What We Cover

- We will cover:
  - A fair amount of Haskell syntax.
  - Some things we do in similar ways in both Haskell and Ruby.
  - Some neat and powerful things Haskell does that Ruby doesn't.
- These slides and the example code files are all available in the github repo:

  `https://github.com/cjs-cynic-net/ruby2haskell-tutorial`
- If you know Ruby reasonably well (or can pick it up quickly!), most of the the Haskell code presented here should be easy to read and understand.
- If something seems too weird, well, remember that before you knew Ruby, this was weird, too:

  ```
  a.collect { |x| ... }
  ```

1. For this presentation, we assume the audience has a reasonably good knowledge of Ruby, particularly Ruby-ish idioms. That said, it shouldn't be too hard for users of other languages to get some sense of the Ruby idioms, and learn the Haskell ones.
2. It turns out, quite a number of things we do in Haskell are not so different from what we do in Ruby!
3. There's much more to Haskell than we have time for in this presentation, but hopefully what you see here will motivate you to learn more.

1. REPL stands for "Read Evaluate Print Loop."
2. In listings in this presentation I show Ruby code in red, Haskell code in blue, shell commands in green, and output in black. Sometimes listings will be shown with Ruby and Haskell code side by side.
3. I use the `-v0` option to `ghci` to reduce verbosity to make the output fit the slide; you should feel free to leave it out.
4. I encourage you to bring up these interpreters in windows on your laptops and type along as we go. If anybody has any problems making things run, stop and ask for help.

Some Simple Syntax

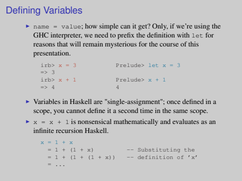
- Ruby uses `#` for comments; Haskell uses `--`, as below.
- (Haskell also offers nesting open/close comment delimiters that may span multiple lines, `{- like this -}`.)
- Basic constant expressions in Haskell are awfully similar to those in Ruby:

```
2           -- integer
3.1415      -- floating point number
[1,2,3]     -- list of integers
'x'         -- character (a type Ruby doesn't have)
"Hello."    -- list of characters, or string
```

- Expressions with binary operators are pretty much the same, though the particular operators are sometimes different:

```
2 + 3                   => 5
2 * (3 + 4)             => 14
[1,2] ++ [3,4]          => [1,2,3,4]
"Hello, " ++ "world."   => "Hello, world."
```

`++` is the list concatenation operator in Haskell. Why do we use it for strings? Because a `String` is just a list of characters, of course!



1. The reason for the use of `let` is that in the GHC interpreter we're operating in an evaluation environment, just as Ruby is all the time. ("`def f n; n + 1; end`" is actually code that's executed as it's read in Ruby.) But a Haskell program in a file is a list of definitions that's usually compiled; only when we run the Haskell program do we start evaluating those definitions.
2. Here we show the Ruby code (and results) on the left and the equivalent Haskell on the right, leaving blank lines where the Haskell interpreter prints no output. You'll note that here our definition was merely accepted, not evaluated.
3. In `ghci` you can re-define variables: `let x = 2` followed by `let x = 3` will work. This is because each `let` creates a new (sub-)environment, so the second definition will shadow the first.

Function or Variable?

- But wait, function definitions and variable definitions look awfully similar:

```
f x = 7         -- The 'x' argument is ignored.
g   = 7
```

- Is `g` a variable or a function with no arguments?
- Like Ruby, it mostly doesn't matter; often functions look like variables and you can use them as such.
- Unlike Ruby, it *really* doesn't matter: there is no difference!
- You can think of "variables" as functions with no arguments, if you like.
- But really, all functions are stored in variables, just like any other objects.

1. Ruby uses syntatic sugar to make things like `o.x = 1` be read as a function call if `o` has an `x=(val)` method on it. Syntactic sugar is generally considered a hack, albeit in many circumstances more welcome than its lack.

More Complex Expressions

- In Ruby (think `collect` instead of `map` if you're Smalltalk-oriented):

```
(0..7).map { |x| x*3+1 } .select { |x| x.even? }
=> [4, 10, 16, 22]
```

- In Haskell we prefix things, so the "flow" runs the other way around:

```
filter (\x -> even x) (map (\x -> x*3+1) [0..7])
[4,10,16,22]
```

- Haskell's `\x -> ...` is lambda syntax, similar to Ruby:

```
f = lambda { |x| x + 1 }     # Ruby
f = \x -> x + 1              -- Haskell
```

1. Here we apply filter to two arguments. The first is the selector function for the filter. The second is the result of the application of `map` to its two arguments.

