

Exercise 1 (Programming in Haskell):

(6 + 9 + 9 + 14 = 38 points)

We use the following data structure `Graph a` to represent *undirected* graphs in Haskell by a list of edges, where the nodes are labeled by values of type `a`.

```
data Graph a = Edges [(a,a)]
```

So the edges are represented as pairs of type `(a,a)`. For example, the graph from Fig. 1 can be represented by:

```
gExmp :: Graph Int
gExmp = Edges [(1,2),(1,4),(1,5),(2,3),(2,4),(3,4),(4,5),(4,6)]
```

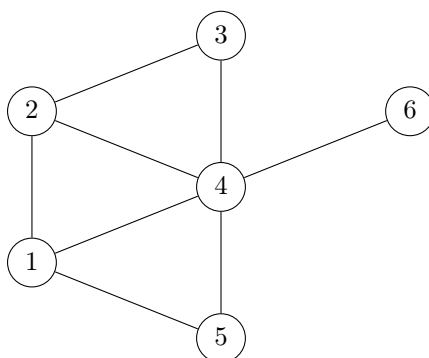


Figure 1: Graph of type `Graph Int`

In each of the following exercises you can use functions from the `Haskell Prelude` and from previous exercises even if you did not implement them. Moreover, you can always implement auxiliary functions.

- Write a Haskell-function `neighbors` and also give its type declaration. Its first argument is a graph `g :: Graph a` for a type `a` from the type class `Eq`, and its second argument is an element `v :: a`. The call `neighbors g v` then results in the list of nodes directly connected by an edge to `v` in `g`. For example, `neighbors gExmp 2 == [1,3,4]`. The order of the resulting list is *irrelevant* and it does not matter if the list contains duplicates.
- Consider the following function which computes a list of all values stored in a graph.

```
nodes :: Eq a => Graph a -> [a]
nodes (Edges es) = removeDuplicates (concat (map (\(x,y)->[x,y]) es))
  where
    removeDuplicates = foldr (\x ys -> x:(filter (x /=) ys)) []
```

Hence, `nodes gExmp` results in `[1,2,4,5,3,6]`.

Write a Haskell-function `existsPath` and also give its type declaration. Its first argument is a graph `g :: Graph a` for a type `a` from the type class `Eq`, and its second and third argument are nodes `v :: a` and `w :: a`. The call `existsPath g v w` results in `True` if and only if there is a path connecting `v` and `w` in `g`. There is a path from `v` to `w` of at most length `n` if and only if `v == w` or there is a neighbor `u` of `v` and a path of at most length `n-1` from `u` to `w`. For example, `existsPath gExmp 6 3 == True`.

Note that a path from `v` to `w` exists if and only if there exists a path from `v` to `w` of at most length `(nodes g)`.

Hints:

- You may use the function `nodes`.
- The function `elem :: Eq a => a -> [a] -> Bool` checks membership in a list, i.e., `elem x xs == True` if and only if `x` occurs in `xs`.

- c) Write a Haskell-function `isConnected` and also give its type declaration. Its only argument is `g :: Graph a` for a type `a` from the type class `Eq`. It returns `True` if and only if for every two elements `v` and `w` of `nodes g`, we have `existsPath g v w == True`. For example, `isConnected gExmp == True`.

Hints:

- You may again use the function `nodes` from part b) of the exercise.
 - List comprehensions are particularly suitable to compute all pairs (v,w) of (distinct) elements of `nodes g`.
 - The function `and :: [Bool] -> Bool` conjuncts all elements in the input, e.g., `and [True, True, True] == True` but `and [True, False, True] == False`.
- d) Implement a function `railNetwork :: Graph String -> IO ()`. This function simulates a network of railway connections as a `Graph String` between cities. The user can either add a connection to the network or check if there exists a connection between two cities.

Furthermore, the following data declaration is given.

```
data NetworkInput = Track (String, String) | Connection (String, String) | Exit
```

When `railNetwork g` is called, it asks the user for an input. If the input is

- `"Track:Start;End"` then an edge from `Start` to `End` is added to `g` and the function `railNetwork` calls itself on the transformed graph again.
- `"Connection:Start;End"` then the user is informed whether there is a path from `Start` to `End` in `g`. The function `railNetwork` then calls itself again with the same argument.
- something else, then the program terminates.

