|  |  |  |
| --- | --- | --- |
| **Rules:**  – The output depends on the inputs only!  – No side effects are allowed!  Haskell functions are pure!  **(f 42) – (f 42) = 0 IMMER!**  – For every specific input a function always computes exactly the same output!  name :: type  name = expression  size :: Integer  size = (7 - 3) \* 2  **Function Application / Evaluation**  negSum (times2 3) (times2 (-4))  ~> negSum (2 \* 3) (times2 (-4)) using (t2)  ~> negSum 6 (times2 (-4)) arithmetic  ~> negSum 6 (2 \* (-4)) using (t2)  ~> negSum 6 (-8) arithmetic  ~> - (6 + (-8)) using (ns)  ~> - (-2) arithmetic  ~> 2 arithmetic  **Basic Types** (Types that cannot be made up of other types)  – Bool (&& || not)  – Char ('a')  – Int (-2^63 to 2^63-1)  – Integer (all Integers - more memory,slower computation)  – Double (floating point numbers)  **Enumerations**  **data** Color = Red **|** Yellow **|** Green  deriving (Show)  **Aggregated Types**  – Tuples (GPS Coordinates: an aggregation of (Longitude, Latitiude)), A tuple is a finite sequence of components of possibly different type (T1, T2, ...,Tn)    sndInt :: (Int,Int) -> Int sndInt (x, y) = y  fstInt :: (Int,Int) -> Int fstInt (x, y) = x    – Lists (Path / Directions: the way from Brugg to Zurich as a list of GPS Coordinates / waypoints  **Type Synonyms**  type Coord = (Int, Int)  xCoord :: Coord -> Int  xCoord (x, y) = x  functionname :: Inputtype -> Outputtype  **Basic classes**  **Eq** – equality types  – Contains types whose values can be compared for equality and inequality  – methods: (==), (/=)  **Ord** – ordered types  – Contains types whose values are totally ordered  – methods: (<), (<=), (>), (>=), min, max  **Show** – showable types  – Contains types whose values can be converted into strings of characters  – method show :: a -> String  **Num** – numeric types  – Contains types whose values are numeric  – methods: (+), (-), (\*), negate, abs, signum  **Integral** – integral types  – Contains types that are numeric but of integral value  – methods: div, mod  **Fractional** – fractional types  – Contains types that are numeric but of fractional value  – methods: (/), recip  Use fromIntegral to convert from Int / Integer back to Num a  fromIntegral :: (Integral a, Num b) => a -> b | | **Lists**  **filter** :: (a -> Bool) -> [a] -> [a]  filter is a function that takes a predicate (a predicate is a function that tells whether something is true or not, so in our case, a function that returns a boolean value) and a list and then returns the list of elements that satisfy the predicate.  filter even [2,3,4] ~> [2,4]  Input: filter (>5) [1,2,3,4,5,6,7,8]  Output: [6,7,8]  Input: filter odd [3,6,7,9,12,14]  Output: [3,7,9]  Input: filter (\x -> length x > 4) ["aaaa","bbbbbbbbbbbbb","cc"]  Output: ["bbbbbbbbbbbbb"]  positives :: [Int] -> [Int]  positives xs = filter pos xs  where pos x = x > 0  ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]  [5,6,4]  ghci> filter (==3) [1,2,3,4,5]  [3]  ghci> filter even [1..10]  [2,4,6,8,10]  ghci> let notNull x = not (null x) in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]  [[1,2,3],[3,4,5],[2,2]]  ghci> filter (`elem` ['a'..'z']) "u LaUgH aT mE BeCaUsE I aM diFfeRent"  "uagameasadifeent"  ghci> filter (`elem` ['A'..'Z']) "i lauGh At You BecAuse u r aLL the Same"  "GAYBALLS"  **map** :: (a -> b) -> [a] -> [b]  map takes a function and a list and applies that function to every element in the list, producing a new list.  map length ["ha", "skel", "l" ] ~> [2,4,1]  ghci> map (+3) [1,5,3,1,6]  [4,8,6,4,9]  ghci> map (++ "!") ["BIFF", "BANG", "POW"]  ["BIFF!","BANG!","POW!"]  ghci> map (replicate 3) [3..6]  [[3,3,3],[4,4,4],[5,5,5],[6,6,6]]  ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]  [[1,4],[9,16,25,36],[49,64]]  ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]  [1,3,6,2,2]  **Let**  cylinder :: Float -> Float -> Float  cylinder r h =  **let** sideArea = 2 \* pi \* r \* h  topArea = pi \* r ^2  **in** 2 \* topArea + sideArea  **Where** (increase readability)  brainpower :: Double -> Double -> String  brainpower mentalAge age  | **iq** < 85 = "lower than average"  | **iq** < 115 = "average"  | **iq** < 135 = "higher than average"  | otherwise = "gifted (top 1 %)"  **where** **iq** = (mentalAge / age) \* 100  cylinder :: Float -> Float -> Float  cylinder r h = 2 \* topArea + sideArea  where sideArea = 2 \* pi \* r \* h  topArea = pi \* r ^ 2  initials :: String -> String -> String  initials firstname lastname = [f] ++ ". " ++ [l] ++ "."  where (f:\_) = firstname  (l:\_) = lastname  Where bindings are visible within the whole function clause. If the visibility (the scope) of a binding should be narrower, use let  **Case**  **case** expression **of**  pattern -> result  pattern -> result  describeList :: [a] -> String  describeList xs = "The list is " ++ **case** xs **of**  [] -> "empty."  [x] -> "a singleton list."  xs -> "a longer list."  **If-Then-Else**  **if** a < b **then** a **else** b (a = b, gleicher Type)  **if** a == b  **then** "Eq"  **else** "Not Eq"  **Guards**  abs :: (Num a, Ord a) => a -> a  abs n = **case** n **of**  m **|** m < 0 -> -m  **|** **otherwise** -> m  **Präzedenz**    **Higher Order Functions**  A function can also take another function as its argument  filter :: **(**a -> Bool**)** -> [a] -> [a]  Functions are just values! They can be  – input and output to functions  – put into and retrieved from data structures  Functions that have functions as input and/or output are called higher-order functions  **Function Composition: Pipelines of functions**  Functions can be combined into pipelines if the types line up  f :: a -> b g :: b -> c  (.) :: (b -> c) -> (a -> b) -> (a -> c)  g . f = \x -> g (f x)  **Operators**  An Operator can be used as a function by wrapping it with parentheses **(**op**)**  (+) 1 2  A function can be used as an operator by quoting it with backticks `fun`  9 `div` 2  **Sections**  Infix operators can be partially applied as well.  (2+) is interpreted as \y -> 2+y  (+3) is interpreted as \x -> x+3  **Lambdas**  \p -> e  🡭 🡬  parameter / pattern result, defined in terms of parameter  prod1 x y z = x \* y \* z  prod4 = \x -> \y -> \z -> x \* y \* z |
| **List Functions** | |
| **S:** | (++) :: [a] -> [a] -> [a] |
| **EX:** | [1,2]++[3,4,5] ~> [1,2,3,4,5] "Hallo " ++ show 12 ~> "Hallo 12" |
| **B:** | Hängt zwei Listen aneinander. Wird entsprechend verwendet um Strings zu konkatenieren. |
| **S:** | take :: Int -> [a] -> [a] |
| **EX:** | take 3 ['a','b','c','d','e'] ~> ['a','b','c'] |
| **B:** | Nimmt die ersten n Elemente |
| **S:** | drop :: Int -> [a] -> [a] |
| **EX:** | drop 1 [1,2,3] ~> [2,3] |
| **B:** | Wirft die ersten n Elemente weg. |
| **S:** | (!!) :: [a] -> Int -> a |
| **EX:** | (!!) [1,2,3] 1 ~> 2 |
| **B:** | Gibt das Element an Index x zurück. Start bei 0 |
| **S:** | last :: [a] -> a |
| **EX:** | last [1 , 2, 3] ~> 3 |
| **B:** | Gibt das letzte Element zurück. |
| **S:** | init :: [a] -> [a] |
| **EX:** | init ['a','b','c','d'] ~> ['a','b','c'] |
