**Programming Language exercises:**

**Exercises 1.**

What are the types of the following values?

[’a’, ’b’, ’c’]

* [Char]

(’a’, ’b’, ’c’)

* (Char, Char, Char)

[(False, ’0’),(True, ’1’)]

* [(Bool, Char)]

([False, ’0’], [True, ’1’])

* ([Bool],[Char])

[tail, init, reverse ]

* [Function]

Construct the above values by explicitly using the constructors

[ ] :: [a ]

(:) :: a → [a ] → [a ]

(,) :: a → b → (a, b)

What are the types of the following functions?

second xs = head (tail xs)

* second :: [a] -> a

swap (x , y) = (y, x )

* swap :: (a,b ) -> (b,a)

pair x y = (x , y)

* pair :: a -> b -> (a,b)

double x = x ∗ 2

* double :: Num a => a -> a

palindrome xs = reverse xs ≡ xs

* palindrome :: Eq a => [a] -> Bool

twice f x = f (f x)

* twice :: (a->a)->a->a

What are the types of the following expressions? What do they do?

(:) ’a’

(,) 5



(+) 2

Give the implementation of a function orElse :: Maybe t → t → t such that

*expression ‘orElse‘ fallback*

equals fallback if expression is constructed using Nothing, and the value v if it is constructed using Just v.

Give examples of functions with the following types. (Your functions are allowed to be more general, but should work with the following types).

f :: Num a ⇒ (a → a) → a

g :: Num a ⇒ a → (a → a)

h :: Num a ⇒ (a → a) → (a → a)

Give a definition of a function sign ::Num a ⇒ a → a which returns 1 if its argument is positive, −1 if its argument is negative, and 0 otherwise.

Suggest possible types for the following functions

one x = 1

apply f x = f x

compose f g x = f (g x )

**Exercises 2.**

**Boolean functions**

The concept of truth tables for a boolean function is well-known. Below, we see the truth table of ∧ (AND).

|  |  |  |
| --- | --- | --- |
| A | B | a∧b |
| F | F | F |
| F | T | F |
| T | F | F |
| T | T | T |

In Haskell, we can make our own implementation of ∧ using pattern matching

andImpl :: Bool → Bool → Bool

andImpl False False = False

andImpl False True = False

andImpl True False = False

andImpl True True = True

We can shorten it using wildcards

andImpl :: Bool → Bool → Bool

andImpl True True = True

andImpl = False

or

andImpl :: Bool → Bool → Bool

andImpl True a = a

andImpl = False

Implement ¬ (NOT), ∨ (OR), ⊕ (XOR) and NAND using pattern matching and wildcards:

notImpl :: Bool → Bool

* notImpl False = True
* notImpl True = False

orImpl :: Bool → Bool → Bool

* orImpl True False = True
* orImpl True True = True
* orImpl False True = True
* orImpl False False = False

xorImpl :: Bool → Bool → Bool

* xorImpl True False = True
* xorImpl True True = False
* xorImpl False True = True
* xorImpl False False = False

nandImpl :: Bool → Bool → Bool

* nandImpl True False = True
* nandImpl True True = False
* nandImpl False True = True
* nandImpl False False = True

**A few simple functions**

Define a function eeny that returns the string "eeny" for even inputs – and "meeny" for odd inputs.

eeny :: Integer → String

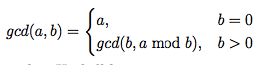
Define a function fizzbuzz that returns "Fizz" for numbers divisible by 3, "Buzz" for numbers divisible by 5, and "FizzBuzz" for numbers divisible by both. For other numbers it returns the empty string.

fizzbuzz :: Integer → String

You can use the function mod to compute modulo.

**Recursive functions on integers**

The mathematical function



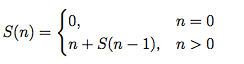
can be implemented in Haskell by:

gcd :: Integer → Integer → Integer

gcd a 0 = a gcd a b =

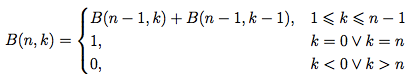
gcd b (a ‘mod‘ b)

The sum of all numbers up to n can be defined using recursion:



Implement a function sumTo :: Integer → Integer in Haskell, which implements this definition.

Binomial coefficients can be defined using recursion:



Implement a function binomial :: Integer → Integer → Integer in Haskell, which implements this definition.

Use recursion to define a function power ::Integer → Integer → Integer . power n k should compute n k .

Use recursion to define a function ilog2 :: Integer → Integer . ilog2 n should be the number of times you can halve the integer n (rounding down) before you get 1.

**Recursive functions on lists**

A list in Haskell is constructed using the following primitives

[ ] :: [a ] -- the empty list

(:) :: a → [a ] → [a ] -- the cons operator

A non-empty list is of the form (x : xs) where x is the first element of the list, and xs is the tail of the list.

As an example of a recursive function on lists, see

flattenMaybe :: [Maybe a ] → [a ]

flattenMaybe [ ] = [ ]

flattenMaybe (Just x : xs) = x : flatten xs

flattenMaybe (Nothing : xs) = flatten xs

This function returns a list of all values which are wrapped in Just - e.g. flatten [Just 1, Nothing, Just 2] ≡ [1, 2].

Please try to implement the following functions using only recursion, pattern matching and the list constructors [ ] and (:). You should not use list functions from Haskell’s standard library. If time permits, you can afterwards find ways to solve the exercises using helper functions from the standard library.

Create a function which drops all the zeros from a list:

dropZeros :: [Int ] → [Int ]

dropZeros [−1, 0, 1, 2, 0] ≡ [−1, 1, 2]

Create a function which flattens a list of lists, i.e.

flatten :: [[a ]] → [a ]

flatten [[1], [ ], [1, 2, 3], [2]] ≡ [1, 1, 2, 3, 2]

Create function which repeats each element of the list:

twiceAll :: [a ] → [a ]

twiceAll [1, 2, 3, 4] ≡ [1, 1, 2, 2, 3, 3, 4, 4]

Create a function which flips the sign of every other element of the list.

alternate :: [Int ] → [Int ]

alternate [1, 2, 3, 4, 5] ≡ [1, −2, 3, −4, 5]

Create a function which repeats each element of the list a specified number of times:

replicateAll :: Int → [a ] → [a ]

replicateAll 2 [1, 2, 3, 4] ≡ [1, 1, 2, 2, 3, 3, 4, 4]

replicateAll 3 [1, 2, 3, 4] ≡ [1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4]

Create function which computes the cumulative sum of a list.

cumulativeSum :: [Int ] → [Int ]

cumulativeSum [1, 2, 3, 4] ≡ [1, 1 + 2, 1 + 2 + 3, 1 + 2 + 3 + 4] ≡ [1, 3, 6, 10]