Expressions

Bindings

Value binding in Haskell:

```
x = 3
```

Compare to C:

```
int x = 3;
```

Question 1

What are we missing in the Haskell case?

Answer: *Type binding*

```
x :: Int
x = 3
```

Haskell compiler (`ghc`) can infer types for you based on their values, but:

- Improves readability when you include them in your code
- It can only do so much – for example, it could figure out that `x = 3` means `x` is a number of some sort, but not that you want it to specifically be an `Int` (as opposed to `Float`)

However, there are some cases where you may prefer that the compiler infers some types for you (but this approach is less common).

Immutability

Can't do this:

```
x = 3
-- x = 4 -- value bindings are immutable, cannot do this!
```

Or this:

```
x :: Int
-- x :: Float -- not allowed, type bindings are also immutable
```

Simple Function

```
f x = x + 1
```

We're again missing a type binding.

```haskell
f :: Int -> Int
f x = x + 1
```

Note how similar this looks to the `x = 3` example. You can think of `x = 3` as a *function with 0 arguments*; also appropriate (and perhaps more natural) to refer to it as a *value*.

Multiple Arguments

Now let's write a function with multiple arguments.

Goal: `add` function

- *Input*: Two `Int` args
- *Output*: Sum of the inputs

Question 2

What should the value binding look like?

Answer:

```haskell
add x y = x + y
```

Question 3

Okay, what about the type binding? Think about how we went from `x :: Int` to `f :: Int -> Int`...

Answer:

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

Guards

Guards are the equivalent of `if`/`else`.

Goal: `max'` function

- *Inputs*: Two `Float`s
- *Output*: The larger of the two inputs

Note: The `max` function is predefined in Haskell, so we'll append a single-quote to the function name and call ours `max'`, which Haskell allows.

Question 4

What should the type binding for this function look like?

Answer:

```
max' :: Float -> Float -> Float
```

Here's one way to define this function:

```
max' :: Float -> Float -> Float
max' x y
    | x >= y = x
    | otherwise = y
```

The guard statement determines the appropriate *value binding* when this function is called. Example functions calls:

```
m = max' 3.5 2.0 -- binds 3 to m, also binds `Float` type
n = max' 3.5 4.0 -- binds 4 to n, also binds `Float` type
```

Fibonacci

Haskell supports recursion. Given that + above info, write a fibonacci function.

Input: Index of the fibonacci number to calculate.

Output: Fibonacci number at that index (*not* a list, just a single value).

Type signature?

```
fib :: Int -> Int
```

Assumptions:

- `fib 0` and `fib 1` equal `1`.
- The input is non-negative, so we don't have to handle cases like `fib (-1)`.

```
fib :: Int -> Int
fib n
    | (n == 0) || (n == 1) = 1
    | otherwise            = fib (n - 1) + fib (n - 2)
```

Alternative way to write this: *pattern match* on the input value.

```
fib' :: Int -> Int
fib' 0 = 1
fib' 1 = 1
fib' n = fib (n - 1) + fib (n - 2)
```

We can do this whenever we have a *concrete value* to test the input value against (in this case, `n == 0` or `n == 1`).