OPISY WYBRANYCH FUNKCJI W JĘZYKU HASKELL

AUTOR: Rafał Kaleta, Wrocław

funkcja i jej typ	użycie	przykład
undefined a	wywołanie błędu	
(,) a → b → (a, b)	utworzenie pary z elementów	(,) ”a” 1 = (”a”, 1)
show Show a => a → String	przekształcenie do postaci drukowalnej	Show [a] show [251, 10] = ”[251, 10]”
($) (a → b) → a → b	aplikacja argumentu do funkcji (działanie o najniższym priorytecie)	show $ 13 = ”13”
(.) (b → c) → (a → b) → (a → c)	złożenie funkcji	(.) head tail = \xs -> head (tail xs)
flip (a → b → c) → (b → a → c)	zamień kolejność przyjmowania argumentów w funkcji dwuargumentowej	flip (,) = \x -> \y -> (y, x)
curry ((a, b) → c) → (a → b → c)	rozwiń funkcję	curry fst = \x -> \y -> x
uncurry (a → b → c) → ((a, b) → c)	zwiń funkcję	uncurry (++) = \(xs, ys) -> xs++ys
fst (a, b) → a	pierwszy element pary	fst (1, 7) = 1
snd (a, b) → b	drugi element pary	snd (1, 7) = 7
[ ] [a]	lista pusta	

(:) a → [a] → [a]	dodanie elementu na przód listy	(:) 5 [31, 178, 3] = [5, 31, 178, 3]
null [a] → Bool	sprawdza, czy lista jest pusta	null [”40”] = False
length [a] → Int	wylicza długość listy	length [4, 7, 16, -8, 10] = 5
head [a] → a	pierwszy element listy (głowa listy)	head [14, 93, 77] = 14
tail [a] → [a]	lista bez pierwszego elementu (ogon listy)	tail [14, 93, 77] = [93, 77]
init [a] → [a]	lista bez ostatniego elementu	init [14, 93, 77] = [14, 93]
tails [a] → [[a]]	kolejne sufiksy listy	tails [14, 93, 77] = = [[14, 93, 77], [93, 77], [77], []]
inits [a] → [[a]]	kolejne prefiksy listy	inits [14, 93, 77] = = [[], [14], [14, 93], [14, 93, 77]]
iterate (a → a) → a → [a]	począwszy od elementu wykonuj kolejno funkcję, aplikując do niej w następnym kroku uzyskany wynik	iterate reverse [7, 0] = = [[7, 0], [0, 7], [7, 0], [0, 7], ...]
repeat a → [a]	tworzy nieskończoną listę zawierającą dany element	repeat 16 = [16, 16, 16, 16, ...]
take Int → [a] → [a]	weź elementy listy do danego indeksu	take 2 [0, 1, 2, 3] = [0, 1]
drop Int → [a] → [a]	odrzuć elementy listy do danego indeksu	drop 3 [0, 1, 2, 3, 4 ,5] = [3, 4, 5]

