



# *Fun With Haskell: Introduction*

Nathaniel Wesley Filardo

January 10, 2012



## *Course Metadata*

### *Who am I?*

#### Handles:

- Nathaniel “Wes” Filardo
- nwf (“noof”)

#### Relevant Attributes:

- Fifth year Ph.D. candidate,
- Working for Jason Eisner on Dyna.
- Programming Language Enthusiast,
- A bit of a Haskell snob.

#### Course web page:

- <http://www.acm.jhu.edu/~nwf/fwh/>



## *Course Metadata*

### *Who are you?*

As of last count, 31 of you:

- 1 grad student
- 5 seniors
- 9 juniors
- 5 sophomores
- 11 freshmen

Everybody should have

- Some experience programming
- A laptop with the Haskell Platform



## *Course Metadata*

### *Course Goals*

- Experience with “functional” ways of thinking.
  - Living in the “Kingdom of Verbs” [6].
  - Change the way you think about programming (Perlis).
- Exposure to the Haskell language ...
  - Syntax (of course) but more *semantics*,
  - Strong, static, polymorphic typing,
  - Effect management (enter *monads*),
  - ...
- ... and ecosystem.
  - GHC(i): (Interactive) Compiler,
  - Hackage, Cabal: Package management
  - QuickCheck, HUnit, (Lazy)SmallCheck: Test frameworks
  - ...



## *Course Metadata*

### *Where do we go from here?*

- I think I know what needs to be presented, if I am to claim to have “shown Haskell.”
- The world of things relevant to Haskell and maybe this course is *much* larger!
- If people want, we can be
  - Theoretical, tea-sipping and monocled types,
  - Test-driven-development, benchmark-everything, dashes-everywhere focused,
  - Metaprogramming everything under the sun (Haskell, C, silicon, ...)
  - Statically-safe web developers.
- I know a little about each, and will learn anything to teach it if there's interest.



## *Course Metadata*

## *Course Structure*

- Lots of different backgrounds,
- Never enough time to cover everything,
- So I'm going to try playing it by ear.
- Using a mix of slides and interactive execution.
  - That means you should feel free to stop me with questions any time.



*What is Haskell?*

*What kind of a name is that, anyway?*

- Named after “Haskell Brooks Curry” (1900-1982)
- A pioneer in mathematical (combinatory) logic
  - SK calculus, Curry-Howard  $\simeq$ , Curry paradox, . . .
  - “Currying” is named after him (we’ll get there).



## *What is Haskell?*

### *Taxonomy*

Haskell is a programming language; it is . . .

- Functional, focused on actions to data;
- (mostly) Pure, eschewing side-effects;
- (mostly) Referentially transparent, for equational reasoning;
- Lazy, doing computation only when necessary;
- Statically typed, at compile time;
- Polymorphic, allowing generic functions (and data!);
- Type-classed, defining “interfaces” to different data;
- Monadic (in its library), for management of side-effects.

Don't worry if that's all new to you. We'll get there.





## *What is Haskell?*

### *Taxonomy*

If all you have is Java, perhaps the most immediately mind-blowing thing is *purity*. Data structures in Java are focused on mutation:

- Adding something to an `ArrayList<Foo>` *changes* the list.
- Iterators have exotic admonitions against modifying the thing being traversed. (e.g. “fail-fast” vs. “fail-safe”)

In Haskell, data is *data*. It's *just there*. It's **immutable**.

- The integer 23 is just that. It doesn't change. (Java's, too.)
- The list of integers `[1,2,3]` is just that. It doesn't change.



# *What is Haskell?*

## *Taxonomy*

Purity (or “purity by default”) is . . .

- More like math (easier to think about).
- Easier to *optimize*.
- Easier to *parallelize*! (The future is again multicore. . .)



## *What is Haskell?*

### *History of the “Ancient” variety*

An excellent paper captures more than I have time for: “A History of Haskell: Being Lazy With Class” [4]. To give some hilights:

- 1950s: John McCarthy invents Lisp.
- 1960s: Researchers (Landin, Strachey, Scott) use Church’s  $\lambda$  calculus for capturing semantics of programs.
- 1970s: Sussman and Steele’s Scheme: Lisp closer to  $\lambda$ .
- 1976: Lazy evaluation enters the scene.
- 1979: Milner invents ML and its type system.
- 1987: Lots of lazy, functional languages and implementations.
  - Mostly small groups. Nobody had “critical mass.”
- 1987: Peyton Jones and Hudak hold a meeting at FPCA.



