| First name | Last name | Matriculation number |
|---|---|---|
| | | |



## Exercise 1 (4 + 3 + 4 + 5 = 16 points)

The following data structure represents polymorphic lists that can contain values of *two* types in arbitrary order:

```
data DuoList a b = C a (DuoList a b) | D b (DuoList a b) | E
```

Consider the following list `zs` of integers and characters:

$$[\,4,\ 'a',\ 'b',\ 6\,]$$

The representation of `zs` as an object of type `DuoList Int Char` in Haskell would be:

```
C 4 (D 'a' (D 'b' (C 6 E)))
```

Implement the following functions in Haskell.

(a) The function `foldDuo` of type

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

works as follows: `foldDuo f g h xs` replaces all occurrences of the constructor `C` in the list `xs` by `f`, it replaces all occurrences of the constructor `D` in `xs` by `g`, and it replaces all occurrences of the constructor `E` in `xs` by `h`. So for the list `zs` above,

```
foldDuo (*) (\x y -> y) 3 zs
```

should compute

```
(*) 4 ((\x y -> y) 'a' ((\x y -> y) 'b' ((*) 6 3))),
```

which in the end results in `72`. Here, `C` is replaced by `(*)`, `D` is replaced by `(\x y -> y)`, and `E` is replaced by `3`.

```
foldDuo f g h (C x xs)    = f x (foldDuo f g h xs)
foldDuo f g h (D x xs)    = g x (foldDuo f g h xs)
foldDuo f g h E           = h
```

| First name | Last name | Matriculation number |
|---|---|---|
| | | |



(b) Use the `foldDuo` function from (a) to implement the `cd` function which has the type `DuoList Int a -> Int` and returns the sum of the *entries* under the data constructor `C` and of the *number of elements* built with the data constructor `D`.

In our example above, the call `cd zs` should have the result `12`. The reason is that `zs` contains the entries `4` and `6` under the constructor `C` and it contains two elements `'a'` and `'b'` built with the data constructor `D`.

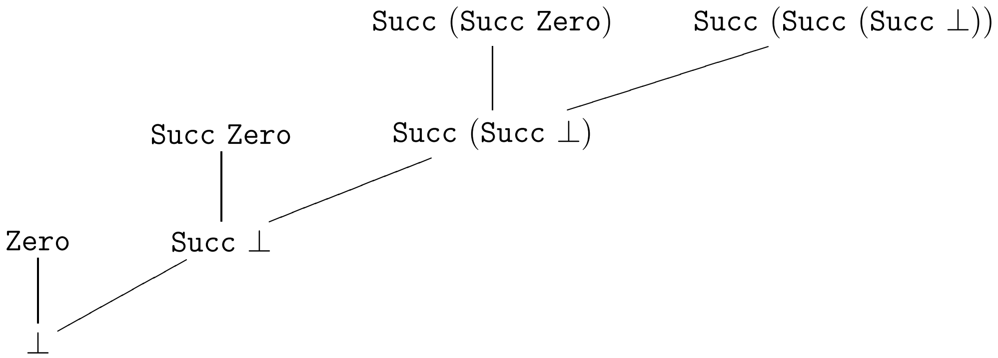
```
cd = foldDuo (+) (\x y -> y + 1) 0
```

| First name | Last name | Matriculation number |
| --- | --- | --- |
| | | |

(c) Consider the following data type declaration for natural numbers:

```
data Nats = Zero | Succ Nats
```