In this exercise, you can assume that there is a function `parseNetworkInput :: String -> NetworkInput` that translates a `String` into a `NetworkInput`, i.e., `parseNetworkInput "Track:Start;End" == Track ("Start", "End")`, `parseNetworkInput "Connection:Start;End" == Connection ("Start", "End")`, and `parseNetworkInput s == Exit` for all other strings `s`.

An example run should look as follows. Here, inputs of the user are written in *italics*.

```
*Main> railNetwork (Edges [])
Enter track or check connection
```

Track:Berlin;Wolfsburg

```
Track added
Enter track or check connection
```

Track:Wolfsburg;Cologne

```
Track added
Enter track or check connection
```

Connection:Cologne;Berlin

```
There is a connection from Cologne to Berlin
Enter track or check connection
```

Connection:Cologne;Munich

```
This connection does not exist
Enter track or check connection
```

Stop

*Main>

Hints:

- The function `putStrLn::String -> IO()` prints a string and performs a line break.
- The function `getLine::IO String` reads a string from the keyboard.

Solution:

```

import Data.Char

data NetworkInput = Track (String, String) | Connection (String, String) | Exit

trim :: String -> String
trim = f . f
  where f = reverse . dropWhile isSpace

parseNetworkInput :: String -> NetworkInput
parseNetworkInput input =
  | lhs == "Track" = Track ((trim (drop 1 start)),
                           (trim (drop 1 end)))
  | lhs == "Connection" = Connection ((trim (drop 1 start)),
                                       (trim (drop 1 end)))
  | otherwise = Exit
  where
    (lhs, rhs) = span (/= ':') (trim input)
    (start, end) = span (/= ';') (trim rhs)

data Graph a = Edges [(a,a)]

nodes :: Eq a => Graph a -> [a]
nodes (Edges es) = removeDuplicates (concat (map \(x,y)->[x,y]) es)
  where
    removeDuplicates = foldr (\x ys -> x:(filter (x /=) ys)) []

a) neighbors :: Eq a => Graph a -> a -> [a]
neighbors (Edges es) v = [n | (n, m) <- es, m == v] ++ [m | (n, m) <- es, n == v]

b) existsPath :: Eq a => Graph a -> a -> a -> Bool
existsPath (Edges es) x y = existsPathHelp (length (nodes (Edges es))) (Edges es) x y
  where
    existsPathHelp 0 _ x y = (x == y)
    existsPathHelp n (Edges es) x y = let neigh = neighbors (Edges es) y in
      (elem x neigh) || ((filter (existsPathHelp (n-1) (Edges es) x) neigh) /= [])

c) isConnected :: Eq a => Graph a -> Bool
isConnected g = let vertices = nodes g in
  and [(existsPath g v w) | v <- vertices, w <- vertices, v /= w]

d) railNetwork :: Graph String -> IO ()
railNetwork (Edges tracks) = do
  putStrLn "Enter track or check connection"
  input <- getLine
  case (parseNetworkInput input) of
    Track (start, end) -> do
      putStrLn "Track added"
      railNetwork (Edges ((start,end):tracks))
    Connection (start, end) -> do
      if (existsPath (Edges tracks) start end) then

```

```

    putStrLn ("There is a connection from " ++ start ++ " to " ++ end)
  else putStrLn "This connection does not exist"
railNetwork (Edges tracks)
Exit -> return ()

```

Exercise 2 (Semantics):

((9 + 11) + (5 + 7 + 4) + 7 = 43 points)

- a) i) Let D be a domain and \sqsubseteq a complete partial order on D . Let $f : D \rightarrow D$ be a continuous function w.r.t. \sqsubseteq and $\text{Fix}(f) \subseteq D$ its set of fixpoints. Let $\sqsubseteq|_{\text{Fix}(f)}$ be the restriction of \sqsubseteq to $\text{Fix}(f)$, i.e., for every $x, y \in \text{Fix}(f)$ let $x \sqsubseteq|_{\text{Fix}(f)} y$ hold iff $x \sqsubseteq y$. Prove that $\sqsubseteq|_{\text{Fix}(f)}$ is complete on $\text{Fix}(f)$.
- ii) Let D be a domain, \sqsubseteq a complete partial order on D , and $f : D \rightarrow D$ a continuous function w.r.t. \sqsubseteq . By i), f has a least fixpoint $\text{lfp } f$. Prove the following claim.

