

Exam

1. Rewrite the following definitions into a point-free form, without using lambda-expressions or list comprehensions or enumeration:

- $f\ x\ y = 3 + y / x$
- $g\ x\ y = [y\ z \mid z \leftarrow [1..x]]$

2. Give the types of the following expressions:

- (a) $(.)(:)$
- (b) $(:)(.)$
- (c) $((.)(:)$
- (d) $((:)(:)$
- (e) $(.)(.)$

3. Explain the concept of a *spark* in Haskell. How does it relate to the following three functions

`seq, pseq, par :: a -> b -> b`

Explain what do they do.

4. The class `Functor` defines a generalization of the function `map`:

`fmap :: Functor f => (a -> b) -> f a -> f b`

Assuming that `m` is a monadic object, show how `fmap f m` can be implemented with a `do`-expression. Rewrite then your `do`-expression using `bind`.

5. Are the following two expressions:

`[(x,y) | x <- [1..10000], x == 2000, y <- [1..100], odd y]`

and

`[(x,y) | x <- [1..10000], y <- [1..100], x == 2000, odd y]`

equivalent? Motivate your answer.

6. Define a type `Tree` where all nodes of a tree, including its leaves, can keep a string value; then define a predicate

`subTree t1 t2`

returning `True` when tree `t1` is a subtree of tree `t2`.


```

{-
A list of selected functions from the Haskell modules:
Prelude
Data.List
Data.Maybe
Data.Char
-}
-----
-- standard type classes
class Show a where
  show :: a -> String

class Eq a where
  (==), (/=) :: a -> a -> Bool

class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a

class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a

class (Num a, Ord a) => Real a where
  toRational :: a -> Rational

class (Real a, Enum a) => Integral a where
  quot, rem :: a -> a -> a
  div, mod :: a -> a -> a
  toInteger :: a -> Integer

class (Num a) => Fractional a where
  (/) :: a -> a -> a
  fromRational :: Rational -> a

class (Fractional a) => Floating a where
  exp, log, sqrt :: a -> a
  sin, cos, tan :: a -> a

class (Real a, Fractional a) => RealFrac a where
  truncate, round :: (Integral b) => a -> b
  ceiling, floor :: (Integral b) => a -> b
-----
-- numerical functions

```

```

even, odd :: (Integral a) => a -> Bool
even n    = n `rem` 2 == 0
odd       = not . even

-----
-- monadic functions
sequence :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [])
  where mcons p q = do x <- p; xs <- q; return (x:xs)

sequence_ :: Monad m => [m a] -> m ()
sequence_ xs = do sequence xs; return ()

-----
-- functions on functions
id :: a -> a
id x = x

const :: a -> b -> a
const 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

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

-----
-- functions on Booleans
data Bool = False | True

(&&), (||) :: Bool -> Bool -> Bool
True && x = x
False && _ = False
True || _ = True
False || x = x
not :: Bool -> Bool
not True = False
not False = True

-----

```

```

-- functions on Maybe
data Maybe a = Nothing | Just a

isJust      :: Maybe a -> Bool
isJust (Just a) = True
isJust Nothing = False

isNothing   :: Maybe a -> Bool
isNothing = not . isJust

fromJust    :: Maybe a -> a
fromJust (Just a) = a

maybeToList :: Maybe a -> [a]
maybeToList Nothing = []
maybeToList (Just a) = [a]

listToMaybe :: [a] -> Maybe a
listToMaybe [] = Nothing
listToMaybe (a:_) = Just a

-----
-- a hidden goodie

instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)

-----
-- functions on pairs

fst :: (a, b) -> a
fst (x, y) = x

snd :: (a, b) -> b
snd (x, 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)

-----
-- functions on lists

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

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

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

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

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

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

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

init [x] = []
init (x:xs) = x : init xs

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

length :: [a] -> Int
length [] = 0
length (_:_) = 1 + length _

(!!) :: [a] -> Int -> a
(x:_) !! 0 = x
(_:xs) !! n = xs !! (n-1)

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

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

take, drop   :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs

drop n xs | n <= 0 = xs
drop _ [] = []
drop n (x:xs) = drop (n-1) xs

splitAt      :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)

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

dropWhile p [] = []
dropWhile p xs@(x:xs')
  | p x
  | otherwise = dropWhile p xs'

lines, words :: String -> [String]
-- lines "apa\nbepa\ncepa\n" == ["apa", "bepa", "cepa"]
-- words "apa bepa\n cepa" == ["apa", "bepa", "cepa"]

unlines, unwords :: [String] -> String
-- unlines ["apa", "bepa", "cepa"] == "apa\nbepa\ncepa"
-- unwords ["apa", "bepa", "cepa"] == "apa bepa cepa"

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 x    = any (== x)
notElem x = all (/= x)

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

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

maximum, minimum :: (Ord a) => [a] -> a
maximum [] = error "Prelude.maximum: empty list"
maximum xs = foldl1 max xs

minimum [] = error "Prelude.minimum: empty list"
minimum xs = foldl1 min xs

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

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)) ([],[])

nub :: [a] -> [a]
nub [] = []
nub (x:xs)
  :: (Eq a) => [a] -> [a]
  = x : nub [ y | y <- xs, x /= y ]

delete :: Eq a => a -> [a] -> [a]
delete y [] = []
delete y (x:xs)
  :: Eq a => a -> [a] -> [a]
  = if x == y then xs else x : delete y xs

(\\) :: Eq a => [a] -> [a] -> [a]
(\\) = foldl (flip delete)

union :: Eq a => [a] -> [a] -> [a]
union xs ys = xs ++ (ys \\ xs)

```

```

intersect      :: Eq a => [a] -> [a] -> [a]
intersect xs ys = [ x | x <- xs, x `elem` ys ]

intersperse   :: a -> [a] -> [a]
-- intersperse 0 [1,2,3,4] == [1,0,2,0,3,0,4]

transpose     :: [[a]] -> [[a]]
-- transpose [[1,2,3],[4,5,6]] == [[1,4],[2,5],[3,6]]

partition     :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs = (filter p xs, filter (not . p) xs)

group         :: Eq a => [a] -> [[a]]
-- group "aapabbbeee" == ["aa","p","aa","bbb","eee"]

isPrefixOf, isSuffixOf :: Eq a => [a] -> [a] -> Bool
isPrefixOf [] _       = True
isPrefixOf _ []       = False
isPrefixOf (x:xs) (y:ys) = x == y && isPrefixOf xs ys

isSuffixOf x y       = reverse x `isPrefixOf` reverse y

sort           :: (Ord a) => [a] -> [a]
sort          = foldr insert []

insert        :: (Ord a) => a -> [a] -> [a]
insert x []   = [x]
insert x (y:xs) = if x <= y then x:y:xs else y:insert x xs

-----
-- functions on Char

type String = [Char]

tolpper, toLower :: Char -> Char
-- tolpper 'a'   == 'A'
-- toLower 'Z'   == 'z'

digitToInt :: Char -> Int
-- digitToInt '8' == 8

intToDigit :: Int -> Char
-- intToDigit 3   == '3'

ord :: Char -> Int
chr  :: Int -> Char

```