| **B:** | Gibt alle Elemente einer Liste ausser dem letzten zurück |
| **S:** | reverse :: [a] -> [a] |
| **EX:** | reverse [1,2,3] ~> [3,2,1] |
| **B:** | Dreht eine Liste um. |
| **S:** | elem :: Eq a => a -> [a] -> Bool |
| **EX:** | elem 1 -> [1,2,3] -> True |
| **B:** | Ist a Element von der Liste |
| **S:** | maximum :: Ord a => [a] -> a minimum :: Ord a => [a] -> a |
| **EX:** | maximum [1,4,3] ~> 4 minimum [1,4,3] ~> 1 |
| **B:** | Gibt den grössten bzw den kleinsten Wert zurück. |
| **S:** | sum :: Num a => [a] -> a product :: Num a => [a] -> a |
| **EX:** | sum [1,2,3] ~> 6 product [2,3,4] ~> 24 |
| **B:** | Gibt die Summe/ das Produkt zurück. Die Listenelemente müssen von einem Zahlen Typen sein (Num). |
| **S:** | zip :: [a] -> [b] -> [(a,b)] |
| **EX:** | zip [1,2] [3,4] ~> [(1,3),(2,4)] |
| **B:** | Macht Tuple aus zwei Listen |
| **S:** | concat :: [[a]] -> [a] |
| **EX:** | concat [[1],[2,3],[4]] ~> [1,2,3,4] concat ["abc","def"] ~> "abcdef" |
| **B:** | Hängt Listen aneinander zu einer Liste |
| **S:** | zipWith :: (a -> b -> c) -> [a] -> [b] -> [c] |
| **EX:** | zipWith (+) [1,2,3] [10,11,12] ~> [11,13,15] zipWith (++) ["Ha","Ec"] ["llo","ho"] ~> ["Hallo","Echo"] |
| **B:** |  |
| **Curry / Uncurry:**  aadd :: (Int, Int) -> Int  aadd (a,b) = a+b  curry :: ((a,b) -> c) -> a -> b -> c  curry f = \a -> \b -> f(a,b)  add :: Int -> Int -> Int  add a b = a+b  uncurry :: (a -> b -> c) -> (a,b) -> c  uncurry f = \(a,b) -> f a b | |

|  |  |
| --- | --- |
| **Entfernt alle doppelte Einträge aus einer Liste**  diy :: Eq a => [a] -> [a]  diy [] = []  diy (x:xs) = x : diy (diy' x (xs))  diy' :: Eq a => a -> [a] -> [a]  diy' a [] = []  diy' a (y:xs)  | a == y = diy' a xs  | otherwise = y : diy' a xs  rem n is = concat (map (\i -> if i == n then [] else [i]) is)  **Quicksort**  qsort [] = []  qsort (p:xs) = qsort lesser ++ [p] ++ qsort greater  where lesser = filter (< p) xs  greater = filter (>= p) xs  quicksort :: (Ord a) => [a] -> [a]  quicksort [] = []  quicksort (x:xs) =  let smallerSorted = quicksort [a | a <- xs, a <= x]  biggerSorted = quicksort [a | a <- xs, a > x]  in smallerSorted ++ [x] ++ biggerSorted  **Insertion Sort**  sort :: (Ord a) => [a] -> [a]  sort [] = []  sort (x : xs) = insert x (sort xs)  insert :: (Ord a) => a -> [a] -> [a]  insert x [] = x : []  insert x (y : ys) = insert' (compare x y) x y ys  insert' :: (Ord a) => Ordering -> a -> a -> [a] -> [a]  insert' LT x y ys = x : (y : ys)  insert' \_ x y ys = y : insert x ys  **Factorial**  factorial 0 = 1  factorial n = n \* factorial (n - 1)  **Maximum**  maximum' :: (Ord a) => [a] -> a  maximum' [] = error "maximum of empty list"  maximum' [x] = x  maximum' (x:xs) = max x (maximum' xs)  **Replicate**  replicate' :: (Num i, Ord i) => i -> a -> [a]  replicate' n x  | n <= 0 = []  | otherwise = x:replicate' (n-1) x  **Take**  take' :: (Num i, Ord i) => i -> [a] -> [a]  take' n \_  | n <= 0 = []  take' \_ [] = []  take' n (x:xs) = x : take' (n-1) xs  **Insert At**  insertAt :: a -> [a] -> Int -> [a]  insertAt x xs 0 = x : xs  insertAt x xs i = head xs : insertAt x (tail xs) (i - 1)  -- ins a n as = take n as ++ [a] ++ drop n as  **ReverseAll**  reverseAll :: [a] -> [a]  reverseAll xs = head xs : tail (reverse (tail xs)) ++ [head (reverse xs)]  -- rev (a:as) = [a] ++ reverse (init as) ++ [last as]  **Largest Divisible**  largestDivisible :: (Integral a) => a  largestDivisible = head (filter p [100000,99999..])  