```haskell
{-------------------------------------------------
  AUSZUG aus prelude.hs

  Standardfunktionen, Standardlistenfunktionen,
  Monadische Ein-/Ausgabe
-------------------------------------------------}
fst             :: (a,b) -> a
fst (x,_)       = x
snd             :: (a,b) -> b
snd (_,y)       = y

curry           :: ((a,b) -> c) -> (a -> b -> c)
curry f x y     = f (x,y)

uncurry         :: (a -> b -> c) -> ((a,b) -> c)
uncurry f p     = f (fst p) (snd p)

id              :: a -> a
id    x         = x

(.)             :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x       = f (g x)

flip            :: (a -> b -> c) -> b -> a -> c
flip f x y      = f y x

($)             :: (a -> b) -> a -> b
f $ x           = f x

until           :: (a -> Bool) -> (a -> a) -> a -> a
until p f x     = if p x then x else until p f (f x)

-- Standard list functions {PreludeList} ------
head            :: [a] -> a
head (x:_)      = x

last            :: [a] -> a
last [x]        = x
last (_:xs)     = last xs

tail            :: [a] -> [a]
tail (_:xs)     = xs

init            :: [a] -> [a]
init [x]        = []
init (x:xs)     = x : init xs

null            :: [a] -> Bool
null []         = True
null (_:_)      = False

(++)            :: [a] -> [a] -> [a]
[]      ++ ys   = ys
(x:xs) ++ ys    = x : (xs ++ ys)

map             :: (a -> b) -> [a] -> [b]
map f xs        = [ f x | x <- xs ]

filter          :: (a -> Bool) -> [a] -> [a]
filter p xs     = [ x | x <- xs, p x ]

concat          :: [[a]] -> [a]
concat          = foldr (++) []

length          :: [a] -> Int
length          = foldl' (\n _ -> n + 1) 0

(!!)            :: [b] -> Int -> b
(x:_)  !! 0     = x
(_:xs) !! n | n>0 = xs !! (n-1)
(_:_)  !! _     = error "Prelude.!!: negative index"
[]     !! _     = error "Prelude.!!: index too large"

foldl           :: (a -> b -> a) -> a -> [b] -> a
foldl f z []    = z
foldl f z (x:xs) = foldl f (f z x) xs

foldl1          :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs

scanl           :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q xs    = q : (case xs of
      []    -> []
      x:xs -> scanl f (f q x) xs)

foldr           :: (a -> b -> b) -> b -> [a] -> b
foldr f z []    = z
foldr f z (x:xs) = f x (foldr f z xs)

foldr1          :: (a -> a -> a) -> [a] -> a
foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)

scanr           :: (a -> b -> b) -> b -> [a] -> [b]
scanr f q0 []   = [q0]
scanr f q0 (x:xs) = f x q : qs
        where qs@(q:_) = scanr f q0 xs

iterate         :: (a -> a) -> a -> [a]
iterate f x     = x : iterate f (f x)

repeat          :: a -> [a]
repeat x        = xs where xs = x:xs

replicate       :: Int -> a -> [a]
replicate n x   = take n (repeat x)

cycle           :: [a] -> [a]
cycle []        = error "Prelude.cycle: empty list"
cycle xs        = xs' where xs'=xs++xs

take            :: Int -> [a] -> [a]
take 0 _        = []
take _ []       = []
take n (x:xs) | n>0 = x : take (n-1) xs
take _ _        = error "Prelude.take: negative argument"

drop            :: Int -> [a] -> [a]
drop 0 xs       = xs
drop _ []       = []
drop n (_:xs) | n>0 = drop (n-1) xs
drop _ _        = error "Prelude.drop: negative argument"

splitAt         :: Int -> [a] -> ([a], [a])
splitAt 0 xs    = ([],xs)
splitAt _ []    = ([],[])
splitAt n (x:xs) | n>0 = (x:xs',xs'')
        where (xs',xs'') = splitAt (n-1) xs
splitAt _ _     = error "Prelude.splitAt: negative argument"

takeWhile       :: (a -> Bool) -> [a] -> [a]
takeWhile p []  = []
takeWhile p (x:xs)  | p x       = x : takeWhile p xs
                    | otherwise = []

dropWhile       :: (a -> Bool) -> [a] -> [a]
dropWhile p []  = []
dropWhile p xs@(x:xs') | p x       = dropWhile p xs'
                       | otherwise = xs

span, break     :: (a -> Bool) -> [a] -> ([a],[a])
span p []       = ([],[])
span p xs@(x:xs')  | p x       = (x:ys, zs)
                   | otherwise = ([],xs)
                    where (ys,zs) = span p xs'
break p         = span (not . p)

lines    :: String -> [String]
lines "" = []
lines s  = let (l,s') = break ('\n'==) s
              in l : case s' of []     -> []
                                (_:s'') -> lines s''
words    :: String -> [String]
words s  = case dropWhile isSpace s of
      "" -> []
      s' -> w : words s''
        where (w,s'') = break isSpace s'

unlines  :: [String] -> String
unlines  = concatMap (\l -> l ++ "\n")

unwords  :: [String] -> String
unwords [] = []
unwords ws = foldr1 (\w s -> w ++ ' ':s) ws

reverse  :: [a] -> [a]
reverse  = foldl (flip (:)) []

and, or  :: [Bool] -> Bool
and      = foldr (&&) True
or       = foldr (||) False

any, all  :: (a -> Bool) -> [a] -> Bool
any p     = or  . map p
all p     = and . map p

elem, notElem   :: Eq a => a -> [a] -> Bool
elem            = any . (==)
notElem         = all . (/=)

lookup          :: Eq a => a -> [(a,b)] -> Maybe b
lookup k []     = Nothing
lookup k ((x,y):xys) | k==x       = Just y
                     | otherwise = lookup k xys

sum, product    :: Num a => [a] -> a
sum             = foldl' (+) 0
product         = foldl' (*) 1

maximum, minimum :: Ord a => [a] -> a
maximum         = foldl1 max
minimum         = foldl1 min

concatMap       :: (a -> [b]) -> [a] -> [b]
concatMap f     = concat . map f

zip             :: [a] -> [b] -> [(a,b)]
zip             = zipWith  (\a b -> (a,b))

zipWith         :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs) = z a b : zipWith z as bs
zipWith _ _     _       = []

unzip   :: [(a,b)] -> ([a],[b])
unzip   = foldr (\(a,b) ~(as,bs) -> (a:as, b:bs)) ([], [])

-- Predefined Monadic I/O Functions ---------------
putChar :: Char ->  IO ()
getChar :: IO Char

putStr, putStrLn :: String -> IO ()
getLine :: IO String
```