UNIVERSITETET I BERGEN

KANDIDAT

315

PRØVE

INF122 0 Funksjonell programmering

Emnekode	INF122
Vurderingsform	Skriftlig eksamen
Starttid	21.11.2022 14:00
Sluttid	21.11.2022 17:00
Sensurfrist	--
PDF opprettet	03.05.2024 11:00

Eksamen i INF122

Oppgave	Tittel	Oppgavetype
ℹ	Velkommen	Informasjon eller ressurser

Flervalgsoppgaver

Oppgave	Tittel	Oppgavetype
1	Verdier	Sammensatt
2	Typer og kinds	Sammensatt

Kortsvarsoppgaver

Oppgave	Tittel	Oppgavetype
3	Enkel IO	Programmering
4	Listefunksjoner	Programmering
5	product	Programmering
6	hasLength	Programmering
7	Records	Programmering

1 Verdier

a) Hva er verdien av uttrykket: sum [1..4]

10

b) Hva er verdien av uttrykket: div 10 3

3

c) Hva er verdien til uttrykket: Map.lookup "foo" (Map.fromList [("foo",4),("bar",5)])
Velg ett alternativ

- 4
- Nothing
- 5
- Just 4

d) Hva er typen til length fra standardbiblioteket?
Velg ett alternativ

- [a] -> Integer
- Int -> [a]
- [Integer]
- [a] -> Int

e) Hva er verdien til uttrykket: length [x | x <-[1..10] , odd x]
Velg ett alternativ

- 0
- [1,2,3,4,5,6,7,8,9,10]
- [1,3,5,7,9]
- 5

f) Hva er verdien til uttrykket: map head (words "rolling on floor laughing")

Husk at words :: String -> [String].

Velg ett alternativ

- 'r'
- "rolling"
- "rofl"

Maks poeng: 12

2 Typer og kinds

a) Hva er riktig type til uttrykket Just (Right "Haskell")?
Velg ett alternativ

- Either (Maybe String) a
- Maybe String
- Maybe (Either a String)
- Maybe (Right String)

b) Hva er riktig type til uttrykket (\x -> x ++ " world") ?

Velg ett alternativ

- String
- Integer -> String
- String -> String -> String
- String -> String

c) Hvilken kind har Maybe?
Velg ett alternativ

- Funktor
- Monad
- * → *
- * → * → *

d) Hvilken kind har Either?

Velg ett alternativ

- (•) * → * → *
- ( ) Functor
- ( ) Monad
- ( ) * → *

e) Hvilke av typene er riktige typinger av funksjonen nedenfor?

f x y = x == y+1

Velg de to riktige typingene

- [x] f :: (Eq a, Num a) => a -> a -> Bool
- [ ] f :: a -> a -> Bool
- [ ] f :: (Int -> Int) -> Bool
- [ ] f :: String -> String -> Bool
- [x] f :: Integer -> Integer -> Bool

f) Hvilke av typene er riktige typinger av funksjonen nedenfor?

f g x = map (g x)

Velg de to riktige typingene

- [ ] f :: (a -> a) -> [a] -> [a]
- [ ] f :: a -> b -> [a] -> [b]
- [x] f :: (a -> a -> a) -> a -> [a] -> [a]
- [x] f :: (a -> a -> b) -> a -> [a] -> [b]
- [ ] f :: (a -> b) -> [a] -> [b]

Maks poeng: 12

3 Enkel IO

I denne oppgaven skal du vise at du behersker enkel IO og do-notasjon

Oppgave: Skriv en main :: IO () verdi som spør brukeren om fornavnet sitt og svarer med å hilse brukeren med navnet skrevet baklengs. Husk å korrigere slik at navnet får stor forbokstav.

Eksempelkjøring av programmet:

Hva heter du til fornavn?
Kari
Hei, Irak!

Funksjoner du kan ha bruk for:

toUpper :: Char -> Char
toLower :: Char -> Char

head :: [a] -> a
tail :: [a] -> [a]
init :: [a] -> [a] (init fjerner siste element i en liste)
last :: [a] -> a

getLine :: IO String
putStrLn :: String -> IO ()

Svar her:

```haskell
main :: IO ()
main = do
    putStrLn "Hva heter du til fornavn?"
    name <- getLine
    let newName = changeCase $ reverseStr name
    putStrLn $ "Hei, " ++ newName ++ "!"


reverseStr String -> String
reverseStr str = foldl (\x y -> (y : x)) [] str

changeCase :: String -> String
changeCase [] = []
changeCase [c] = [toUpper c]
changeCase str = do
    let firstLetter = toUpper $ head str
    let lastLetter = toLower $ last str
    ([firstLetter] : drop 1 $ init str : [lastLetter])
```

