Haskell Tutorial for C Programmers

Eric Etheridge

July, 2005

Contents

Chapter 1

Introduction

1.1 Abstract

Many people are accustomed to imperative languagues, which include C, C++, Java, Python, and Pascal. For computer science students, Haskell is weird and obtuse. This tutorial assumes that the reader is familiar with C/C++, Python, Java, or Pascal. I am writing for you because it seems that no other tutorial was written to help students overcome the difficulty of moving from C/C++, Java, and the like to Haskell.

I write this assuming that you have checked out the Gentle Introduction to Haskell, but still don't understand what's going on.

Haskell is not 'a little different,' and will not 'take a little time.' It is very different and you cannot simply pick it up, although I hope that this tutorial will help.

If you play around with Haskell, do not merely write toy programs. Simple problems will not take advantage of Haskell's power. Its power shines mostly clearly when you attack difficult tasks. Haskell's tools dramatically simplify your code.

I am going to put many pauses in this tutorial because learning Haskell hurt a lot, at least for me. I needed breaks, and my brain hurt while I was trying to understand.

Haskell has both more flexibility and more control than most languages. Nothing that I know of beats C's control, but Haskell has everything C does unless you need to control specific bytes in memory. So I call Haskell powerful, rather than just 'good.'

I wrote this tutorial because Haskell was very hard for me to learn, but now I love it. "Haskell is hard!" "You can't write code the way I know how!" "My brain hurts!" "There aren't any good references!" That's what I said when I was in college. There were good references, but they didn't cover the real problem: coders know C.

This abstract was pieced together by Mark Evans, here, from my own work. I have had no contact with Mark Evans, but since he did't contact me when he editted together this abstract from my work and posted it on lambda-the-ultimate, I doubt he'll care that I've taken that edit and used it as my abstract here. If he wishes, he may contact me regarding the legal status of this work. For now, I assume I still hold the copyright on all of it, including the abstract (but see the license section below).

1.2 Downloads

1.3 License

I've decided that I should be specific about the license for this tutorial. To sum up, you can do whatever you want with this tutorial as long as my name is still on it, including modifying it, redistributing it, or selling derivative works. **Specifically, you can use it to educate people in a commercial setting, such as in-house training or consulting.** I would love to hear that a company considering Haskell used this tutorial to train its workforce. So feel free. Any derivative works also carry this license, by the way. "Share alike". Thank you, Creative Commons. The link leads to the legal wording.

1.4 This Tutorial's Purpose and Other Online References

Many people are accustomed to imperative languagues, which include C, C++, Java, Python, and Pascal. In fact most languages in common usage are imperative, other than LISP, Scheme, ML, or OCaml. For computer science students, it is virtually guaranteed that Haskell is weird and obtuse. I first encountered Haskell in the classroom when I was a freshman at UT Austin, and then in another class two years later. I was only familiar with C/C++, Pascal, and QBASIC, and all of the Haskell tutorials and books seemed to assume more of my education. This tutorial assumes that the reader is familiar with C/C++, Python, Java, or Pascal. Specifically, this tutorial is for **students** of computer science, people in their first few years of college, or even in high school. I am writing for you because it seems that no other tutorial was written to help students overcome the difficulty of moving from C/C++, Java, and the like to Haskell.

I write this assuming that you have checked out the following tutorial, the Gentle Introduction to Haskell, but found that you still don't understand what's going on:

http://www.haskell.org/tutorial/

That tutorial is a good reference for basic syntax. In this tutorial we will skip most of that until later when it is clear why function definition is so important in Haskell. Here is another tutorial, the Tour of the Haskell Syntax, with much more specific information:

http://www.cs.uu.nl/ afie/haskell/tourofsyntax.html

Refer to it, too. It has the appropriate syntax for most of the things I discuss, without requiring you to know Haskell's depths. Please read both of these after, during, or even before reading this tutorial.

One of the best references is the source code for the Prelude, the file "Prelude.hs", where all of the general-purpose functions are defined. You can find it in Hugs's install directories someplace. If any function shows up that you don't understand, you can look up its definition and figure out what it's really doing. This is a very good practice for those unfamiliar with general Haskell use.

Another resource is the GHC Hierarchical Libraries documentation. These have the types and module names for everything in GHC, and most of the time have meaningful commentary:

http://www.haskell.org/ghc/docs/latest/html/libraries/index.html

There is also a newer version of that documentation for the 'newAPI' in the most recent CVS version of GHC. Since writing the first draft of this tutorial, HOpenGL moved to the main version, so this link is not necessary for finding HOpenGL documentation. However, I still include it because some of GHC's newest extensions, such as HOpenAL (note the 'AL') are here, as of 7/3/05:

http://www.haskell.org/HOpenGL/newAPI/

Finally, if you don't already have them, get and install both Hugs and GHC. Hugs will let you play around and learn, and GHC will be necessary for your more advanced tasks later on. GHCi (GHC interactive) comes with GHC, and is a close substitute for Hugs. It is a little harder (I think) to play around with. It may be equal in utility now, so check it out, too. If you use Debian, Hugs and GHC are packages. For everyone else, the homepages are here:

http://www.haskell.org/hugs/ http://www.haskell.org/ghc/

1.5 Relationship to Our Other Tutorials

You may be here because you're trying to use HOpenGL. The tutorial you're reading was written along with our other tutorial on rendering and texture examples in HOpenGL, and if you are familiar with Haskell you can go directly there:

Dave Morra's HOpenGL tutorial

Otherwise, read this tutorial and those mentioned above first, then check out our HOpenGL tutorial if you like. If we've written well enough you should be able to not only use Haskell but HOpenGL as well. On with the tutorial.

1.6 Preface and Style Notes

I am not writing a Haskell reference. This is a tutorial designed to take someone having trouble understanding Haskell and help them. This tutorial is for people who, like me, needed to learn enough concepts to understand the code covered in a classroom. Haskell allows things to be done easily and clearly, but it is not easy or clear, and it can be extremely challenging for a novice. You cannot pick up Haskell code and understand it. What I have attempted to write is a tutorial that covers the common aspects of Haskell that are the most obtuse.

As the tutorial progresses, one thing should become clear about Haskell: its real power comes into play when you attack difficult problems. Because of this, I use some difficult problems in this tutorial. Don't worry if you don't understand the solutions after reading the tutorial once. Haskell is not a toy language, and even a moderately sized set of functions will include several of Haskell's complicated tools all working together. This has left educators with a dilemma: do I use ridiculously simple code in order to cover a single topic at once, or do I use something actually useful and try to explain all the pieces and how they all fit together? Many tutorials and lessons have chosen the former, but I prefer the latter. That means that each example requires a lot of explaining. Often concepts must be explained once in extremely simplistic terms, and then explained again later after other related topics have also been briefly discussed. As you read this tutorial, remember this: Haskell's real power is the fact that all of its pieces fit so well together, not just that they are good pieces.

Chapter 2

Section I: What the Heck is Going On?

2.1 Part I: Haskell's Oddity

First of all, Haskell has no update operator. If that sentence did not make sense, then please keep reading, because this tutorial was written with you in mind. By 'update operator', I mean that the following does not happen in normal Haskell:

```
int a
a := 4
print a
a := 5
print a



> 4
> 5
```

The above programming style, i.e. 'making a variable, putting data in it, using it, then replacing the data in it, using it again' does not happen in normal Haskell. Those of you who have used LISP or Scheme will be familiar with this concept, but I am sure that the rest of you are probably baffled. Here is how Haskell works, again in pseudo-code:

```
print a

int a
a = 5

> 5
or
int a
```

```
a = 5

print a

> 5
```

The order to these operations does not matter. There is also a reason that the first example used ':=' and the second example used '='. In 'imperative' languages, storing data is an operation, and it happens in a sequence. In 'funtional' languages like Haskell, the equal sign means exactly that. In other words, each variable is equal to its value not only after the assignment statement is reached in sequence, but in fact at all points during execution.

Now, some of you may be saying, "That's nice, Eric, but what the `%$@!` good is a language where everything is hardcoded? Wouldn't I have to define every variable with its correct value as I coded? Isn't 'computing' values the whole point of a 'computer'?" And you would be right; knowing results ahead of time would make computing weird. The 'redeeming' feature of Haskell is that you don't need to store data to return a result.

I am going to put many pauses in this tutorial because learning Haskell hurt a lot, at least for me. I needed breaks, and my brain hurt while I was trying to understand. Anyway, let's look at that statement again: you don't need to store data to return a result. Here is an example of a function in C:

```
int foo (int bar) {
    int result;
    result = bar * 10 + 4;
    return result;
}
```

Now, the important part is the expression in the middle. It can also be written thus:

```
int foo (int bar) {
    return bar * 10 + 4;
}
```

These are the same, but the second is shorter, and also clearer. With a function like this, you could state the following: "The value of foo(x) is equal to (x * 10 + 4)." Or, more simply, "foo(x) = x * 10 + 4". I know you're saying, "Most functions aren't that simple." That is true, but bear with me. Haskell has much more powerful tools for writing functions that most other languages, and a lot of complicated operations look this simple in Haskell. The key to using those tools will be changing the way you think from 'make data, then alter it' to 'define function that would return result, then apply to inputs'.

