

# **Haskell on All Eight Cylinders**

**John Goerzen**

**Haskell on All Eight Cylinders**  
by John Goerzen

Copyright (C) 2004 John Goerzen. All rights reserved. Redistribution in whole or part is prohibited.

# Table of Contents

<b>1. Freedom .....</b>	<b>1</b>
The 1971 Nova .....	1
<b>2. Getting Acquainted.....</b>	<b>2</b>
<b>3. The Engine.....</b>	<b>3</b>
The Nature of Haskell .....	3
Test Drive .....	3
An Imperative grep .....	4
A Haskell grep .....	4
Compiling grep .....	4
What grep Tells Us About The World .....	5
Analyzing Haskell grep .....	5
A New, Expanded Haskell grep .....	5
Haskell grep Broken Down .....	7
Test Drive Results .....	8
<b>4. Safety Belts: The Type System.....</b>	<b>9</b>
A Brief Typing Backgrounder .....	9
Strong vs. Weak Typing .....	9
Static vs. Dynamic Checking .....	9
Overview of Haskell's Type System .....	9
Type Inference .....	10
Some Quick Experiments with ghci .....	10
Some Typing Experiments with ghci .....	11
<b>5. The Fuel: Functions.....</b>	<b>13</b>
Our Functions Actually Do Something .....	13
<b>Index.....</b>	<b>14</b>

# Chapter 1. Freedom

How do you think it would feel to program in a language that has no notion of a loop? This language would have no `for`. No `do`. No `while`. Not even `foreach`. In fact, it doesn't even have a notion of a variable that can be modified.

Do you think it would be painful? Maybe primitive? Does it remind you of punched cards in the 1960s?

What do you think it would feel like to program in a language that has no *need* for a loop? No *need* for a `do`, `while`, or a variable that can be modified? A language that has so completely obsoleted these things from programming that adding them to the language would make the language feel primitive?

Do you still think it would be primitive? I'm here to tell you that it will make you feel *free*. You'll have more freedom than ever before. Your programs will look completely different. You'll wonder why on earth you ever wrote a loop.

What's the catch?

You have to learn a completely different way of thinking about everything.

OK, that was a little dramatic. You only have to learn a completely different way of thinking about computing.

## The 1971 Nova

I first learned how to drive a car in high school, like most Americans. My parents owned a rather boring 1986 Pontiac. It was a family car with a family car engine, which means it didn't have a lot of enthusiasm for acceleration.

My parents had bought a car for me to drive about 10 years before I could even drive legally. It was a 1971 Nova. A *big* car. Twice as heavy as the Pontiac and twice as old. It was painted a deep orange -- which worked well, since then the rust wasn't as obvious. It had been stored in a barn for 10 years. Mice had eaten away at part of the seats, and it smelled bad inside. My dad got it cheap because it had been in at least one serious wreck already, though it had been mostly repaired.

My dad got the car out and running for me. Like everything else about the car, its engine was big -- 350 cubic inches. I remember hating the car at first. Its primitive carburetor required just the right touch to keep the engine from stalling -- much less forgiving than the fuel-injected car I was used to.

But before long I discovered something -- this car was fast. Despite being old, and in not very good condition, if I pressed the accelerator the wrong (or right?) way, it would take off so quick I'd have to hit the brakes to avoid hitting something.

That car must have been really something when it was new. State of the art, with a massive engine and gleaming paint, and comfortable expansive seats. Yet even in its state, it was so powerful that it could run on less than its full eight cylinders and one might never even notice a problem.

I want to show you just the right touch to using Haskell. The touch that will leave you stunned at how you created something amazing almost before you knew it.

I want you to feel the fun of using a no-holds-barred language. A language that has no loops because *we don't need them*.

# Chapter 2. Getting Acquainted

## *Installing Compilers*

First thing to do is familiarize yourself with the engine of Haskell -- the compilers or interpreters that you'll be using.

Haskell can be either compiled or interpreted. For an interpreter, you could use Hugs or GHC. For a compiler, you could use GHC or nhc98. GHC provides both an interpreter and compiler.

I'm going to use GHC in these examples, though you can usually substitute Hugs or nhc98. You can find it in your operating system distribution in many cases. Or you can find it at <http://www.haskell.org/ghc/>.

