**Haskell programs to try in replit.com updated May 26th 2022**

<https://replit.com/languages/haskell>

In window on the right, type ghci at the prompt to put the environment into interactive mode, i.e.repl mode. The module Prelude is loaded for this purpose.

>ghci

**Anonymous function**

\x -> x\*x

**Function application**

(\x -> x\*x) 6

Prelude: (\x -> x\*x) 6

36

**Functions (Slide 9)**

Prelude>1 + 2

3

Prelude> (+) 1 2 **function (+) two arguments.**

3

Prelude>:t (+) **to get type of (+)**

(+) :: Num a => a -> a -> a **(The return type is the last item in the declaration and the parameters are the first two. :: means has type)**

**=> separates two parts of a type signature:**

**On the left, typeclass constraints - Num a means (+) works with any type a that is an instance of the Num class (a is a type variable). This is an example of parametric polymorphism.**

**On the right, the actual type**

**Num is a typeclass. A typeclass is a sort of interface that defines some behavior.**

Prelude> 2\*3

6

Prelude> (\*) 2 3

6

**Illustration of conciseness of FP languages (Slide 13)**

abs x | x >= 0 = x

| x < 0 = -x

Prelude: abs x | x >= 0 = x | x < 0 = -x

**Need to enclose negative numbers in parentheses in Haskell. Don't need these for non-negative numbers**

Prelude Data.Char> abs (-4)

4

**Partial functions (Slide 25)**

Prelude> add1 = (+) 1 **creates a partial function add1 (Slide 9). can also write this as add1 = (1 +)**

Prelude> add1 2

3

Prelude> mul2 = (\*) 2

Prelude> mul2 3

6

**Types and typeclasses**

Prelude> :t ‘a’

'a' :: Char (:: means has type)

Prelude> square x = x \* x

Prelude> square 5

25

Prelude> square 5.5

30.25

Prelude> :t square

square :: Num a => a -> a **(A function has a type i.e. a type signature)**

**:: means has type**

**=> separates two parts of a type signature:**

**On the left, typeclass constraints - Num a means square works with any type a that is an instance of the Num class (a is a type variable). This is an example of parametric polymorphism.**

**On the right, the actual type**

**Num is a typeclass. A typeclass is a sort of interface that defines some behavior.**

Prelude> :t (+)

(+) :: Num a => a -> a -> a **(The return type is the last item in the declaration and the parameters are the first two)**

Prelude> double y = 2 \* y

Prelude> double 5

10

**Composition (Slide 9)**

Prelude> double(square 4) **(is an example of composition)**

32

Prelude> (double.square) 4 **(Alternative way of expressing composition)**

32

**: introduces a command in the case that follows, a multistatement block**

**action is a user-defined name. I could just as easily written queenOfSheba (an identifier defined by the user must begin with a lowercase letter)**

Prelude> :{

Prelude| action :: (a -> a) -> a -> a **(The return type is the combination of the last two items in the declaration and the parameters are the combination of the first two)**

Prelude| action f x = f x

Prelude| :}

**To try this yourself I have repeated the statements below without "Prelude>" so that you may copy and paste**

:{

action :: (a -> a) -> a ->

action f x = f x

:}

Prelude> action (\x -> x + 3) 4 **Parameterise everything is functional style (Slide 9)**

7

Prelude> :{

Prelude| applyTwice :: (a -> a) -> a -> a

Prelude| applyTwice f x = f(f x)

Prelude| :}

**To try this yourself I have repeated the statements below without "Prelude>" so that you may copy and paste**

:{

applyTwice :: (a -> a) -> a -> a

applyTwice f x = f(f x)

:}

Prelude> applyTwice (\x -> x^2) 3

81

Prelude> applyTwice (\x -> x^2) 3.6

167.96160000000003

**The above shows parametric polymorphism**

Prelude>map (+1) [1,2,3,4] **(Slide 10 and slide 27)**

[2,3,4,5]

Prelude>map add1 [1,2,3,4] **(Requires add1 to have been defined)**

[2,3,4,5]

Prelude> import Data.Char

Prelude Data.Char> :t toUpper

toUpper :: Char -> Char

Prelude Data.Char> map words ["hello world", "the sun has got its hat on"]

[["hello","world"],["the","sun","has","got","its","hat","on"]]

Prelude Data.Char>:quit

>ghci

Prelude> filter even [1..100] **(Slide 31)**

[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,56,58,60,62,64,66,68,70,72,74,76,78,80,82,84,86,88,90,92,94,96,98,100]

Prelude> map (\*2) $ filter odd [1..100] **Expression to right of $**

**evaluated first then passed as argument to expression to the left**

Prelude> map (\*2) $ filter odd [1..100]

[2,6,10,14,18,22,26,30,34,38,42,46,50,54,58,62,66,70,74,78,82,86,90,94,98,102,106,110,114,118,122,126,130,134,138,142,146,150,154,158,162,166,170,174,178,182,186,190,194,198]

Prelude> foldl (+) 0 [1,2,3,4]

10

Prelude> :{

Prelude| factorial :: Integer -> Integer

Prelude| factorial 0 = 1

Prelude| factorial i = foldr (\*) 1 [2..i]

Prelude| :}

Prelude> factorial 3

6

**To try this yourself I have repeated the statements below so that you may copy and paste**

:{

factorial :: Integer -> Integer

factorial 0 = 1

factorial i = foldr (\*) 1 [2..i]

:}

**Lazy evaluation (Slide 9)**

**To try this yourself copy and paste the statements**

:{

inFact :: [Integer] **(Stores a list of integers – Integer typeclass is unbounded)**

inFact = map factorial [0..] **(Calculates factorial of 0, 1, 2, etc. Requires factorial to have been predefined)**

:}

Prelude> inFact **(You will have to breakout of the execution by pressing Crl C. Scroll though the list to see that the individual results are separated by commas)**

**To try this yourself copy and paste the statements**

:{

inFact :: [Integer] **(Stores a list of integers)**

inFact = map factorial [0..3] **(Calculates factorial of 0, 1, 3)**

:}

**Interfaces: Let’s take Single Responsibiliy Principle Principle and the Interface Segregation Principle to the extreme then every interface should have only one method. An interface with only one method is just a function type.**

Type this in, don’t copy and paste

:{

getInt :: IO Int

getInt = readLn

main = do x <- getInt

y <- getInt

print (x+y)

:}

Prelude>main

4

5

9

Function application of function which takes x to x + 1

Prelude> (\x->x + 1) 3

4

Make a function that takes a single argument n and returns a function \m -> n + m

Prelude> addn = \n -> (\m -> n + m)

Make a function that takes a single argument m and returns a function \m -> 1 + m

Prelude> add1 = addn 1

Evaluate function \m -> 1 + m for m = 2, i.e. function application (\m -> 1 + m) 2

Prelude> add1 2

3

In Lambda calculus:

Lambda expression λ x.λy.x + y applied ( λ x.λy.x + y) (1 5) or (λ xy.x + y) (1 5) evaluates to 6

Prelude> (\x -> (\y -> x + y)) 1 5

6

Lambda expression: λ x.λy. λz.x + y + z applied (λ x.λy. λz.x + y + z) (1 5 3) or (λ xyz.x + y + z) (1 5 3) evaluates to 9

Prelude> (\x->(\y->(\z->x+y+z))) 1 5 3

9