Maks poeng: 8

4 Listefunksjoner

I denne oppgaven skal du vise at du mestrer behandling av lister i Haskell.

a) Skriv en funksjon sumOfSquares :: [Integer] -> Integer **ved hjelp av listekomprehensjon.** Funksjonen tar inn en liste med tall og regner ut summen av kvadratene av tallene i listen. For eksempel sumOfSquares [1,2,3] = $1^2 + 2^2 + 3^2 = 14$

b) Skriv samme funksjon som i a) men erstatt listekomprehensjonen med bruk av funksjonen map.

c) Skriv en funksjon duplicate :: [a] -> [a] som dupliserer hvert element i en liste. For eksempel duplicate "Hei" = "HHeeii". Velg selv hvilken fremgangsmåte du vil bruke.

d) Regn ut verdien av uttrykket: f "abcdefg" hvor f er funksjonen nedenfor. Du behøver ikke gi hele utregningen i denne deloppgaven, kun svaret.

```
f = map snd . filter (odd . fst) . zip [0..]
```

Svar på alle deloppgaver i denne kodeboksen:

```haskell
sumOfSquares :: [Integer] -> Integer

-- A)
sumOfSquares xs = sum [x * x | x <- xs]

-- B)
sumOfSquares xs = sum $ map (\x -> x * x) xs

-- C)
duplicate :: [a] -> [a]
duplicate [] = []
duplicate (x:xs) = (x : x : duplicate xs)

-- D)
"bdf"
```

Maks poeng: 14

5 product

I denne oppgaven skal du vise at du kan bruke definisjonen av en rekursivt definert funksjon til å vise at den har en gitt egenskap. Samt vise at du mestert fold.

Husk at vi definerer funksjonen product rekursivt som følger:

product :: [Integer] -> Integer
product [] = 1
product (x : xs) = x * (product xs)

a) Vis ved utregning at for hvert heltall a, b og c så holder følgende:

product [a,b,c] = a*b*c

b) Gi en alternativ definisjon av product som bruker foldr istedet for rekursjon.

Husk at foldr har typen (a -> b -> b) -> b -> [a] -> b

Svar på alle deloppgavene i denne kodeboksen:

```
1
2
3  -- A)
4   product [a,b,c] =
5   product (A : [B,C]) = A * product [B,C]
6   A * (product (B : C)) = A * ( B * product [C])
7   A * ( B * product (C : [])) = A * ( B * ( C * product []))
8   A * ( B * ( C * product [])) = A * ( B * ( C * ( 1 )))
9
10  = A * ( B * ( C * 1 ))
11  = A * ( B * C )
12  = A * B * C
13
14
15  -- B)
16  product :: [Integer] -> Integer
17  product xs = foldr (*) 1 xs
```

Maks poeng: 8

6 hasLength

I denne oppgaven skal vi se på to ulike måter å sjekke om en liste har en gitt lengde.

Husk at lengdefunksjonen, length, er definert ved rekursjon som følger:

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

Vi definerer så funksjonen:

hasLength :: Int -> [a] -> Bool
hasLength n list = length list == n

a) Hva får vi hvis vi evaluerer hasLength 2 [1,2,3]? Du behøver ikke gi utregningen, bare svaret.

b) Forklar hva som skjer dersom man prøver å evaluere (i GHCi for eksempel) uttrykket hasLength 2 [1,2..]

c) Skriv en alternativ implementasjon av hasLength slik at hasLength 2 [1,2..] evaluerer til False.
Hint: Bruk rekursjon og patternmatching.

d) Skriv en utregning som viser at for din hasLength så evaluerer hasLength 2 [1,2..] til False.

Skriv ditt svar på alle deloppgavene i denne kodeboksen:

```
1
2
3   -- A)
4   False
5
6   -- B)
7   [1,2..] er en uendelig lang liste som øker med 1.
8   når du bruker length på en uendelig lang liste vil du få en stackoverflow ( med minc
        kjøre programmet), fordi length vill aldri stoppe rekursjonen.
9
10  -- C)
11  hasLength :: Int -> [a] -> Bool
12  hasLength i [] = i == 0
13  hasLength i (x:xs) = if (i < 0)
14      then False
15      else hasLength (i-1) xs
16
17
18  -- D)
19  hasLength 2 [1,2..] = hasLength 1 [2,3..] -- (i >= 0 så vi går til else)
20  hasLength 1 [2,3..] = hasLength 0 [3,4..] -- (i >= 0 så vi går til else)
21  hasLength 0 [3,4..] = hasLength -1 [4,5..] -- (i >= 0 så vi går til else)
22  hasLength -1 [4,5..] = False -- (i < 0 så vi går til then som returnerer False)
```