Build it and install it -- it's not hard.

These examples are also going to use my MissingH library. You can download it from <http://quux.org/devel/missingh>. The distribution contains installation instructions, which should be no trouble to follow. You'll need 0.4.2 or above.

That was easy. You don't even have to replace the oil.

# Chapter 3. The Engine

## *Using the Compiler and Basic Syntax*

Your new Haskell engine is a little different from ones you've used before. Haskell is a *lazy, purely functional* language.

### The Nature of Haskell

It's *lazy* because it doesn't do any work until it really needs to. That sounds simple, but it's a tremendously powerful concept that is pervasive in the language.

It's *functional* because Haskell deals a lot with functions. Functions are not just chunks of reusable code. In Haskell, they're also things we pass around as arguments. We create them out of thin air and manipulate them with the same ease we'd manipulate strings in some other languages<sup>1</sup>. Think about that for a second. In Haskell, you pass and manipulate *code*. Being functional also means that Haskell is not imperative. An imperative language is one where you tell the computer what actions to perform and in what order. Examples of imperative languages include C, C++, Java, Python, Perl, Assembler, BASIC, C#, Pascal, Fortran, COBOL, etc. You can tell we're onto something odd here when I tell you that Haskell is, in all likelihood, not in the same class of languages as *every language you have ever used* (except perhaps for Make, but that's not functional either). For the record, some other functional languages include Lisp, Scheme, and OCaml.

Haskell is *pure functional* because there are no side-effects. Every time you call a function with certain parameters, the function returns the same thing. Every. Single. Time. There are no exceptions<sup>2</sup>. Oh, functions also do not modify state. They don't change the world. They don't change some global variables or counters. In Haskell, functions don't change things. They just return an altered copy of reality. And this is a good thing, too, because Haskell's reality would be a scary thing indeed if functions could just up and modify it.

You may be wondering how in the world we do I/O in Haskell then. In C, a function like `fgets()` is bound to return different things even when called with the same input. I/O changes the world. A specific call to `fprintf()` could cause this printer to start spewing out paper. but Haskell functions don't modify state. They don't change the world. Well, Haskell has a way to deal with I/O, and it involves *actions*. We'll get to that in a bit.

### Test Drive

Let's take Haskell for a spin. From here on out, I'm going to show you various Haskell examples using GHC. I'm also going to use Python to illustrate a traditional, imperative approach to the same problems. That's not because I'm trying to say that Python is better than your favorite imperative language. Half the time I'm not even showing you idiomatic Python -- I'm illustrating typical iterative algorithms, not showing off Python. I'm just using Python because it should be easily understood by a wide audience, even if your favorite language is Java, C, or Perl.

1. OK, we admit that manipulating strings is not that easy in some languages. We're not talking about those languages here.
2. OK, I lied. GHC has some "unsafe" extensions. But they are not for mere mortals to use. They're more there for the people that write compilers. So pay no attention to the man behind the curtain.

Now then. Let's say we want to write an implementation of the Unix utility **grep**. The **grep** command is used to search for data in files. It will simply display every line containing a particular search term.

## An Imperative grep

We're going to start with a very simple version. Our first **grep** simply reads from standard input and prints all lines containing the word "Haskell" to standard output. Simple enough.

Here's how an imperative programmer might approach it:

```
#!/usr/bin/python2.3
# simplegrep.py
import sys

while True:
    line = sys.stdin.readline()
    if line == "":
        break
    if line.find("Haskell") != -1:
        sys.stdout.write(line)
```

That looks fairly normal. We have a `while` loop. Each time through the loop, we read a line from standard input. If it's empty (meaning we reached the end of the file), we `break` out of the `while` loop, terminating the program.

Otherwise, we check to see if the line contains the word Haskell. If it does, we display it. Simple, right?

## A Haskell grep

Here's how we could do that in Haskell:

```
-- simplegrepl.hs
import MissingH.List

main = do
    c <- getContents
    putStr (unlines(filter (\line -> contains "Haskell" line) (lines c)))
```

**Tip:** An experienced imperative programmer might look at this code, and the discussion below, and conclude that it's terrible because it reads the entire file into memory before processing it. That would be a perfectly valid complaint about a program like this in an imperative language.

