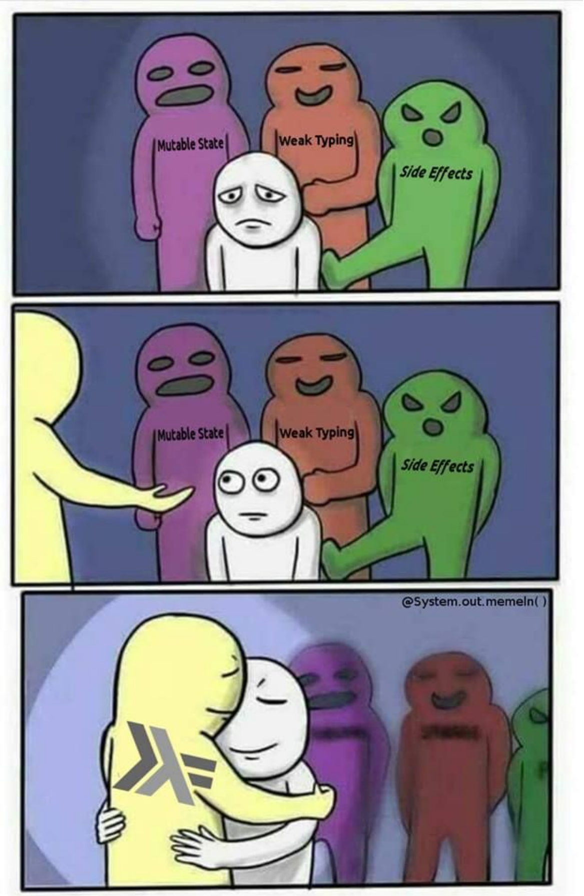
Haskell

`mapM_ putChar "Josh Bicking"`

What's Haskell?

- Haskell is a functional, lazy, pure language.
- Functional
  - Program logic is functions and data (and functions *as* data).
  - Focused on statelessness: instead of changing variables, you call functions, which call other functions, and so forth.
- Lazy
  - Nothing is evaluated until it's needed.
  - The value of unused variables isn't calculated.
  - `x = 1/0` won't throw an error, unless you try to use `x`!
- Pure
  - Variable and function names can't be overwritten once set.
  - `x = x+1` makes no sense.



Try Haskell yourself!

- Any lines starting with λ> can be given to a Haskell interpreter.
  - You can follow along and try things yourself at https://repl.it/languages/haskell
    - Be sure you type into the interpreter (the terminal prompt). The left part is for writing executables.
  - If you're feeling more adventurous, download and install Haskell through stack: https://www.haskellstack.org/
    - Once it's complete, open an interpreter with `stack ghci`

Syntax and Structure

Haskell

Goodbye, S-expressions!

- Lisp haters rejoice: Haskell tries to avoid those dreaded parentheses.
- Some functions, like **+**, have a special prefix and infix notation.
    - Most just have a prefix notation.
    - Functions without a special infix notation may be used infix by surrounding them with backticks.

```
λ> 2 + 2
4
λ> (+) 2 2  -- prefix notation
4
λ> quot 33 5
6
λ> 33 `quot` 5  -- using functions as infix
6
```

Type Signatures

- Structure
  - 0 or more inputs that result in an output.
  - Can specify data types or type restrictions.
- Data types
  - Takes data of that input type.
- Type restrictions
  - Takes data that satisfies the category restrictions placed on the input.
- Higher Order Functions
  - Functions given as data are subject to the same type signatures.

```
λ> :t replicate
replicate :: Int -> a -> [a]
λ> :t (+)
(+) :: Num a => a -> a -> a
λ> :t (< 3)
(< 3) :: (Num a, Ord a) => a -> Bool
λ> :t map
map :: (a -> b) -> [a] -> [b]
```

Definitions

- Everything is a function. Mostly.
  - x is just data. However, its type signature suggests it's a function that takes no arguments and returns a number.
  - Functions and "data" are declared the same way.
- Note: The interpreter will let you "redefine" x. This is for convenience in the interpreter, and not allowed in compiled Haskell.
  - Also, `x= x+1` still won't work like it does in other languages.

```
λ> x = 5
λ> :t x
x :: Num t => t
λ> squaredAdd a b = a^2 + b^2
λ> :t squaredAdd
squaredAdd :: Num a => a -> a -> a
λ> squaredAdd 2 3
13
```