2.2 Part II: Input and Output

We'll come back to the frame of mind later. Haskell is such a difference from C/C++ that many concepts only make sense in conjunction with others. I need to cover the basics of several concepts before I can explain each of them fully.

Moving on, the question on everybody's mind is probably either, "How does I/O work?" or "What are these tools?" IO is one of the complicated parts of Haskell, and later I'll describe how it works as well as a setup for programming with GHC. Until then, use GHCi or Hugs to try the examples. They have an interactive prompt where you type in expressions, like a function with its parameters. Then they print the evaluated result. Also, variable bindings such as 'a = 4' hang around while you're using them, so examples I use here should work just fine. To write your own functions, you need to write a Haskell source file and load it first. Using GHC itself requires knowing some of Haskell's more complicated tools, so we'll put it off until then.

To use Hugs and GHCi with your own functions, you have to write a Haskell source file and load it into the interpreter. Generally, this works as follows:

1. Open a text editor and write Haskell code.
2. Save that code as a file with the extension '.hs', for instance, 'test.hs'.
3. With the source file in the current directory, run Hugs or GHCi.
4. In Hugs or GHCi, at the prompt type ':l test.hs'. That's a lowercase 'L'.
5. Source code that needs modules, say Data.Maybe, should include 'import Data.Maybe' at the top.

Note that the module 'Prelude' is automatically imported. This module covers most basic elements of the language.

2.3 Part III: Very Basic Intro to Types

Moving on again, let's talk about those tools. Haskell's greatest strength lies in the power a programmer has to define useful functions easily and clearly. Again, our earlier example from C:

```
int foo (int bar) {
    return bar * 10 + 4;
}
```

In Haskell, to write this function named foo, you would write the following:

```
foo bar = bar * 10 + 4
```

That's all, except for the type:

```
foo :: Int -> Int
```

The type reads, "foo is of type Int to Int", meaning that it takes an Int and returns an Int. Together, you write:

```
foo :: Int -> Int
foo bar = bar * 10 + 4
```

Defining functions and types is the majority of the work in Haskell, and usually they require equal amounts of time. Haskell has a variety of ways of defining functions, and the tutorials mentioned above do not adequately introduce them. We will discuss them later, once we have some tools under our belt.

2.4 Part IV: Haskell's Lists and List Comprehensions

Those of you familiar with C know that pointers are the primary object in the language, and for almost all imperative languages, the most useful structure is the Array, a randomly accessible sequence of values (usually) stored in order in memory. Haskell has arrays, but the most-used object in Haskell is a list. A list in Haskell is accessible only at the front, and is not stored in order in memory. While this may sound atrocious, Haskell has such weird abilities that it is more natural to use a list than an array, and in fact faster. Let us start with the C code to compute fibonacci numbers starting with zero:

```
int fib (int n) {
    int a = 0, b = 1, i, temp;
    for (i = 0; i < n; i++) {
        temp = a + b;
        a = b;
        b = temp;
    }
    return a;
}
```

This is fine for computing a particular value, but things get ugly when you want to create the sequence:

```
int * fibArray(int n) {
    int * fibs;
    fibs = (int *)malloc((sizeof int) * n);
    for (i = 0; i < n; i++) {
        fibs[i] = a;
        temp = a + b;
        a = b;
        b = temp;
    }
    return fibs;
}
```

When I say 'get ugly', I mean that something is included in that function which shouldn't be there: the size of the list. The fibonacci sequence is infinite, and the code above does not represent it, only a part of it. This doesn't sound so bad, unless you don't know how many values you need initially.

In Haskell, 'fib', the function to compute a single fibonacci value, can be written as follows:

```
fib :: Int -> Int
fib n = fibGen 0 1 n

fibGen :: Int -> Int -> Int -> Int
fibGen a b n = case n of
    0 -> a
    n -> fibGen b (a + b) (n - 1)
```

This is a slight improvement over the C code, but not much. Note that the type of fibGen is "Int to Int to Int to Int", meaning that it takes three Ints and returns an Int. More on that later. Also note that this uses a recursive function. Recursion is everywhere in Haskell. Most of your 'looping' functions will involve recursion instead, unless they use more sophisticated tools like the one below. Don't worry, the GHC compiler optimizes the heck out of Haskell code. I wrote a Pascal compiler in Haskell, and it was faster than all of my classmate's C code, plus it was correct. End of side note.

The real improvement over C comes in defining the sequence:

```
fibs :: [Int]
fibs = 0 : 1 : [ a + b | (a, b) <- zip fibs (tail fibs)]
```

Don't be scared. Once you understand this function, you will understand at least half of the intracies of Haskell. Let's take it from the top. In Haskell, lists are written as follows:

```
[ 4, 2, 6, 7, 2 ]
```

This is the list of 4, then 2, then 6, etc. The ':' operator is used to compose a list by sticking a value on the front (left). For instance:

```
temp = 1 : [ 4, 2, 5 ]
```

is the list [1, 4, 2, 5].

That means that in the above code, fibs is a list of Int, and its first two values are zero and one. So far, so good, at least the first two values of 'fibs' will be right, if the thing even compiles. The next part looks really weird, though. It's a Haskell tool that really comes in handy called a 'list comprehension'. Instead of making space and then filling it with the right values, you can you define the right values. That's my statement from Part II, anyway. List comprehensions work like so:

```
[ func x | x <- list, boolFunc x ]
```

In the middle, there's a 'list', and it spits out values called x. These are the values of the list in order. If 'boolFunc x' is True, then x will get used in this new list. No boolFunc is in the 'fibs' example, but I include it here because it can also be extremely handy. 'func x' applies some function to the value of x, and the result is then put next in line in the final result, assuming 'boolFunc x' was true. Here's an example of a list and its use in some list comprehensions:

```
nums :: [Int]
nums = [ 4, 2, 6, 8, 5, 11 ]

[ x + 1 | x <- nums ]
= [ 5, 3, 7, 9, 6, 12 ]

[ x * x | x <- nums, x < 7 ]
= [ 16, 4, 36, 25 ]

[ 2 * x | x <- 9 : 1 : nums ]
= [ 18, 2, 8, 4, 12, 16, 10, 22 ]

[ "String" | x <- nums, x < 5 ]
= [ "String", "String" ]
```

Note that the order was preserved. This is very important for our example. Also, the type of the list comprehension was not necessarily the type of nums, nor did x actually have to be used in the function. Let's return to 'fibs'.

2.5 Part V: Making Sense of 'fibs', and Why Lazy Evaluation is Important

We were working on a definition for the list of Fibonacci numbers. Here is the example again:

```
fibs :: [Int]
fibs = 0 : 1 : [ a + b | (a, b) <- zip fibs (tail fibs)]
```

So what the heck is '(a, b)' and 'zip fibs (tail fibs)' and all that? Well, Haskell has a more expressive type system than most other languages. As in Python, '(a, b)' is a tuple, meaning two values stuck together. It's a convienent way to store and pass multiple values, much more so than structs. Just add parentheses and enough commas, and you pass the group of values around as you please. The only trick is that Haskell expects you to be consistent, and that means having the right type. Clearly, '(a, b)' is of type '(Int, Int)', or:

```
(a, b) :: (Int, Int)
```

Therefore 'zip fibs (tail fibs)' is of type '[(Int, Int)]', a list of 2-tuples of an Int and an Int or more clearly,

```
zip fibs (tail fibs) :: [(Int, Int)]
```

GHCi and Hugs will report these to you with the ':t' command, followed by a variable or function. This comes in handy.

So what is 'zip'? Its type and general meaning are given here:

Prelude, Section: Zipping and Unzipping Lists

'zip', as its name somewhat implies, takes two lists and 'zips' them together, returning a list of tuples, with the left member of each tuple being an item from the first (left) list, and likewise for the right.

```
zip [ 1, 3, 6, 2 ] [ "duck", "duck", "duck", "goose" ]
= [ (1, "duck"), (3, "duck"), (6, "duck"), (2, "goose")
```

And what about '(tail fibs)'? 'tail' is a pretty straightforward function: it chops off the first item of a list and returns the remainder. That statement can be slightly misleading. 'fibs' doesn't get altered by using 'tail' on it; as I said before, Haskell doesn't have an update operation. Instead, 'tail' just computes the proper result and returns it, rather than altering 'fibs' itself.

```
tail [ 10, 20, 30, 40, 50 ]
= [ 20, 30, 40, 50 ]
```

Well, that makes it seem like 'zip fibs (tail fibs)' probably has the right type, but what is it?

```
fibs :: [Int]
fibs = 0 : 1 : [ a + b | (a, b) <- zip fibs (tail fibs)]
```

The first paramater to zip is 'fibs', which is the list defined by the expression! What the heck? Can you do that? Yes, you can. See, 'fibs' is the entire list, including the 0 and 1 at the beginning. So the first two tuples in the list created by the zip function will have a 0 and then a 1 on their left. So what is 'zip fibs (tail fibs)'? Well, the first value is definitely (0, 1). Why? Because the first item in fibs is 0, and the first item in (tail fibs) is 1, the second item in fibs. So what's the second value in zip fibs (tail fibs)? It's (1, 1). Where did the right hand 1 come from? It's the third value in fibs, which we just computed. The first value of zip fibs (tail fibs) is (0, 1), which is '(a, b)' in the list comprehension, and so the first value in that comprehension is 0 + 1, or 1. That is also the third value in fibs, etc.