Side Note: Ruby Gets Weirder

- Ruby has "lambdas" and "procs," and they're not the same thing, though they look awfully similar!

```
def call_lambda
    f = lambda { return; };     puts(f.class)
    f.call;                     puts("Printed!")
end
call_lambda  # Prints "Proc" followed by "Printed!"

def call_proc
    f = Proc.new { return; };   puts(f.class)
    f.call;                     puts("Not printed!")
end
call_proc    # Prints just "Proc"!
```

- More scary details at:

  `http://www.skorks.com/2010/05/ruby-procs-and-lambdas-and-the-difference-between-them/`

1. Both `lambda` and `proc` produce `Proc` objects, but these two different types of `Proc` objects that behave differently when called, as demonstrated above. In Ruby 1.8 it doesn't appear to be possible to tell the difference between the two; Ruby 1.9 adds a `lambda?` method to the `Proc` class.
2. This is actually rather typical of the mess Ruby often makes when it could be doing things in a much simpler and more consistent way. Functional languages tend to do a lot better, even at just the syntax level. (For whatever reason, functional languages tend to have simpler syntax than not only Ruby but even languages such as Java or Javascript.)

Lack of Order Dependency Makes Haskell Easy

- In Haskell, without order dependency issues, we merely need to look at the nearest scope with the definition.

```
x   = 100
f   = x + 1
g x = x + 1
h   =  let x = 300  -- "local variable"
        in x + 1
```

- Load up this example file in the interpreter and see what evaluating the various definitions gives you:

```
$ ghci -v0 ex/01_defs.hs
Prelude> g 200
201
```

- For any use of `x`, it's easy to find its definition.

1. There's a parallel here between Ruby and Haskell and dynamically scoped (e.g., Emacs Lisp) and lexically scoped (e.g., Scheme) languages. It's all about how much you need to remember about the current environment when examining an evaluation.

How Does Pattern Matching Work?

- Let's see how this worked:

```
find p (x:xs) = if p x then x else find p xs
```

- `:` is the operator we use to prepend (cons) an element on to a list, constructing a new list:

```
Prelude> 2:1:[]
[2,1]
Prelude> 3:[2,1]
[3,2,1]
```

- The pattern `x:xs` above does the reverse operation, deconstructing the components of the list in to the given variables. The head goes in to `x` and the tail in to `xs`.
- This is not special compiler support for the list constructor; this works on all user-defined data types.
- We use this a *lot*.

1. Programmers who use languages with pattern matching not only use it a lot, but also tend to look with pity on those without it. It's one of those "blub" things where you just can't imagine living without it once you've used it.

The Missing Pattern Found

- Though it's not much different from what's happening anyway, let's make `find` throw an exception with a better error message on failure.

```haskell
import Control.Exception

find _ [] = throw (PatternMatchFail "Not found.")
find p (x:xs) | p x       = x
              | otherwise = find p xs

main = print (find (\x -> x > 7) [1,2,3,4,5])
```

- We can have multiple definitions of a function so long as their patterns don't overlap. The first definition with a matching pattern is used.
- The underscore as the pattern for the first argument matches anything, and says that we're ignoring it.

1. There are better ways of handing this than throwing an exception at runtime, in particular, informing the programmer at compile time that he needs to deal with this potential error condition. We'll see how this is done later on.
2. The observant among you will note that chosing the first matching definition of multiple definitions breaks the "order doesn't matter" rule. Well, nobody's perfect.

Type-checking Advantages: Efficiency

- A type-checked program can be more efficient at run-time because it can do work at compile time to avoid run-time checks.
- Consider: in Ruby, a function's parameter can be any object (in this sense, variables are untyped), and so an object must be queried at run time to see if it supports an operation.
- If the check has been done at compile time, we know at run time what the object must be, because the compiler guarantees it.
- We can sometimes use a (more efficient) wider type at runtime knowing some values will not be present, e.g., a signed int that holds a smaller range of unsigned values (e.g., an enumeration). This is known as *type erasure*.

1. The run-time type query may be automatic, as when you attempt to call a method on an object, or done by the programmer, as when you query an object with `respond_to?`.

