Code Making, Code Breaking
Introduction to Haskell
Some more introduction and practice

Part 1: setting up

You generally have two options for executing Haskell code. You can either enter it directly into the Haskell interpreter, which you can run from the command line, or you can you can write a '.hs' file and load it inside the interpreter. For more complicated tasks, you'll want to do the latter, so that you don't have to enter lots of code individually line-by-line.

To practice with the code to follow, enter the interpreter (on a Mac, this is done by opening Terminal, typing `ghci`, and pressing Enter). If you want to code in an external '.hs' file as plain text, make sure that you're in the directory that contains this file. The following practice, however, is all done within the Haskell interpreter.

Part 2: getting started

Programs and values. Haskell makes a basic distinction between *values* and *types*. Values are the things that programs (i.e., expressions) evaluate to: when you execute a program in Haskell, you get a value as the result. For many kinds of values, you can display them in your interpreter by typing them in and pressing Enter. Try typing

```
1
```

into your interpreter, and pressing Enter. Here's what should pop out:

```
1
```

In Haskell, 1 is a program, but not a very interesting one, since nothing happens when you execute it: it is already a value. Here's a slightly more interesting program you can run. Instead, type the following into your interpreter, and press Enter.

```
1 + 1
```

Now, something different should pop out: this time you should get

```
2
```

So, when you run a program in Haskell, Haskell evaluates it until it turns into a value, something which, depending on what kind of value it is, it can print on the screen.

Types. In addition to programs (like '1 + 1') and the values that they evaluate to (like '2'), Haskell has *types*. Every program and every value in Haskell has a type. A type in Haskell says what kind of thing a particular value (or program) is;

e.g., whether or not it is a character, a number, a string, or a list. To see what type something has, you just write ':type' (or ':t' for short) before it in the interpreter and press enter. For example, try writing

```
:t 'd'
```

in the interpreter and pressing Enter. What it should print is

```
'd' :: Char
```

telling you that 'd' is a Character.

Beyond atomic types like Char, Haskell has function types like Char→Char, which is the type of functions from things of type Char to things of type Char. For example, we can make a function 'identity' that takes a Char and just returns that Char by typing the following into the interpreter and pressing enter.

```
let identity :: Char -> Char; identity a = a
```

If you then type

```
identity 'd'
```

and press Enter, the interpreter should print 'd'. This is because, by typing 'identity 'd'', you have *applied* the function 'identity' to the character 'd'. In general, you have Haskell evaluate a function f on an argument a by writing '$(f\ a)$' (or '$f\ a$', omitting the parentheses). This syntactic juxtaposition of a function with its argument is known as an *application*. Moreover, because the domain and range of 'identity' is Char, you can't apply it to things of other types, e.g., numbers like 1. If you now type in

```
identity 1
```

for example, you will get an error message indicating that the type of 'identity' makes it inappropriate for applying it to 1.

Type classes. You can also look at the types Haskell assigns to numbers by entering the following into the interpreter.

```
:t 2
```

The result will actually be slightly different than it was for the case of characters. Rather than, e.g., '2 :: Int', Haskell should print

```
2 :: Num t => t
```

The reason for this is that the type of '2' is, in fact, ambiguous: Haskell does not commit to whether it is an integer, as opposed to, say, a decimal floating point number—only that, whatever type it is, that type is in the class Num. This brings us to *type classes*. In addition to programs, values, and types, various types may be sorted into classes that acknowledge something about their behavior. For example, if t is in the class Num, then values of type t should be able to be added together. Specifically, Num comes with a *method* '(+)' whose type is t→t→t for any t belonging to it. If you enter

```
:t (+)
```

into the interpreter, it will reveal the following type assignment.

```
(+) :: Num a => a -> a -> a
```

This type signature indicates that, for any a in the type class Num, if you apply '(+)' to two things of type a, it will return another thing of type a. One thing, as we saw, of a type in the class Num is 1. Hence, we may enter

```
(+) 1 1
```

into the interpreter, which will cause it to print

```
2
```

Notice that '(+) 1 1' is just an alternate way of evaluating '1 + 1', as we did above. In general, functions that by default appear with *infix* notation, like '(+)', can be made to appear as a prefix by surrounding them with parentheses.

λ-abstraction. One of the defining features of Haskell is a special syntax it provides—appropriated from its theoretical underpinnings in typed λ-calculus—for expressing functions anonymously within code. The convention used in this syntax is known as *λ-abstraction*, and it is exemplified by functions like the following. Lets say that we want to express the function that takes a number (something whose type is in the class Num) and adds 1 to it. We may express this function as follows.

```
(\x -> x + 1)
```

This expression of the function is called "anonymous" because, rather than defining a constant with a name, say, 'plusOne', as in

```
let plusOne a = a + 1
```

and then, afterwards, using this function in a particular piece of code, we have skipped the "naming" step altogether and used the syntax native to Haskell—in particular, λ-abstraction—to denote the function. Then, just like we might enter

```
plusOne 1
```

into the interpreter and have it evaluate to 2, we can also enter

```
(\x -> x + 1) 1
```

and have it evaluate to 2. Intuitvely, to evaluate this program, Haskell is getting rid of the '\x →' prefixed to the function (which we therefore say is expressed as an *abstraction*) and substituting the function's argument 1 for 'x' in the *body* of this function. This syntax is very general: rather than expressing the addition operation using '(+)', we could also express it using the following abstraction.

```
(\x -> (\y -> x + y))
```

This function, like addition, must be applied to two arguments, e.g.,

```
(\x -> (\y -> x + y)) 1 2
```

If you enter this expression into your interpreter, Haskell first evaluates the left-most application

‘(\x →(\y →x + y)) 1’

to get

```
(\y -> 1 + y) 2
```

and then it evaluates this application to get

```
1 + 2
```

which evaluates to 3. The process of evaluating applications of abstractions to arguments in this way by sticking the arguments into the body of the abstraction where the relevant variable use to be is known as β-reduction. Any application that has the form ‘(\x →M) N’ is known as a β-redex because it is something that can be reduced.

Assignment.

(1) How might you express the function ‘identity’ defined above as a λ-abstraction (i.e., something of the form ‘(\x →. . .)’?

(2) Native to the Num type class is another method ‘(/)’ for doing division. For example, if you enter either

```
1 / 2
```

or

```
(/) 1 2
```

into the interpreter, it will evaluate to 0.5. Using ‘(/)’, how might you express, as a λ-abstraction, the function that takes two numbers and divides the second by the first? In other words, if this abstraction is ‘M’, then writing ‘M 2 1’ should cause the interpreter to print 0.5.