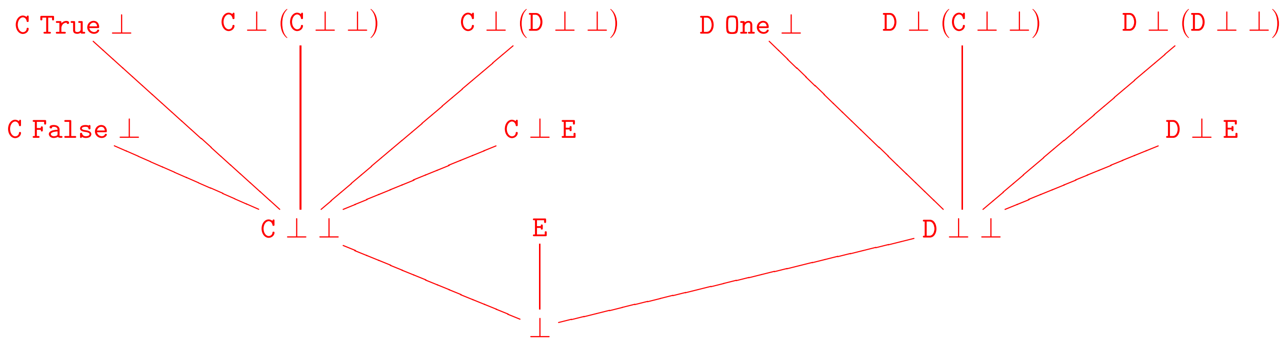
A graphical representation of the first four levels of the domain for `Nats` could look like this:




We define the following data type `Single`, which has only one data constructor `One`:

```
data Single = One
```

Sketch a graphical representation of the first three levels of the domain for the data type `DuoList Bool Single`.



| First name | Last name | Matriculation number |
|---|---|---|
| | | |



(d) Write a Haskell function `printLength` that first reads a line from the user, then prints this string on the console and in the end also prints the length of this string on the console. Also give the type declaration for your function.

You may use the `do`-notation, but you are not obliged to use it. Some of the following pre-defined functions can be helpful:

- `getLine :: IO String` reads a line from the user
- `length :: String -> Int` has the length of a string as its result
- `show :: Int -> String` converts a number to a string
- `putStr :: String -> IO ()` writes a string to the console

An example run should look as given below. Here the string "`foo`" was read from the user.

```
Main> printLength
foo
foo3
```

```
-- without do-notation
printLength :: IO ()
printLength = getLine >>= \s -> putStr s >> putStr (show (length s))

-- alternative: with do-notation
printLength2 :: IO ()
printLength2 = do s <- getLine
                  putStr s
                  putStr (show (length s))
```

| First name | Last name | Matriculation number |
|---|---|---|
| | | |



## Exercise 2 (4 + 5 = 9 points)

Consider the following Haskell declarations for the `square` function:

```
square :: Int -> Int
square 0     = 0
square (x+1) = 1 + 2*x + square x
```

(a) Please give the Haskell declarations for the higher-order function `f_square` corresponding to `square`, i.e., the higher-order function `f_square` such that the least fixpoint of `f_square` is `square`. In addition to the function declaration(s), please also give the type declaration of `f_square`. Since you may use full Haskell for `f_square`, you do not need to translate `square` into simple Haskell.

```
f_square :: (Int -> Int) -> (Int -> Int)
f_square square 0 = 0
f_square square (x+1) = 1 + 2*x + square x
```

(b) We add the Haskell declaration `bot = bot`. For each $n \in \mathbb{N}$ please determine which function is computed by $\texttt{f\_square}^n$ `bot`. Here "$\texttt{f\_square}^n$ `bot`" represents the $n$-fold application of `f_square` to `bot`, i.e., it is short for $\underbrace{\texttt{f\_square (f\_square} \ldots \texttt{(f\_square}}_{n \text{ times}} \texttt{bot)} \ldots)$.

Let $f_n : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$ be the function that is computed by $\texttt{f\_square}^n$ `bot`.
Give $f_n$ in **closed form**, i.e., using a non-recursive definition.

$$(\texttt{f\_square}^n(\perp))(x) = \begin{cases} x^2, & \text{if } 0 \leq x < n \\ \perp, & \text{otherwise} \end{cases}$$

| First name | Last name | Matriculation number |
|---|---|---|
| | | |

6

## Exercise 3 (6 points)

Let $D_1, D_2$ be domains, let $\sqsubseteq_{D_2}$ be a complete partial order on $D_2$. As we know from the lecture, then also $\sqsubseteq_{D_1 \to D_2}$ is a complete partial order on the set of all functions from $D_1$ to $D_2$.