Did you catch all that? The definition of fibs is evaluating itself while it is computing itself. So why didn't some sort of error happen because of undefined values? The trick to all of this is Haskell's laziness. Also, the evaluation is always one step behind the computation, so evaluation can always proceed exactly as far as needed for the next computation. Finally, this list is infinite. Of course, no computer can hold an infinite amount of data. So how much is really there? The answer is simple:

until you read some values from fibs and print them, there's only the 0 and the 1, plus the function to generate more of the list. After you read some, fibs will be evaluated to that point, and no further. Since fibs is defined globally, it will remain defined in memory, making reading further values very quick. Try this with Hugs or GHCi and you see'll what I mean.

```
fibs !! 2
fibs !! 4
fibs !! 30
fibs !! 30
fibs !! 6
fibs !! 20
fibs !! 30
take 10 fibs
```

'!!' is the 'index' operator for lists in Haskell. It walks down the list and returns the nth item, zero-indexed like C/C++ and Python. 'take 10 fibs' will return the first 10 values of fibs. Be careful, fibs has infinite length. If you just type 'fibs', the output could go forever.

And why does the list only get evaluated as far as you print it? Haskell is 'lazy', meaning that it doesn't do work that it doesn't have to. C programmers know that the boolean operators '&&' and '||' are 'short-circuit', meaning that the right hand side is not evaluated unless it's needed. This allows for all kinds of neat tricks, like not dereferencing a null pointer. **The entire language of Haskell has this short-circuit behavior, including the functions that you write yourself**. This sounds strange, and will become even more important when we get to Haskell's tools.

This also brings us to one of the other odd things about Haskell: **It is often easier to code the general definition for something than to write a function that generates a specific value**. This is one of those things you have to get used to, and you will probably come back to it again. And again.

Well, give yourself a pat on the back. If you got all that, or at least you will after playing around in Hugs, then you understand about half of the common usages of Haskell. Ready for the other half? Maybe take a break, and play around a bit in Hugs or GHCi.

Chapter 3

Section II: Towards Functions

3.1 Part I: The Order of Operations as a Programmer

A programming note for recursive functions and Haskell:

Concerning the fib / fibGen example here:

```
fib :: Int -> Int
fib n = fibGen 0 1 n

fibGen :: Int -> Int -> Int -> Int
fibGen a b n = case n of
    0 -> a
    n -> fibGen b (a + b) (n - 1)
```

I wrote the type of fib first, then the type and definition of fibGen, then finally the definition of fib.

For those of you who are not accustomed to writing recursive functions, Haskell programming quite often requires them. Often these recursive functions need subsidiary functions, which either 'frame' the main operation of recursion, or perform a simple task that keeps the main recursion function clean. In either case, the subsidiary functions can be written later, after the major recursive operation is clearly defined including end conditions. It is generally a good idea to concentrate on the most crucial aspect of a piece of code when programming, but Haskell's design greatly reinforces this. The definition of subsidiary functions, such as the 'setup' where fib calls fibGen with parameters '0 1 n', can wait until the function itself has been written. This is true even though the type of fib was obvious from the beginning. Likewise, Haskell makes writing trivial functions like that so quick that they can generally be ignored while thinking about the larger picture. This is likely to change the way that you code, and probably in a good way.

3.2 Part II: Functions, But Really a Sidetrack to Types

As we move on, the other half of Haskell's general usage looms. This half is about functions.

So what is a function? As this tutorial's alignment indicates, we'll compare C/C++ to Haskell. In C, a function is a sequence of commands that have their own namespace, are called during execution and passed parameters, inherit the namespace of the scope in which they are written, and return a value to their caller. In Haskell, most of that is true, except of course functions in Haskell are not sequences of events, but expressions and definitions. There is a major difference between C and Haskell, however, and it concerns the amount of flexibility that functions have. In C, functions take parameters and return a single value. We've already seen that Haskell has many ways to group values, like several other languages.

The two most common of these are lists and tuples, and these can be the return type from a function. To sum them up, in Haskell lists are variable length and hold values of the same type, and tuples are fixed length and can hold values of different types. Here is an example of a function type that returns a tuple:

```
splitAt :: Int -> [a] -> ([a], [a])
```

'splitAt' takes an Int and a list and returns a tuple. The left value in the tuple is the first n values in the list, and the right value is the rest of the list, where n is the first parameter. This function is in the Prelude, and its description can be found here:

Prelude, Section: Sublists

We've already seen lists in a type:

```
fibs :: [Int]
```

Since the fibonacci numbers grow rapidly, but 'Int' is 32-bit, it would probably have been better to use 'Integer', Haskell's built-in infinite-precision integer storage.

```
fibs :: [Integer]
```

And this is a function type. It takes zero parameters and returns a list of Integers. This isn't a trick of Haskell's syntax, 'fibs' really is a function that, when evaluated will return numbers comprising the fibonacci sequence. That kind of logic is what lets Haskell's compilers make code run very fast, and lets Haskell programmers write code very efficiently and quickly.

3.3 Part III: More Types, Because Haskell Is 'Polymorphic'

It's time for a brief [not so brief] digression about types. As you've noticed, the trend seems to be to call everything a 'function'. And that's true. Take '4' for example. When you use a number '4' hardcoded into your code, it looks to you like the number 4. But what is it to Haskell? Type ':t 4' into Hugs or GHCi. What do you get? You get some weird junk:

```
4 :: Num a => a
```

That looks like it's a function that's taking a parameter. It's not, and the key is the '=>' arrow rather than the '->' arrow. The type given is read: "four is of type 'a', where 'a' is in the class 'Num'." What's the class Num? Well, it's the class that all numbers belong to. The real answer is that Haskell has something C doesn't: true polymorphism. Most C++ programmers are familiar with the term 'overloading', which means that a function is defined for more than one set of parameter types. For instance, addition and multiplication in C/C++ are overloaded, allowing the following combinations to occur:

```
int a = 4, b = 5;
float x = 2.5, y = 7.0;

cout << a + b;  //9
cout << a + y;  //11
cout << y + a;  //11.0
cout << x + y;  //9.5
cout << b * a;  //20
cout << b * x;  //12.5
cout << y * b;  //35.0
cout << x * y;  //17.5
```

In C/C++, this is accomplished by defining all of the following overloaded definitions for '+' and '*':

```
operator+ (int, int);
operator+ (int, float);
operator+ (float, int);
operator+ (float, float);
operator* (int, int);
operator* (int, float);
operator* (float, int);
operator* (float, float);
```

The C compiler picks the appropriate type at compile time. The key here is that, in C/C++, each combination must be written separately. Also, in C/C++, any other function that uses either an int or a float must specify which one it expects, or must _itself_ be overloaded. This bring us to the idea of classes. For what types is '+' defined? In C/C++ it is possible to overload the operator for new types, but those new types will not be interchangeable with ints, floats, or other numeric types. For instance, sort functions such as mergeSort and quickSort would need to be rewritten to sort arrays of the new value. In constrast, here is the type of mergeSort in Haskell:

```
mergeSort :: Ord a => [a] -> [a]
```

What is going on? Again, there are two parameters, not three. The first thing that appears to be a parameter is actually a class restriction. 'mergeSort', as you would expect, takes a list of objects of some type (type 'a'), and returns a list of objects of the same type. So why is the following type not sufficient?:

```
mergeSortBadType :: [a] -> [a]
```

The reason is that at some point in mergeSort the items in the list will need to be compared to each other using a comparator such as '>', '<', '>=', or '<='. In Haskell, those operators are part of a class definition. The only values for which '>' and so on are defined are those which are members of class 'Ord', so named because an 'order' can be determined for them. Many numeric types are of type Ord, as are characters and strings. So mergeSort must clarify its type by stating that its parameter must be a list of objects for which '<' and so on are defined. It would also be okay to make the type more specific, but this is not necessary and generally bad technique.

And what about '4'? How come four is of type 'a', where 'a' is a member of class 'Num'? Can't it just be a Num? Or an Int? It can be an Int if we specifically say it is, like so:

```
a = (4 :: Int) + 2
```

Here '4' is an Int. That is how you specify the type of something inside of an expression. But without that, 4 is of type 'a', where 'a' is in class 'Num', or more simply, 4 is of type 'a' in class 'Num'. And that is important, because '+' is defined for all member types of class Num, meaning that '4' is definitely a legal parameter for this function:

```
doubleIt :: Num a => a -> a
doubleIt n = n + n
```

'-' and '*' are also defined for all member types of Num, so 4 is also allowed for this function:

```
fib :: Num a, Num b => a -> b
fib n = fibGen 0 1 n

fibGen :: Num a, Num b => b -> b -> a -> b
fibGen a b n = case n of
    0 -> a
    n -> fibGen b (a + b) (n - 1)
```

That is our first Haskell fib function, but with the types changed. The first type is read, "fib is of type 'a' to 'b', where 'a' is a member of class Num and 'b' is a member of class Num." There is only one '=>' arrow because there is only ever one section of the type that describes class restrictions. Why would we do this? Shouldn't we pick a type? Well, what if you worked on a group project, and two people need to calculate fibonacci numbers? And for reasons of their own, one needed an Int returned and the other needed an Integer? Or a Double? Would you write the code twice with different types? If you were using C you would. Picking the most general types allows code reuse. Class definitions allow code reuse.

