Haskell for Readers

Joachim Breitner[*], DFINITY Foundation[†]

Exercise 1

What associativity do you expect for `(+)` and `(-)`? Verify your expectation.

Exercise 2

Look up the precedences of the other arithmetic operations, and see how that corresponds to the PEMDAS rule.

Exercise 3

Can you predict the result of the following?

```
Prelude> 1 + const 2 3 + 4
```

Exercise 4

What is the result of

```
Prelude> (-) 5 $ div 16 $ (-) 10 $ 4 `div` 2
```

Exercise 5

Discuss: Think of other programming language that have concepts called functions. Can you always replace a function call with the function definition? Does it change the meaning of the program?

Exercise 6

Write a function `absoluteValue` with one parameter. If the parameter is negative, returns its opposite number, otherwise the number itself.

Exercise 7

Write a function `isHalfRound` that checks if a number is divisible by 5, by checking whether the last digit is 0 or 5.

Exercise 8

Write a function `isEven` that checks if a number is divisible by 2, by checking whether the last digit is 0, 2, 4, 6, 8.

Exercise 9

Write the function `sumDigits` that sums up the digits of a natural number.

Exercise 10

[*] http://www.joachim-breitner.de/
[†] https://dfinity.org/

Write a (recursive) function `fixEq` so that `fixEq f x` repeatedly applies `f` to `x` until the result of `f` is the same as its argument.

Exercise 11

Use this function and `sumDigits` to write a function `isMultipleOf3` so that `isMultipleOf3 x` is true if repeatedly applying `sumDigits` to `x` results in 3, 6 or 9.

Exercise 12

Which other recent definitions can be changed accordingly?

Exercise 13

What do you think is the type of `id`?

Exercise 14

A great example for the power of polymorphism is the following type signature:

```
(a -> b -> c) -> b -> a -> c
```

There is a function of that type in the standard library. Can you tell what it does? Can you guess its name? You can use a type-based search engine like Hoogle[1] or Hayoo[2] to find the function.

Exercise 15

Consider the following definition:

```
data Wat = Wat Wat
```

Is this legal? What does it mean? Which occurrences of `Wat` are terms, and which are types? Can you define a value of type `Wat`?

Exercise 16

How many values are there of type `Maybe (Maybe Bool)`. When can it be useful to nest `Maybe` in that way?

Exercise 17

How could you represent the Riemann numbers from the previous section using only these predefined data types?

Exercise 18

Write functions

```
fromEitherUnit :: Either () a -> Maybe a
```

and

```
toEitherUnit :: Maybe a -> Either () a
```

that are inverses to each other.

In only one of these type signatures you can replace `()` with a new type variable `b`, and still implement the function. In which one? Why?

[1] https://www.haskell.org/hoogle/?hoogle=%28a+-%3E+b+-%3E+c%29+-%3E+b+-%3E+a+-%3E+c
[2] http://hayoo.fh-wedel.de/?query=%28a+-%3E+b+-%3E+c%29+-%3E+b+-%3E+a+-%3E+c

Exercise 19

What does this program print?

```
theAnswer :: IO Integer
theAnswer = do
  putStrLn "Pondering the question..."
  return 23
  return 42

main :: IO ()
main = do
  a <- theAnswer
  putStrLn (show a)
```

Exercise 20

What does this program do?

```
foo :: Integer -> IO () -> IO ()
foo 0 a = putStrLn "Done"
foo n a = do
   if n == 1 then putStrLn "Almost done"
             else return ()
   a
   foo (n-1) a

main :: IO ()
main = do
   foo 4 (putStrLn "Hooray!")
   foo 0 (putStrLn "And up she rises.")
```

Exercise 21

Write an `Eq` instance for `Employee`, using record accessors. Is there a problem with this code?

Exercise 22

Look up the `Semigroup` type class, and find its laws.

Exercise 23

Can you think of a `Semigroup (Tree a)` instance? Or maybe even more than one? How can you be sure it is a lawful instance?

Exercise 24

The `Monoid` class[3] extends the `Semigroup` class with an operation `mempty :: Monoid a => a` that is supposed to be a neutral element of `(<>)`.

Given a function signature `summarize :: Monoid a => Tree a -> a`, can you guess what it does? What would be the implementation you expect?

[3] http://hackage.haskell.org/package/base/docs/Prelude.html#t:Monoid

With that implementation, can you use `summarize` to distinguish trees that differ in shape, but have the same elements in the same order? What does this imply for search trees?

Exercise 27

Implement `forM :: Monad m => [a] -> (a -> m b) -> m [b]` using do-notation.

Exercise 25

Give definitions of the operators up to `join` using `(>>=)`, `return`, and operators you already defined. (You will have to change the constraints to `Monad` for this to typecheck.)

Exercise 26

Why is the type not `when :: Bool -> m a -> m a`?

Exercise 28

You might observe that `Tagged` is simply the existing pair type. There is an `Applicative` instance for pairs. Look it up! How does that fit to what I just said?