Flow Control

- We have a familiar looking `if`.
- Also have an interesting `case`.
  - Uses **Pattern Matching**: data matches a specific structure.
- Conditionals and pattern matching can also be used as part of a function definition.
- *Side note*: Outside of the interpreter, Haskell is structured with indentation and line breaks.

```
λ> if 5 == 6 then "foo" else "bar"
"bar"

λ> isEmpty l = case l of; [] -> True; otherwise -> False
λ> isEmpty [1,2,3]
False
λ> isEmpty []
True
```

```
-- Case statements look nicer outside of the interpreter.
isEmpty l =
  case l of
    [] -> True
    otherwise -> False
```

. and $

- Haskell lets you keep structure, without throwing in tons of parentheses.
  - `($)`: Give precedence to the right of the $.
  - `(.)`: Chain functions together: take the output of the right, and apply it as an argument to the left.
    - Meant to look like the mathematical function composition operator, ∘.

```
a b c d e  -- "Call function a, with arguments b, c, d, e"

a b (c d e)  -- "Call function a, with arguments b, and the result of calling c with arguments d, e."
a b $ c d e  -- The same as above.

a (b (c d))  -- "Call c with arguments d, e. Apply the result of c to b, then apply the result of b to a.
(a . b . c) d  -- The same as above.
a . b . c $ d  -- The same as above.
```

Why use Haskell?

The Theory Underneath

- Haskell is based off some really cool constructs!
  - Category Theory
  - Theoretical Computer Science
  - Programming language theory
- I won't go too deep into these, just why they help Haskell do what it can do.
  - The theoretical constructs give Haskell a lot of practical advantages.

Strict, extensive type system

- No casting
  - Turning an Integer into a Double requires a function that takes an Integer and returns a Double.
- Typeclasses are (optionally) inferred by the compiler.
  - Typeclass is determined by how the data is used.

```
λ> fun1 a = a
λ> :t fun1
fun1 :: p -> p

λ> fun2 a b = a < b
λ> :t fun2
fun2 :: Ord a => a -> a -> Bool

λ> fun3 a = a < 3
λ> :t fun3
fun3 :: (Ord a, Num a) => a -> Bool
```

Referential Transparency

- You may substitute the right hand side of a declaration, in any context.
  - The meaning doesn't change.
- Immutability guarantees a function's result is determined **only** by its input.
  - No concept of state!
- Cool use case: "Hotswapping Haskell" for Facebook's spam filter
  - Functions are updated on the fly.
    - New objects are swapped in.
    - Old objects are marked for garbage collection.

```
λ> f a b = a + b
λ> x = 3
λ> y = 5

λ> f x y
8
λ> f 3 5  -- Substitute x and y
8
λ> x + y  -- Substitute f
8
```

Parallelism is Easy!

- Functions don’t modify each other, so we can run them simultaneously without worrying.
    - `par a b` lets you evaluate a and b simultaneously.

```
import Control.Parallel (par)

factorial n = product [1..n]

let
  x = factorial 20000
  y = factorial 30000
in
  par x (par y (x - y))
```

For those of you in an interpreter (this probably won’t work on repl.it):

```
λ> import Control.Parallel (par)

λ> factorial n = product [1..n]
λ> let { x = factorial 20000; y = factorial 30000 } in par x (par y (x - y))
```

