CMSC 113: Computer Science I
First Haskell Exercises

Although it is possible to install Haskell on your computer, we want to get started quickly, so we will be using an online Haskell programming system (also called an IDE, for Integrated Development Environment).

> Go to `https://repl.it/languages/haskell` to start a fresh Haskell project.

You should see two vertical panes. Write the following in the left pane:

```haskell
main = putStrLn "Hello, world!"
```

This is a very short Haskell program that prints `Hello, world!` to the screen. (*putStrLn* is Haskell's equivalent of `System.out.println`.)

To test it, we will be using the Glasgow Haskell Compiler (or GHC, for short). GHC comes with an interactive environment, called GHCi.

> Press the "run" button toward the top of the screen to load and run your code. You should see your output in the pane on the right.

After `Hello, world!` is printed, you should see a prompt in the right pane. It is at this prompt that you will type commands to GHCi. I will abbreviate this prompt with $\lambda>$, which is typically how the GHCi prompt is shown in writings about Haskell.

Just to make sure all is working:

> Type `main` at the prompt and press Enter.

(I'm going to leave off the "press Enter" part of instructions from now on.) You should see `Hello, world!` printed again. That's how you know things are working.

Now, you will write a function in your file:

> Write the following code in the file (the pane on the left):
>
> ```haskell
> doNothing x = x
> ```
>
> Reload your file by pressing the "run" button again.

This function does, well, nothing. It takes an argument *x* and returns it. But let's test it anyway.

Only by pressing the "run" button will the window on the right get your updated code. From now on, I will assume that you reload the file before using any new definitions.

Enter *doNothing* 5 at the prompt.

Writing a Haskell expression at the prompt evaluates the expression and prints the result of evaluation. You should see 5 displayed back at you.

In Haskell, we don't think of *running functions*, we think of *evaluating expressions*. This is much closer to mathematics, where we can define, say, $f(x) = x^2 + 4x - 1$ and then evaluate $f(3)$ to be 20. Indeed, we can even do this in Haskell:

Add this definition to your file:

```
f x = x^2 + 4 * x - 1
```

After reloading, type *f* 3 at the prompt.

You should see 20. This is one of the beauties of Haskell: it's just like math! Type in a few mathematical expressions in GHCi. You should see the results you expect. (Haskell does not use parentheses around function arguments as you would in math, however.)

Types

Haskell is a *strongly-typed* language, meaning that nonsense expressions are rejected.

Write this at the prompt: *True* + `"hi"`

See what you get.

Add a definition *x* = *True* + `"hi"` and reload

You should see a similar error. Note that you cannot even ask what the value of *x* is; just loading the file shows you the error.

Java is also strongly-typed, but Haskell comes equipped with *type inference*, so that you never need to write a type in your program if you don't want to. In GHCi, you can ask for the type of something with the `:type` command.

Remove the erroneous *x* from your file and add this one:

```
choose x = if x then "hi" else "bye"
```

Then write `:type choose` at the prompt.

You should see that the type of *choose* is $Bool \to String$. That is, *choose* is a function that takes a *Bool* argument and returns a *String*. You can add type signatures to functions if you like. (This sometimes results in better error messages when you make a mistake.) For example, you could add $f :: Int \to Int$ to your file to declare that your function *f* takes an *Int* and returns an *Int*.

Once you have all of this working, you're ready to start writing your own code in the Haskell file. 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!

Basics

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. If you want to write a type signature for *add1*, it would be $add1 :: Int \to Int$ (on a line by itself).

2. Write a function named $always0 :: Int \to Int$. The return value should always just be 0.

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

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*.

Conditionals

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!)

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.

Strings

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!"
```

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 on strings.

- Use *show* to convert most types into *String*s. For example, *show* 3 is `"3"`.

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.)

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*.

Recursion

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.

[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*.

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

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).

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:

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

14. Triangular numbers are the sum of consecutive numbers. They are called triangular because, if you have a triangular number of pillows, then you can make a triangle of pillows. 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.

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`.

Algebraic datatypes

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*.

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

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]

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*.

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!

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 :("
```

[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.

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!

Now, hold up here until we learn about lists.

Lists

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

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

24. Write a function *myReverse* :: [*a*] → [*a*] that reverses a list. You will probably want to use ++, which appends (concatenates) two lists.

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

26. Write a function *myLast* :: [*a*] → *Maybe a* that returns the last element of a list, if such an element exists.

27. Write a function *palindrome* :: *String* → *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 [*Char*]), so you can use, say, *myReverse* if you want.

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`
- This tutorial, in particular, seems worthwhile: `https://www.schoolofhaskell.com/school/starting-with-haskell/introduction-to-haskell`