If $d \in D$ with $f(d) \sqsubseteq d$ then $\text{lfp } f \sqsubseteq d$.

Hints:

- Use the characterization of the least fixpoint from the Fixpoint Theorem and perform an induction proof.

- b) i) Consider the following Haskell function `f`:

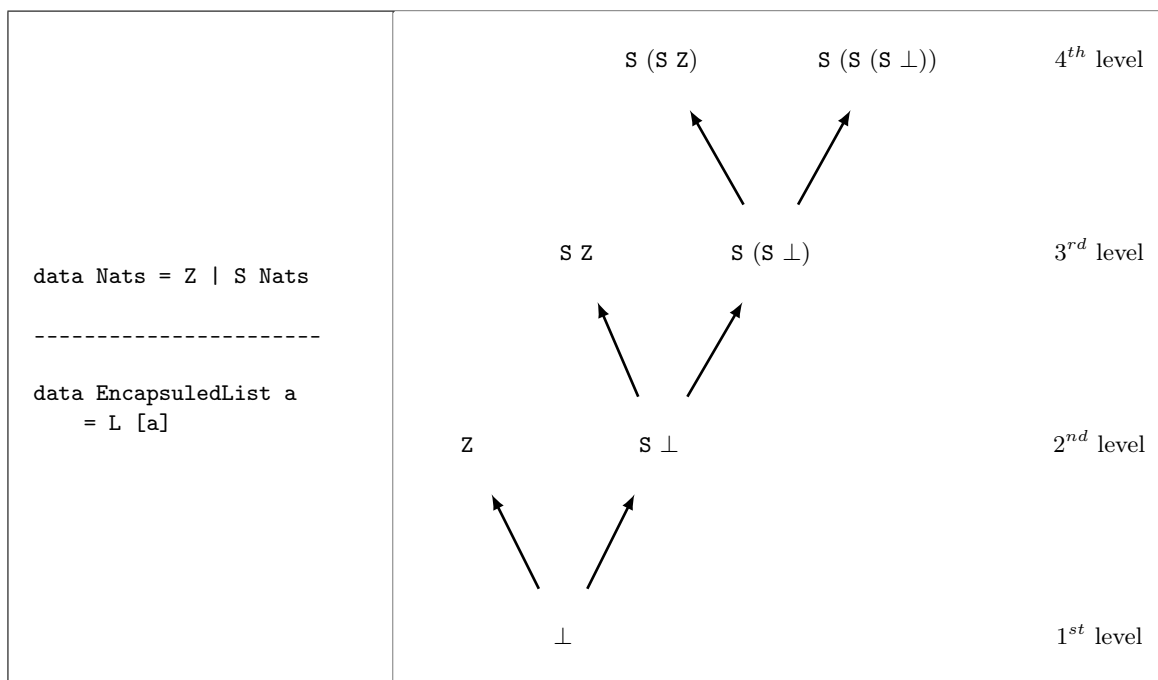
```

f :: (Int, Int, Int) -> Int
f (x, y, 0) = 1
f (x, y, n) = x * f(x, y, n-1) + y * f(x, y, n-1)

```

Please give the Haskell declaration for the higher-order function `ff` corresponding to `f`, i.e., the higher-order function `ff` such that the least fixpoint of `ff` is `f`. In addition to the function declaration, please also give the type declaration for `ff`. You may use full Haskell for `ff`.

- ii) Let ϕ_{ff} be the semantics of the function `ff`. Give the definition of $\phi_{ff}^m(\perp)$ in closed form for any $m \in \mathbb{N}$, i.e., give a non-recursive definition of the function that results from applying ϕ_{ff} m -times to \perp . Here, you should assume that `Int` can represent all integers, so no overflow can occur.
- iii) Give the definition of the least fixpoint of ϕ_{ff} in closed form.
- c) Consider the data type declarations on the left and, as an example, the graphical representation of the first four levels of the domain for `Nats` on the right:



Give a graphical representation of the first four levels of the domain for the type `EncapsuledList Bool`.