However, Haskell is lazy, so your complaint is not valid with this program. You'll see why later.

## Compiling grep

To compile this, use a command such as:

```
$ ghc --make -package MissingH -o simplegrep1 simplegrep1.hs
```

This runs the Haskell compiler, telling it to make an executable that uses the `MissingH` package. It will produce an executable named `simplegrep1` based on the source code in `simplegrep1.hs`. You can now experiment with this executable.

## What grep Tells Us About The World

Let's look back at the source code for `grep`. From this example, there are some things to note:

- The main entry point of every Haskell program is an action named `main`.
- Here, you can see we have used a `do` block. This is as close as Haskell gets to imperative programming. A `do` block is used to sequence actions, and is usually used with I/O.
- In this, and many other, Haskell examples, the amount of indentation you use is syntactically significant. Python programmers will find this concept quite familiar. For those of you that have a contempt for this style, let me say two things: First, this style is optional, and plenty of Haskell programmers don't use it. Second, I'm going to be using this style throughout this tutorial so you can see what it looks like. I encourage you to try it, and decide whether to use it after you've tried it.
- Haskell has two types of comments. Two dashes are used to start a comment that extends to the end of a line. You can enclose a multi-line comment between `{- and -}`.
- Function calls don't need parentheses, though sometimes parentheses are necessary to separate elements of a call.

## Analyzing Haskell grep

Let's look at this program. Its first line has a comment. The next line is an `import` statement. These statements tell the compiler "make available all the names defined in this module." You can find a reference to these modules at GHC's Hierarchical Libraries page<sup>3</sup> and at the `MissingH` reference area<sup>4</sup>. In this case, we are importing `MissingH.List` for the `contains` function. All the other functions we use are available by default. Technically speaking, they're defined in the `Prelude` module, which is always imported by default.

After the `import`, there's the definition of `main`, which consists of a `do` block of two statements. The first statement is `c <- getContents`. That says, effectively, "evaluate the `getContents` action and store the result in `c`". Like I said, we'll go into more detail on actions later. For now you should just know that the only (common) way to "run an action" is to store it in some variable using the `<-` operator.

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

4. <http://gopher.quux.org:70/devel/missingh/html/index.html>

## A New, Expanded Haskell grep

The next line is the real fun one. Before I explain it, I'm going to show you a new version of the Haskell grep. This version does exactly the same thing as the first one, but splits things out to make them easier to understand.

```
-- simplegrep2.hs
import MissingH.List

filterfunc line = contains "Haskell" line

main = do
  c <- getContents
  let inputlines = lines c
      outputlines = filter filterfunc inputlines
      outputstring = unlines outputlines
  putStr outputstring
```

Let's look at this new example. It starts with a comment and an `import`, just like before. Next, we have a line defining `filterfunc`. `filterfunc` is a function that takes one parameter, named `line`. The return value if the function is the one obtained by calling `contains "Haskell" line`. This calls the `contains` function, passing it two parameters. `contains` returns `True` or `False` depending on whether or not the word `Haskell` occurred in the given line.

Next, we move on into `main`.

First thing we do is call `getContents`. The `getContents` action returns a string representing all data read from `stdin`. Some imperative languages have a similar function, which would read the entire contents into memory, then return it. That's bad -- if you're grepping through a 5GB file, your system better have 5GB of RAM or things will grind to a halt.

Because Haskell is lazy, it doesn't have this problem. It only asks for data from standard input when the program needs it, and it releases the memory as soon as it knows the program will never need the data again. Clever, eh?

Next, we call `lines c`. The `lines` function takes a string parameter and returns a list of strings. It simply splits up the input string by newline boundaries, so the resulting list is a list of lines. Remember, Haskell is lazy, so we're not really storing a massive list of all lines of the file in memory -- they're just consumed as needed. The `let inputlines =` part of the statement assigns the result of this call to the variable named `inputlines`.

Next, we call `filter filterfunc inputlines`. This is a cool function. All your friends will be impressed by `filter`. Well, maybe they'll be jealous of it.

`filter` is our first opportunity to really see the functional nature of Haskell at work. It takes two parameters. The first parameter is actually a function! This function is supposed to take an item and return `True` or `False`. The second parameter is a list. It also returns a list. `filter` will call the supplied function for each item in the list. If the function returns `True`, the item is in the returned list. If it returns `False`, the item isn't. In this case, we pass along our `filterfunc`, which will filter out the lines that don't contain `Haskell`.

