05.12.2012

ALP I Funktionale Programmierung
WS 2012/2013
Prof. Dr. Margarita Esponda

Zwischenklausur

Name: .... **Vorname:**............................... **Matrikel-Nummer:**..................

Die maximale Punktzahl ist 100.

Aufgabe	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**Summe**	**Note**
Max. Punkte	8	10	16	16	14	18	18	100	
Punkte									

Wichtige Hinweise:

1) Schreiben Sie in allen Funktionen die entsprechende Signatur.
2) Verwenden Sie die vorgegebenen Funktionsnamen, falls diese angegeben werden.
3) Die Zwischenklausur muss geheftet bleiben.
4) Schreiben Sie Ihre Antworten in den dafür vorgegebenen freien Platz, der unmittelbar nach der Frage steht.

Viel Erfolg!

1. Aufgabe (8 Punkte)

Was ist der **Wert** und der **Datentyp** folgender Ausdrücke? Schreiben Sie mindestens einen Zwischenschritt Ihrer Berechnungen.

a) (mod 5 (-2)) - (rem 5 (-2)) **:: Integer** (1 Punkt)

=> (-1) - (1)

=> -2 (1 Punkt)

b) True && (not False || undefined) **:: Bool** (1 Punkt)

=> True && (True || undefined)

=> True && True

=> True (1 Punkt)

c) [(n,m) | n<-[1..2], m<-[3,2..0], n/=m] **:: [(Integer, Integer)]** (1 Punkt)

=> [(n,m) | n<-[1,2], m<-[3,2,1,0], n/=m]

=> [(1,3),(1,2),~~(1,1)~~,(1,0),(2,3),~~(2,2)~~,(2,1),(2,0)]

=> [(1,3),(1,2),(1,0),(2,3),(2,1),(2,0)] (1 Punkt)

d) (map (*2) . filter odd) [2,3,4,5] **:: [Integer]** (1 Punkt)

=> map (*2) (filter odd [2,3,4,5])

=> map (*2) ([3, 5])

=> [3*2, 5*2]

=> [6, 10] (1 Punkt)

2. Aufgabe (10 Punkte)

Definieren Sie eine Funktion **catalanNum**, die bei Eingabe einer natürlichen Zahl n die entsprechende Catalan-Zahl berechnet. Verwenden Sie folgende Formel für die Berechnung:

$$C_0 = 1 \quad und \quad C_{n+1} = \frac{2 \cdot C_n \cdot (2n+1)}{n+2} \quad für \quad alle \quad n \geq 1$$

1. Lösung

```
{-- einfache Rekursion --}
catalanNum :: Integer -> Integer                            (2 Punkte)
catalanNum 0 = 1                                            (1 Punkt)
catalanNum (n+1) = div (2*(catalanNum n)*(2*n+1)) (n+2)
                                                            (7 Punkte)
```

2. Lösung

```
{-- einfache Rekursion --}
catalanNum :: Integer -> Integer                            (2 Punkte)
catalanNum 0 = 1                                            (1 Punkt)
catalanNum n = div (2*(catalanNum (n-1))*(2*(n-1)+1)) (n+1)
                                                            (7 Punkte)
```

3. Lösung

```
{-- mit Endrekursion --}
catNum :: Integer -> Integer
catNum 0 = 1
catNum (n+1) = catNum'  1    1
                  where
                  catNum'  acc  k
                     | k<(n+1) = catNum' (div (2*acc*2*k) (k+1)) (k+1)
                     | otherwise  = acc
```

3. Aufgabe (16 Punkte)

Betrachten Sie folgende rekursive Funktion, die die maximale Anzahl der Teilflächen berechnet, die entstehen können, wenn ein Kreis mit **n** geraden Linien geteilt wird.

```
maxSurfaces :: Int -> Int
maxSurfaces 0 = 1
maxSurfaces n = maxSurfaces (n - 1) + n
```

a) Definieren Sie eine Funktion, die mit Hilfe einer endrekursiven Funktion genau die gleiche Berechnung realisiert.

b) Welche Vorteile hat die endrekursive Funktion gegenüber der nicht endrekursiven Lösung?

a) Lösung: (12 Punkte)

```
end_maxSurfs :: Int -> Int
end_maxSurfs n = end_maxSurfs' 0 n
                 where
                 end_maxSurfs' acc 0 = acc + 1
                 end_maxSurfs' acc n = end_maxSurfs' (acc + n) (n-1)
```

b) Lösung: (4 Punkte)

Die Ausdrucke werden nicht großer, weil die Argumente reduziert werden bevor die Berechnung wieder in die Rekursion hinein geht (Pattern-Matching). Dadurch wird zwischendurch weniger Speicherplatz verbraucht.
In beide Funktionen ist die Komplexität linear O(n), weil nur n Rekursive Aufrufe und 2*n Additionen/Subtraktionen stattfinden.

```
maxSurfaces 4 => (maxSurfaces 3) + 4
              => ((maxSurfaces 2) + 3) + 4
              => (((maxSurfaces 1)+ 2) + 3) + 4
              => ((((maxSurfaces 0) + 1)+ 2) + 3) + 4
              => (1 + 1)+ 2) + 3) + 4
              => ....

end_maxSurfs 5  => end_maxSurfs' 0 4
                => end_maxSurfs' 4 3
                => end_maxSurfs' 7 2
                => end_maxSurfs' 9 1
                => end_maxSurfs' 10 0
                => 11
```

4. Aufgabe (16 Punkte)

Definieren Sie eine rekursive, polymorphe Funktion **mapUntil**, 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 **nicht** erfüllen, die Funktion **f** auf die Elemente der Liste anwendet und diese in der Ergebnisliste einfügt.