Solution: _____

- a) i) • By the Fixpoint Theorem, $\text{lfp } f$ exists, i.e., $\text{Fix}(f)$ has a smallest element w.r.t. $\sqsubseteq_{|\text{Fix}(f)}$.
- Let $S \subseteq \text{Fix}(f)$ be a chain. Then $f(S) = \{f(s) \mid s \in S\} = \{s \mid s \in S\} = S$ as S only consists of fixpoints. As \sqsubseteq is complete on D , $\text{lub } S \in D$ exists. As f is continuous we have $f(\text{lub } S) = \text{lub } f(S) = \text{lub } S$, i.e., $\text{lub } S \in \text{Fix}(f)$. So, every chain has a least upper bound. This completes the proof.
- ii) By the Fixpoint Theorem we have $\text{lfp } f = \text{lub } \{f^n(\perp) \mid n \in \mathbb{N}\}$. We will show by induction that $f^n(\perp) \sqsubseteq d$ for any $n \in \mathbb{N}$. Then, d is an upper bound on the set $\{f^n(\perp) \mid n \in \mathbb{N}\}$, hence, $\text{lfp } f = \text{lub } \{f^n(\perp) \mid n \in \mathbb{N}\} \sqsubseteq d$ as it is the least upper bound.
- I.B. We have $\perp \sqsubseteq d$ by definition.
- I.H. Assume $f^n(\perp) \sqsubseteq d$ for some fixed $n \in \mathbb{N}$.
- I.S. We have

$$\begin{aligned}
 f^{n+1}(\perp) &= f(f^n(\perp)) \\
 &\sqsubseteq f(d) \sqsubseteq d. \quad (\text{By continuity} \Rightarrow \text{monotonicity, I.H and assumption.})
 \end{aligned}$$

- b) i) `ff :: ((Int, Int, Int) -> Int) -> ((Int, Int, Int) -> Int)`
`ff g (x, y, 0) = 1`
`ff g (x, y, n) = x * g(x, y, n-1) + y * g(x, y, n-1)`

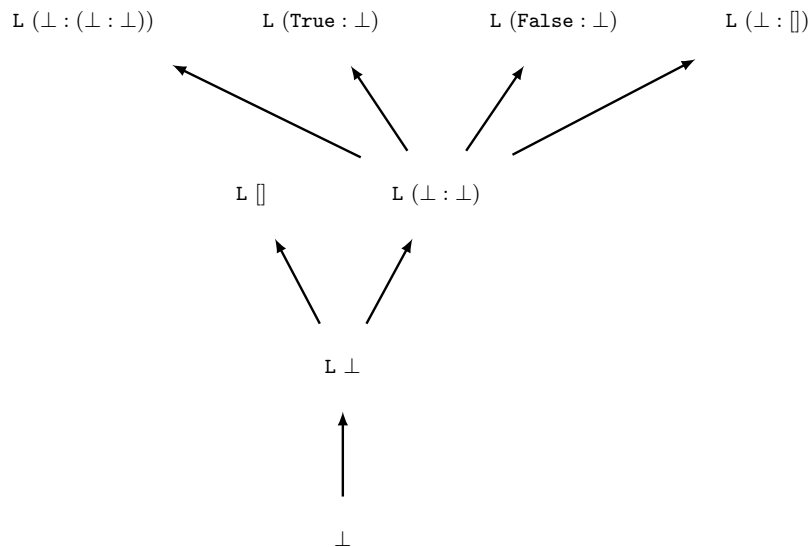
ii) $\phi_{ff}^0(\perp) = \perp$

$$\phi_{ff}^1(\perp)(x, y, n) = \begin{cases} 1, & n = 0 \\ \perp, & \text{otherwise} \end{cases}$$

$$\phi_{ff}^2(\perp)(x, y, n) = \begin{cases} 1, & n = 0 \\ (x + y)^n, & 0 < n < 2 \wedge x, y \neq \perp \\ \perp, & \text{otherwise} \end{cases}$$

$$\phi_{ff}^m(\perp)(x, y, n) = \begin{cases} 1, & n = 0 \wedge m > 0 \\ (x + y)^n, & 0 < n < m \wedge x, y \neq \perp \\ \perp, & \text{otherwise} \end{cases}$$

iii) $\text{lfp } \phi_{ff}(x, y, n) = \begin{cases} 1, & n = 0 \\ (x + y)^n, & 0 < n \wedge x, y \neq \perp \\ \perp, & \text{otherwise} \end{cases}$