Prove that $\sqsubseteq_{D_1 \to D_2}$ is also a complete partial order on the set of all *constant* functions from $D_1$ to $D_2$. A function $f : D_1 \to D_2$ is called *constant* iff $f(x) = f(y)$ holds for all $x, y \in D_1$.

> *Hint:* The following lemma may be helpful:
>
> If $S$ is a chain of functions from $D_1$ to $D_2$, then $\sqcup S$ is the function with:
>
> $$(\sqcup S)(x) = \sqcup\{f(x) \mid f \in S\}$$

We need to show two statements:

a) The set of all constant functions from $D_1$ to $D_2$ has a smallest element $\perp$.

   Obviously, the constant function $f$ with $f(x) = \perp$ for all $x \in D_1$ satisfies this requirement.

b) For every chain $S$ on the set of all constant functions from $D_1$ to $D_2$ there is a least upper bound $\sqcup S$ which is an element of the set of all constant functions from $D_1$ to $D_2$.

   Let $S$ be a chain of constant functions from $D_1$ to $D_2$. By the above lemma, we have $(\sqcup S)(x) = \sqcup\{f(x) \mid f \in S\}$. It remains to show that the function $\sqcup S : D_1 \to D_2$ actually is a constant function. For all $x, y \in D_1$, we have:

$$\begin{array}{ll} & (\sqcup S)(x) \\ = & \sqcup\{f(x) \mid f \in S\} \\ = & \sqcup\{f(y) \mid f \in S\} \qquad \text{since the elements of } S \text{ are constant functions} \\ = & (\sqcup S)(y) \end{array}$$

   Therefore, also $(\sqcup S)(x)$ is a constant function.

$\square$

| First name | Last name | Matriculation number |
|---|---|---|
| | | |



## Exercise 4 (6 points)

We define the following data structures for natural numbers and polymorphic lists:

```
data Nats = Zero | Succ Nats
data List a = Nil | Cons a (List a)
```

Consider the following expression in complex Haskell:

```
let length Nil          = Zero
    length (Cons x xs) = Succ (length xs)
    in length
```

Please give an equivalent expression `let length = ... in length` in **simple** Haskell.

Your solution should use the functions defined in the transformation from the lecture such as $\texttt{sel}_{n,i}$, $\texttt{isa}_{\underline{\texttt{constr}}}$, and $\texttt{argof}_{\underline{\texttt{constr}}}$. However, you do not have to use the transformation rules from the lecture.

let length = \ys -> if ($\texttt{isa}_{\texttt{Nil}}$ ys)<br>
then Zero<br>
else Succ (length ($\texttt{sel}_{2,2}$ ($\texttt{argof}_{\texttt{Cons}}$ ys)))<br>
in length

| First name | Last name | Matriculation number |
|---|---|---|
| | | |



## Exercise 5 (4 + 5 = 9 points)

Consider the following data structure for polymorphic lists:

```
data List a = Nil | Cons a (List a)
```

(a) Please translate the following Haskell-expression into an equivalent lambda term (e.g., using $\mathcal{L}am$). Recall that pre-defined functions like `even` are translated into constants of the lambda calculus.

It suffices to give the result of the transformation.

```
let f = \x -> if (even x) then Nil else Cons x (f x)
    in f
```

$(\texttt{fix}\ (\lambda f\ x.\ \texttt{if}\ (\texttt{even}\ x)\ \texttt{Nil}\ (\texttt{Cons}\ x\ (f\ x))\ )\ )$

| First name | Last name | Matriculation number |
|---|---|---|
| | | |



(b) Let $\delta$ be the set of rules for evaluating the lambda terms resulting from Haskell, i.e., $\delta$ contains at least the following rules:

$$\begin{aligned} \texttt{fix} &\to \lambda f.\ f\ (\texttt{fix}\ f) \\ \texttt{plus}\ 2\ 3 &\to 5 \end{aligned}$$