Next, we come to `unlines`. Can you guess that it does the opposite of `lines`? In fact, `unlines` simply takes a list of strings, taken to represent a line, and returns one big string with all those lines separated by the newline character.

The imperative programmer in you probably wants to yell at me about this again -- what an inefficient waste of memory! But the Haskell programmer in you should remember that Haskell is lazy. The function works incrementally, building things as it goes, and never has to store the whole thing in memory.

Finally, we come to `putStr outputstring`. The `putStr` function simply writes a string to standard output. Since it's lazy, it can be thought of as the "driver" for the while process. It demands data from `unlines`, which demands data from `filter`, which demands it from `lines`, which demands it from `getContents`. Get this -- if we didn't have `putStr` there, the program would never read any data at all, because nothing in the program would require it to be read! That's laziness for you. Isn't it great to be a slacker?

## Haskell grep Broken Down

Above, I expanded the `grep` example. Recall that I earlier mentioned that `do` is the only real imperative-like feature in Haskell. That's true. But it seems like we're doing an awful lot of work in imperative mode here, doesn't it? Well, actually only two lines are doing work in "imperative mode": the first and last lines in the `do` block. Let me show you another example to illustrate this:

```
-- simplegrep3.hs
import MissingH.List

filterfunc line = contains "Haskell" line

processfunc1 inputdata =
    let inputlines = lines inputdata
        outputlines = filter filterfunc inputlines
    in
        unlines outputlines

processfunc2 inputdata =
    unlines(filter (\line -> contains "Haskell" line) (lines inputdata))

processfunc3 inputdata =
    unlines . filter filterfunc . lines $ inputdata

main = do
    c <- getContents
    putStr (processfunc3 c)
```

Here, I provided three pure, non-imperative `processfuncs`. They all take a `String` parameter and return a `String` result. They all are non-imperative -- they are an expression, not a list of commands.

Some people have trouble understanding the difference. Consider the mathematical equation  $3 * 4 + 5$ . This is an expression. Yet to evaluate this expression, we know that we should first multiply 3 and 4, then add 5 -- not add 4 and 5, then multiply by three. There are rules to evaluating an expression.

The same is true in Haskell, but the concept expands. In `processfunc1`, we have only one expression making up the body of the function: `unlines outputlines`. But to evaluate it, Haskell must evaluate `outputlines`. And to do that, it must evaluate `inputlines`. Even though we give it only definitions and an expression, we still get things done right. By the way, every function in Haskell has only one expression. You will never see a Haskell function (aside from actions -- a whole different story) that has more than one expression.

The `processfunc2` function does the exact same job as `processfunc1`, it just does it without using any names for things. That's how our first Haskell `grep` worked. Hopefully, this can help illustrate how this is just an expression evaluation. In Haskell, saying `let inputlines = lines inputdata` doesn't evaluate `lines inputdata` and store the result in `inputlines`. It says, "Hey, Haskell -- if you ever need to figure out what `inputlines` means, here's how."

Notice also in `processfunc2` the little backslash line arrow thingy. That's an anonymous function definition. An anonymous function is a function that has no name. (Clever, eh?) We use them a lot with things like `filter`. This anonymous function is the same function as `filterfunc` above, just defined differently.

Then, `processfunc3` shows you a little bit of what we will cover in (FIXME: ADD LINK TO CHAPTER). You'll see over there how we can write this entire program with less than one line of Haskell code. (Well, one line plus an import, geez, you're so picky!) We do so by doing what I like to call "adding functions". It's pretty slick. Yeah, we take functions and add them up as if they were numbers. (Unix geeks might liken this to pipes.)

## Test Drive Results

I hope you enjoyed your little Haskell test drive. I've shown you a lot quickly. Don't worry if you don't fully understand everything yet. We'll be using the `grep` example more soon. And, by the way, you haven't even seen the really cool, mind-blowing stuff yet.

But you should have a little taste of what I mean when I say that Haskell is a really different way to look at computing. Did you notice that this entire program doesn't have even one loop? It doesn't even have a reference to `if`! (Yes, Haskell does have `if . . then . . else`, but us Haskell types don't use it all that often because, well, we don't need to all that often.)

