, the string "foo" isn't a tuple, so you cannot apply fst to it.

Solution 3 Solution: map isLower "aBCde"

Solution 4 Solution: length (filter isLower "aBCde")

Solution 5 foldr max 0 [5,10,2,8,1]. You could also use foldl. The foldr case is easier to explain: we replace each cons with an application of max and the empty list with 0. Thus, the inner-most application will take the maximum of 0 and the last element of the list (if it exists). Then, the next-most inner application will return the maximum of whatever was the maximum before and the second-to-last element. This will continue on, carrying to current maximum all the way back to the beginning of the list.

In the foldl case, we can think of this as looking at each element in the list in order. We start off our "state" with 0. We pull off the first element and check to see if it's bigger than our current state. If it is, we replace our current state with that number and the continue. This happens for each element and thus eventually returns the maximal element.

Solution 6 fst (head (tail [(5,'b'),(1,'c'),(6,'a')]))

Index