where p x = x `mod` 3829 == 0  **Types:**  f1 [] = ’’  f1 (x:xs) = x :: String -> Char  f2 [] = false  f2 (x:\_) = x :: [Bool] -> Bool  f3 (x:xs) = (x,True) :: [a] -> (a,Bool)  f4 (x:xs) = (3,xs) :: [a] -> (Int,[a])  f5 \x -> \y -> (x,y) :: a -> b -> (a,b)  f6 \x -> \y -> (y,x) :: a -> b -> (b,a)  f7 \x -> \y -> \z -> y+1 :: a -> Int -> b -> Int  f8 \x -> \y -> \z -> y==True :: a -> Bool -> b -> Bool  f9 [[x,\_],[\_,y]] -> x+y :: [[Int]] -> Int  f10 ([x,\_],[\_,y]) -> x+y :: ([Int],[Int]) -> Int  f11 = map init :: [a] -> [a]  f12 = map head :: [[a]] -> [a]  f13 = zipWith (.) :: [b -> c] -> [a -> b] -> [a -> c]  f14 = zipWith filter :: [a -> Bool] -> [[a]] -> [[a]]  f15 = map snd :: [(a,b)] -> [b]  f16 = map map :: [a -> b] -> [[a] -> [b]]  f17 = drop 3 :: [a] -> [a]  f18 = (\x -> x > 9) 6 :: Bool  f19 = (\x -> tail x) :: [a] -> [a]  f20 = (\(a,b) -> b++ a :: ([a],[a]) -> [a]  f21 = (\t -> fst) :: c -> ((a,b) -> a)  f22 = (\(x:xs) -> x :: [a] -> a  f23 = (\x y -> head y) 2 :: Int -> [a] -> a  f24 = \(a,b) -> fst a ++ b :: (([a],b),[a]) -> [a]  **Lambda expressions:**  incAll :: [Int] -> [Int]  incAll xs = map (\x -> x + 1) xs  incAll,incAll2,incAll3 :: [Int] -> [Int]  incAll = map (\x -> x+1)  incAll2 a = map (\x -> x+1) a  incAll3 = \a -> map (\x -> x+1) a  addToAll …addToAll4 :: Int -> [Int] -> [Int]  addToAll a = map (\x -> a+x)  addToAll2 a b = map (\x -> a+x) b  addToAll3 = \a -> \b -> map (\x -> a+x) b  addToAll4 = \a -> map (\x -> a+x)  keepOld …keepOld3 :: [Int] -> [Int]  keepOld = filter (\x -> x >= 90)  keepOld2 a = filter (\x -> x >= 90) a  keepOld3 = \a -> filter (\x -> x >= 90) a  dropShort …dropShortSS :: [String] -> [String]  dropShort = \a -> filter (\x -> length x > 1) a  dropShortS ss = filter (\x -> length x > 1) ss  dropShortSS = filter (\x -> length x > 1) | **Spamfilter**  normalize :: String -> [String]  normalize [] = []  normalize b = filter laenge spittedBetreff  where laenge wort = length wort >= 3  splittedBetreff = words (map toLower betreff)  rateWord :: String -> Int  rateWord [] = 0  rateWord word  | elem word spam = -10  | otherwise = 2  rateWords :: [String] -> Int  rateWords words = sum (map rateWord words)  isSpam :: String -> Bool  isSpam betreff = rateWords (normalize betreff) < 0  **Globalwarming**  tempDiffs :: [Messwert] -> [TempDiff]  tempDiffs [] = []  tempDiffs (x:[]) = []  tempDiffs ((w0,t0):(w1,t1):xs) = ((w0,w1), t0-t1) : tempDiffs xs  findDiff :: Int -> [TempDiff] -> [Double]  findDiff \_ [] = []  findDiff week (((w0,w1),diff):xs)  | week >= w0 && week < w1 = [diff]  | otherwise = findDiff week xs  diffSmme :: Int -> Int -> [TempDiff] -> [Double]  diffSumme x y diffs = sumDiffs (findDiff x diffs) (findDiff y diffs)  where sumDiffs [] \_ = []  sumDiffs \_ [] = []  sumDiffs (x:\_) (y:\_) = [x+y]  **Bank**  **accountsByCustomer** c acc = filter (\a -> customer a == c) accs  **totalByCustomer** c accs = sum (map balance (accountsByCustomer c accs))  where balance (\_,\_,b) = b  **find greatest**  findGt :: Int -> [Int] -> Int  findGt \_ [] = 0  findGt e (i:is)  | i > e = i  | otherwise = findGt e is  **Filme**  mediaByAge x ms = filter (\m -> ageLimit m <= a) ms  titles x ms = map (\(t,\_,\_) -> title) (mediaByAge x ms)  **dropElemAt** 0 (a:as) = as  dropElemAt x [] = []  dropElemAt x (a:as) = a : dropElemAt (x-1) as  **startsWith** \_ [] = False  startsWith [] \_ = True  startsWith (a:as) (b:bs) | a == b = startsWith as bs  | a == ‘?’ = startsWith as bs  | otherwise = False  **split** [] = ([],[])  split (i:is) = if i >= 0 then (n, i:p) else (i:n,p)  where (n,p) = split is |