Functional Programming: Assignment 2

Group 60

Lucas van der Laan
s1047485

1

1)

```haskell
module Swap where

swap :: (Int, Int) -> (Int, Int)
swap (x,y) = (y,x)

apply :: (Int, Int) -> (Int, Int)
apply (x,y) = (x + y, y)

fibStep :: (Int, Int) -> (Int, Int)
fibStep (x,y) = (y,y + x)
```

2)

The original definition of swap is still valid, because we do not access the values of the variables. The other two will no longer be valid, as they go from Int -> Int instead of swapping, there is only 1 definition possible with (a,b) $\rightarrow$ (b,a).

3)

swap :: (b, a) -> (a, b)
apply :: Num b => (b, b) -> (b, b)
fibStep :: Num b => (b, b) -> (b, b)

4)

(Int, (Char,Bool)) is a tuple inside of a tuple, while (Int,Char,Bool) is just a normal tuple.

```haskell
convert :: (Int, Char, Bool) -> (Int, (Char, Bool))
convert (a, b, c) = (a, (b, c))
```

2

f0 :: (Char, Char) -> Bool

f1 :: [Char] -> [Char]

f2 :: b -> (c,a) -> (a,b,c)

f3 :: Char -> Char

f4 :: [Char] -> [Char] -> [Char]

f5 :: Bool -> a -> a -> (a,a)

f6 :: a -> b -> a

f7 :: [Char] -> [Char]

The functions all produced what I expected them to produce.

3

1)

```haskell
pow2 :: Integer -> Integer
pow2 0 = 1
pow2 n = 2 * pow2 (n-1)
```

2)

```haskell
pow2 :: (Ord n, Num n, Num a) => n -> a
pow2 0 = 1
pow2 n = 2 * pow2 (n-1)
```

3)

- Integer: Infinite
- Int: 62
- Double: 1023
- Float: 127

4

1)

f8 and f11.

2)

f8 :: ad-hoc polymorphic, because it uses the Ord type class that needs to be specified per type. The types it can be depends on if the types of x and y have implemented the Ord type class.

f9 :: not polymorphic, becuase it is just 2 bools (Bool -> Bool -> Bool).

f10 :: not polymorphic, it is (Ord n -> Num n -> n) as it x and y can only ever be numbers.

f11 :: parametric polymorphic, because it never accesses x or y and just returns them, although it only ever returns x. 1 can be any type, as the value of x and y are never used.

5

1)

```haskell
(~~) :: String -> String -> Bool
(~~) a b = map toLower a == map toLower b
```

2)

```haskell
reverseCase :: String -> String
reverseCase = map (\c -> if isUpper c then toLower c else toUpper c)
```

3)

```haskell
shift :: Int -> Char -> Char
shift n c = if isAsciiUpper c then asciiShift n c else c
    where
        asciiShift :: Int -> Char -> Char
        asciiShift n c
            | ascii + n > ord 'Z' = chr (result - 26)
            | otherwise = chr result
            where
                ascii = ord c
                result = ascii + n
```

4)

```haskell
caesar :: Int -> String -> String
caesar n = map (shift n . toUpper)
```

5)

```haskell
msg :: String
msg = "ADMNO D HPNO NKMDIFGZ TJP RDOC AVDMT YPNO"

decode :: String
decode = innerDecode msg 25
  where
    innerDecode :: String -> Int -> String
    innerDecode s n
      | n == 1 = result ++ "\n"
      | otherwise = result ++ "\n" ++ innerDecode result (n - 1)
      where
        result = caesar 1 s

decodeResult :: IO ()
decodeResult = putStr decode
```

FIRST I MUST SPRINKLE YOU WITH FAIRY DUST

6

See Database.hs