Type Declarations

- But often we want to declare our types, either to avoid ambiguity and confusion or because we trust ourselves to write the type more reliably than we can write the code.
- In `ex/05_typedecl.hs` we have our first examples of this:

```
go :: Color
go = Green

aoi :: Color -> Bool
aoi Red   = False
aoi Green = True
aoi Blue  = True
```

- The first declaration says, "`go` is bound to a value of type `Color`."
- The second, "`aoi` is bound to a function that, when applied to a value of type `Color`, evaluates to a value of type `Bool`."

1. In `ex/05_typedecl.hs` we add a `deriving` statement to the type declaration. This asks the compiler to automatically write a `show` function for us that can generate strings representing the values of the type.

Point-free Style

- In the previous slide, we didn't use an explicit parameter like this:

```
matchingBlue :: Color -> Bool
matchingBlue c = matching Blue c
```

- But no need for the syntax: functions are first-class values.
- Just as these are algebraically equivalent:

$$f(x) = g(x)$$
$$f = g$$

  so is this to the above:

```
matchingBlue = matching Blue
```

- This is referred to as *point-free* (or *tacit*) style.
- Point-free style moves the focus from the *data* to the *functions*.

1. In the previous slide, you'll notice that we could have used an explicit parameter with our partial application bound to `matchingBlue`.
2. Since we can treat functions just like any other object, there's no need to use syntax to indicate the as-yet unused arguments for these unapplied or partially applied functions.
3. The term *point-free* originated in topology, a branch of mathematics which works with spaces composed of points, and functions between those spaces. So a 'points-free' definition of a function is one which does not explicitly mention the points (values) of the space on which the function acts.

The Point of Point-free Style

- First, a bonus function: `.` for function composition. Just as in math $(f \cdot g)(x) = f(g(x))$, in Haskell:

```
(f.g) x == f (g x)
```

- Point-free style is more concise and, once you're used to it, often more clear.

```
sum = foldr (+) 0      vs.  sum xs = foldr (+) 0 xs
fn = f . g . h         vs.  fn x = f (g (h x))
membr = any.(==)       vs.  membr x lst = any (==x) lst
```

- Intermediate form of `membr`:

```
membr x = any (==x)
```

1. `f.g` of course means something quite different in Ruby, and it make take a while to re-orient how you read it to its meaning in Haskell. If it helps, add some spaces: `f . g`.
2. That last one might be difficult for the uninitiated. An intermediate form, after we've applied `(==)` to the first argument, and `any` to the function resulting from that, is `membr x = any (==x)`. But you probably need to take some time work through this carefully yourself.
3. GHC can't automtaically infer the type of that last definition, so if you want to play with it you'll need to annotate it:

```
membr :: Eq a => a -> [a] -> Bool
membr = any . (==)
```

Type Constructor Parameters I

- As with constructors in Ruby, type constructors in Haskell can take parameters for the specific data that they “store”, and we typically extract these values by deconstructing via pattern matching. `06_Length.hs`:

```
data Length = Length Int

(|+|) :: Length -> Length -> Length
Length x |+| Length y = Length (x+y)

main = print $ Length 3 |+| Length 7
```

- Bonus function: `$` is another way of doing precedence without parens. These two expressions evaluate identically:

```
head   ( [1] ++ [2] )
head $   [1] ++ [2]
```

1. We make up a new infix operator, `|+|`, for our function to add two lengths.
2. `$` is actually just a regular function that you could write yourself:

   ```
   infixr 0 $
   f $ x = f x
   ```

   It applies the function in the left-hand argument to the value on its right. It works as a “paren-remover” because it has its precedence set very low so that more stuff on the right side is “grabbed” to be evaluated before the `$` function itself is evaluated.

Type Constructor Parameters II

- Constructors may have multiple params (`07_Point.hs`):

```haskell
data Point = Cart Double Double | Polar Double Double

toCart :: Point -> Point
toCart xy@(Cart _ _) = xy
toCart    (Polar r θ) = Cart (r * cos θ) (r * sin θ)

distanceTo :: Point -> Point -> Double
distanceTo p1 p2 = sqrt $ (x2-x1)^2 + (y2-y1)^2
    where Cart x1 y1 = toCart p1
          Cart x2 y2 = toCart p2
```

- Here we use two different constructors for different representations of the same value.
- More bonus syntax: `@` to bind the constructed value to a pattern, `where` for local definitions, and ability to use Unicode alphabets in GHC.