Also notice that in the initial call to 'fibGen', the third parameter is 'n', the first parameter of 'fib', and that the types of 'fib' and 'fibGen' seem to make note of this. The reason I wrote 'fib' with a different return type from its parameter is that the following would be common:

```
fib :: Int -> Integer
```

We only need Int-sized storage of our counter variable, but we may need Integer-sized storage of the result. So, two separate types. Also notice how types flow in 'fibGen'. The math does not mix parameters of type 'a' and 'b', and a parameter of type 'b' is also used as the final return value. The types match not only externally but internally. Following types in this manner will be important for debugging.

Continuing onward, in the fib example we used 'tail'. Here is its type:

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

In C, tail would have to be reimplemented for every type of list you used. That sounds slightly contrived, so what about '!!', the index operator? Well, in most other languages, indexing a list is builtin, because it has to work for every kind. And so on. Everything in C is either overloaded, built in, or works for only one type. Well, there are a few exceptions, generally involving casting to or from '(void *)'. The point is, you're going to see 'Num a =>' at the beginning of type signatures, as well as 'a' and 'b' inside type signatures. 'a' and 'b' are type variables in this case, used by the compiler solely to determine proper types for compilation. Occassionally you will get messages such as 'can't determine type', or 'type mismatch'. The second means the you've done something wrong, but the first usually means that a type variable can't be pinned down to a single type for a function that you've written. This can happen for the simplest of reasons:

```
main = putStrLn (show 4)
```

'putStrLn' takes a string and puts it on the screen. 4 has a 'polymorphic' type, i.e. it is a member of a type class, not defined as a specific type. 'show' takes anything that can be turned into a string (basically), and so it doesn't specify a type for '4' either. This leaves the compiler in a quandry, because no specific type is indicated anywhere, and it will complain. To resolve it, add the type definition like so:

```
main = putStrLn (show (4 :: Int))
```

Or Integer, or Double, or whatever. This will be handy when you try to test generalized functions, and you'll need it in a few other weird cases as well.

One last note, you can define the type of multiple functions simultaneously:

```
addOne, subtractOne :: Int -> Int
```

This can be handy.

3.4 Part IV: Functions Already

But we were talking about functions. As you may have noticed, it seems like anything can work as a parameter or return value for a function. This is absolutely true, as long as the types match. For instance, let's take a look at the extremely useful 'map' function:

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

By now you can probably read that, strange as it may be. "map is of type function a to b followed by a list of type a and returns a list of type b". It's taking a function as a parameter. Not only that, but a polymorphic function, with no type restrictions. And look at the other two items. The function it takes is from a to b, and then it takes a list of type a and returns a list of type b. With a name like 'map', it's pretty clear what should happen when you use it:

```
fooList :: [Int]
fooList = [3, 1, 5, 4]

bar :: Int -> Int
bar n = n - 2

map bar fooList
= [1, -1, 3, 2]
```

Nothing to it. Notice I had to give at least 'fooList' or 'bar' a specific type, or Hugs and GHC would complain that the types could not be fully determined.

You can write functions which take functions as parameters. This can be fun, and also very useful. Now let's try something stranger:

```
subEachFromTen :: [Int] -> [Int]
subEachFromTen = map (10 -)
```

What the heck? First, there is absolutely supposed to be parentheses around the '-' and the '10'. Second, what does this do? Again, one step at a time. '(10 -)' is a function. It takes a number and returns ten minus that number. Use ':t' in Hugs or GHCi to find this out:

```
(10 -) :: Int -> Int
or rather

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

Second, 'map' takes a function as its first parameter. There's a reason that Haskell uses arrows to define types, rather than a parenthesized list. 'map', applied to '(10 -)' has the following type (again, check in Hugs and GHCi):

```
map (10 -) :: Num a => [a] -> [a]
```

It takes a list of Num members (all the same type of Num members, mind you) and returns a list of the same type (again, the same type member of Num). This is called 'partial evaluation'. You take a function, give it some of its parameters, and you've got a function that takes the rest of the parameters. You can even name this 'in-between' state, since it is just a function like anything else. So, here is 'subEachFromTen' in action:

```
subEachFromTen [4, 6, 7, 11]
= [6, 4, 3, -1]
```

It does what you think it should, given how I named it. Remember that applying subEachFromTen to a list, even a named list, does not change that list, but merely returns the result.

Take some time now to play around with partial evaluation, in addition to the list functions mentioned before and list comprehensions. Remember that functions grab their parameters 'eagerly', so you have to put parentheses around any parameters that are composed of a function with its own parameters.

Chapter 4

Section III: Now Let's Really Write Functions

4.1 Part I: Did You Take That Break? Here Are Patterns

Hopefully you are now comfortable defining and using functions in Haskell, with your choice of Hugs, GHC, or GHCi. Now it's time to talk about all the ways in which functions can be defined. If not, well, get that way. The rest of this will only help you if you're trying it for yourself.

All functions have a type, even those with no parameters ('global variables' and 'local variables'). It is not always necessary to write this type, as Haskell compilers can determine it, but it is very good practice and sometimes a necessity. After reading this section, it would be a good idea to read the Tour of the Haskell Syntax linked here:

http://www.cs.uu.nl/ afie/haskell/tourofsyntax.html

There are a few examples we can go through to make that page clearer. First, a simple function that walks down a list and sums its members:

```
sumAll :: Num a => [a] -> a
sumAll (x:xs) = x + sumAll xs
sumAll [] = 0
```

This is a recursive function. It takes a list of 'a' in Num and return an 'a'. However, there seem to be two definitions for sumAll. And there are. This is how pattern matching works. The two definitions have different specifications for their parameters, and each time sumAll is called, whichever pattern matches the parameters will be evaluated. Let's look at each definition. The second definition is the clearest. '[]' is the empty list, and sumAll of an empty list is defined here as zero. The middle line has something odd about it, though. '(x:xs)' is listed as a parameter, as if we were trying to stick something on the front of a list. In essence we are, because this pattern takes apart its input. There are a few patterns which do this, and this feature of Haskell makes lists very easy to use. To restate, when '(x:xs)' is written as a parameter in a function definition, it will only match lists which have an item stuck on the front. In other words, it will match lists with at least one member. The choice of variable names 'x' and 'xs' is completely arbitrary, but since 'x' will be 'bound' to the first member of the list and 'xs' will be 'bound' to the remainder of the list, it is somewhat natural to write one 'x', and then the remaining 'xs' when describing the list.

When the pattern 'sumAll (x:xs)' matches the input, that is, when 'sumAll' is called with a list that has at least one value, the first definition will be evaluated. This will return the result of adding said first value to the result of calling sumAll on the rest of the list. Patterns are checked top-to-bottom, so when 'sumAll (x:xs)' fails to match, 'sumAll []' is checked. Since the only pattern that could fail to match 'sumAll (x:xs)' is an empty list, 'sumAll []' will definitely match, and it returns zero. This is the end condition for the recursion.

This sort of function is very common in Haskell. Pattern matching lets us avoid complicated 'switch' statements or 'if' blocks in favor of simply writing separate definitions for distinct inputs. This allows code to be clear and concise. Patterns can also be more specific. The fib example can be rewritten as follows:

```
fib :: Num a, Num b => a -> b
fib n = fibGen 0 1 n

fibGen :: Num a, Num b => b -> b -> a -> b
fibGen a _ 0 = a
fibGen a b n = fibGen b (a + b) (n - 1)
```

Here a literal ('0') is used to match a pattern, and that is fine, too. Also note the underscore ('_') in the first definition. The underscore matches anything, just like a variable name, but does not get 'bound' to its parameter. This can make code clearer when a parameter is not used. In this case, we do not care about the second parameter, since we are only matching against the third and returning the first.

4.2 Part II: After Patterns, Guards

Sometimes functions need more complicated work to choose between definitions. This can be done with guards, shown here:

```
showTime :: Int -> Int -> String
showTime hours minutes
    | hours == 0    = "12" ++ ":" ++ showMin ++ " am"
    | hours < 11    = (show hours) ++ ":" ++ showMin ++ " am"
    | otherwise     = (show (hours - 12)) ++ ":" ++ showMin ++ " pm"
    where
    showMin
        | minutes < 10  = "0" ++ show minutes
        | otherwise     = show minutes
```

Ignore the lines that start with 'where' for the moment. 'showTime' has only one definition clause, but it is broken up into three guards. Each guard has a boolean expression, and they are checked in order. The first which is found to evaluate to True will have its corresponding expression evaluated and returned. The function will be equal to that expression for the case that that guard is true. 'otherwise' is equal to True, and will therefore always be accepted if it is reached. 'otherwise' is a convenience, not a necessity. '++' is the list concatenation operator. It is used here for the following reason:

```
String :: [Char]
```

So all of that makes sense except for the 'where' stuff. Here I threw in something extra, and used a 'where' clause to define a local function, in this case a variable called 'showMin'. 'showMin' is a variable in the traditional sense, but it is also a function here in Haskell, so instead of an 'if' statement or 'case' statement, I used guards again to describe its two definitions.

In all, this function takes an hour (hopefully from 0 to 23) and minutes (hopefully from 0 to 59) and prints the time from them. 'showMin' is a local variable/function, defined in the where clause. Guards are used both to define 'showTime' and 'showMin'.

It is important to note that the variables defined in 'where' clauses and their cousins, 'let' clauses, are only in scope for the pattern in which they exist. So a function defined with multiple patterns can't use a 'where' clause to define a local variable across all of them.

4.3 Part III: 'If'

I mentioned 'if' statements just above, and Haskell has them, but they are always if-then-else. Haskell doesn't have much use for a function that doesn't return some kind of value, so 'if-then' doesn't work. That said, here's a simple example:

```
showMsg :: Int -> String
showMsg n = if n < 70 then "failing" else "passing"
```

Not much to it. Since 'showMsg' has a return type of String, the values in both the 'then' clause and the 'else' clause have to also be of that type. 'if' does not need to be the entire definition of a function. For instance:

```
showLen :: [a] -> String
showLen lst = (show (theLen)) ++ (if theLen == 1 then " item" else " items")
    where
    theLen = len lst
```

4.4 Part IV: Indention Syntax

You may have noticed that I use indention to indicate lines that are part of a block of source code. This is not simply good style, but it is also part of Haskell's syntax. Indentation denotes structure.

Specifically, changing the indentation from one line to the next indicates that a block of code has begun or ended. Also, Haskell will not let you place the definitions for two functions one line after another. Instead it demands a blank line to indicate that the definition of a function has truly finished. This all enforces good style, and it greatly reduces the amount of junk syntax in the language, such as '', '', and ';'.

4.5 Part V: And Lambda Functions

Given a programmer's ability to define functions locally in Haskell, you might ask, "Is there an even more brief way of defining functions?" Unlike this tutorial, Haskell does let you finish quickly:

```
(\x y -> x * 4 + y) :: Num a => a -> a -> a
```

What is this you ask? It all starts with the '(ŕight at the beginning. '́, you see, looks a little like the greek letter lambda, which happens to be the symbol used for Haskell itself:

http://www.haskell.org/

'Lambda calculus' is a branch of mathematics that deals with functions created on the fly. That's all I really know about it, and there's a lot more to be found online, etc. But the point is that when you write '(́ you have started a 'lambda expression', which looks a lot like the one above, generally. It has '(́ then some variable names (usually short ones), a '->' arrow, and an expression which uses those variables. And of course a ')'. I'm sure you can figure out what this does. It defines a function and uses it right where it is defined. For example:

```
map (\x -> "the " ++ show x) [1, 2, 3, 4, 5]
= ["the 1", "the 2", "the 3", "the 4", "the 5"]
```

I mention lambda functions because they come in handy, but also because they are used often in the more complicated examples. Of course, lambda functions cannot be recursive, since they would need a name in order to refer to themselves. So they are good for those cases where a function is needed only once, usually to define another function.

4.6 Part VI: Polymorphic Types and Type Constructors

The simplest Haskell program is the following:

```
main = return ()
```

The variable 'main' is a reserved word, and whatever its value is, the program executes. Here is the obligatory "Hello World" program:

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

And the type of 'main':

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

This is another way of saying "main is of type something or other". Well, that's how it looks anyway. What we have here are examples of two strange things at once. That happens a lot when you're learning Haskell, and that's why I'm writing such a long tutorial in the first place. '()' is a type. The only value of type '()' is written '()'. Another word for this type and its singular value is 'null'. So this type can be read "main is of type IO null". But why is it read that way? What does it mean to put one type after another? IO is not a type. The word 'IO' is only part of a type. 'IO a' is a type. 'IO' is a type constructor which takes a type as a parameter. Deep breath.

This is not the same thing as a class. When we were talking about classes, I said that functions were polymorphic, meaning that they could operate on values of any type provided that the proper functions were defined for it. You can create a type and make it a member of Num, as long as it has '+', '-', and/or '*' defined for it, as well as having equality defined. If you do that correctly, any function which has 'Num a =>' at the beginning of its type will accept your new type and everything will work fine. But 'IO' isn't a class, or a polymorphic function. This is something stranger. It is a polymorphic type.

Did that make sense? A type that takes another type as a parameter? Let's look at an example from the standard libraries:

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

This type can be found here:

http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data.Maybe.html

This is a 'data' type definition. The values on the right are separated by '|', the pipe symbol, which can be read here as "or". This type says, "a value of type 'Maybe a' can be 'Nothing', or can be 'Just a'", that is 'Just' followed by a value of type 'a'. Here's an example using Maybe in pattern matching:

```
showPet :: Maybe (String, Int, String) -> String
showPet Nothing      = "none"
showPet (Just (name, age, species)) = "a " ++ species ++ " named " +
+ name ++ ", aged " ++ (show age)
```

'showPet' has two patterns. The first matches a value of 'Nothing', the first 'data constructor' for Maybe. There are no variables after 'Nothing', just as there are no types listed after 'Nothing' and before the '|' in the type definition for 'Maybe a'. The second matches a value of 'Just', the second 'data constructor' for Maybe. 'Just' does have a tuple after it, just like in the type definition, and parentheses are used to group these two things in the pattern. The words 'Just' and 'Nothing' are arbitrarily chosen, although good choices. It is important that the first letter is capitalized. As you may have noticed, throughout Haskell, variables have lowercase first letters and types have uppercase first letters. This is a part of Haskell's syntax. Data constructors are not variables, and so the convention is extended to require their first letters to also be capitalized.

4.7 Part VII: The IO Monad

So let's get back to 'main' and its type, 'IO ()'. IO is a polymorphic type. But what is it? Unfortunately, I cannot show you its definition. 'IO a' is a part of the Haskell standard, and most of its implementation is 'under the hood'. Its implementation is also very large, and involves a lot of low-level code. 'IO' is a member of the class Monad, which means (very briefly) that it lets you write sequential-looking code using the 'do' notation which I'll show in minute. 'IO' is also short for Input/ Output of course, and that means that the 'under the hood' functions of IO let you read and write the world state. So 'IO' is how Haskell interacts with the real world. Again:

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

Monads are odd things, and a function of type 'IO a' will perform an 'IO' operation and return a value of type 'a'. In this case, main will return a value of type '()', namely '()', or 'null'. So main is always "an IO operation that returns null". Here is short main function, with types. Note that it is never necessary to specify the type of main.

```
someFunc :: Int -> Int -> [Int]
someFunc .........

main = do
    putStr "prompt 1"
    a <- getLine
    putStr "prompt 2"
    b <- getLine
    putStrLn (show (someFunc (read a) (read b)))
```

Here's what's going on: The IO Monad binds blah blah blah that's all in the other tutorials and yet you're reading this. Let's try again.

Here's what's going on: All of that 'variable equal at all times' stuff I mentioned above doesn't work with I/O. After all, you have to call 'getLine' several times, and it doesn't always return the same value. You can't say that the function 'getLine' is "equal" to anything, since its value may be different every time it is referenced. This is in contrast to a value, like '4', which will always be the number 4. So IO is hard to do if you want a language that has true "equality" when you write an assignment. This has actually been the biggest stumbling block for functional language design since the idea of functional languages arose. Haskell has a solution.

Most functional languages break their 'functional model' to handle IO, but Haskell does something weirder, of course. There's an obscure branch of mathematics, namely monads, that concerns state transformation functions. The authors of Haskell used it to let you write mathematical functions that denote changes in the world's state without breaking that nice equality stuff. [How's that for simple? See how I made it all make sense by appealing to a higher authority? Haskell works by magic! -Eric]

Briefly, the IO monad takes the state of the world, alters it with some function, and passes it forward to be the input for another IO function. In the example, the functions 'putStr', 'getLine', and 'putStrLn' are members of IO. 'Binding' them to 'main' by using the 'do' notation means that when 'main' is evaluated, they will be evaluated in order, just like you'd expect. Thus the main function of the program above will put a prompt on a screen and read in a string, and then do it again, then print something. The '(read x)' function takes a string and returns a number of whatever type is needed, assuming the string parses. The 'show' function will take the result from 'someFunc' and turn it into a string so that putStrLn can display it. I will return to this code later.

The description of IO may sound like 'imperative' behavior, and it is, with a twist. Input and output need to happen in a certain sequence in a program, but a mathematical function's parts can be determined in any order as long as it eventually finishes. Sequence doesn't matter for the evaluation of a mathematical definition, and most Haskell code is written in that manner. So what about input and output? Can they be mathematically described? As it turns out, they can. Monad theory means that a function can be "equal" to the act of transforming a state. The mathematical framework for monads captures the idea of sequenced actions, and that let Haskell's creators give 'IO a' type that could be used just like any other type. When a sequence of monad functions are evaluated, for instance when 'main' is evaluated, the monad functions are applied in sequence to the initial state. Before that evaluation, Haskell gets to treat functions of type 'IO a' just like anything else, i.e. as expressions waiting to be evaluated. That means that the 'IO a' type doesn't have to be a special, builtin part of Haskell, even though input and output have to be part of the standard libraries and implemented with system calls. It does mean that doing IO in Haskell means using a polymorphic type as well as a mathematical theory that is rather obtuse. That's why IO is so far down in this tutorial.

All Haskell functions carry around this mathematical framework ("equality") and unless otherwise noted, they are lazy. This means that the only thing that forces evaluation is binding an 'IO monad' function to 'main'. **Nothing is ever evaluated unless it is going to be the return value of an evaluated 'monad' function, or unless it is needed to compute such a return value**. That the 'IO monad' forces evaluation isn't really important, but it will explain some odd behavior if you start doing "out there" coding in Haskell. There are eager, or 'strict' functions in Haskell which, when evaluated, will first fully evaluate their parameters. These are usually marked in the documentation.

It is possible to use the IO monad, illustrated above, to write imperative-style programs in Haskell. That is a direct result of the behavior of the IO monad. Doing so would be inappropriate, because Haskell has much more power than that. I mention that because no one else has, and it is important.

Everything that deals with the rest of the computer is part of the IO monad, such as driver calls, network libraries, file access, threading, and system calls. There are other monads in Haskell's libraries, and you can also write your own. Writing your own monad for a major project will probably be the other hard thing you need to do to fully understand Haskell, after understanding 'foldr' and variants. It's pretty hard, but not because it's complicated. Writing your own monad is hard because there's so little to do that you'll have to work to understand why that little bit of code is all you need. But that's quite some time in the future.

4.8 Part VIII: Dissecting the IO Example

Here's the example main program again:

```
someFunc :: Int -> Int -> [Int]
someFunc ........

main = do
    putStr "prompt 1"
    a <- getLine
    putStr "prompt 2"
    b <- getLine
    putStrLn (show (someFunc (read a) (read b)))
```

This is a general framework for learning Haskell. Aside from altering the number of parameters to 'someFunc' (perhaps zero), this is all you will really need for a main function until you feel comfortable with Haskell. It is good enough for most simple tasks, and you can use it to test out ideas in GHC by replacing the definition of 'someFunc' with whatever function you're trying to write. You won't need it for working in Hugs or GHCi, but you will if you compile with GHC. In Hugs and GHCi, you only need to make a source code file that includes someFunc's definition.

What I said earlier about the indentation technique removing extraneous syntax isn't quite true. '', '', and ';' do exist in Haskell. They are an alternative to the 'do' notation used here, and are sometimes but rarely preferable. The 'do' notation is very simple, and the example shows most of its syntax.

The '(read x)' items use the 'read' function found here:

Prelude: the 'read' function

'someFunc' is whatever you want to test. Since its return value is a parameter to 'show', 'someFunc' can be defined with a variety of return types, such as 'Int', '[Int]', 'String', or even '(String, [Int])'. The type of 'show' is given here:

Prelude: the 'show' function

'show' is a class method defined for members of the class 'Show'. This is just like how '+' and '*' are class methods defined for members of 'Num'. You can see those here:

Prelude, Section: the Num class

The types of the IO functions, specifically 'putStr', 'getLine', and 'putStrLn', are given here:

System.IO, Section: Special cases for standard input and output

and also here, in the Prelude, which is why they are in scope normally:

Prelude, Section: Simple I/O operations

As you can see from the documentation, when putStrLn is applied to a String, you get a value of type 'IO ()'. The 'null' means that no useful information is returned by the function. It altered the state of the IO monad by putting something on the screen, and no value comes back from it.

The '<-' arrow is used to bind the result of an IO operation to a variable. 'getLine' has a type of 'IO String'. The 'do' notation says that a monad function like 'getLine' can be prefaced by a variable name and the left arrow. That variable name comes into scope at that point in the function and when the monad function is evaluated, and its return value will be assigned to the variable. If a function is not prefaced by a left arrow and a variable, then its return value is ignored, although whatever that function did to the state carried by the monad still happens. For instance, if you do not bind the return value of 'getLine' to a variable, you don't store the line the user typed in, but it is still read in and buffers are messed with, etc., meaning that another getLine won't return it.

There is one exception to this 'ignoring' of return values not bound to variables. It is no accident that the last line in the sequence has the same return type as main. When using monads to compose sequences of operations, the last line in a function must have the same IO type as the function itself. When this is not natural, use 'return' with an appropriate value:

```
getName :: IO String
getName = do
    putStr "Please enter your name: "
    name <- getLine
    putStrLn "Thank you.  Please wait."
    return name
```

'putStrLn' has a type String -> IO (), but IO () doesn't match the type of 'getName', which is IO String. 'return name' is used to end the function with a function of the proper type, and to return the data we wanted to return. Without the reply message, this function can be written much more succintly:

```
getName :: IO String
getName = do
    putStr "Please enter your name: "
    getLine
```

As you can see, calling an IO function which returns a value is the same as binding that value to a variable and then returning that value.

This whole monad issue looks imperative, and in some ways, it is. Once you call 'someFunc', you get away from all that, and Haskell's laziness and equality become the norm. Is all of this necessary? Is it a good idea to have imperative-looking code calling lazy, functional code? In my opinion, it can be. You get to specify the order of execution for those operations that require it (such as IO), and you get powerful functional tools the rest of the time. You also get a hurt head. So take a break. I need one from writing all of this in two days so far. [Now with three days of editing. -Eric] The next and hopefully last section will be on advanced type declarations.

Chapter 5

Section IV: Haskell and You

5.1 Part I: Where Are the 'For' Loops?

As you may have noticed, the aren't any 'for' loops in Haskell. You could write them using IO monad functions, but I said that wasn't the right thing to do. So where are they? If you've already figured Haskell out, you can skip this section, but for those you like me that needed a lot of assistance, read on.

'for' loops are unnecessary. Not just in Haskell, but in general. The only reason that you have ever used 'for' loops is that you had to 'do' something to a chunk of memory to store the right value in the right spot. Haskell frees you from this, and when you got lost and wonder where your 'for' loops are, check here:

```
bar :: Int -> Int
bar = ...

foo :: [Int] -> [Int]
foo (x:xs) = bar x : foo xs
foo [] = []
```

That looks a little too easy, and is actually equivalent to 'foo = map bar'. Here's a less contrived example. What if you were implementing a physics simulation, gravity for example, and you needed to calculate a new position of each object based on the other objects' positions? The following function calculates part of the process, which is finding, for each object, the sum of its mass times another object's mass divided by distance, over all other objects. In C, this would be accomplished by a pair of nested for loops, the outer one reading the mass and position from an array of objects, and the inner one computing mass1 times mass2 over the distance and summing it. Here are the types for that operation in Haskell:

```
type Mass = Double
type Pos = (Double, Double, Double) --x, y, z
type Obj = (Mass, Pos)

{-
Takes a list of objects.
Returns a list of (object, sum of mass times other object mass over distance for all o
Order is preserved.
-}

calcMassesOverDists :: [Obj] -> [Double]
```

That's definitely the setup. '--' indicated that the rest of the line is a comment. '{-' and '-}' open and close block comments.

```
calcMassesOverDists objs = calcHelper objs objs

distXYZ :: Pos -> Pos -> Double
distXYZ (x1, y1, z1) (x2, y2, z2) = sqrt (xd * xd  + yd * yd + zd * zd)
    where
    (xd, yd, zd) = (x1 - x2, y1 - y2, z1 - z2)

calcHelper :: [Obj] -> [Obj] -> [Double]
calcHelper (obj:objs) objList  = (sum (calcMMoD obj objList)) : calcHelper objs objL
calcHelper [] _ = []

calcMMoD :: Obj -> [Obj] -> [Double]
calcMMoD obj@(m1, pos1) ((m2, pos2):rest)  =  safeValue : calcMMoD obj rest
    where
    dist = distXYZ pos1 pos2
    safeValue = if pos1 == pos2 then 0 else m1 * m2 / dist
calcMMoD _ [] = []
```

Okay, I threw something extra in there, too. 'obj@' in front of '(m1, pos1)'. The '@' is read 'as', and it means that 'obj' will refer to the whole value of type 'Obj', while '(m1, pos1)' will pattern match against the values in 'obj'. It's handy, because otherwise I would have to write '(m1, pos1)' again when I called 'calcMMoD' recursively, and I might make a typo when I did. Also, it's clearer. Note also that I did not have to completely 'take apart' obj. I left the value of type 'Pos' bound to 'pos1'. And note that I put all the sub-computations for 'distXYZ' into one line to save space. I defined the tuple '(x1 - x2, y1 - y2, z1 - z2)', and then bound it to the pattern of '(xd, yd, zd)', thus defining 'xd', 'yd', and 'zd' how I needed to. Finally note that dist is not evaluated in a call to calcMMoD if pos1 == pos2, and neither is m1 * m2 / dist, which avoids the division by zero.

I can compare equality between pos1 and pos2 because a tuple of types which are all in the class 'Eq' is also in the class 'Eq'. The definitions allowing that are here, although you have to scroll down about four pages:

Prelude, Section: the Eq class

What you're looking for is a line like this:

