Exam questions and learning resources

Functional Languages

MO

May 2020

Instructions

At the beginning of your exam, you are randomly given an item from the list below. After preparation time, you have to answer the questions and instructions in the item. The instructor may also ask questions from items other than the randomly chosen.

1. **What is a function? What is λ (lambda) calculus? Syntax of λ calculus.What is β (beta) reduction, and how does a function application evaluate in λ calculus?What is currying? What is the normal form of an expression? When do we say that an expression is reducible?**

```
-- In Haskell, Functions has one input and one output

-- add 1 2 == 3
add :: Int -> Int -> Bool
even   =   \x  ->  (x `mod` 2 == 0)


-- lambda calculas
-- indication that we are started to define a function, also known as
anonymous function
-- \a.a  \(input).(output)



-- lambda calculus syntax
-- variable              x
-- abstraction           (\a.b) x    (a -> b) (x)
-- application           f a   f(a)



-- beta reduction
-- take function and applying it to its argument
-- ((\a.a) \b.\c.b)(x)\e.f
--   = (\b.\c.b)   (x)\e.f
--   = (\c.x)    \e.f
--   = x
--  beta normal form
```

```
-- Currying
f :: a -> b -> c
-- to
f :: (a -> b) -> c   -- curried function
--   ^^^^^^^
--   one arg


-- normal form expression : x = x + 1, 1 + 1,
-- reducible expression : (\x -> x + 1)(7)  == 8
```

2.Present the syntax of Haskell functions and constants. Show step-by-step evaluation of a function application. Show how functions can be used in infix and prefix form. What is precedence and associativity of an operator? What is the difference between left and right associativity?

```
-- Present the syntax of Haskell functions and constants.
-- let x = 1  -- constant

-- f x = x
-- f :: Int -> Int


-- Show step-by-step evaluation of a function application.
length' :: [Int] -> Int
length'    []      = 0
length'    (x:xs)      = 1 + length' xs



-- Show how functions can be used in infix and prefix form.
add :: Int -> Int -> Int
-- add      x      y  = x + y     -- infix  form
add      x      y  = (+) x y      -- prefix form



-- What is precedence and associativity of an operator?
-- :info ^    precedence level 8,  associativity right
-- :info *    precedence level 7, associativity left



-- What is the difference between left and right associativity?
-- left  associative : (1 - 2) + 3 == 2
-- right associative : 1 - (2 + 3) == -4
```

3. **Define the notion of type. What are the predefined numeric types in Haskell? What is a type class? Present Num, Integral, Fractional, Eq, Ord and Show type classes.When do we say that a type has an instance of a type class?Define a new data type and make the type an instance of a type class without deriving. Present how function types are written in Haskell. What is currying in Haskell?**

```
-- Define the notion of type. What are the predefined numeric types in
Haskell?
-- predefined numeric types
-- Integer
-- Double
-- String
-- Bool



-- What is a type class?
-- it is a set of functions like Num, Ord :info Num

-- Present Num, Integral, Fractional, Eq, Ord and Show type classes.
-- :info   Num
-- :info Integral


-- When do we say that a type has an instance of a type class?


-- Define a new data type and make the type an instance of a type class
without deriving.



-- Present how function types are written in Haskell. What is currying in
Haskell?
f  :: Int -> Int -> Int
f = undefined

ff :: Num a => a -> a
ff  x = x



filter' :: (a -> Bool) -> [a] -> [a]     -- currying
--         ^^^^^^^^^^^
--        arg as a function
filter'         f         []      = []
filter'         f        (x:xs)
  | f x         = x : filter' f xs
  | otherwise   = filter' f x
```

4. Present tuple types, their construction in Haskell expressions and how a function can pattern match on tuples. Present the list type, its data constructors, and how a function can pattern match on lists. What is the difference between tuples and lists? What is a list comprehension and how does it evaluate? How is the String type defined in Haskell?

```
-- tuples pattern matching
ftuple :: (Int, String, Char) -> (String, Int, Char)
ftuple    (x, str, chr)       =  (str, x, str)


-- list pattern matching
flist :: [Int] -> [Int]
flist     xs   = xs


flist :: [a] -> Int
flist    xs    = length xs


-- a can be String, Char, Int

Tuples : can mix many types        (Int, String)
List   : can use only one type     [Int, Int]


-- list comprehension
countTo10 :: [Int]
countTo10 = [x | x <- [1..10]]


-- String can be defined as [Char] == ['d'] or String == "d"
```