Next, you're going to see Haskell's slick, strong, static typing system. I haven't even shown it to you yet, though it's already working (and even caught a bug in my code as I was writing this chapter). It's as strong as Java's, as unintrusive as Perl's or Python's, and as powerful as -- well, there's no comparison. It's more powerful than anything you've seen.

After that, we're going to talk about functions. They do a lot in Haskell, and they are *everywhere*. Haskell programmers do very little without thinking of functions somewhere.

Hopefully, if that didn't get you excited, you'll at least read on to be able to prove to me, and all your imperative friends, just how wrong you think I am :-). (Yes, I do still have some imperative friends. They're nowhere near as lazy as I am.)

# Chapter 4. Safety Belts: The Type System

Every modern language you use -- except, perhaps, for assembler -- has a typing system. A typing system is used to identify what sort of data a particular variable holds. You're probably familiar with these typing systems -- types such as ints, floats, strings, lists, etc. are common in languages. A typing system acts as your seat belt. It helps prevent mistakes such as `53 + "asdf"`. There's no real way to add an integer to a string like that, and a typing system helps make sure you don't generate odd results for trying.

In this chapter, you'll learn about both Haskell's type checking and its various data types.

## A Brief Typing Backgrounder

Now, this is a very practical, hands-on tutorial, but I've got to go into theory for just a moment to help you understand how Haskell is different from the languages you know if. It won't be painful, I promise.

When we talk about typing systems, we generally talk about two different distinctions in typing systems: strong vs. weak, and static vs. dynamic.

### Strong vs. Weak Typing

Strongly-typed languages tend to keep track of types tightly. It knows that this thing over here is an int, that thing over there is a string, and you can't just add them together without converting one to the other. One almost universal characteristic of strongly-typed languages is that you must declare the type of every variable you use and the expected types for every function parameter and return value. Examples of strongly-typed languages include Python, C, and Java.

Weakly-typed languages may have an internal notion of types, but they let you trample all over them if you want. You may be able to add the string "5" to the integer 12, but whether you get the string "512" of the integer 17 out of it is going to be language-dependant. Some weakly-typed languages go so far as to not generate an error if you use a variable you've never set before. Examples of weakly-typed languages include Perl and PHP.

### Static vs. Dynamic Checking

Statically-typed language perform type checking at compile time. That is, with few or no exceptions, every possible type error is caught before you ever even try to run the program. The exceptions to this rule generally occur with casting, where it's possible to do some really weird things. Examples of statically-typed languages include C and Java.

Dynamically-typed language perform type checking while the program is running. You could be perfectly safe passing an int to a function expecting a string, as long as that function never tries to use it. This lets you get away with things, but at the same time the program can blow up unexpectedly if there's a typing problem in a little-used part of code. Examples of dynamically-typed languages include Python and Perl.

## Overview of Haskell's Type System

Haskell is a strong, statically-typed language with type inference, polymorphism, and classing.

You understand some of that already, but there are some new terms here. We'll learn about them shortly.

Because Haskell is statically-typed, type errors are caught at compile time, before you ever run your program. Because it's strongly-typed, you don't ever have to worry about type errors at runtime.

### Type Inference

But remember how I said that almost all strongly-typed languages force you to declare the types of your variables? Recall from the examples in Chapter 3 that you never actually declared any types. Not only that, but you never even saw types anywhere. How can that be?

The answer is type inference. Let's look at one of those examples again:

```
-- simplegrep2.hs
import MissingH.List

filterfunc line = contains "Haskell" line

main = do
  c <- getContents
  let inputlines = lines c
      outputlines = filter filterfunc inputlines
      outputstring = unlines outputlines
      putStr outputstring
```

Haskell works backwards and forwards to figure out what types things must be. First, Haskell knows that `filterfunc` takes a `String` and returns a `Bool` because of the type of arguments and return value of the `contains` function.

Then, in `main`, it knows that `c` is a `String` because that's what `getContents` returns. It similarly knows that `inputlines` is a list of `Strings` because that's what `lines` returns. It can track these types all the way down. If you made a mistake somewhere -- say, you wrote `unlines c` instead of `unlines outputlines`, the compiler would have noticed the type error and aborted. We'll see a lot more on this later.