```
(Eq a, Eq b) => Eq (a, b)
```

When listed in the 'Instances' section of the Eq class, that means that somewhere in the source code for the Prelude, there is an instance of Eq defined for '(a, b)' with the condition that both a and b are also members of Eq. The definition of that instance is fairly straightforward; the point is that it is there.

'sqrt' is a class function, and is defined (separately) for all types in the class 'Floating', which of course includes 'Double'. The type of 'sqrt' can be found here:

Prelude, Section: the Floating class

'sum' is a polymorphic function, and is defined (once) for all types in the class 'Num', which also of course includes 'Double'. The type of 'sum' can be found here:

Prelude, Section: Special Folds

I could have had calcMMoD return the sum, but code will compile to a more efficient result if the tasks are broken up, since 'sum' in the prelude is based on a tail-recursive and highly optimized function. The definition of 'sum' is in the Prelude, and more about 'tail recursion' can hopefully be found here:

http://en.wikipedia.org/wiki/Tail_recursion

So where are the 'for' loops? Each recursive function iterates over a list, and the two together act as a nested pair of 'for' loops. This code is good, but there is a quicker way to write 'calcMassesOverDists' and 'calcMMoD', with the same types, and using a much simpler helper function for 'calcMMod':

```
calcMassesOverDists :: [Obj] -> [Double]
calcMassesOverDists objList = map (\obj -> sum (calcMMod obj objList)) objList

calcMMoD :: Obj -> [Obj] -> [Double]
calcMMoD obj objList = map (mMoDHelper obj) objList

mMoDHelper :: Obj -> Obj -> Double
mMoDHelper (m1, pos1) (m2, pos2) = if pos1 == pos2 then 0 else m1 * m2 / distXYZ pos

distXYZ :: Pos -> Pos -> Double
distXYZ (x1, y1, z1) (x2, y2, z2) = sqrtDouble (xd * xd  + yd * yd + zd * zd)
     where
     (xd, yd, zd) = (x1 - x2, y1 - y2, z1 - z2)
```

I could have also avoided writing 'mMoDHelper' by using a lambda function:

```
calcMassesOverDists :: [Obj] -> [Double]
calcMassesOverDists objList = map (\obj -> sum (calcMMod obj objList)) objList

calcMMoD :: Obj -> [Obj] -> [Double]
calcMMoD (m1, pos1) objList = map (\(m2, pos2) -> if pos1 == pos2 then 0 else m1 * m2 /

distXYZ :: Pos -> Pos -> Double
distXYZ (x1, y1, z1) (x2, y2, z2) = sqrtDouble (xd * xd  + yd * yd + zd * zd)
     where
     (xd, yd, zd) = (x1 - x2, y1 - y2, z1 - z2)
```

Or I could have avoided writing calcMMoD, but at that point it gets ridiculous:

```
calcMassesOverDists :: [Obj] -> [Double]
calcMassesOverDists objList = map
   (\obj1@(m1, pos1) -> sum (map (\(m2, pos2) -> if pos1 == pos2 then 0 else m1 * m2 /
     objList

distXYZ :: Pos -> Pos -> Double
distXYZ (x1, y1, z1) (x2, y2, z2) = sqrtDouble (xd * xd  + yd * yd + zd * zd)
     where
     (xd, yd, zd) = (x1 - x2, y1 - y2, z1 - z2)
```

In any case, the division by zero will not be evaluated if pos1 == pos2. In the first example, note that mMoDHelper starts with a lowercase letter as all functions and variables do. Also note that none of these take much code to write. Haskell is like that. mMoDHelper is partially applied in this example. It is given only one of its two parameters where it is written in calcMMoD. The type of 'mMoDHelper obj' in that expression is:

```
mMoDHelper obj :: Obj -> Double
```

This in turn is suitable as a parameter to 'map', as it only takes one parameter, rather than two.

Not every instance of a for loop should be turned into a 'map'. In this case, there is the Prelude function 'sum', which will take the list generated by the 'map' in calcMMoD and return the sum of its values. There is not always a pre-built function for your purposes. List functions are part of Haskell's core tools, and there are more advanced functions to use when you need them, such as 'foldr' and its variants.

Learning how to use 'foldr' and 'foldl' is a rite of passage for Haskell programmers. You will learn in time, by studying their definitions (in the Prelude) and definitions of other functions defined in terms of 'foldr' and 'foldl', such as (oddly enough) 'map'. For loops were just the beginning. When you get away from sequential behavior, real power comes out. Also, keep in mind that since foldr and map are used everywhere, GHC has heavily optimized them, and functions like them, so it's a good idea to use them.

5.2 Part II: Remember Lazy Evaluation? It's Still There

This section fleshes out the flexibility of the Haskell type system in a way that I haven't seen described for newcomers before. Hopefully, your are at least familiar with the concept of a 'Tree', that is, a data structure which has nodes that contain values and may also point to further nodes in the structure. 'Binary Trees' are one of the common forms, i.e. trees where each node has at most two children. For those of you for whom this is new, note this isn't a graph. Trees aren't supposed to loop back on themselves; they just branch. Anyway, here's my example.

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

This is a relatively complicated type. It is a 'data' type, defined with the reserved word 'data', like 'Maybe a' was. It is also polymorphic like 'Maybe a'; you can tell because the type, 'Tree a', takes a type variable. It has two data constructors, 'Null' and 'Node'. 'Null' has no arguments and 'Node' has three. The way it is named indicates that it is a tree, and on inspection it is a binary tree, having two children in 'Node'. So how would this work? First, let's write a value of this type:

```
t :: Tree Int
t = Node 3 (Node 2 Null Null) (Node 5 (Node 4 Null Null) Null)
```

If we were to graph this tree, not including Null branches, it would look like this:

```
    3
   / \
  2   5
     /
    4
```

The first node has both a 'left' child and a 'right' child, and the 'right' child of the first node also has a 'left' child. Let us review 'constructors'. In this example 'Tree a' is a type, and 'Node' and 'Null' are 'data constructors' for that type. A data constructor, often simply called a constructor, acts like a function that groups together objects to form an object of a datatype. They only exist for types defined using 'data' or 'newtype'. Note that they are different from type constructors, such as IO and Maybe. Data constructors act like functions, and do not belong in type signatures. Type constructors act like functions on types, and only belong in type signatures. They do not live in the same namespace, so often a 'data' type will have a data constructor of the same name out of convenience.

Constructors are also used to 'deconstruct' objects of 'data' types, like so:

```
inOrderList :: Tree a -> [a]
inOrderList Null    = []
inOrderList (Node item left right)  =
    inOrderList left ++ [item] ++ inOrderList right
```

'inOrderList' uses pattern matching to determine which constructor its parameter uses. Further, it 'deconstructs' a value that uses the constructor 'Node' and binds its member values to the variables 'item', 'left', and 'right', which are of types 'a', 'Tree a', and 'Tree a', respectively. We know these types because Node's definition reads, "Node a (Tree a) (Tree a)", and 'a' is not further specified in this example. For those of you unfamiliar with trees, 't' above is a 'binary search tree', and evaluating 'inOrderList t' will result in the following:

```
inOrderList t
= [2, 3, 4, 5]
```

Those values are in ascending order, since that is the definition of a 'binary search tree'. Read up on them if you're not familiar with them already.

There's another funny thing about the definition of 'Tree a'. It's recursive. It used 'Tree a' to define each child. You can do that in Haskell as well as C, but as usual, there's a twist, and again as usual it involves lazy evaluation. In C/C++, to use a type in its own definition, you must declare a pointer to an object of its type rather than including it directly. In Haskell, pointers aren't used like that, so the type is included directly, as in the tree example. So what about this definition:

```
foo :: Int -> Tree Int
foo n = Node n (foo (n - 1)) (foo (n + 1))

t2 = foo 0
```

What is the value of 't2'? And what is its type? The value of 't2' requires an infinite amount of space to store. The first few levels of the trees could be drawn like this:

```
         0
       /   \
    -1       1
    / \     / \
  -2   0   0   2
  / \ / \ / \ / \
```

And so on. But the type of 't2' is simple, and should be clear from the type of 'foo':

```
t2 :: Tree Int
```

If you ever get an error message (not if ever, but when you get an error message) that says "can't do such-and-such, unification would result in infinite type", it means that the type would require an infinite amount of space to store. Typically this happens with pattern matching on lists. In that case a careful check will show that something incorrect in the code would have resulted in nested lists infinitely deep, meaning a type that looks like this:

```
[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[....
```

And so on. But back to 't2'. Its type is not infinite, even though its type is defined recursively and has a value that, when evaluated, would require infinite space to store. Because its type is defined recursively, Haskell (and you) know that values of type 'Tree a' can contain other values of type 'Tree a'. The fact that each value of type Tree a which is 'in' t2 uses the constructor 'Node' is not stored in the type of t2, so the type of t2 is finite and Haskell can use it. Also, multiple values of type 'Tree Int' can exist, some infinite, some not, and they all have the same type. Okay?

5.3 Part III: The Point(s)

I wrote all this about types to drive two points home. First, Haskell uses lazy evaluation. For example, the following type is valid and Haskell will not complain about it:

```
data Forever a = AThing a (Forever a)
```

It is a polymorphic type, it has a single constructor 'AThing', and it is recursive. Values of this type will always be infinite if fully evaluated. That's okay, and you could use this in a program if you wanted to. Haskell would not attempt to fully evaluate an object of this type unless you told it to.

The second point is that learning and using Haskell requires unlearning and rethinking the basic assumptions most coders have developed about programming. Unlike so many languages, Haskell is not 'a little different', and will not 'take a little time'. It is very different and you cannot simply pick it up, although I hope that this tutorial will help.

If you play around with Haskell, do not merely write toy programs. Simple problems will not take advantage of Haskell's power. Its power shines mostly clearly when you use it to attack difficult tasks. Once Haskell's syntax is familiar to you, write programs that you would ordinarily think of as too complicated. For instance, implement graph-spanning algorithms and balanced-tree data types. At first, you'll probably trudge ahead using 'imperative thought processes'. As you work, you'll hopefully see that Haskell's tools let you dramatically simplify your code.

A big tip is, "Don't use arrays." Most of the time, you need to access members of a list one at a time, in order. If that is the case, you do not need an array, and it would take you a lot of work to set one up. Use a list. If you are truly storing data that needs random access, still don't use arrays. Use lists and '!!'. When you need speed, then use arrays. Use Arrays, and then switch to IOArrays. But only use them if you actually need speed. Your class projects absolutely will not need arrays unless you are writing a game, or working on something for at least three months. Then, you might need fast random-access data. Maybe. Use lists.

Chapter 6

Section V: Final Commentary

6.1 Part I: Why is 'Referential Transparency' Worth Anything?

Functional languages have often been accused of generating slow code. This is not the case with Haskell, except for the Array type. Switch to IOArrays if you need speed. Anyway.

Referential transparency gives Haskell programs two big boosts: better execution speed and memory use, and shorter, clearer code.

First, the reasons for better speed and memory use. No code runs faster than code evaluated during compilation, except code optimized away during compilation. The easiest way to evaluate during compilation is to find where a value is assigned to a variable, find where that variable is next read, and hardcode the value in that place. This avoids a memory write, possibly a read, and also potentially some memory allocation.

In both C and Haskell (for example), functions inherit the namespace of their scope. In C, this means that any variable in scope can be accessed at any time. Worse, you could have assigned a pointer the value of the memory space of a local variable, or be running a multi-threaded application. Or you could do pointer arithmetic and mess with anything. In those cases, your compiler has to give up most of its hope of knowing how variables will be updated during execution. Since it is possible to write code that uses those techniques to update a variable when unexpected, the C compiler has to actually read a variable every time it is referenced unless it can prove that no update will touch that variable between two points in a program. Every function call potentially alters every variable in scope, within some limits. Certainly all data passed to a function in another module through a pointer must be re-read after the function call. That happens often in large projects.

What about Haskell? Haskell is 'lazy and purely referentially transparent'. If you pass a value to a function, it will not be updated. That means that GHC can assume that. Specifically, there is never a need to copy a value when it is passed into a function, or ever re-read it. Also, a function which takes a list and returns part of that list hasn't copied anything. Since no update is possible, there is no harm in referencing the original. That's a big savings in memory, which translates into less memory reads and less page faults. Also, since the order of evaluation is unimportant to the programmer, the compiler can determine the fastest order.

Other things make Haskell code incredibly fast. First, Haskell's laziness adds an extra kick to the speed. Second, there are only a few pieces to Haskell: functions, if-then-else, polymorphic types, exceptions, the IO code, and a garbage collector. Most library list functions are built on foldr or a variant, if they do any real work at all. This is a very small number of pieces, and that makes Haskell easy to compile and optimize. Finally, GHC does cross-module optimization, even for library functions (I think).

Contrast that to C/C++, with any number of libraries, most of them pre-compiled, and the only thing fast is memory manipulation. If you need memory manipulation, C is what you use. Python makes code-writing quick. LISP makes logical manipulation easy. Java makes code safely portable. Haskell is great (in my opinion) for the major target of industry: big projects with a lot of parts.

Second, there is the source code itself. If you don't care about the order of execution, you can avoid using syntax to specify it. The best way to discover this is to try it yourself. If you're having trouble breaking your imperative habits, read other Haskell code until you can understand why it is so short. The source code of the Prelude itself is a good place to start.

6.2 Part II: Feedback and Notes

Dave and I are thinking about writing practice problems for Haskell aspirants. Please check back later for them.

If you liked this tutorial, it came in handy, it helped you understand Haskell, or you think I talk to much, write me at etherson@yahoo.com.

Now we're working on a video game in Haskell using HOpenGl and HOpenAL, and we've written a short tutorial [short like this one :P -Eric] on HOpenGL. The tutorial here is meant to provide a basic framework for understanding Haskell. The HOpenGL tutorial hopefully does not assume much more knowledge of the programmer. If something too great is assumed by the HOpenGL tutorial, please let me know, because these are supposed to go together.

The HOpenGL tutorial we wrote (Dave, specifically):

Dave's HOpenGL tutorial

The Gentle Introduction Haskell, useful as a reference:

http://www.haskell.org/tutorial/

The Tour of the Haskell Syntax, also a good reference:

http://www.cs.uu.nl/ afie/haskell/tourofsyntax.html

GHC's Hierarchical libraries, an excellent reference once you know the basics:

http://www.haskell.org/ghc/docs/latest/html/libraries/index.html

Finally, GHC's newer API, needed for the docs for the under-development HOpenAL libraries, as well as some others:

http://www.haskell.org/HOpenGL/newAPI/

You should probably read the Gentle Introduction to Haskell and the Tour of the Haskell Syntax thoroughly. Hopefully a lot more should make sense now. There are a lot of little picky things in every programming language, and Haskell is no exception. Particularly, there are a few cases where strictness is an issue, and it's important to know how Haskell's syntax is interpeted. Good luck. Let me know if there's another tutorial I need to write.

6.3 Part III: Ranting

During this tutorial, I refer to Haskell's 'power'. I am cribbing from an old equation about programming languages:

```
flexibility * control of detail = a constant
```

That statement should really be written as follows:

```
flexibility * control of detail = power
```

Haskell has both more flexibility and more control than most languages. Nothing that I know of beats C's control, but Haskell has everything C does unless you need to control specific bytes in memory. So I call Haskell powerful, rather than just 'good'.

I wrote this tutorial because Haskell was very hard for me to learn, but now I love it. I haven't seen tutorials that addressed the difficulty that computer science students usually face with Haskell. I had to take two semesters of college that included Haskell before I really got a grip on it, and I only passed the first because one of my friends knew Haskell and helped me. I kept thinking that other people, maybe the haskell.org people, would write a tutorial aimed at C programmers, but it didn't happen. I understand why, now. As far as I can tell, Haskell is maintained and contributed to mostly by professors, and they have already learned LISP or other functional languages. Also, Haskell is not generally taught to whiny undergrads that throw a fit when faced with something this new and different. UT Austin is somewhat of an exception, and out of my classmates, I was the exception in actually liking it, let alone learning it. So a relatively small number of people have been in my position, and it seems like none of them have spoken up. Well, here it is. **"Haskell is hard!" "You can't write code the way I know how!" "My brain hurts!" "There aren't any good references!**" That's what I said when I was in college. There were good references, but they didn't cover the real problem: coders know C. We, as students just getting through our second or third advanced computer science course, expect sequential behavior for code. It isn't good enough for a book to say that Haskell does not execute commands in sequence. We can tell that. What we don't (and what I didn't) understand is why on Earth you would bother coding in a language that wasn't sort of like C, and how you would even think about it. We could read all these tutorials and a few books about the syntax of Haskell and how Haskell is lazy and has pure referential transparency, but none of that helps. We don't know how to put functions together. We don't know what will be evaluated, and we don't know how the pieces of Haskell syntax fit together. Haskell is so different from the industry standard (C/C++) that we can't look at complicated Haskell code and understand it. And if we don't, Haskell will always be an academic language. Of course, if a small horde of college kids start making little programs in Haskell, the language will survive, and maybe there will be a company besides ours that uses Haskell to make games. I suppose there might be already. With an OpenGL binding and OpenAL on the way, that's a guarantee. But only if students are using it, not just professors. Why didn't you Haskell people see this already? That's okay, you've been busy making a wonderful and extremely powerful language and libraries, and I thank you. Hopefully someday with cash. But only if we're successful. Just like the other college students that you need to encourage, as this hopefully has. Well, I'm done here. Thanks for reading all of this.

Eric Etheridge etherson@yahoo.com