First Haskell Exercises Solutions

Wilmington Charter School
November 1, 2013

Richard Eisenberg
eir@cis.upenn.edu

Using your favorite text editor (I can suggest `pico` if you don't have a favorite), make a new file named `Exercises.hs`. For example, with `pico`, you would write

```
> pico Exercises.hs
```

at the command prompt. (You don't write the `>` — that's there to represent the prompt!)

Put the following code in your file to start yourself off:

```
module Exercises where
doNothing :: Int → Int
doNothing x = x
```

Then, in another window, start up GHCi (the Glasgow Haskell Compiler, interactive mode) with this:

```
> ghci Exercises.hs
```

GHCi will introduce itself to you, and you should have a line like `Prelude>` staring at you. This is where you enter either commands or other Haskell expressions in for evaluation. By long-standing tradition, the GHCi prompt is written λ> on this paper. (That's the Greek letter lambda there, which is a part of the Haskell logo.) To make sure GHCi is working for you, type in a mathematical expression, such as $3 + 8$:

```
λ> 3 + 8
11
```

As the example shows, you should see 11. Now, try this:

```
λ> True + "hi"
```

You should see an error saying, essentially, that these two things (*True* and `"hi"`) can't be added together with +.

If everything is working up until now, you're ready to load the `Exercises.hs` file. Say this:

```
λ> :load Exercises.hs
Ok, modules loaded: Exercises.
```

Now, you should be able to use the *doNothing* function that you wrote in the Haskell file. It should do nothing at all to its argument:

```
λ> doNothing 10
10
```

Once you have all of this working, you're ready to start writing your own code in the Haskell file. To reload the code into GHCi, use :*reload* (or :*r* for short). And, make sure to save the Haskell file before trying to reload!

For each exercise below, write the function requested in your Haskell file. After each function, switch over to GHCi to test your function and make sure it works on several different inputs. Good luck!

1. Write a function named *add1* that takes an *Int* and returns an *Int* that is one greater than its input. For example, if we compute *add1* 5, we should get 6. The type declaration for *add1* should be

   ```
   add1 :: Int → Int


   add1 x = x + 1
   ```

2. Write a function named *always0* that takes an *Int* and returns an *Int*. The return value should always just be 0.

   ```
   always0 :: Int → Int
   always0 n = 0
   ```

3. Write a function *subtract* that takes two numbers (that is, *Int*s) and subtracts them.v

   ```
   subtract :: Int → Int → Int
   subtract a b = a − b
   ```

4. Write a function *addmult* that takes three numbers. Let's call them *p*, *q*, and *r*. *addmult* should add *p* and *q* together and then multiply the result by *r*.

   ```
   addmult :: Int → Int → Int → Int
   addmult p q r = (p + q) ⋆ r
   ```

The next function will also need you to use an **if** expression in Haskell. Here is an example of **if** in action:

```
greaterThan0 :: Int → String
greaterThan0 n = if n > 0 then "Yes!" else "No :("
```

You can copy that function definition into your file and try it in GHCi if you want. Note that both the **then** part *and* the **else** part are required in Haskell. If you left the **else** out, what would happen if *n* weren't greater than 0? There's no good answer to that question, so the **else** is always required.

5. Write a function *myAbs* that computes absolute value. (Don't use the built-in function *abs* — that's cheating!)

   ```
   myAbs :: Int → Int
   myAbs n = if n < 0 then (−n) else n
   ```

6. Write a function *pushOut* that takes a number and returns the number that is one step further from 0. That is, *pushOut* 3 is 4, *pushOut* (-10) is (-11), and *pushOut* 0 is 0. That last one is because we don't know which direction to go! Note that, in Haskell, you *always* have to put parentheses around negative numbers.

   Hint: Use == for equality checking in Haskell, just like Java.

   ```
   pushOut :: Int → Int
   pushOut n = if n > 0
                 then n + 1
                 else if n < 0
                        then n − 1
                        else 0
   ```

All of the functions so far have dealt only with numbers. Now, we'll look at *String*s, which are chunks of printable text. *String*s are written in double-quotes in Haskell:

```
exampleString :: String
exampleString = "Hello there!"
```

(You might notice that the *greaterThan0* example uses *String*s.) There are two interesting operations on *String*s (for now):

- Use the ⧺ (written like ++) operator to concatenate *String*s. To concatenate is to put one after the other. For example, `"Hi " ⧺ "there!"` is `"Hi there!"`. This is quite like + in Java.

- Use *show* to convert most types into *String*s. For example, *show* 3 is `"3"`. This is quite like *toString* in Java, but Java also automatically converts things to *String*s when you use + with another *String*.

