Funktionale Programmierung
WS 2020/2021
Prof. Dr. Margarita Esponda

Zwischenklausur

Wichtige Hinweise:

1) Speichern Sie regelmäßig die Zwischenergebnisse der einzelnen Fragen und kontrollieren Sie nach der Speicherung vor allem die Korrektheit der Einrückung aller Haskell-Programme.
2) Im Antwortfeld öffnen Sie den *Rich-Text Editor*, indem Sie dafür den entsprechenden Knopf oben rechts sehen können, um ihre Antworten korrekt zu formatieren. Die Haskell-Funktionen sollen mit Hilfe von Leerzeichen korrekt eingerückt werden, sonst werden Punkte abgezogen.
3) Öffnen Sie die Klausur nur in einem Tab/Browserfenster und auf einem einzigen Gerät.
4) Wenn Sie früher fertig sind, geben Sie bitte die Zwischenklausur explizit ab, bevor Sie das Fenster zu machen, andernfalls können Sie technische Probleme bekommen.
5) Verwenden Sie die vorgegebenen Funktionsnamen und/oder Variablennamen für die Aufgaben.
6) Die Zwischenklausur ist zeitaufwendig. Bei der Korrektur wird eine relative Einstufung für die Benotung verwendet. D.h. wenn die besten Studierenden z.B. überwiegend nur 80 Prozent der Punkte erreicht haben, werden die Noten zwischen 0-80 Punkte gleichmässig verteilt.

Viel Erfolg!

1. Aufgabe (8 Punkte)

Definieren Sie eine **squares** Funktion in Haskel, die als Eingabe einer **x**, **y** Koordinate und **size** (Seitenlänge eines **n**x**n** Zeichenbild) bekommt, mit folgender Signatur:

squares :: (Int, Int, Int) -> Char

die bei Eingabe des Ausdrucks

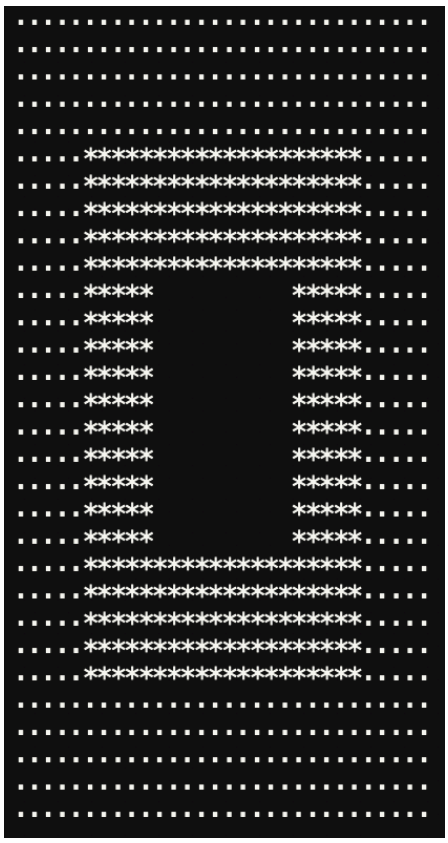
paintChars squares 30

folgendes Zeichenbild-Muster produziert:

Hier ist die **paintChars** Funktion, die die **squares** Funktion als Argument bekommt, um das Bild zu generieren und auf dem Bildschirm auszugeben:

```
paintChars f size = putStrLn (genChars f size)

genChars f size = paint size (map f [(x,y,size) | y <- [1..size], x <- [1..size]])
                  where
                  paint 0  []     = []
                  paint 0 (c:cs)  = '\n' : (paint size (c:cs))
                  paint n (c:cs)  = c: (paint (n-1) cs)
```



Lösung 1, G1:

```
square :: (Int, Int, Int) -> Char
square (x, y, size) | inSmallSquare = ' '
                    | inBigSquare   = '*'
                    | otherwise     = '.'
                          where
                          s6 = div size 6
                          inBigSquare  = x>s6 && x<=5*s6 && y>s6 && y<=5*s6
                          inSmallSquare = x>2*s6 && x<=4*s6 && y>2*s6 && y<=4*s6
```