## Some Quick Experiments with `ghci`

I'm going to start by introducing a new program to you. It's called **ghci** and is installed by default with the GHC distribution. **ghci** is the GHC interpreter, and lets you easily experiment with Haskell code. If your system doesn't have **ghci**, you can also try `hugs`<sup>1</sup>, which has many of the same commands. You should be able to just type **ghci** to start up the interpreter.

1. <http://www.haskell.org/hugs>

Let's experiment a bit with **ghci**. Try typing in these examples. You should get similar results as me:

```
Prelude> 5 + 3
8
```

Yep, Haskell can do simple math.

```
Prelude> "asdf"
"asdf"
```

We put strings in double quotes.

```
Prelude> "Haskell" ++ "is" ++ "cool"
"Haskelliscool"
Prelude> ['H', 'a', 's', 'k', 'e', 'l', 'l']
"Haskell"
```

Strings in Haskell are just lists of characters. `++` is the list concatenation (combining) operator, and since strings are lists, it's also the strong concatenation operator.

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

You can define a list by putting its elements in square brackets, separated by commas.

```
Prelude> filter (\x -> x `mod` 2 == 0) [1, 2, 3, 4, 5]
[2,4]
```

Remember `filter` and anonymous functions? We can use them in the interpreter, too. Here we get a list of the even numbers in the list.

```
Prelude> filter (\x -> x `mod` 2 == 0) [1..20]
[2,4,6,8,10,12,14,16,18,20]
```

You can also use the `..` operator to create sequences. Remember, Haskell is lazy, so we aren't causing 20 things to spring into memory at once.

## Some Typing Experiments with ghci

Now that you've tried a few things with **ghci**, let's try some typing experiments. In **ghci**, you can type `colon-t (:t)` to have it show you the type, rather than the result, of an expression. Let's try a few.

```
Prelude> :t 'H'
'H' :: Char
Prelude> :t "Haskell"
"Haskell" :: [Char]
```

The first line showed you that the type of `'H'` is `Char`. The second line shows that the type of `"Haskell"` is `[Char]`. The brackets mean that the type is a list of characters. Every element in a Haskell list has to have the same type.

# Chapter 5. The Fuel: Functions

I told you back in the Section called *The Nature of Haskell* in Chapter 3 that Haskell is a functional language. In Haskell, functions are not just a way towards code reuse. We manipulate them with ease and use them everywhere. We can dissect them, combine them, subtract them, spontaneously create them, return them, pass them, save them. Haskell slices, it dices, it cuts, it chops, it can even make a salad. French dressing, yum.

## Our Functions Actually Do Something

OK, back to functions. You already have some idea of what a function is (subroutine to your Perlers out there, method to you Java fans... it's all the same thing, really.) A function is a thing that takes something and returns something.

Now, in some languages like C, a function might take nothing. It might return nothing. It may even take nothing and return nothing. That sort of function either does nothing or relies on side effects to accomplish something useful. Maybe it increments a global variable or writes something out to the terminal. Well, a function that takes nothing or returns nothing doesn't exist in Haskell. A Haskell function is pure -- it performs a computation based on its parameters and returns something new. A Haskell function that takes no arguments could perform no useful computation, and a Haskell function that returns nothing couldn't do that either. So those don't exist.

As an example, in Python, one could do this:

```
def fiveplusthree():  
    return 5 + 3
```

Now, in Haskell, one could do this:

```
fiveplusthree = 5 + 3
```

That's not a function, though. It's just a number that Haskell could calculate for you. It's similar to saying this in Python:

```
fiveplusthree = 5 + 3
```

Though, of course, this is not a function -- and Python is forced to calculate it for you, whether or not you ever need it, because Python is not lazy.

# Index

## C

Compilers  
  Installing, 2

## D

Dynamic Typing, 9

## F

Functional Languages, 3  
  Pure, 3

## G

GHC  
  Installing, 2  
ghci, 10

## L

Lazy Languages, 3

## M

MissingH  
  Installing, 2

## S

Static Typing, 9  
Strong Typing, 9

## T

Type Inference, 10  
Type System, 10

## W

Weak Typing, 9