1. The pattern `xy@(Cart x y)` binds three variables: `x` and `y` to the two deconstructed values, and `xy` to the whole constructed value. In our example, we ignore the deconstructed values because we just want to match any value constructed with that pattern.
2. GHC, with the `-XUnicodeSyntax` option, also allows us to use Unicode characters for synatx if we want to look really nice, e.g.,

```haskell
notEq :: (Num α) ⇒ α → α
notEq x y = x ≠ y
```

Variables and Functions at the Type Level

- Real functional languages have variables and functions, and so of course our type language in Haskell has these too:

```
data List a = Cons a (List a) | Nil
```

- `List` is a type-level function that, given an `a` (which is a type variable), creates a new type based on whatever `a` it's given.

```
l0 :: List Int
l0 = Nil

l1 :: List Int
l1 = Cons 1 (Cons 0 Nil)

la :: List Char
la = Cons 'c' (Cons 'b' (Cons 'a' Nil))
```

- Note also that this data type is recursive! The `Cons` constructor to create a `List` value takes a `List` value!

1. Lists are often implemented as *cons* cells, a name that comes from Lisp. A cons cell is a pair of values, the first of which is an element of the list, and the second of which is a pointer to the next item in the list. If the second value is "nil,", that indicates that this is the last cell in the list.

Infix Type Operators and Nicer List Construction

- We have infix operators in the type language, too; any punctuation sequence starting with a colon (`:`) is an infix type function.
- Let's use this to make a nicer syntax for constructing `List`s:

```
data List a = a :. (List a) | Nil
infixr 2 :.

l1 :: List Int
l1 = 1 :. 0 :. Nil
```

1. The `infixr 2 :.` statement makes the `:.` operator right-associative (it defaults to left-associative, otherwise) and sets the precedence to a fairly low value.

Our Favorite Type Class

- Let's create a class of types where a value of each type is the favorite value. For each type we'll have a function `isFavorite` that tells us if a value is the favorite value or not:

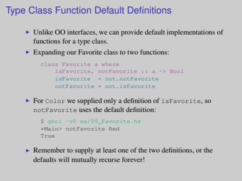
```
class Favorite a where
    isFavorite :: a -> Bool

data Color = Red | Green | Blue
instance Favorite Color where
    isFavorite Green = True
    isFavorite _     = False
```

- A quick test in `ghci`:

```
$ ghci -v0 ex/09_Favorite.hs
*Main> isFavorite Blue
False
*Main> isFavorite Green
True
```

1. We declare a type class called `Favorite` taking a single type as a parameter, *a*. (Type classes can involve more than one type, which is extremely powerful, but we won't get in to that here.)
2. For every type *a* that's a member of this class, there must exit a function `isFavorite` that takes a value of type *a* and tells us if that value is the favorite or not.
3. The semantics of the `isFavorite` function aren't known to the compiler; that's the responsibility of the programmer.
4. We declare a type and, using the `instance` keyword, declare it to be an instance of class `Favorite`. We supply an `isFavorite` function that does the appropriate thing for our data type.



1. Again, the function composition operator. For `isFavorite`, we first apply `notFavorite` to the argument, then apply `not` to the result of that.

Type Classes are Open

- Unlike types, which are *closed* and cannot be changed once defined, type classes are *open*.
- This means that you can add any type to any type class at any time, even if neither the type class nor the type were created by you, and even if the original creators of the type and type class were entirely unaware of each other.
- Let's make a pre-defined type a member of our new type class:

```
instance Favorite Int where
    isFavorite = (== 42)
```

- And test it:

```
$ ghci -v0 ex/09_Favorite.hs
*Main> isFavorite (42::Int)
True
```

1. Note the use of point-free style for this definition of `isFavorite`.
2. When we use the `favorite` function in the `print` commands, the compiler can't infer which `favorite` we're trying to call because, in this situation, it doesn't have enough clues. So we supply an in-line type anotation to help it out.

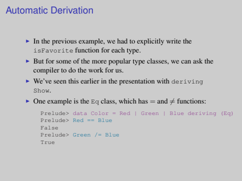
Duck Typing in Haskell

- We can make functions that operate on values of any type in a class, so long as the function restricts itself to operations available in that class:

```
hasFavorite :: Favorite a => [a] -> Bool
hasFavorite (x:xs) = isFavorite x || hasFavorite xs
hasFavorite []     = False
```

- This works even on members of the class that are defined long after the function was written.
- Key point: it's statically type checked, so it can't break.

1. The `Favorite a => ...` syntax says that type *a* must be a member of the class `Favorite`.



1. There's special magic in the compiler that does this. However, there's work being done on extending this to user-supplied type classes. One is the *derive* program, another is the work on generic programming.