takeWhile (a → Bool) → [a] → [a]	bierz elementy listy dopóki warunek prawdziwy	takeWhile (\x -> x<10) [4, 9, 52, 7] = = [4, 9]
dropWhile (a → Bool) → [a] → [a]	odrzucaj elementy listy dopóki warunek prawdziwy	dropWhile (\x -> x>10) [95, 4, 73] = [4, 73]
(++) [a] → [a] → [a]	złącz dwie listy	(++) [1, 2] [10, 6] = [1, 2, 10, 6]
concat [[a]] → [a]	złącz listy	concat [[1], [3, 40]] = [1, 3, 40]
map (a → b) → [a] → [b]	wykonaj funkcję na każdym elemencie listy	map fst [(17, 2), (9, 13), (34, 81)] = = [17, 9, 34]
concatMap (a → [b]) → [a] → [b]	wykonaj funkcję na każdym elemencie listy, po czym złącz wyniki funkcji	concatMap tail [[6, 1], [2, 71, 5]] = = [1, 71, 5]
reverse [a] → [a]	odwróć listę	reverse [44, 0, 19, 5] = [5, 19, 0, 44]
filter (a → Bool) → [a] → [a]	wybierz z listy elementy spełniające warunek	filter (\x -> x>10) [2, 56, 71, -8, 34] = = [56, 71, 34]
all (a → Bool) → [a] → Bool	sprawdź, czy wszystkie elementy listy spełniają warunek	all (\x -> x>10) [2, 56, 71, -8, 34] = = False
any (a → Bool) → [a] → Bool	sprawdź, czy istnieje element listy spełniający warunek	any (\x -> x>10) [2, 56, 71, -8, 34] = = True
zip [a] → [b] → [(a, b)]	utwórz listę par elementów list o tych samych indeksach aż do zakończenia jednej z list	zip [1, 3, 12] [9, 8, 74, 32] = = [(1, 9), (3, 8), (12, 74)]
unzip [(a, b)] → ([a], [b])	utwórz listy zawierające odpowiednio pierwsze oraz drugie elementy par	unzip [(1, 9), (3, 8), (12, 74)] = = ([1, 3, 12], [9, 8, 74])

zipWith (a → b → c) → [a] → [b] → [c]	utwórz listę wyników działania funkcji na elementach list o tych samych indeksach aż do zakończenia jednej z list	zipWith (*) [11, 3, 20, 14] [9, 8, 4] = = [99, 24, 80]
foldr (a → b → b) → b → [a] → b	za pomocą funkcji dołączaj kolejne elementy listy (od prawej strony) do elementu początkowego	foldr (+) 100 [1, 4, 12, 25] = 142
unfoldr (b → Maybe (a, b)) → b → [a]	za pomocą funkcji rozdziel element na parę: pierwszy element pary dołącz do listy, a drugi rozłączaj dalej aż do zwrócenia ”Nothing”	unfoldr (\x -> if x>0 then Just (x, x `div` 10) else Nothing) 12345 = = [12345, 1234, 123, 12, 1]
foldl (b → a → b) → b → [a] → b	za pomocą funkcji dołączaj kolejne elementy listy (od lewej strony) do elementu początkowego	foldl (*) 10 [2, 3, 9, 5] = 2700
scanr (a → b → b) → b → [a] → [b]	począwszy od elementu wykonuj kolejno funkcję, aplikując do niej kolejny element listy (od prawej strony) i poprzedni wynik	scanr (+) 0 [5, 8, 14] = [27, 22, 14, 0]
scanl (b → a → b) → b → [a] → [b]	począwszy od elementu wykonuj kolejno funkcję, aplikując do niej kolejny element listy (od lewej strony) i poprzedni wynik	scanl (*) 1 [2, 4, 7] = [1, 2, 8, 56]
(>>=) Monad m => m a → (a → m b) → m b	wyłuskaj wynik z monady i wykonaj przejście do kolejnego obliczenia monadowego za pomocą funkcji	Monad [a] (>>=) [[4, 76, 12], [13, 0, 3]] init = = [4, 76, 13, 0]
return Monad m => a → m a	utwórz monadę zawierającą element	Monad [a] return 45 = [45]
fail Monad m => String → m a	wywołanie błędu w obliczeniach monadowych	Monad [a] fail ”error” = [ ]
mzero MonadPlus m => m a	element neutralny łączenia monad	Monad [a] mzero = [ ]
mplus MonadPlus m => m a → m a → m a	połącz wyniki kolejnych obliczeń monadowych	Monad [a] mplus [1, 4, 67] [13, 6, 52] = = [1, 4, 67, 13, 6, 52]

