Functional programming - tutorial 1

Arnold Meijster

Dept. computer science (university of Groningen)

September 11, 2018

3.5 nAnd function

Give two different definitions of the nAnd function

```
nAnd :: Bool -> Bool -> Bool
```

which returns the result `True` except when both arguments are `True`.

3.5 nAnd function

Give two different definitions of the nAnd function

```
nAnd :: Bool -> Bool -> Bool
```

which returns the result `True` except when both arguments are `True`.

```
nAnd :: Bool -> Bool -> Bool
nAnd x y = not (x && y)
```

3.5 nAnd function

Give two different definitions of the nAnd function

```
nAnd :: Bool -> Bool -> Bool
```

which returns the result True except when both arguments are True.

```
nAnd :: Bool -> Bool -> Bool
nAnd x y = not (x && y)

nAnd2 :: Bool -> Bool -> Bool
nAnd2 False False = True
nAnd2 False True  = True
nAnd2 True False  = True
nAnd2 True True   = False
```

Give two different definitions of the nAnd function

```
nAnd :: Bool -> Bool -> Bool
```

which returns the result `True` except when both arguments are `True`.

```
nAnd :: Bool -> Bool -> Bool
nAnd x y = not (x && y)

nAnd2 :: Bool -> Bool -> Bool
nAnd2 False False = True
nAnd2 False True  = True
nAnd2 True False  = True
nAnd2 True True   = False

nAnd3 :: Bool -> Bool -> Bool
nAnd3 False y = True
nAnd3 True y = not y
```

3.8 `mystery` function

Explain the effect of the function defined here:

```
mystery :: Integer -> Integer -> Integer -> Bool
mystery m n p = not((m==n) && (n==p))
```

3.8 mystery function

Explain the effect of the function defined here:

```
mystery :: Integer -> Integer -> Integer -> Bool
mystery m n p = not((m==n) && (n==p))
```

Answer: not all numbers are the same (in other words, at least two values differ)

3.14 min and minThree functions

Give definitions of the functions `min` and `minThree` which calculate the minimum of two and three integers, respectively. [Note: `min` is a built-in function from the `Prelude`, therefore we choose the name `min2`]

```
min2:: Int -> Int -> Int
```

```
minThree :: Int -> Int -> Int -> Int
```

3.14 min and minThree functions

Give definitions of the functions `min` and `minThree` which calculate the minimum of two and three integers, respectively. [Note: `min` is a built-in function from the `Prelude`, therefore we choose the name `min2`]

```
min2:: Int -> Int -> Int

minThree :: Int -> Int -> Int -> Int

min2:: Int -> Int -> Int
min2 x y
  | x < y     = x
  | otherwise = y
```

3.14 min and minThree functions

Give definitions of the functions `min` and `minThree` which calculate the minimum of two and three integers, respectively. [Note: `min` is a built-in function from the `Prelude`, therefore we choose the name `min2`]

```
min2:: Int -> Int -> Int
```

```
minThree :: Int -> Int -> Int -> Int
```

```
min2:: Int -> Int -> Int
min2 x y
  | x < y     = x
  | otherwise = y
```

```
minThree :: Int -> Int -> Int -> Int
minThree x y z = min2 x (min2 y z)
```

3.17 `charToNum` function

Define the function `charToNum` which converts a digit like '8' to its value 8. The value of non-digits should be taken to be 0.

3.17 charToNum function

Define the function charToNum which converts a digit like '8' to its value 8. The value of non-digits should be taken to be 0.

Note that we can import Data.Char to use ord, or use the prelude function fromEnum :: Char -> Int

```
charToNum :: Char -> Int
charToNum x
  | x < '0' = 0
  | x > '9' = 0
  | otherwise = ord x - ord '0'

charToNum2 :: Char -> Int
charToNum2 x
  | x < '0' = 0
  | x > '9' = 0
  | otherwise =  fromEnum x - fromEnum '0'
```

3.22 `numberNDroots` function

Write a function `numberNDroots` that given the coefficients of the quadratic `a`, `b` and `c`, will return how many (real) roots the equation has. You may assume that `a` is non-zero.

Write a function `numberNDroots` that given the coefficients of the quadratic `a`, `b` and `c`, will return how many (real) roots the equation has. You may assume that `a` is non-zero.

```
numberNDroots :: Float -> Float -> Float -> Integer
numberNDroots a b c
  | discr < 0      = 0
  | discr == 0     = 1
  | otherwise      = 2
    where discr = b*b - 4*a*c
```

3.23 `numberRoots` function

Using your answer to the last question, write a function

```
numberRoots :: Float -> Float -> Float -> Integer
```

that given the coefficients of the quadratic `a`, `b` and `c`, will return how many (real) roots the equation has. In the case that the equation has every number a root you should return the result 3.

Using your answer to the last question, write a function

```
numberRoots :: Float -> Float -> Float -> Integer
```

that given the coefficients of the quadratic a, b and c, will return how many (real) roots the equation has. In the case that the equation has every number a root you should return the result 3.

```
numberRoots a b c
  | a /= 0     = numberNDroots a b c
  | b /= 0     = 1
  | c == 0     = 3
  | otherwise = 0
```

3.24 `smallerRoot` and `largerRoot` functions

The formula for the roots of a quadratic is

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Write definitions of the functions

```
smallerRoot :: Float -> Float -> Float -> Float
largerRoot :: Float -> Float -> Float -> Float
```

which return the smaller and larger real roots of the quadratic. In the case that the equation has no real roots or has all values as roots you should return zero as result of each of the functions.