Exercise 3 (Lambda Calculus):

((5 + 5) + 8 + 6 = 24 points)

- a) Consider the following function computing the difference of two natural numbers (if the first argument is not smaller than the second). Here, natural numbers are represented by the data structure `Nats` defined by `data Nats = Z | S Nats`.

```

minus :: Nats -> Nats -> Nats
minus x Z = x
minus (S x) (S y) = minus x y

```

- i) Please give an equivalent function in *simple Haskell*.
- ii) Implement the function `minus` in the *lambda calculus*, i.e., give a lambda term q such that, for all natural numbers m, n the term $q (S^m Z) (S^n Z)$ can be reduced to

$$\begin{cases} S^{m-n} Z, & \text{if } m \geq n \\ \text{bot}, & \text{otherwise} \end{cases}$$

via WHNO-reduction with the $\rightarrow_{\beta\delta}$ -relation and the set of rules δ as introduced in the lecture to implement Haskell.

Hints:

- You can use infix notation for predefined functions like `(&&)` in both simple Haskell and the lambda calculus.
- You do *not* have to use the transformation algorithms presented in the lecture. It is sufficient to just give an equivalent simple program and an equivalent lambda term.

- b) Let

$$t = \lambda g n. \text{if } (n \leq 0) 1 (g n)$$

and

$$\begin{aligned} \delta = \{ & \text{if True} \rightarrow \lambda x y. x, \\ & \text{if False} \rightarrow \lambda x y. y, \\ & \text{fix} \rightarrow \lambda f. f(\text{fix } f) \} \\ \cup \{ & x \leq y \rightarrow \text{True} \mid x \leq y \in \mathbb{Z} \} \\ \cup \{ & x \leq y \rightarrow \text{False} \mid x > y \in \mathbb{Z} \} \end{aligned}$$

Please reduce `fix t 0` by WHNO-reduction with the $\rightarrow_{\beta\delta}$ -relation. List **all** intermediate steps until reaching weak head normal form, but please write “ t ” instead of

$$\lambda g n. \text{if } (n \leq 0) 1 (g n)$$

whenever possible.

- c) Consider the representation of natural numbers in the pure lambda calculus presented in the lecture (i.e., $n \in \mathbb{N}$ is represented by the term $\bar{n} = \lambda f x. f^n x$) and the representation of Booleans (i.e., $\overline{\text{True}} = \lambda x y. x$ and $\overline{\text{False}} = \lambda x y. y$). Give a pure lambda term for the function `isZero`, such that `isZero`($\lambda f x. f^n x$) can be reduced to $\overline{\text{False}}$ if $n > 0$ and to $\overline{\text{True}}$ if $n = 0$.

Explain your solution shortly. You may give a reduction sequence as explanation.

Hints:

- A function $\lambda x.t$ where x does *not* occur as a free variable in the term t is a constant function. Then applying $\lambda x.t$ to a term multiple times still evaluates to t .

Solution:

- a) i)

```

minus = \x -> \y ->
  if (isa_Z y) then x
  else if (isa_S x) && (isa_S y) then minus (argof_S x) (argof_S y)
  else bot

```
- ii)

```

fix (\lambda g x y. if (isa_Z y) x (if ((isa_S x) && (isa_S y)) (g (argof_S x) (argof_S y)) bot)