Maks poeng: 10

7 Records

I denne oppgaven skal vi se på induktive datatyper, Maps og rekursjon. Her er noen funksjoner fra Data.Map som kan være nyttige:

Map.union :: Ord k => Map k a -> Map k a -> Map k aSource

O(n+m). The expression (`union t1 t2`) takes the left-biased union of `t1` and `t2`. It prefers `t1` when duplicate keys are encountered, i.e. (`union == unionWith const`). The implementation uses the efficient *hedge-union* algorithm. Hedge-union is more efficient on (bigset `union` smallset).

```
union (fromList [(5, "a"), (3, "b")])
      (fromList [(5, "A"), (7, "C")])
 == fromList [(3, "b"), (5, "a"), (7, "C")]
```

Map.unionWith :: Ord k => (a -> a -> a) -> Map k a -> Map k a -> Map k aSource
O(n+m). Union with a combining function. The implementation uses the efficient *hedge-union* algorithm.

```
unionWith (++) (fromList [(5, "a"), (3, "b")])
               (fromList [(5, "A"), (7, "C")])
   == fromList [(3, "b"), (5, "aA"), (7, "C")]
```

Noen programmeringsspråk, slik som JavaScript, har innebygget støtte for nøstede strukturer med nøkkel–verdi par. Disse kan for eksempel se slik ut:

```
norway={ pop : 5425270;
         big-cities: {oslo: {pop: 702543;
                             est: 1048}
                     bergen: {pop: 285911;
                              est: 1070}
                     trondheim: {pop: 210496;
                                 est: 997 } } }
```

I Haskell kan vi selv definere slike strukturer, for eksempel med denne induktive datastrukturen:

```
data Record key value
  = Record (Map key (Either (Record key value) value))
```

Et element i denne strukturen, en record, inneholder et map fra nøkler til enten en *underrecord* eller *en verdi*.

For eksempel kan en enkel record slik som:
x = {A: 3; B: {C: 3}}

representeres med (det litt lange) uttrykket:

```
x :: Record String Integer
x = Record
```

```
 $ Map.fromList [("A",Right 3)
                 ,("B",Left (Record (Map.singleton "C" (Right 3))))]
```

For å gjøre det litt lettere å lese disse uttrykkene kan vi lage en Show instans:

```
instance (Show key, Show value) => Show (Record key value) where
  show (Record m)
     = "{" ++ concat (intersperse ";" [ show k ++ ":"  ++ either show show v
             | (k,v) <- Map.toList m]) ++ "}"
```

```
ghci> x
{"A":3;"B":{"C":3}}
```

Vi ser på to måter å sette sammen records på: *overfladisk union* og *dyp union*.

Den overfladiske unionen tar to records og slår de sammen slik at hvis en nøkkel forekommer i en record så tas den med i unionen. Dersom en nøkkel forekommer i begge tas kun med verdien eller underrecorden fra den første.

Den dype unionen fungerer som den overfladiske, bortsett fra at dersom en nøkkel forekommer i begge og i begge tilfeller er en underrecord, så slås disse underrecordene sammen med dyp union igjen.

For å illustrere forskjellen på union og deepUnion, la x og y være to recorder:

```
x = {"A":3;"B":{"C":3}}
y = {"A":4;"B":{"D":3}}
```

så skal vi hvis vi bruker henholdsvis union og deepUnion få:

```
shallowUnion x y
 = {"A":3;"B":{"C":3}}
```

```
deepUnion x y
= {"A":3;"B":{"C":3;"D":3}}
```

a) Skriv en funksjon som beregner den overfladiske unionen av to records.

```
shallowUnion :: (Ord key)
                => Record key value
                -> Record key value
                -> Record key value
```

b) Skriv en funksjon som beregner den dype unionen av to records:

```
deepUnion :: (Ord key)
                => Record key value
                -> Record key value
                -> Record key value
```

Svar på alle deloppgavene i denne kodeboksen:

```haskell


-- A)

shallowUnion :: (Ord key)
    => Record key value
    -> Record key value
    -> Record key value
shallowUnion r1 r2 = Map.union r1 r2



-- B)

deepUnion :: (Ord key)
    => Record key value
    -> Record key value
    -> Record key value
deepUnion r1 r2 = Map.unionWith helperFunc r1 r2

helperFunc :: Either (Record key value) value -> Either (Record key value) value ->
helperFunc (Left x) (Left y) = deepUnion x y
helperFunc (Right x) (Left y) = y
helperFunc (Right x) _ = x
helperFunc (Left x) _ = x
```

Maks poeng: 6