Laziness: It’s a good thing

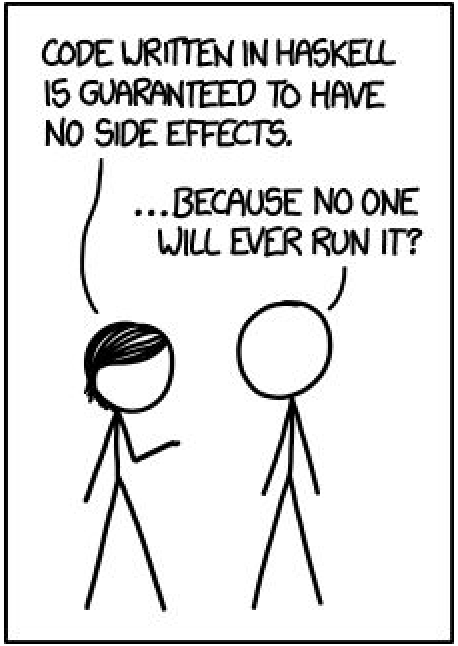
- Elements that are never used are never evaluated.
- Declare a huge, or infinite list, and take what you need from it.
    - *A program to solve Sudoku*, by Richard Bird
        - `sudoku :: Board -> [Board]`
        - For any board configuration, compute all possible ways to fill it.

```
λ> x = [1..]
λ> x !! 10
11

λ> take 5 x
[1,2,3,4,5]

λ> show x -- This would loop forever!
```

Why use Haskell?

A program becomes a number of side-effect free, strongly typed functions.

This leaves very little room for runtime errors.

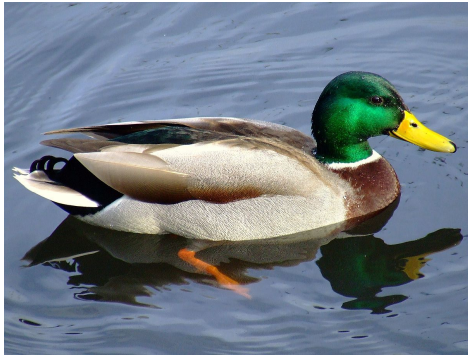


A Touch of Theory: The Type System

Haskell

Duck Typing on Steroids

- **Duck typing**: “If it waddles and quacks like a duck, then it’s probably a duck.”
    - The type of data is inferred: it doesn’t have to be specified.
- Let’s say we have a `Duck d`. It can waddle.
    - Python
        - `d.waddle()` - ✓
        - `d.ribbit()` - Runtime error
    - Haskell
        - d is a Duck data type, and Duck is part of the Waddles typeclass. - ✓
        - d is a Duck data type, and Duck is not part of the Ribbits typeclass. - *Compile time* error.



Python also uses “duck typing”.

```
>>> x = 3
>>> type(x)
<class 'int'>
>>> x = 3.0
>>> type(x)
<class 'float'>
```

Category Theory in the Type System

Because there aren't any papers on Duck Typing Theory.

- If a data type can implement what's necessary to be in a typeclass, then it belongs to that typeclass.
  - In Haskell, typeclasses are defined with **class** (not to be confused with a Java class).
  - To be in Eq, a data type must implement (==) and (/=), and their results must not be equal to each other.
- Offers data encapsulation and polymorphism without an OOP model.

```haskell
class  Eq a  where
    (==), (/=) :: a -> a -> Bool

        -- Minimal complete definition:
        --      (==) or (/=)
    x /= y     =  not (x == y)
    x == y     =  not (x /= y)
```

Something is missing…

I’ve left out an essential part of learning a new programming language.

Printing requires IO, and IO is a side effect: it changes the state of a system.

Haskell abstracts away side effects through monads.

```haskell
module Main where

main :: IO ()
main = putStrLn "Hello world!"
```

Monads: Bundling State

Let's look at some JavaScript

- This code is riddled with null checks.
  - Is there any way we can remove them?
- Haskell has a `Maybe` data type.

```
data Maybe t = Just t | Nothing
```

- A `Maybe` has some value wrapped in a `Just`, or it has no value, `Nothing`.

```
var person = {
    "name":"Homer Simpson",
    "address": {
        "street":"123 Fake St.",
        "city":"Springfield"
    }
};

if (person != null && person["address"] != null) {
    var state = person["address"]["state"];
    if (state != null) {
        console.log(state);
    }
    else {
        console.log("State unknown");
    }
}
```

Maybe in JavaScript

- Now we have a *unit function*
  - Returns a `Nothing` object if given `null` or `undefined`.
  - Returns a `Just` function if given a value, which returns the original value.

```
var Nothing = {};
var Maybe = function(value) {
  var Just = function(value) {
    return function() {
      return value;
    };
  };

  if (typeof value === 'undefined'
      || value === null)
    return Nothing;

  return Just(value);
};
```

Maybe some Examples

```
Maybe(null) == Nothing;   // true
typeof Maybe(null);   // 'object'

Maybe('foo') == Nothing;   // false
Maybe('foo')();   // 'foo'
typeof Maybe('foo');   // 'function'
```

```
var Nothing = {};
var Maybe = function(value) {
  var Just = function(value) {
    return function() {
      return value;
    };
  };

  if (typeof value === 'undefined'
      || value === null)
    return Nothing;

  return Just(value);
};
```

And just like that...