```

b)

$$\begin{aligned}
& \text{fix } t \ 0 \\
& \rightarrow_{\delta} (\lambda f. (f (\text{fix } f))) \ t \ 0 \\
& \rightarrow_{\beta} t (\text{fix } t) \ 0 \\
& \rightarrow_{\beta} (\lambda n. \text{if } (n \leq 0) \ 1 \ ((\text{fix } t) \ n)) \ 0 \\
& \rightarrow_{\beta} \text{if } (0 \leq 0) \ 1 \ ((\text{fix } t) \ 0) \\
& \rightarrow_{\delta} \text{if } \text{True} \ 1 \ ((\text{fix } t) \ 0) \\
& \rightarrow_{\delta} (\lambda x \ y.x) \ 1 \ ((\text{fix } t) \ 0) \\
& \rightarrow_{\beta} (\lambda y.1) \ ((\text{fix } t) \ 0) \\
& \rightarrow_{\beta} 1
\end{aligned}$$

c)

$$\text{isZero} = \lambda n.n \ (\lambda z.\overline{\text{False}}) \ \overline{\text{True}}$$

The representation of n applies the function f n times to some x . If $n = 0$ then the first argument is simply ignored, i.e., it can be reduced to True . If $n > 0$ then the constant function $\lambda z.\overline{\text{False}}$ is applied n -times to True which results in $\overline{\text{False}}$.

$$\begin{aligned}
& \text{isZero } \bar{0} \\
& \rightarrow_{\beta} \bar{0} \ (\lambda z.\overline{\text{False}}) \ \overline{\text{True}} \\
& \rightarrow_{\beta}^* \overline{\text{True}} \\
& \\
& \text{isZero } \bar{n} \\
& \rightarrow_{\beta} \bar{n} \ (\lambda z.\overline{\text{False}}) \ \overline{\text{True}} \\
& = (\lambda f \ x.f^n \ x) \ (\lambda z.\overline{\text{False}}) \ \overline{\text{True}} \\
& \rightarrow_{\beta}^* (\lambda z.\overline{\text{False}})^n \ \overline{\text{True}} \\
& \rightarrow_{\beta}^* \overline{\text{False}}
\end{aligned}$$

(Hint)

Exercise 4 (Type Inference):

(15 points)

Using the initial type assumption $A_0 := \{f :: \forall a.a \rightarrow \text{Int}, g :: \forall a.a \rightarrow a \rightarrow a\}$, infer the type of the expression $\lambda y.g \ (f \ y)$ using the algorithm \mathcal{W} .

Hints:

- When writing $\mathcal{W}(A, t) = (\theta, t)$, you do not have to give the full substitution θ , but it is enough to give the parts of θ that concern the free variables in the type schemas of A .

Solution: _____

$$\begin{aligned}
& \mathcal{W}(A_0, \lambda y. g \ (f \ y)) \\
& \quad \mathcal{W}(A_0 + \{y :: b_1\}, g \ (f \ y)) \\
& \quad \quad \mathcal{W}(A_0 + \{y :: b_1\}, g) \\
& \quad \quad = (id, b_2 \rightarrow b_2 \rightarrow b_2) \\
& \quad \quad \mathcal{W}(A_0 + \{y :: b_1\}, f \ y) \\
& \quad \quad \quad \mathcal{W}(A_0 + \{y :: b_1\}, f) \\
& \quad \quad \quad = (id, b_3 \rightarrow \mathbf{Int}) \\
& \quad \quad \quad \mathcal{W}(A_0 + \{y :: b_1\}, y) \\
& \quad \quad \quad = (id, b_1) \\
& \quad \quad \quad \mathbf{mgu} \ (b_3 \rightarrow \mathbf{Int}, b_1 \rightarrow b_4) \\
& \quad \quad \quad = [b_3/b_1, b_4/\mathbf{Int}] \\
& \quad \quad = (id, \mathbf{Int}) \\
& \quad \quad \mathbf{mgu} \ (b_2 \rightarrow b_2 \rightarrow b_2, \mathbf{Int} \rightarrow b_5) \\
& \quad \quad = [b_2/\mathbf{Int}, b_5/\mathbf{Int} \rightarrow \mathbf{Int}] \\
& \quad (id, \mathbf{Int} \rightarrow \mathbf{Int}) \\
& (id, b_1 \rightarrow (\mathbf{Int} \rightarrow \mathbf{Int}))
\end{aligned}$$

Under the type assumption A_0 the most general type of $\lambda y. g \ (f \ y)$ is $b \rightarrow (\mathbf{Int} \rightarrow \mathbf{Int})$.