IMPLEMENTACJE WYBRANYCH FUNKCJI W JĘZYKU HASKELL

AUTOR: Rafał Kaleta, Wrocław

funkcja i jej typ	implementacja
(.) (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
curry ((a, b) → c) → (a → b → c)	curry f = \x -> \y -> f (x, y)
uncurry (a → b → c) → ((a, b) → c)	uncurry f = \(x, y) -> f x y
fst (a, b) → a	fst (x, y) = x
snd (a, b) → b	snd (x, y) = y
null [a] → Bool	null [] = True null (x:xs) = False
length [a] → Int	length [] = 0 length (x:xs) = 1 + (length xs)
head [a] → a	head (x:xs) = x
tail [a] → [a]	tail (x:xs) = xs
init [a] → [a]	init [x] = [] init (x:xs) = x : (init xs)

tails [a] → [[a]]	tails [] = [[]] tails (x:xs) = (x:xs) : (tails xs)
inits [a] → [[a]]	inits [] = [[]] inits (x:xs) = [] : (map (\xs -> x:xs) (inits xs))
iterate (a → a) → a → [a]	iterate f x = (f x) : (iterate (f x))
repeat a → [a]	repeat x = x : (repeat xs)
take Int → [a] → [a]	take 0 xs = [] take n (x:xs) = x : (take (n-1) xs)
drop Int → [a] → [a]	drop 0 xs = xs take n (x:xs) = drop (n-1) xs
takeWhile (a → Bool) → [a] → [a]	takeWhile f (x:xs) = if f x then x : (takeWhile f xs) else []
dropWhile (a → Bool) → [a] → [a]	dropWhile f (x:xs) = if f x then dropWhile f xs else x:xs
(++) [a] → [a] → [a]	(++) [] ys = ys (++) (x:xs) ys = x : ((++) xs ys)
concat [[a]] → [a]	concat [] = [] concat (x:xs) = x ++ (concat xs)
map (a → b) → [a] → [b]	map f [] = [] map f (x:xs) = (f x) : (map f xs)
concatMap (a → [b]) → [a] → [b]	concatMap f [] = [] concatMap f (x:xs) = (f x) ++ (concatMap f xs)

reverse [a] → [a]	reverse [] = [] reverse (x:xs) = (reverse xs) ++ [x]
filter (a → Bool) → [a] → [a]	filter f [] = [] filter f (x:xs) = if f x then x : (filter f xs) else filter f xs
all (a → Bool) → [a] → Bool	all f [] = True all f (x:xs) = if f x then all f xs else False
any (a → Bool) → [a] → Bool	any f [] = False any f (x:xs) = if f x then True else any f xs
zip [a] → [b] → [(a, b)]	zip [] ys = [] zip xs [] = [] zip (x:xs) (y:ys) = (xs, ys) : (zip xs ys)
unzip [(a, b)] → ([a], [b])	unzip [] = ([], []) unzip (x:xs) = aux x (unzip xs) where aux (m, n) (ms, ns) = (m:ms, n:ns)
zipWith (a → b → c) → [a] → [b] → [c]	zipWith f [] ys = [] zipWith f xs [] = [] zipWith f (x:xs) (y:ys) = (f x y) : (zipWith f xs ys)
foldr (a → b → b) → b → [a] → b	foldr f e [] = e foldr f e (x:xs) = f x (foldr f e xs)
unfoldr (b → Maybe (a, b)) → b → [a]	unfoldr f n = case f n of Just (x, m) -> x : (unfoldr f m) Nothing -> []
foldl (b → a → b) → b → [a] → b	foldl f e [] = e foldl f e (x:xs) = foldl f (f e x) xs
scanr (a → b → b) → b → [a] → [b]	scanr f e [] = [e] scanr f e (x:xs) = (f x (head ys)) : ys where ys = scanr f e ys
scanl (b → a → b) → b → [a] → [b]	scanl f e [] = [e] scanl f e (x:xs) = e : (scanl f (f e x) xs)

(>>=) Monad m => m a → (a → m b) → m b	Monad [a] (>>=) [] f = [] (>>=) (x:xs) f = (f x) ++ ((>>=) xs f)
return Monad m => a → m a	Monad [a] return x = [x]
fail Monad m => String → m a	Monad [a] fail s = []
mzero MonadPlus m => m a	Monad [a] mzero = []
mplus MonadPlus m => m a → m a → m a	Monad [a] mplus xs ys = xs++ys