YES WE CAN HASKELL!
self 2014-05-30 – 11:00~11:50

Intro

Introductory deep dive. I will cover numerous aspects of Haskell, all at a surface-level. I assume you are unfamiliar with Haskell and functional programming.
The goal is to give you a feel for Haskell and get you interested/excited about the language.

Let's write a function!

We're going to help the kitty write his Haskell.

Start with Hello World:

```
main :: IO ()        ← Function signature is optional (type inference).
main = putStrLn "Hello, world!"

runhaskell ywch.hs  ← Interpret code.
```

Introduce the REPL.

Next, I'll show you how to write a quick sort function. Example of elegant Haskell code.

```
qs :: [a] -> [a]
qs []     = []
qs (x:xs) = lhs ++ [x] ++ rhs
  where
    lhs = filter (<= x) xs
    rhs = filter (>  x) xs
```

Note that whitespace has syntactic meaning. The horizontal indentation of code delimits code blocks.

```
λ> qs [3,2,1,5,4]← Function application does not require parenthesis like it does in C family.
[2,1,3,5,4]
```

[cont...]

Make the function recursive.

```
qs :: (Ord a) => [a] -> [a]
qs []     = []
qs (x:xs) = qs lhs ++ [x] ++ qs rhs
  where
    lhs = filter (<= x) xs
    rhs = filter (>  x) xs
```

`(Ord a) =>` specifies a constraint on the type variable `a`. `a` must be a type that can be compared in terms of greater than/less than.

`qs [] = []` is what terminates the recursion, because `[]` indicates the end of an empty list:

```
λ> 1:2:3:4:5

<interactive>:2:1:
    No instance for (Num [a0]) arising from a use of `it'
    Possible fix: add an instance declaration for (Num [a0])
    In a stmt of an interactive GHCi command: print it
λ> 1:2:3:4:5:[]
[1,2,3,4,5]
```

So, our Haskell implementation of quick sort is:
elegant
concise / terse (not verbose like Java)

It's nice to be able to use bindings in the body of the function that are actually defined below, in the `where` block.

What is this Haskell?

much high level:
C = hammer and nails
Java = power tools
Haskell = abstractions from mathematics
These abstractions allow us to write our code, and reason about it, at a very high level.
At such a high level, it can be comparatively harder to reason about performance, space, and time.

Is Haskell really so difficult?

Monad, monoid, functor... these abstraction from mathematics are heavily used in Haskell. Most mainstream programmers have never heard of them.
They are just typeclasses, which are very similar to interfaces in OOP languages. They are highly abstract patterns with many practical applications.

The budding Haskell programmer simply needs to become familiar with these patterns, and understand them on the typeclass level. (The Haskell programmer need not understand the mathematics behind them.)

Many Haskell programmers will tell you that understanding monad, monoid, and functor is no more difficult than wrapping your head around the various concepts under the umbrella of OOP.

Haskell is not easy, but it's not impossibly difficult, either. You'll need:
time, determination, willingness to ask the community for help when you get stuck

---

very pure funkshunal

Functional programming is a programming paradigm/style.
It is the only style supported in Haskell; there is no support for OOP.

A good way to begin to understand functional programming is to compare it to imperative programming.

Imperative programming is "programming that makes extensive use of assignment."
Computations are described in terms of statements that change a program state.

In a purely functional program, all bindings are immutable.

```
x = 5
-- ...
x = 6        ← Multiple declarations of x!

x = x + 1  ← Multiple declarations of x!
```

Q: What happens if you have a data structure, and you want to change just part of it?
A: Make a copy of the existing data structure, with that part changed.

[cont...]

Introduce record update syntax:

```
data Record = Record { name :: String
                     , email :: String
                     , age :: Int } deriving (Eq, Show)
```

`deriving (Eq, Show)` automatically implements the typeclasses (which are like interfaces) for us. This is a bit like getting a free `equals()` and `toString()`.

The first `Record` is the name of our new data type. The second `Record` is called a "value constructor." (You can have more than one value constructor, as in `data Color = Red | Green | Blue` – this particular example is like an enum.)

```
jason :: Record
jason = Record "Jason Stolaruk" "jason@detroitlabs.com" 35

λ> jason
Record {name = "Jason Stolaruk", email = "jason@detroitlabs.com", age = 35}
λ> name jason
"Jason Stolaruk"
λ> email jason
"jason@detroitlabs.com"
λ> age jason
35

λ> let olderJason = jason { age = 36 }

λ> olderJason == jason
False

λ> olderJason
Record {name = "Jason Stolaruk", email = "jason@detroitlabs.com", age = 36}

λ> jason
Record {name = "Jason Stolaruk", email = "jason@detroitlabs.com", age = 35}
```

Consider a program in which every binding is immutable (like a Java program in which every variable is "final"). What would that look like?

How can you even write a `for` loop, when a `for` loop requires an index variable that is updated (mutated) with each iteration?
In fact, there are no `for` loops in Haskell! Consider the following:

[cont...]