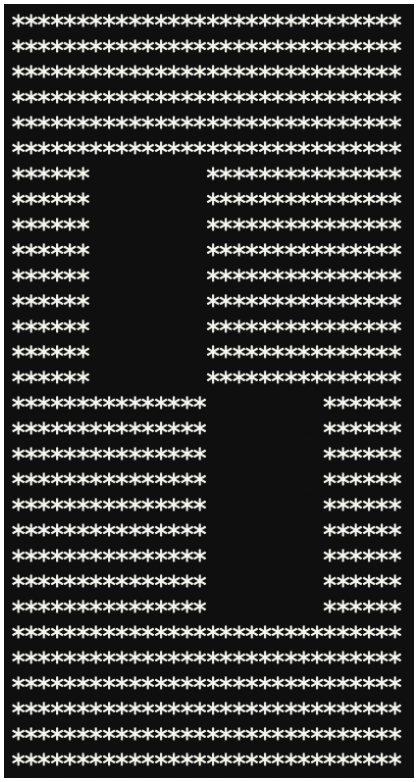
Lösung 2, G1:

```
square :: (Int, Int, Int) -> Char
square (x,y,size) | y<s || y>5*s || x<s || x>5*s = '.'
                  | y<2*s || y>4*s || x<2*s || x>4*s = '*'
                  | otherwise = ' '
                              where
                              s = div size 6
```

Lösung 3, G1:

```
squares :: (Int, Int, Int) -> Char
squares (x, y, size) | inSmallSquare = ' '
                     | inBigSquare   = '*'
                     | otherwise     = '.'
                               where
                               s6 = div size 6
                               s3 = div size 3
                               inBigSquare  = x>s6 && x<=5*s6 && y>s6 && y<=5*s6
                               inSmallSquare = x>s3 && x<=2*s3 && y>s3 && y<=2*s3
```

1. Aufgabe (8 Punkte)



Lösung G2:

```
squares :: (Int, Int, Int) -> Char
squares (x, y, size) | inLeftSquare || inRightSquare = ' '
                     | otherwise     = '*'
                          where
                           s5 = div size 5
                           s2 = div size 2
                           inLeftSquare  = x>s5 && x<=s2 && y>s5 && y<=s2
                           inRightSquare = x>s2 && x<=4*s5 && y>s2 && y<=4*s5
```

2. Aufgabe (12 Punkte)

Betrachten Sie folgende Funktionsdefinitionen:

```
natFold :: (a->a) -> a -> Integer -> a
natFold  h  c  0 = c
natFold  h  c  n = h (natFold  h  c  (n-1))

g :: Integer -> Integer -> Integer
g  n  m  =  natFold (*n) 1 m

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

Reduzieren Sie **schrittweise** folgende drei Haskell-Ausdrücke zur **Normalform**. Schreiben Sie **mindestens vier** Reduktionsschritte pro Ausdruck und begründen Sie Ihre Antwort.

a) take 4 (iterate (g 2) 1)
b) ((foldl (+) 0) . (map (g 2))) [1..4]
c) length [x | xs<-["one", "two", "three"], x<-xs]

2. Aufgabe (12 Punkte)

d) take 5 (iterate (g 1) 2)
e) ((foldl (+) 1) . (map (g 2))) [2..5]
f) length [(x,y) | y<-[1..4], x<-["one", "two", "three"]]

Lösung G1:

a) take 4 (iterate (g 2) 1)

=> take 4 (1: iterate (g 2) ((g 2) 1)))

=> take 4 (1: iterate (g 2) (natFold (*2) 1))

=> take 4 (1: iterate (g 2) ((*2) (natFold (*2) 1 0)))

=> take 4 (1: iterate (g 2) ((*2) 1))

=> take 4 (1: iterate (g 2) 2)

=> take 4 (1:2:(iterate (g 2) ((g 2) 2)))

=> take 4 (1:2:(iterate (g 2) 4))

=> take 4 (1:2:4:(iterate (g 2) ((g 2) 4)))

=> take 4 (1:2:4:16:(iterate (g 2) ((g 2) 16)))

=> [1,2,4,16]

b) ((foldl (+) 0) . (map (g 2))) [1..4]
=> (foldl (+) 0) ((map (g 2) [1..4]))
=> (foldl (+) 0) ((map (g 2) [1, 2, 3, 4]))
=> foldl (+) 0 [(g 2) 1, (g 2) 2, (g 2) 3, (g 2) 4]
=> foldl (+) 0 [2, 4, 8, 16]
=> foldl (+) ((+) 0 2) [4, 8, 16]
=> foldl (+) ((+) 2 4) [8, 16]
=> 30

c) length [x | xs<-["one", "two", "three"], x<-xs]
=> length ['o','n','e', … | xs<-["two", "three"], x<-xs]
=> length ['o','n','e', 't','w','o', … | xs<-["three"], x<-xs]
=> length ['o', 'n', 'e', 't', 'w', 'o' ,'t', 'h', 'r', 'e', 'e']
=> 11

Lösung G2:

a) take 5 (iterate (g 1) 2)
=> take 5 (2: (iterate (g 1) ((g 1) 2)))
=> take 5 (2: (1: (iterate (g 1) ((g 1) 1))))
=> take 5 (2: (1: (1: (iterate (g 1) ((g 1) 1)))))
=> take 5 (2: (1: (1: (1: (iterate (g 1) ((g 1) 1))))))
=> take 5 (2: (1: (1: (1: (1: (iterate (g 1) ((g 1) 1)))))))
=> 2 : (take 4 (1: (1: (1: (1: (iterate (g 1) ((g 1) 1)))))))
=> 2: (1: (1: (1: (1: (take 0 (iterate (g 1) ((g 1) 1)))))))
=> 2: (1: (1: (1: (1: []))))
=> [2, 1, 1, 1, 1]

b) ((foldl (+) 1) . (map (g 2))) [2..5]
=> (foldl (+) 1) ((map (g 2) [2..5])
=> (foldl (+) 1) ((map (g 2) [2, 3, 4, 5])
=> (foldl (+) 1) [(g 2) 2, (g 2) 3, (g 2) 4, (g 2) 5]
=> foldl (+) 1 [4, 8, 16, 32]
=> foldl (+) 5 [8, 16, 32]
=> foldl (+) 13 [16, 32]
=> foldl (+) 29 [32]
=> foldl (+) 61 []
=> 61

c) length [(x,y) | y<-[1..3], x<-["one", "two", „three"]]
=> length [(x,y) | y<-[1, 2, 3], x<-["one", "two", „three"]]
=> length [("one",1),("two",1),("three",1),…| y<-[2,3,4], x<-[„one","two",„three"]]
=> length [(„one",1),("two",1),("three",1),
("one",2),("two",2),("three",2), …| y<-[3], x<-[„one","two",„three"]]
=> length [(„one",1),("two",1),("three",1),
(„one",2),("two",2),("three",2), („one",3),("two",3),("three",3)]
=> 9

3. Aufgabe (10 Punkte)

Betrachten Sie folgende Funktionsdefinitionen:

```
span p [] = ([],[])
span p (x:xs) | p x         = (x:ys, zs)
              | otherwise = ([], (x:xs))
                                where (ys,zs) = span p xs

unfold p f g x | p x = []
               | otherwise = f x : unfold  p  f  g  (g x)
```

Reduzieren Sie folgende Ausdrucke schrittweise zur **Normalform**. Begründen Sie Ihre Antwort und schreiben Sie **mindestens vier** Zwischenschritte pro Ausdruck Ihrer Berechnungen auf.

a) span (/=' ') "Hi world"

b) unfold (4<) ((*2).(`mod`3)) (1+) 1

Lösung G1:

(3 P.) Schritte + Richtige Lösung

a) span (/=' ') "Hi world"

=> ('H':ys, zs) where (ys,zs) = span (/=' ') "i world"

=> ('H':ys, zs) where (ys,zs) = ('i':ys', zs') where (ys',zs') = (span (/=' ') " world"

=> ('H':ys, zs) where (ys,zs) = ('i':ys', zs') where (ys',zs') = ([], " world")

=> ('H':ys, zs) where (ys,zs) = ('i':[], " world")

=> ('H':'i':[], " world")

=> ("Hi", " world")

(2 P.) Begründung

b) unfold (4<) ((*2).(`mod`3)) (1+) 1

=> (((*2).(`mod`3)) 1) : unfold (4<) ((*2).(`mod`3)) (+1) 2

=> 2 : unfold (4<) ((*2).(`mod`3)) (1+) 2

=> 2 : (((*2).(`mod`3)) 2) : unfold (4<) ((*2).(`mod`3)) (+1) 3

=> 2 : 4 : unfold (4<) ((*2).(`mod`3)) (+1) 3

=> 2 : 4 : (((*2).(`mod`3)) 3) : unfold (4<) ((*2).(`mod`3)) (+1) 4

=> 2 : 4 : 0 : unfold (4<) ((*2).(`mod`3)) (+1) 4

=> 2 : 4 : 0 : (((*2).(`mod`3)) 4): unfold (4<) ((*2).(`mod`3)) (+1) 5

=> 2: 4: 0: 2: unfold (4<) ((*2).(`mod`3)) (+1) 5

=> 2: 4: 0: 2: []

=> [2, 4, 0, 2]

3. Aufgabe (10 Punkte)

a) span (>=2) [2, 2, 1, 0, 1]

b) unfold (5<) ((==0).(`mod` 2)) (2+) 1

Lösung G2:

(3 P.) Schritte + Richtige Lösung

a) span (>=2) [2,2,1,0,1]

=> (2:ys, zs) where (ys,zs) = span (>=2) [2,1,0,1]

=> (2:ys, zs) where (ys,zs) = (2:ys', zs') where (ys',zs') = span (>=2) [1,0,1]

=> (2:ys, zs) where (ys,zs) = (2:ys', zs') where (ys',zs') = ([], [1,0,1])

=> (2:ys, zs) where (ys,zs) = (2:[], [1,0,1])

=> (2:2:[], [1,0,1])

=> ([2,2], [1,0,1])

(2 P.) Begründung

b) unfold (5<) ((==0).(`mod` 2)) (2+) 1

=> (((==0).(`mod` 2)) 1) : unfold (5<) ((==0).(`mod` 2)) (2+) 3

=> False : unfold (5<) ((==0).(`mod` 2)) (2+) 3

=> False : (((==0).(`mod` 2)) 3): unfold (5<) ((==0).(`mod` 2)) (2+) 5

=> False : False: unfold (5<) ((==0).(`mod` 2)) (2+) 5

=> False :False: (((==0).(`mod` 2)) 5): unfold (5<) ((==0).(`mod` 2)) (2+) 5

=> False :False: False: unfold (5<) ((==0).(`mod` 2)) (2+) 5

=> False :False: False: []

=> [False, False, False]

4. Aufgabe (10 Punkte)

Schreiben Sie eine Haskell Funktion **serie**, die bei Eingabe einer Gleitkommazahl **x** und einer natürlichen Zahl **n** folgende Seriensumme von **0** bis zum **n**-Wert berechnet. Die Fakultät Funktion muss als Hilfsfunktion definiert werden.

$$serie(x,n) = \sum_{i=0}^{n} i^3 \cdot \frac{x^i}{i!}$$

Lösung G1:

```
serie :: Double -> Int -> Double
serie x 0 = 0
serie x n = (i**3)*((x**i)/(fact i)) + (serie x (i-1))
        where
        i = fromIntegral n
        fact 0 = 1
        fact n = n*(fact (n-1))
```

4. Aufgabe (10 Punkte)

$$serie(x,n) = \sum_{i=0}^{n} \frac{x^{2i}}{(2i)!}$$

Lösung G2:

```
serie' :: Double -> Int -> Double
serie' x 0 = 1
serie' x n = (x**(2*i))/(fact(2*i)) + serie' x (i-1)
          where
           i = fromIntegral n
           fact 0 = 1
           fact n = n*fact (n-1)
```

5. Aufgabe (10 Punkte)

Betrachten Sie folgende Funktionsdefinition, die eine beliebig lange Liste von Primzahlen berechnet:

```
primes = sieb [2..]
         where
           sieb (p:xs) = p:sieb[k | k<-xs, (mod k p)>0]
```

Programmieren Sie damit eine **prime3** Funktion, die die ersten **n** Primzahlen berechnet, die mit der Ziffer **3** enden.

Anwendungsbeispiel: **prime3** 6 => [11, 31, 41, 61, 71, 101]

Lösung G1:

```
prime3 :: Int -> [Integer]
prime3 n = take n [x|x<-primes, (mod x 10)==3]
```

5. Aufgabe (10 Punkte)

Betrachten Sie folgende Funktionsdefinition, die eine beliebig lange Liste von Primzahlen berechnet:

```
primes = sieb [2..]
         where
           sieb (p:xs) = p:sieb[k | k<-xs, (mod k p)>0]
```

Programmieren Sie damit eine **prime7** Funktion, die die ersten **n** Primzahlen berechnet, die mit der Ziffer **7** enden.

Anwendungsbeispiel: **prime7** 6 => [11, 31, 41, 61, 71, 101]

Lösung G2:

```
prime7 :: Int -> [Integer]
prime7 n = take n [x|x<-primes, (mod x 10)==7]
```

6. Aufgabe (10 Punkte)

Definieren Sie eine rekursive, polymorphe Funktion **applyWhile**, die als Argumente eine Funktion **f** (f :: a -> b), eine Prädikat-Funktion **p** (p :: a->Bool) und eine Liste bekommt und solange die Elemente der Liste das Prädikat **p** erfüllen, die Funktion **f** auf die Elemente der Liste anwendet und diese in der Ergebnisliste einfügt.

Anwendungsbeispiel:

applyWhile (`div` 3) (<15) [3, 6, 12, 8, 18, 3, 9] => [1,2,4,2]

Lösung G1 und G2:

```
applyWhile :: (a -> b) -> (a -> Bool) -> [a] -> [b]
applyWhile f p [] = []
applyWhile f p (x:xs) | p x = f x : applyWhile f p xs
                      | otherwise = []
```

7. Aufgabe (14 Punkte)

In einer Liste aus vergleichbaren Elementen wird ein lokales Minimum als das Element definiert, das streng kleiner ist als die Elemente, die unmittelbar davor und danach stehen.

a) Definieren Sie eine endrekursive polymorphe Funktion, die bei Eingabe einer Liste, die Liste aller lokalen Minima berechnet.

Anwendungsbeispiele:

```
listLocalMins [2, 4, 5, 1, 6, 5, 4, 3, 2, 7] => [1, 2]
listLocalMins [1.0, 2.0, 3.0, 3.0, 4.0, 5.0] => []
listLocalMins "local minimum"  =>  "a ii"
```

b) Analysieren Sie die Komplexität ihrer Funktion.

Lösung G1 und G2:

```
listLocalMins :: (Ord a) => [a] -> [a]
listLocalMins xs = listLM [] xs
                where
                  listLM lms [] = lms
                  listLM lms [x] = lms
                  listLM lms [x,y] = lms
                  listLM lms (x:y:z:xs) | x>y && y<z = listLM (lms++[y]) (y:z:xs)
                                        | otherwise = listLM lms (y:z:xs)
```

8. Aufgabe (10 Punkte)

Definieren Sie eine Funktion **sumMultiple**, die bei Eingabe von zwei natürlichen Zahlen **k** und **n**, die Summe aller Mehrfachen von **k** zwischen **1** und **n** unter Verwendung der **foldl** und **filter**-Funktionen berechnet. Sie dürfen in Ihrer Definition keine Listengeneratoren verwenden.

Anwendungsbeispiel: **sumMultiple** 3 100 => 140

Lösung G1:

```
sumMultiple :: Int -> Int -> Int
sumMultiple k n = foldl (+) 0 (filter (multiple k) [1..n])
                  where
                  multiple a b = (mod b a) == 0
```

8. Aufgabe (10 Punkte)

Definieren Sie eine Funktion **sumDivider**, die bei Eingabe einer positiven natürlichen Zahl **n**, die Summe aller Teiler von **n** unter Verwendung der **foldl** und **filter**-Funktionen berechnet. Sie dürfen in Ihrer Definition keine Listengeneratoren verwenden.

Anwendungsbeispiel: **sumDivider 11** => 12

Lösung G2:

```
sumDivider :: Int -> Int
sumDivider n = foldl (+) 0 (filter (divider n) [1..n])
               where
               divider a b = (mod a b) == 0
```

9. Aufgabe (6 Punkte)

Definieren Sie eine **nicht** curryfizierte Version der Haskell **filter-**Funktion.

Lösung G1:

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

9. Aufgabe (6 Punkte)

Definieren Sie eine **nicht** curryfizierte Version der Haskell **foldl-**Funktion.

Lösung G2:

```
foldl :: ((a -> b -> a), a,  [b])  ->  a
foldl (f, c, []) = c
foldl (f, c, (x:xs)) = foldl (f, (f v x), xs)
```

10. Aufgabe (10 Punkte)

Definieren Sie eine Funktion **compress**, die eine Liste von Bits bekommt und diese in kompakter Form zurückgibt, indem sie nebeneinander stehende gleiche Bits mit einer Zahl zusammenfasst. Definieren Sie zuerst einen algebraischen Datentyp **Bits** dafür.

Anwendungsbeispiele:

compress [One, Zero, Zero, One, One, One, One] => [1,2,4]

compress [Zero, Zero, Zero, Zero, One, One, Zero, Zero] => [4,2,2]

Lösung G1 und G2:

Lösung:

data Bit = Zero | One. deriving (Eq, Show)

1. Lösung:

```
compress :: [Bit] -> [Int]
compress bits = map snd (foldr pack' [] bits)
        where
         pack' x [] = [(x,1)]
         pack' x (y:ys) | (fst (y)) == x = (x, (snd (y)) + 1):ys
                        | otherwise      = (x,1):(y:ys)
```

2. Lösung:

```
compress :: [Bit] -> [Int]
compress [] = []
compress (x:xs) = pack' [] (x,1) xs
            where
                pack' cs (x,n) []     = cs ++ [n]
                pack' cs (x,n) (y:ys)
                             | x==y      = pack' cs (x,n+1) ys
                             | otherwise = pack' (cs++[n]) (y,1) ys
```

3. Lösung:

.....