Anwendungsbeispiel:

mapUntil (*3) (>5) [1,5,5,7,1,5] => [3,15,15]

1. Lösung:

```
mapUntil :: (a -> b) -> (a -> Bool) -> [a] -> [b]                 (2 Punkte)
mapUntil f p [] = []                                               (2 Punkte)
mapUntil f p (x:xs) | not (p x)  = f x : mapUntil f p xs           (12 Punkte)
                    | otherwise = []
```

2. Lösung:

```
mapUntil :: (a -> b) -> (a -> Bool) -> [a] -> [b]
mapUntil f p xs  = map f (takeWhile (not.p) xs)
```

5. Aufgabe (14 Punkte)

Definieren Sie eine Funktion **sumPowerTwo**, die die Summe der Quadrate aller Zahlen zwischen **1** und **n** berechnet unter Verwendung der **foldl** und **map**-Funktionen.

1. Lösung

```
sumPowerTwo :: Integer -> Integer                          (2 Punkte)
sumPowerTwo n = foldl (+) 0 (map ^2 [1..n])                (12 Punkte)
```

2. Lösung

```
sumPowerTwo :: Integer -> Integer
sumPowerTwo n = foldl (+) 0 (map g [1..n] )
                where
                g x = x*x
```

3. Lösung

```
sumPowerTwo :: Integer -> Integer
sumPowerTwo n = foldl (+) 0 (map (\a -> a*a) [1..n])
```

4. Lösung

```
sumPowerTwo :: Integer -> Integer
sumPowerTwo n = (foldl (+) 0 . map (^2)) [1..n]
```

5. Lösung

```
sumPowerTwo :: Integer -> Integer
sumPowerTwo n = let ns = [1..n]
                in (foldl (+) 0 . map (^2)) ns
```

6. Lösung

```
sumPowerTwoWhere :: Integer -> Integer
sumPowerTwoWhere n = (foldl (+) 0 . quads) n
                     where
                     quads n = map (^2) [1..n]
```

6. Aufgabe (18 Punkte)

a) Definieren Sie eine Funktion **pack**, 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:

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

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

b) Analysieren Sie die Komplexität der **pack** Funktion.

Einige Lösungen:

```
data Bit = Zero | One
                   deriving Eq
type Bits = [Bit]
```

1. Lösung:

Komplexität: **O(n)**

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

2. Lösung:

Komplexität: **O(n)**

```
pack :: [Bit] -> [Int]
pack 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)
```

3. Lösung:

Komplexität: **O(n²)**

```
pack :: [Bit] -> [Int]
pack [] = []
pack (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
```

7. Aufgabe (18 Punkte)

Betrachten Sie den folgenden algebraischen Datentyp für einfache binäre Bäume:

```
data SBTree = L | N SBTree SBTree
                                deriving Eq
```

Definieren Sie eine Funktion **completely**, die überprüft, ob eine **SBTree**-Baumstruktur ein vollständiger binärer Baum ist. Möglicherweise müssen Sie Hilfsfunktionen dafür schreiben.

1. Lösung:

```
depth :: SBTree -> Integer
depth L = 0
depth (N lt rt) = (max (depth lt) (depth rt)) + 1

completely :: SBTree -> Bool
completely L = True
completely (N lt rt) = (completely lt) && (completely rt) && depth lt == depth rt
```

2. Lösung:

```
size :: SBTree -> Integer
size L = 1
size (N lt rt) = size lt + size rt + 1

completely :: SBTree -> Bool
completely L = True
completely (N lt rt) = (completely lt) && (completely rt) && size lt == size rt
```

3. Lösung:

```
depth :: SBTree -> Integer
depth L = 0
depth (N lt rt) = (max (depth lt) (depth rt)) + 1

size :: SBTree -> Integer
size L = 1
size (N lt rt) = size lt + size rt + 1

completely :: SBTree -> Bool
completely  tree  =  (size tree) == (2^((depth tree)+1)-1)
```

4. Lösung:

```
completely :: SBTree -> Bool
completely tree = fst (balanced tree)
        where
        balanced L = (True, 1)
        balanced (N lt rt) = (bLeft && bRight && dLeft == dRight, 1+dRight)
                             where
                               (bLeft,   dLeft  ) = balanced lt
                               (bRight, dRight) = balanced rt
```