7. Write a function *greet* (with type *String* → *String*) that takes in a person's name and says `"Hi "` to that person. For example, *greet* `"Haskell"` is `"Hi Haskell"`. (The language Haskell is named after a logician, Haskell Curry.)

```
greet :: String → String
greet person = "Hi " ++ person
```

8. Write a function *greet2* that is just like *greet*, but if the name provided is empty, your function should return `"Hi there"`. So, giving an empty string, written `""`, is like giving the string `"there"`. To test a string for emptiness, use the *null* function, of type *String* → *Bool*.[1] *null* `""` is *True*, while *null* `"Esmerelda"` is *False*.

```
greet2 :: String → String
greet2 person = if null person
                then "Hi there"
                else "Hi " ++ person
```

The functions up until now have all been fairly simple. The next function, however, must perform an operation many times. Haskell's way of repeating an operation is *recursion*, the act of a function calling itself. As long as the function's argument(s) keep getting smaller, this doesn't cause a problem — Haskell knows what to do.

For example, here is a function that makes a *String* containing any number of `a`s:

```
makeAs :: Int → String
makeAs n = if n ≡ 0
           then ""
           else "a" ++ makeAs (n − 1)
```

For example, *makeAs* 3 is `"aaa"` and *makeAs* 7 is `"aaaaaaa"`.

9. Write a function *twiceAs* that is like *makeAs*, but it makes twice as many `a`s as requested.

```
twiceAs :: Int → String
twiceAs n = if n ≡ 0
            then ""
            else "aa" ++ twiceAs (n − 1)
```

[1]That type is a tiny white lie. *null* actually works on any kind of list, not just *String*s. But I'm getting ahead of myself.

10. Write a function *countDown* (with type *Int* → *String*) that produces a *String* counting down from a number. For example *countDown* 5 is `"5 4 3 2 1 "`. Note that there is an extra space at the end — that's supposed to make it easier. (Bonus points if you can get rid of the extra space!) If the number passed in is 0 or less, the returned *String* should be `"Too low"`. Remember that *show* converts a number to a *String*.

```
countDown :: Int → String
countDown n = if n ≡ 1
              then "1"
              else if n > 1
                   then (show n) ++ " " ++ countDown (n − 1)
                   else "Too low"
```

11. Write a function *countUp* that goes the opposite way of *countDown*.

```
countUp :: Int → String
countUp n = if n ≡ 1
            then "1"
            else if n > 1
                 then countUp (n − 1) ++ " " ++ (show n)
                 else "Too low"
```

12. Write a function *mult* (with type *Int* → *Int* → *Int*) to multiply two numbers without using built-in multiplication. To do this, you will use repeated addition. Writing it out in mathematical notation:

$$a \cdot b = \begin{cases} 0 & \text{if } b = 0 \\ a + a \cdot (b - 1) & \text{if } b > 0 \end{cases}$$

To compute *mult a b*, check *b*. If *b* is 0, then *mult a b* should be 0. If *b* is greater than 0, *mult a b* should be *a* plus the result of *mult a* (*b* − 1).

```
mult :: Int → Int → Int
mult a b = if b ≡ 0
           then 0
           else a + (mult a (b − 1))
```

13. Write a function *power* that raises a number *a* to the power *b*. This is quite similar to the last exercise. Here is the mathematical notation for it:

$$a^b = \begin{cases} 1 & \text{if } b = 0 \\ a \cdot a^{b-1} & \text{if } b > 0 \end{cases}$$

```
power :: Int → Int → Int
power a b = if b ≡ 0
              then 1
              else mult a (power a (b − 1))
```

14. Triangular numbers are the sum of consecutive numbers. They are called triangular because, if you have a triangular number of squines, then you can make a triangle of squines. Here are the first several triangular numbers:

$$\begin{aligned}1 &= 1\\ 3 &= 1 + 2\\ 6 &= 1 + 2 + 3\\ 10 &= 1 + 2 + 3 + 4\\ 15 &= 1 + 2 + 3 + 4 + 5\end{aligned}$$

    Write a function *triangle* which, when given *n*, computes the *n*th triangular number.

```
triangle :: Int → Int
triangle n = if n ≡ 0
               then 0
               else n + triangle (n − 1)
```

When enough of the room has reached this point, we'll continue by reviewing alebraic datatypes before going on to more exercises. If you get here before your peers, please offer to help them out! Or, check out Haskell online, for example at `planet.haskell.org`.

Type the following definition of *Pet* into your Haskell file:

```
data Pet = Cat String
    | Dog String String
```

Both kinds of *Pet* take a *String* parameter to represent the pet's name. The *Dog* also takes a second *String* parameter to store the dog's breed.