Now let the lambda term $t$ be defined as follows:

$$t = (\texttt{fix}\ (\lambda g\ x.\ \texttt{Cons}\ (\texttt{plus}\ x\ 3)\ \texttt{Nil}))\ 2$$

Please reduce the lambda term $t$ by WHNO-reduction with the $\to_{\beta\delta}$-relation. You have to give **all** intermediate steps until you reach **weak head normal form** (and no further steps).

$$\begin{aligned} & (\texttt{fix}\ (\lambda g\ x.\ \texttt{Cons}\ (\texttt{plus}\ x\ 3)\ \texttt{Nil}))\ 2 \\ \to_\delta\ & ((\lambda f.\ f\ (\texttt{fix}\ f))\ (\lambda g\ x.\ \texttt{Cons}\ (\texttt{plus}\ x\ 3)\ \texttt{Nil}))\ 2 \\ \to_\beta\ & ((\lambda g\ x.\ \texttt{Cons}\ (\texttt{plus}\ x\ 3)\ \texttt{Nil})\ (\texttt{fix}\ (\lambda g\ x.\ \texttt{Cons}\ (\texttt{plus}\ x\ 3)\ \texttt{Nil})))\ 2 \\ \to_\beta\ & ((\lambda x.\ \texttt{Cons}\ (\texttt{plus}\ x\ 3)\ \texttt{Nil})\ 2 \\ \to_\beta\ & \texttt{Cons}\ (\texttt{plus}\ 2\ 3)\ \texttt{Nil} \end{aligned}$$

| First name | Last name | Matriculation number |
|---|---|---|
| | | |



## Exercise 6 (10 points)

Use the type inference algorithm $\mathcal{W}$ to determine the most general type of the following lambda term under the initial type assumption $A_0$. Show the results of all sub-computations and unifications, too. If the term is not well typed, show how and why the $\mathcal{W}$-algorithm detects this.

$$\lambda f.\ (\texttt{Succ}\ (f\ x))$$

The initial type assumption $A_0$ contains at least the following:

$$\begin{array}{lcl} A_0(\texttt{Succ}) & = & (\texttt{Nats} \to \texttt{Nats}) \\ A_0(f) & = & \forall a.\ a \\ A_0(x) & = & \forall a.\ a \end{array}$$

$\mathcal{W}(A_0,\ \lambda f.\ (\texttt{Succ}\ (f\ x))\ )$

$\quad \mathcal{W}(A_0 + \{f :: b_1\},\ (\texttt{Succ}\ (f\ x))\ )$

$\quad\quad \mathcal{W}(A_0 + \{f :: b_1\},\ \texttt{Succ})$

$\quad\quad = (id,\ (\texttt{Nats} \to \texttt{Nats})\ )$

$\quad\quad \mathcal{W}(A_0 + \{f :: b_1\},\ (f\ x)\ )$

$\quad\quad\quad \mathcal{W}(A_0 + \{f :: b_1\},\ f)$

$\quad\quad\quad = (id,\ b_1)$

$\quad\quad\quad \mathcal{W}(A_0 + \{f :: b_1\},\ x)$

$\quad\quad\quad = (id,\ b_2)$

$\quad\quad\quad mgu(b_1,\ (b_2 \to b_3)\ ) = [b_1/(b_2 \to b_3)]$

$\quad\quad = ([b_1/(b_2 \to b_3)],\ b_3)$

$\quad\quad mgu((\texttt{Nats} \to \texttt{Nats}),\ (b_3 \to b_4)\ ) = [b_3/\texttt{Nats}, b_4/\texttt{Nats}]$

$\quad = ([b_1/(b_2 \to \texttt{Nats}), b_3/\texttt{Nats}, b_4/\texttt{Nats}],\ \texttt{Nats})$

$= ([b_1/(b_2 \to \texttt{Nats}), b_3/\texttt{Nats}, b_4/\texttt{Nats}],\ ((b_2 \to \texttt{Nats}) \to \texttt{Nats})\ )$

Resulting type: $((b_2 \to \texttt{Nats}) \to \texttt{Nats})$