- This code is riddled with `Nothing` checks instead.
    - Yay?

```
if (Maybe(person) != Nothing &&
    Maybe(person["address"]) != Nothing) {
  var state = person["address"]["state"];
  if (Maybe(state) != Nothing) {
    console.log(state);
  }
  else {
    console.log("State unknown");
  }
}
```

Back to function composition

- What if we had a functional way to do what && is doing, but with `Nothing`s?
  - If any result is `Nothing`, then stop computing things and just return `Nothing`.

- Introducing bind
  - If we already have a `Nothing`, then return `Nothing`.
  - If we have a `Just value`, output is determined by the given value.

```
// For Nothing
bind: function(fn) { return Nothing; }

// For Just value
bind: function(fn) {
        return Maybe(fn.call(this, value));
    }
```

bind() in action

```
// For Nothing
bind: function(fn) { return Nothing; }

// For Just value
bind: function(fn) {
        return Maybe(fn.call(this, value));
      }
```

```
var address = Maybe(person).bind(
  function(p) {
    return p["address"];
  });
address === Nothing  // false

var fake_address = Nothing.bind(
  function(p) {
    return p["address"];
  });
fake_address === Nothing  // true

var state = Maybe(person).bind(function(p) {
  return p["address"];
}).bind(function(a) {
  return a["state"];
});
state === Nothing  // true
```

Doing something with the result

- If the result is `Nothing`, we should have some sort of fallback or default behavior.
- Otherwise, we should do something with its contents.
  - Extract `value` from `Just value`, and apply it to `fn`.
  - If we just want to print, we can give the identity function as `fn`.

```
// For Nothing
maybe: function(def, fn) {
      return def;
    }

// For Just value
maybe: function(def, fn) {
        return fn.call(this, value);
       }
```

Maybe some more examples

```
Maybe(3).maybe("not a number", function(a) { return a+2; });  // 5

Maybe(null).maybe("not a number", function(a) { return a+2; });  // "not a number"

// Combining two "Maybe"s isn't the prettiest with this implementation, but it's possible.
Maybe(3).maybe("not a number", function(a){
  return Maybe(5).maybe("not a number", function(b){
    return a+b})});  // 8
```

Why do we have to call maybe() twice?

Maybe we have a solution

- The result of each `bind` function is passed forward.
  - If we have something at the `maybe()`, we print it.
  - Otherwise, we print the default.

```
console.log(Maybe(person).bind(function(p) {
  return p["address"];
}).bind(function(a) {
  return a["state"];
}).maybe("State unknown", function(s) {
  return s;
}));
```

The entire Maybe implementation

```
var Nothing = {
  bind: function(fn) { return Nothing; },
  maybe: function(def, fn) {
    return def;
  }
};
var Maybe = function(value) {
  var Just = function(value) {
    return {
      bind: function(fn) { return Maybe(fn.call(this, value)); },
      maybe: function(def, fn) {
        return fn.call(this, value);
      }
    };
  };
  if (typeof value === 'undefined' || value === null)
    return Nothing;
  return Just(value);
};
```

Maybe we have a monad

- Monads allow “packaging” of data.
    - Done in such a way that allows “chainable” usage.
    - Kind of like putting a value in a box, giving it to someone to open, and they place it in another box.
        - However, there’s rules stating how functions should operate when the box is opened in a particular way.

```
λ> :t (>>=)
(>>=) :: Monad m => m a -> (a -> m b) -> m b
λ> :t return
return :: Monad m => a -> m a

λ> Just 3 >>= \x -> return $ (+) 1 x
Just 4
λ> Nothing >>= \x -> return $ (+) 1 x
Nothing
```

```
λ> return 3 :: [Int]
[3]
λ> [] >>= show
""
λ> [1,2,3,4,5] >>= show
"12345"
```

Our solution in Haskell

```
λ> data Person = Person { name :: String , addr :: Maybe String}
λ> buddy = (Just (Person "Buddy" (Just "123 Moon Ave")))
λ> putStrLn $ maybe "No addr" id $ buddy >>= addr
123 Moon Ave
```

- To make things easier to follow, we won't nest an Address data type.
- Both the Person and their address are optional.