15. Write a *speak* function (with type *Pet* → *String*) that uses pattern matching to return `"Meow!"` when given a *Cat* and `"Woof!"` when given a *Dog*.

```
speak :: Pet → String
speak (Cat n)     = "Meow!"
speak (Dog n b) = "Woof!"
```

16. Write a *petName* function (with type *Pet* → *String*) that returns a pet's name.

```
petName :: Pet → String
petName (Cat n)     = n
petName (Dog n b) = n
```

17. Write a *breedString* function (with type *Pet* → *String*) that returns a dog's breed. If given a *Cat*, *breedString* should return `"Cats don't have breeds!"`.[2]

```
breedString :: Pet → String
breedString (Cat n)     = "Cats don't have breeds!"
breedString (Dog n b) = b
```

The next several exercises will involve the type *Maybe*, which optionally stores a value. *Maybe* is built-in to Haskell,[3] so you don't have to put this in your file, but here is its definition for reference:

```
data Maybe a = Nothing
    | Just a
```

18. Write a *breed* function that has type *Pet* → *Maybe String*. It should return *Just* the dog's breed when given a *Dog* and *Nothing* when given a *Cat*.

```
breed :: Pet → Maybe String
breed (Cat n)     = Nothing
breed (Dog n b) = Just b
```

19. Write a *maybeDiv* function that takes two *Int*s and optionally returns an *Int*. It should divide its two arguments (using the built-in function *div* :: *Int* → *Int* → *Int* — don't use /!) only when the second argument is not 0. If the second argument is 0, it should return *Nothing*. Use pattern-matching, not an **if** expression!

```
maybeDiv :: Int → Int → Maybe Int
maybeDiv x 0 = Nothing
maybeDiv x y = Just (div x y)
```

[2]Cats *do* have breeds, of course, but let's pretend.

[3]That's another small lie. *Maybe* is defined in the *Prelude*, which is automatically imported into every Haskell file. This is rather like Java's *java.lang.*∗ package.

20. Rewrite your *pushOut* function (call the new one *pushOut2*) to use pattern guards instead of **if** expressions. For example, here is *greaterThan0* written with guards:

```
greaterThan0'2 :: Int → String
greaterThan0'2 n
  | n > 0     = "Yes!"
  | otherwise = "No :("
```

```
pushOut2 :: Int → Int
pushOut2 0 = 0
pushOut2 n
  | n > 0     = n + 1
  | otherwise = n − 1
```

21. Write a function *maybePlus* :: *Maybe Int* → *Maybe Int* → *Maybe Int* that adds two *Int*s, each of which may or may not exist. If either one is *Nothing*, just return *Nothing*. Remember: no **if** expressions!

```
maybePlus :: Maybe Int → Maybe Int → Maybe Int
maybePlus Nothing y        = Nothing
maybePlus x        Nothing = Nothing
maybePlus (Just x) (Just y) = Just (x + y)
```

Now, hold up here until we learn about lists.

---

22. Write a function *myLength* :: [*a*] → *Int* that computes the length of a list.

```
myLength :: [a] → Int
myLength []      = 0
myLength (a : b) = 1 + myLength b
```

23. Write a function *listSum* :: [*Int*] → *Int* that adds up all the numbers in a list.

```
listSum :: [Int] → Int
listSum []      = 0
listSum (h : t) = h + listSum t
```

24. Write a function $\mathit{myReverse} :: [a] \to [a]$ that reverses a list. You will probably want to use $+\!\!+$, which appends (concatenates) two lists.

```
myReverse :: [a] → [a]
myReverse []      = []
myReverse (h : t) = reverse t ++ [h]
```

25. Write a function $\mathit{listUp} :: \mathit{Int} \to [\mathit{Int}]$ that creates a list from 1 up to the number passed in. For example, $\mathit{listUp}\ 3$ is $[1, 2, 3]$.

```
listUp :: Int → [Int]
listUp 0 = []
listUp n = listUp (n − 1) ++ [n]
```

26. Write a function $\mathit{myLast} :: [a] \to \mathit{Maybe}\ a$ that returns the last element of a list, if such an element exists.

```
myLast :: [a] → Maybe a
myLast []      = Nothing
myLast [x]     = Just x
myLast (h : t) = myLast t
```

27. Write a function $\mathit{palindrome} :: \mathit{String} \to \mathit{Bool}$ that checks if a string is a palindrome or not. (A palindrome is a word, like *level* or *racecar*, that reads the same forward or backward.) In Haskell, a *String* is actually a list of *Char*s (that is, *String* is the same as $[\mathit{Char}]$), so you can use, say, *myReverse* if you want.

```
palindrome :: String → Bool
palindrome s = s ≡ myReverse s
```

Reflect for a moment at how hard these last few would be in Java!

There's plenty more to learn! Here are two books, freely available online, that might be good places to start:

- *Real World Haskell*, by Bryan O'Sullivan, Don Stewart, and John Goerzen

- *Learn You a Haskell for Great Good*, by Miran Lipovača
- The FP Complete *School of Haskell*, at `fpcomplete.com`

You may also want to check out my website, at `http://cis.upenn.edu/~eir`. I will post solutions to today's exercises there shortly, on the page viewable at `http://cis.upenn.edu/~eir/misc/wcs/`.