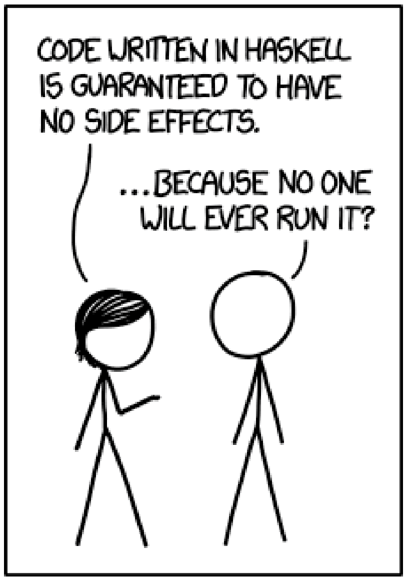
Lab 8. Intro to Haskell

8.1 Getting to know Haskell

Prequisites: having a working haskell environment (Haskell Environment)

Haskell is a general-purpose, purely functional programming language, that we will use for the rest of the semester to showcase functional patterns and programming styles.


CODE WRITTEN IN HASKELL IS GUARANTEED TO HAVE NO SIDE EFFECTS.
...BECAUSE NO ONE WILL EVER RUN IT?

This section is designed for us to get comfortable with haskell syntax, we will use several concept that we learned in Scala, such as tail-recursion, folds and maps, but this time in a purely functional context.

A trip through time

Remember: Lab 1. Introduction to Scala

8.1.1. Implement a tail-recursive function that computes the factorial of a natural number.

```haskell
fact :: Int -> Int

fact = undefined
```

8.1.2. Implement a tail-recursive function that computes the greatest common divisor of two natural numbers.

```haskell
mygcd :: Int -> Int -> Int
mygcd a b = undefined
```

8.1.3. Implement the function `mySqrt` which computes the square root of an integer a.

Lists

The following Scala syntax for working with lists, can be translated to Haskell as follows:

Scala	Haskell cases	Haskell pattern matching	Haskell guards

`def f(l: List[Int]) = l match {`
`  case Nil => ...`
`  case (x::xs) => ...`
`}`
	`f l = case l of`
	`  [] -> ...`
	`  (x:xs) -> ...`
		`f [] = ...`
		`f (x:xs) = ...`
			`f l | l == [] = ...`
			`    | otherwise = ...`

8.1.4. Implement funtions `mymin` and `mymax` that take a list of ints, and return the smallest/biggest value in the list.

8.1.5. Implement a function `unique` that takes a list of ints, and removes all duplicates.

8.1.6. Given a list of ints, return a list of strings where for each element, return:

- **'Fizz'** if the number is divisible by 3
- **'Buzz'** if the number is divisible by 5
- **'FizzBuzz'** if the number is divisible by 3 **and** 5
- a string representation of the number otherwise

8.1.7. Extend the function from **8.1.6.** with the following rules:

- **'Bazz'** if the number is divisible by 7
- **'FizzBazz'** if the number is divisible by 21
- **'BuzzBazz'** if the number is divisible by 35
- **'FizzBuzzBazz'** if the number is divisible by 105

Click to hide ⇱

You can test **8.1.6** and **8.1.7** with the following snippet, if your function is f:

```
f [1..n]
```

In Haskell, the list data type is denote by the type the list holds surrounded by square paranthesis.

```
[Int] -- list of ints
[Double] -- list of doubles
```

```
[[Int]] -- list of lists of ints (matrices)

[] -- !!! not a data type, represents the empty list (Nil in Scala)
```

Types in Haskell

In Haskell, functions are curried by default, **i.e.** a function:

```
f a b = ...
```

is the same as:

```
f = \a -> \b -> ...
```

So, if aa is a `Int` and bb a `Double`, and ff returns a `Char`, it would have the following type:

```
f :: Int -> Double -> Char
```

8.1.8. Check the type signature of the following functions:

- `foldl`
- `foldr`
- `filter`
- `map`

If a function is not ambigous, `ghc` can infer the type signature, for **educational** purposes, going forward you will have to write signatures for all functions you define, this is considered good practice and helps prevent bugs.

In `ghci`, you can check the type of a expression with: `:t`

8.2 A predicate-based implementation for sets

8.2.1. Consider **sets** represented as characteristic functions with signature `s :: Integer → Bool`, where `s x` is true if `x` a member in the set. Examples:

```
s1 1 = True
s1 2 = True
s1 _ = False

s2 x = mod x 2 == 0
```

```
s3 _ = False
```

Above, `s1` is the set $\{1,2\}\{1,2\}$, `s2` is the set of even integers and `s3` is the empty-set. Write a function which tests if an element is a member of a set:

```
mem :: (Integer -> Bool) -> Integer -> Bool

mem = ...
```

8.2.2. Define the set $\{2_n|n\in N\}\{2n|n\in N\}$.
8.2.3. Define the set of natural numbers.

8.2.4. Implement the intersection of two sets. Use lambdas.

```
intersection :: (Integer -> Bool) -> (Integer -> Bool) -> (Integer -> Bool)
```

8.2.5. Write intersection in another way, (without using lambdas).

```
intersection' :: (Integer -> Bool) -> (Integer -> Bool) -> Integer -> Bool
```

8.2.6. Write a function which takes a list of integers, and returns the set which contains them.

```
toSet :: [Integer] -> (Integer -> Bool)
```

8.2.7. Implement a function which takes a list of sets and computes their intersection. Use fold.

```
capList :: [Integer -> Bool] -> Integer -> Bool
```

8.2.8. Implement the function described in **8.2.7** but without using fold.

```
capList' :: [Integer -> Bool] -> Integer -> Bool
```

8.2.9 Write a function that takes a list of sets and an operation over sets (like intersection or union) and applies on the given list. Use fold.

```
setsOperation :: [Integer -> Bool] -> ((Integer -> Bool) -> (Integer -> Bool) ->
(Integer -> Bool)) -> (Integer -> Bool)
```

8.2.10 Write a function that applies a set to a list of integers, resulting a list containing the elements from the given list that are a part of the set.

```
applySet :: (Integer -> Bool) -> [Integer] -> [Integer]
```

8.2.11. Implement a function that takes a set and a list of integers and returns a tuple of list. The first element of the tuple is a list that contains the elements from the given list that are part of the set, and the second element contains the elements that are not part of the set. After implementing your version, check out partition from Data.List and use it!

```
partitionSet :: (Integer -> Bool) -> [Integer] -> ([Integer], [Integer])
```

8.3 Brain Twisters

8.3.1. Implement `map` using `foldl` and `foldr`.

```
mymapl :: (a -> b) -> [a] -> [b]

mymapr :: (a -> b) -> [a] -> [b]
```

8.3.2. Implement `filter` using `foldl` and `foldr`.

```
myfilterl :: (a -> Bool) -> [a] -> [a]

myfilterr :: (a -> Bool) -> [a] -> [a]
```

8.3.3. Implement `foldl` using `foldr`.

```
myfoldl :: (a -> b -> a) -> a -> [b] -> a
```

8.3.4. Implement `bubbleSort`.

```
bubbleSort :: [Int] -> [Int]
```

8.3.5. Implement `quickSort`.

```
quickSort :: [Int] -> [Int]
```