5. Show how new data types can be defined in Haskell. What is a type constructor and data constructor? Present an example of new data type in Haskell. Show how functions can pattern match on values of the new data type. When do we say that a function is total or partial? Present the definition of the list type in Haskell. Make the type an instance of a type class without deriving.

```
-- let mike = Person {firstName = "Mike", age = 20}
-- mike
-- show mike

data Person = Person { firstName :: String
                     , age       :: Int
                     }  deriving (Show, Read, Eq)
```

```
type Name          = String
type PhoneNumber   = String
type PhoneBook     = [(Name, PhoneNumber)]

inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook    name      pnum        pbook      = (name, pnum) `elem` pbook


data TrafficLight = Red | Yellow | Green -- deriving (Show)

instance Show TrafficLight where
  show Red      = "Red light"
  show Yellow   = "Yellow"
  show Green    = "Green"




-- total function   : involves passing all arguments
-- partial function : involves passing less than the arguments

add' :: Int -> Int -> Int
add'     x     y    = x + y   -- total

addOne = add' 1               -- partial



-- list type
-- ff :: [Int] -> [Int]
-- ff :: [Char] -> Bool
-- ff :: String -> String
-- ff :: Num a => [a] -> Bool
-- ff :: [Bool]
```

6. What is a recursive function? Define a recursive function over lists and explain its behavior using step-by-step evaluation. What is a base case? Present the three step principle of recursively solving a problem. Show how the order of function clauses affects the behavior of the function.

```
-- recursive functions are the functions that repreat itself / apply itself


{-recursive with list-}
-- List itself is a recursive data structure
myLength :: [Int] -> Int
myLength     []   = 0
myLength    xs    = 1 + myLength xs

-- myLength [1,2] = 1 + myLength [2]
--                = 1 + 1 + myLength []    <- 0


-- base case is a non-recursive function


{-Present the three step principle of recursively solving a problem.-}

-- break down a problem to similar subproblem
-- solve sub problem recursivly
-- combine the solutions of subproblems and adjust them to solve the
original problem.


{-recursion without lists-}
-- fact 3 == 1*2*3
fact :: Int -> Int
fact     0  = 1
fact     x  = x * fact (x - 1)

-- fact 2 = 2 * fact(2-1)
--        = 2 * fact (1)
--        = 2 * 1 * fact (0) <- 1
--        = 2
```

7. Present the three categories of types in Haskell. What is polymorphism and what kinds of polymorphism are supported in Haskell? Define a function for each kind of polymorphism. What makes a function polymorphic in Haskell? Demonstrate parametricity in Haskell. Present numeric literals in Haskell and their types. What operations can be used on a value given only the type class instances of its type but not the type itself?

```
-- Types in Haskell
-- 1. Integral Type       --> Integer(limitless) --> Int(fixed precision)
-- 2. Fractional Type    --> Float, Double, Rational scintefic
-- 2. Functional Type    --> Integer -> Integer -> Integer




-- polymorphisim types
-- 1. concerete          f :: Integer -> Integer
-- 2. parametric         f :: a -> a
-- 3. constraind         f :: Num a => a -> a



-- polymorphisim
-- Polymorphic functions can take arguments and return results of multiple
types



-- Operations
-- Num == +, -, *
-- Ord == >,<, <=, >=


-- numeric literals
-- :info Num

-- typeclasses : Num, Ord, Eq, Show

-- :t +d (+)
```

8. What is a higher order function? Define a higher order function in Haskell. Define the function composition operator in Haskell together with its type and demonstrate its use in an expression. Define map and filter. What is an anonymous function? What are sections in Haskell?

```
{-
higher order functions -> functions that take functions as an arguments
comp :: (b -> a) -> (a -> b) -> a -> c


-}



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


-- map' (+1) [1,2,3]         == [2,3,4]
-- map' (\x -> x + 1) [1,2]  == [2,3]

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



filter' :: (a -> Bool) -> [a] -> [a]
filter'          f         []         = []
-- filter'           f        xs      = [y | y <- xs, f y]
filter'              f     (x:xs)
  |f x            =  x : filter' f xs
  | otherwise     = filter' f xs



{- -anonymous functions -> functions without names also called lambda}
map' (\x -> x + 1) [1,2]
```

9. What does it mean to fold a list? What is the difference between folding from left and right? Define foldr with type declaration and demonstrate its use in an example. Define a recursive function over list using foldr. Define foldl with type declaration and demonstrate its use in an example.

```
-- fold a list means replace comma, with the + sign for example
-- folds use function and default value


-- foldl --> (-) 0 [1,2,3]
-- ((0-1) - 2 ) - 3


-- foldr --> (-) 0 [1,2,3]
-- 1 - (2 - (3 - 0))


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



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







map'' :: (a -> b) -> [a] -> [b]
map''       f        []  =  []
map''       f        xs  = foldr (\x xs -> (f x):xs) [] xs




filter''' :: (a -> Bool) -> [a] -> [a]
filter'''         f         []  = []
filter'''         f       xs    = foldr (\x xs -> if f x then x:xs else xs)
[] xs
```