## *What is Haskell?*

### *History of the specification*

- 1990 (April 1): Haskell Report, Version 1.0.
- February 1999: The Haskell 98 Report.
  - More or less specifies the language we know and love as Haskell.
- December 2002: Revised Haskell 98 Report.
  - Mostly fixes minutiae.
- November 2009: Haskell 2010 announced.
  - Mostly agreeable minor changes to the language
  - That had been implemented already
- Ongoing: Haskell Prime: “an ongoing process to produce revisions to the Haskell standard, incorporating mature language extensions and well-understood modifications to the language.” [1]



*What is Haskell?*  
*Why use Haskell?*

- Declarative, functional programming model.
  - More expressive yet simpler code.
  - Better maintainability.
  - Smaller “semantic gap.”
- Strong, static typing with an expressive type system.
  - Type inference means types usually sit out of your way.
- Good performance in most cases. Ongoing research for
  - better single-core performance,
  - automatic multi-core handling,
  - GPGPU,
  - ...
- Improved programmer productivity?
- Moving out of being a “boutique” “research” language.



*What is Haskell?*

*Am I going to be all alone if I use this?*

There is a growing, vibrant community:

- <http://www.haskell.org/>;
- <http://planet.haskell.org/> (blog aggregator);
- <irc://chat.freenode.net/#haskell>;
- [haskell-cafe@haskell.org](mailto:haskell-cafe@haskell.org);
- Reddit, StackOverflow, ...

with a wide range of members:

- Casual users & hobbyists, researchers, industrial users, ...
- “The Haskell Elders” are on IRC, -cafe, and Reddit.



## *What is Haskell?*

### *A Little Name Dropping Never Hurt Anybody*

- Industry:
  - Code processing: Facebook
  - Embedded Programming: Eaton, NASA
  - Game engines: iPwn Studios
  - HFT: Allston Trading
  - Infrastructure: Google
  - Modeling: Credit Suisse
  - Verification: Galois, MITRE
- Researchers:
  - Papers in every major, modern conference.
  - Dedicated Implementors Workshop and Symposium at ICFP.
- Serious web development frameworks:
  - We've got three: HAppS, Snap, and Yesod.



*What is Haskell?*  
*Am I going to be all alone if I use this?*

Some tools are just so important they deserve front-and-center mention:

- The Haskell Platform: Haskell, Batteries Included.
- Hackage: the Haskell package repository
  - <http://hackage.haskell.org/>
- Hoogle: type-directed search engine
  - <http://www.haskell.org/hoogle/>
- Cabal and cabal-install
  - Want something from hackage?
  - `cabal update && cabal install $PACKAGE`





## *Let's get started*

### Disclaimer:

*Much of today's material and presentation ordering comes from the first part (sections 3 and 4) of Hal Daumé's excellent "Yet Another Haskell Tutorial" [5].*

### Another disclaimer:

*Many of you probably already know some or all of this. Sorry. If you feel that something could be said more clearly, please don't hesitate to speak up.*



*Let's get started*

Ladies and Gentlemen, the canonical introduction:

HelloWorld.hs

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

```
$ runhaskell HelloWorld.hs  
Hello, World!
```



*Let's get started*  
*Haskell as a Pocket Calculator*

Let's play with some **expressions** in ghci:

```
Prelude> 3+4*5
23
Prelude> (1 + sqrt 5) / 2
1.618033988749895
Prelude> lcm 112358 1248
70111392
```

Note lack of parentheses for arguments to “sqrt” and “lcm”.



## *Let's get started*

### *Bindings*

What if we actually want to use something later? We can give something a name with a **let** binding:

```
Prelude> let phi = (1 + sqrt 5) / 2
Prelude> phi^2
2.618033988749895
Prelude> let fi = 112358 in lcm fi 1248
70111392
Prelude> fi
<interactive>:1:1: Not in scope: 'fi'
```

Haskell variables always begin with a lower-case letter (any lower-case unicode will do), and can involve alphanumerics, unicode, underscore, and single quotes.



## *Some Data*

### *Booleans*

Boolean values tell us whether something is **True** or **False**.

```
Prelude> 1 < 2
True
Prelude> 2 < 1
False
Prelude> if 1 < 2 then "Yes!" else "No!"
"Yes!"
```

(Like Java's `boolean` (and unlike C): booleans are not numbers; try `1 + True`; the error message might not make sense right now.)



## *Some Data Pairs*

Having single numbers is great and all, but what about **pairs** of things?

```
Prelude> ("phi", phi)
("phi",1.618033988749895)
Prelude> fst ("phi", phi)
"phi"
Prelude> snd ("phi", phi)
1.618033988749895
```



*Some Data  
Pairs*

We can make bigger things:

```
Prelude> (1, "phi", phi, 'f')  
(1,"phi",1.618033988749895,'f')
```

And **nested** things:

```
Prelude> ( (1,2), (phi, "phi") )  
((1,2),(1.618033988749895,"phi"))
```

Behold: a pair of pairs.



## *Some Data Lists*

What about arbitrarily-sized collections of things? For that, we need **lists**:

```
Prelude> let xs = 1:2:4: []  
Prelude> head xs  
1  
Prelude> tail xs  
[2,4]
```

`[]` denotes *the* empty list. (There's really only one!)