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

{-
"sumThem [1..5]" expands like so:

sumThem 1 + sumThem [2,3,4,5]
sumThem 1 + 2 + sumThem [3,4,5]
...and so on
-}
```

Why immutability?

Mutability introduces a lot of complexity into a program. Variables can and will vary. How can you be certain that a given variable will have the value you expect? (Think of all the NPEs you've been burned by...)

In Haskell, there are no variables. There are only functions and bindings.

Referential transparency:
Every time you call a given function with the same argument/value, it is guaranteed to return the same result.
There is no dependency between a function's behavior and global state. The two are entirely decoupled.

Thus embracing immutability makes it easier to reason about your code, and feel confident about your code.

(More) very pure funkshunal

Let's toss around some functions!

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

A string is the same as a list of characters.

```
λ> ['a', 'b', 'c'] == "abc"
True

λ> import Data.Char

λ> :t toUpper
toUpper :: Char -> Char

λ> map toUpper "Hello, world!"
"HELLO, WORLD!"

λ> :t ord
ord :: Char -> Int
```

```
λ> map ord "Hello, world!"
[72,101,108,108,111,44,32,119,111,114,108,100,33]

λ> map (odd . ord) "Hello, world!"
[False,True,False,False,True,False,False,True,True,False,False,False,True]
```

Partial application:

```
λ> map (*2) [1..10]      ← (*) is just a function.
[2,4,6,8,10,12,14,16,18,20]

λ> :t compare
compare :: Ord a => a -> a -> Ordering

λ> compare 'a' 'b'
LT
λ> compare 'a' 'a'
EQ
λ> compare 'b' 'a'
GT

λ> 'a' `compare` 'b'
LT

λ> map (`compare` 'o') "Hello, world!"
[LT,LT,LT,LT,EQ,LT,LT,GT,EQ,GT,LT,LT,LT]
```

Of course, we can define our own functions and map them.

```
λ> let bang s = s ++ "!"  ← Type inference! We don't explicitly give the type of s.

λ> :t bang
bang :: [Char] -> [Char]

λ> bang "yes"
"yes!"

λ> bang "Yes we can Haskell"
"Yes we can Haskell!"

λ> map bang (words "Yes we can Haskell")
["Yes!","we!","can!","Haskell!"]
```

---

such lazy eval

Q: Why might it be a sad thing that `bar` is called?
A: `bar` might not have any side effects, in which case it would be a waste to call it, as its results are never used.

```
main :: IO ()
main = let x = 11
       in putStrLn (foo x (bar x))

foo :: Int -> Int -> Int
foo a b = if a > 10 then a else b

bar :: Int -> Int
bar x = x -- Do some time consuming operations on x...
```

The above code will not compile, because `putStrLn` must take a string as an argument. However, `foo` returns an integer.

Haskell is statically typed. The types of all bindings and expressions are known at compile time.
Haskell is strongly typed. There is no implicit type conversion.
All the types in your code must "line up," or your code won't even compile.
You will never get runtime errors due to type mismatching. *Code you can believe in.*

Let's fix this error:

```
main :: IO ()
main = let x = 11
       in putStrLn (show (foo x (bar x)))
```

`show` is the `toString()` of Haskell.

Let's get rid of all those parenthesis and make the code more Haskell-like.

```
main :: IO ()
main = let x = 11
       in (putStrLn . show . foo x . bar) x   ← Use $ here.
```

Now we can run our code in the REPL.

The question remains: is `bar` ever called? We can get our answer by using `undefined`.

```
λ> undefined
*** Exception: Prelude.undefined
```

The return type of `undefined` is entirely polymorphic, so we can use it anywhere.

[cont...]

```
bar :: Int -> Int
bar x = undefined -- Do some time consuming operations on x...
```

Sure enough, `bar` throws an exception when called:

```
λ> bar 11
*** Exception: Prelude.undefined
```

Right. So, what happens when we run `main`?

More illustrations of lazy evaluation:

```
λ> (0,'a')
(0,'a')
λ> :t fst
fst :: (a, b) -> a
λ> fst (0,'a')
0
λ> fst (undefined, 'a')
*** Exception: Prelude.undefined
λ> fst (0, undefined)
0
```

```
λ> :t take
take :: Int -> [a] -> [a]
λ> take 2 [1..5]
[1,2]
λ> take 2 [1,2,undefined]
[1,2]
```

(More) such lazy eval: How does it work?

What is the mechanism for this? Is lazy evaluation some kind of black magic?
Actually, it's very simple...

Thunks can be layered in the sense that the results of an unevaluated expression can depend on the results of other as-of-yet unevaluated expressions, and so on.

The top (root) of this layering can extend all the way up to the `main` function (the entry point of the program).

(More) such lazy eval: It's good to be lazy?

Efficiency:

Return to the sample Java program. Unnecessary, time-consuming work will be done. This is a classic example of how lazy can be more efficient than strict.

HOWEVER, laziness does introduce overhead due to the creation of thunks.

Infinite data structures:

Infinite lists are the classic example.

```
[1..]
100 `elem` [1..]

λ> ['a'..] !! 25 ← Zero-based index.
'z'
```

Note that `['a'..]` will actually stop when it reaches the end of the Unicode table.

Simpler, more elegant code:

```
λ> take 3 $ reverse [10000,9999..1] ← Both efficient and concise.
[1,2,3]
```

Strict evaluation would reverse the entire list before reaching the `take` method. If the list is very long, the operation would be significantly costly.

```
import Data.List
take 3 $ sort xs
```

Here, too, strict evaluation would sort the entire list before reaching the `take` method.
In the case of lazy evaluation, we only need sort to the extent that we identify the first 3 elements of the result. (Overall efficiency depends on the implementation of the sort function.)