Relevant data types:

```
Person :: String -> Maybe String -> Person
Just :: a -> Maybe a
putStrLn :: String -> IO ()
maybe :: b -> (a -> b) -> Maybe a -> b
id :: a -> a
```

Monads, this time with sheep

- We have some `Sheep` datatype. We also have a sheep family tree database.
    - `father` returns the father of the sheep, if we know the father.
    - `mother` returns the mother of the sheep, if we know the mother.

```
type Sheep = ...

father :: Sheep -> Maybe Sheep
father s = ...

mother :: Sheep -> Maybe Sheep
mother s = ...
```

How far can we go?

- Going two generations isn't too bad.
- However, it quickly gets ugly.
- We don't need all these checks: a `Nothing` at any step results in a `Nothing`.

```
maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s =
  case (mother s) of
    Nothing -> Nothing
    Just m  -> father m

mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s =
  case (mother s) of
    Nothing -> Nothing
    Just m  -> case (father m) of
      Nothing -> Nothing
      Just gf -> father gf
```

Using >>=

- Binding results together makes for a much cleaner solution.
- Any `Nothing` along the chain of binds will result in a `Nothing` being returned.

```
maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s = mother s >>= father

mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = mother s >>= father >>= father
```

Monads and Lists

```
(>>=) :: [a] -> (a -> [b]) -> [b]   -- In the
case of the List monad.

lst >>= f = concat (map f lst)
```

- Many structures in Haskell are represented as monads, to allow for composition.
  - Lists
    - Could be the [ ], [a], [a , a]...
    - We can operate on what's inside them in a similar way, if we want.
    - We can also think of it as "extracting" a value from its "list" context.
- Bind works differently for different monads, but produces the same result.
  - A value is extracted from the monad, and then placed in it again.

```
λ> :t replicate
replicate :: Int -> a -> [a]
λ> ["sheep"] >>= replicate 3
["sheep","sheep","sheep"]

λ> [1,2,3] >>= (\x -> return $ 3 + x)
[4,5,6]
```

Monads, Haskell, and sweet flow control

- Haskell gives the programmer more control over state, and composition of state.
  - Bundling state into monads means it changes in a trackable, predictable way.
  - Structure hands itself nicely to using closures and continuations.

```
exp = x >>= (f1 >>= f2) >>= f3
-- At each point, the exp is equal to:
exp = closure >>= continuation
```

- **Continuation**: representation of control flow
- **Closure**: a function with contextual information, given from its state.
  - Some value with context is fed into an environment that requires that value to complete.
  - *We can see these values at each step.*
    - Check them for validity.
    - Record them, allowing us to track state and *undo that state*, if necessary.

References and Further Information

- Free resource (plenty of introductions for concepts): https://en.wikibooks.org/wiki/Haskell
- Much more comical, free resource: http://learnyouahaskell.com
- Hoogle: A search engine for functions and type signatures: https://www.haskell.org/hoogle
- Building a small parser: https://wiki.haskell.org/Parsing_a_simple_imperative_language
- Facebook's Haskell spam filter:
  https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell
  https://simonmar.github.io/posts/2017-10-17-hotswapping-haskell.html
- Maybe monad in Javascript: http://sean.voisen.org/blog/2013/10/intro-monads-maybe
- Building a Sudoku solver: http://www.cs.tufts.edu/~nr/cs257/archive/richard-bird/sudoku.pdf
- xmonad, a tiling window manager written and configured in Haskell: http://xmonad.org
- Backtracking with monads:
  https://www.schoolofhaskell.com/user/agocorona/the-hardworking-programmer-ii-practical-backtracking-to-undo-actions