## *Some Data Lists*



A list is an odd creature: it has only a head and a tail, where the tail is itself a creature with only a head and a tail, and so on. [2]



## *Some Data Lists*

The formal definition of a list takes some unpacking:

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

- “A list of things (“a”) is either empty ([]) or a thing followed by list of things.’
- The **constructors** [] and : are called “nil” and “cons”.
- The arguments to cons are the **head** “thing” and **tail** list.
- We’ll come back to the mysterious “a”.

## *Some Data*

### *Lists*

Of course, we have a library of functions for manipulating lists:

```
Prelude> let xs = 1:[2,4,8]
Prelude> length xs
4
Prelude> sum xs
15
Prelude> all (<10) xs
True
Prelude> filter (>5) xs
[8]
```



## *Some Data*

### *Lists*

Strings are lists of **Chars**. We can append lists with the ++ operator:

```
Prelude> let hw = 'H':"ello, " ++ "World!"
Prelude> hw
"Hello, World!"
```

Non-string things can (often) be made into strings by **showing** them:

```
Prelude> "The number is " ++ show (1*2*3)
"The number is 6"
```



## *Functions*

- Changing gears!
- Thus far: “stuff”-oriented introduction.
  - Have to have at least *some* stuff; stuff is handy.
- Now: “doing things” to “stuff.”

## *Functions*

A first function:

### FirstFunc.hs

```
addtwo x = 2 + x
```

And now

```
*Main> addtwo 3  
5
```

And that's it. So let's go write some interesting functions...



## *Functions*

### *Pattern Matching*

Suppose I have a pair of numbers and I want to add one to each. How do I do this?

```
Prelude> let mypair = (1,2)
Prelude> mypair + 1
... No instance for bla bla bla
Prelude> mypair + (1,1)
... No instance for different bla bla bla
```



## *Functions*

### *Pattern Matching*

Suppose I have a pair of numbers and I want to add one to each.

Need to **deconstruct** (**match**) the pair and get at the (delicious) numbers inside.

```
Prelude> let mypair = (1,2)
Prelude> case mypair of (a,b) -> (a+1,b+1)
(2,3)
```

Don't *have* to package them back up. Maybe I want to add them together:

```
Prelude> case mypair of (a,b) -> a+b
3
```





## *Functions*

### *Pattern Matching*

Pattern matching is a popular thing for functions to do:

#### FirstFuncMatch.hs

```
myfunc pab = case pab of
  (a,b) -> "The answer is: " ++ show (a+b)
```

And then

```
*Main> myfunc (3,4)
"The answer is: 7"
```



## *Functions*

### *Pattern Matching*

In fact, it's so common to write things like this that there's syntactic sugar:

#### FirstFuncMatchSugar.hs

```
myfunc (a,b) = "The answer is: " ++ show (a+b)
```

And still

```
*Main> myfunc (3,4)  
"The answer is: 7"
```



## *Functions*

### *Passing Functions Around*

Suppose we find ourselves manipulating the first element of a pair by itself frequently:

```
foo (a,b) = (a+1,b)
bar (a,b) = (2*a,b)
{- ... -}
```

That's a lot of the same thing over and over.



## *Functions*

### *Passing Functions Around*

Want to be able to somehow say “do *something* to the first element” and later fill in the something:

```
mapFst f (a,b) = (f a, b)
```

Now we can write things like `mapFst (+1) (3,4)`.



## *Functions*


### *Passing Functions Around*


- This is a key part of “functional” programming: functions are “stuff” (formally: values) as well as being functions.
  - In C, you can have “function pointers” which are sort of close.
  - In Java, you have to box up a function in a class as a method. Ick!
- In fact, it’s so common to want to have a function in Haskell that there’s sugar for **anonymous** functions:


```
Prelude> map (\x -> x*x) [1,2,3]
[1,4,9]
```


- Can even take multiple arguments:  $\lambda x y \rightarrow x + (2*y)$ .

## Bib

 Available from: <http://hackage.haskell.org/trac/haskell-prime/>.

 Available from: <http://wadler.blogspot.com/2009/11/list-is-odd-creature.html>.

 Available from: <http://courses.cms.caltech.edu/cs11/material/haskell/index.html>.

 Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler.

A history of haskell: being lazy with class.

In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

○○○  
○○  
○  
○○○○  
○○  
○○  
○○○○○○○○○  
○○○

Available from: <http://doi.acm.org/10.1145/1238844.1238856>,  
doi:<http://doi.acm.org/10.1145/1238844.1238856>.



Hal Daumé III.

Yet another haskell tutorial.

2002–2006.

Available from: <http://www.cs.utah.edu/~hal/htut/>.



Steve Yegge.

Execution in the kingdom of nouns, 2006.

Available from: <http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html>.