3.24 smallerRoot and largerRoot functions

The formula for the roots of a quadratic is

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Write definitions of the functions

```
smallerRoot :: Float -> Float -> Float -> Float
largerRoot :: Float -> Float -> Float -> Float
```

which return the smaller and larger real roots of the quadratic. In the case that the equation has no real roots or has all values as roots you should return zero as result of each of the functions.

```
smallerRoot a b c
  | nr == 0 = 0
  | nr == 3 = 0
  | otherwise = (-b - sqrt(b*b - 4*a*c))/(2*a)
    where nr = numberRoots a b c
```

3.24 smallerRoot and largerRoot functions

The formula for the roots of a quadratic is

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Write definitions of the functions

```
smallerRoot :: Float -> Float -> Float -> Float
largerRoot :: Float -> Float -> Float -> Float
```

which return the smaller and larger real roots of the quadratic. In the case that the equation has no real roots or has all values as roots you should return zero as result of each of the functions.

```
smallerRoot a b c
  | nr == 0 = 0
  | nr == 3 = 0
  | otherwise = (-b - sqrt(b*b - 4*a*c))/(2*a)
    where nr = numberRoots a b c

largerRoot a b c
  | nr == 0 = 0
  | nr == 3 = 0
  | otherwise = (-b + sqrt(b*b - 4*a*c))/(2*a)
    where nr = numberRoots a b c
```

4.17 rangeProduct function

Define the function `rangeProduct` which when given natural numbers `m` and `n` returns the product `m*(m+1)*...*(n-1)*n`. The function should return 0 when `n` is smaller than `m`.

Define the function `rangeProduct` which when given natural numbers `m` and `n` returns the product `m*(m+1)*...*(n-1)*n`. The function should return 0 when `n` is smaller than `m`.

```
rangeProduct :: Integer -> Integer -> Integer
rangeProduct m n
  | n < m      = 0
  | n == m     = n
  | otherwise = m*rangeProduct (m + 1) n
```

4.17 rangeProduct function

Define the function `rangeProduct` which when given natural numbers `m` and `n` returns the product `m*(m+1)*...*(n-1)*n`. The function should return 0 when `n` is smaller than `m`.

```
rangeProduct :: Integer -> Integer -> Integer
rangeProduct m n
  | n < m     = 0
  | n == m    = n
  | otherwise = m*rangeProduct (m + 1) n

rangeProduct2 :: Integer -> Integer -> Integer
rangeProduct2 m n
  | n < m     = 0
  | otherwise = product [m..n]
```

4.18 `fac` function

As `fac` is a special case of `rangeProduct`, write a definition of `fac` which uses `rangeProduct`.

As fac is a special case of rangeProduct, write a definition of fac which uses rangeProduct.

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = rangeProduct 1 n
```

4.32 pow2 function

Suppose we have to raise 2 to the power `n`. If `n` is even, `2*m` say, then

$$2^{\mathtt{n}} = 2^{2*\mathtt{m}} = (2^{\mathtt{m}})^2$$

If `n` is odd, `2*m+1` say, then

$$2^{\mathtt{n}} = 2^{2*\mathtt{m}+1} = (2^{\mathtt{m}})^2 * 2$$

Give a recursive function to compute $2^{\mathtt{n}}$ which uses these insights.

4.32 pow2 function

Suppose we have to raise 2 to the power n. If n is even, 2*m say, then

$$2^{n} = 2^{2*m} = (2^{m})^{2}$$

If n is odd, 2*m+1 say, then

$$2^{n} = 2^{2*m+1} = (2^{m})^{2} * 2$$

Give a recursive function to compute 2^{n} which uses these insights.

```
pow2 :: Integer -> Integer
pow2 m
  | m == 0              = 1
  | m `mod` 2 == 0      = sqr (pow2 (m `div` 2))
  | otherwise           = 2*pow2 (m-1)
  where sqr n = n*n
```

5.1 maxOccurs function

Give a definition of the function

```
maxOccurs :: Integer -> Integer -> (Integer,Integer)
```

which returns the maximum of two integers, together with the number of times it occurs.

Using this, define the function

```
maxThreeOccurs :: Integer -> Integer -> Integer -> (Integer,Integer)
```

which does a similar thing for three arguments.

5.1 maxOccurs function

Give a definition of the function

```
maxOccurs :: Integer -> Integer -> (Integer,Integer)
```

which returns the maximum of two integers, together with the number of times it occurs.

Using this, define the function

```
maxThreeOccurs :: Integer -> Integer -> Integer -> (Integer,Integer)
```

which does a similar thing for three arguments.

```
maxOccurs a b
  | a == b    = (a,2)
  | a > b     = (a,1)
  | otherwise = (b,1)
```

5.1 maxOccurs function

Give a definition of the function

```
maxOccurs :: Integer -> Integer -> (Integer,Integer)
```

which returns the maximum of two integers, together with the number of times it occurs.

Using this, define the function

```
maxThreeOccurs :: Integer -> Integer -> Integer -> (Integer,Integer)
```

which does a similar thing for three arguments.

```
maxOccurs a b
  | a == b    = (a,2)
  | a > b     = (a,1)
  | otherwise = (b,1)

maxThreeOccurs a b c
  | c < m     = (m,cnt)
  | c == m    = (m,cnt+1)
  | otherwise = (c,1)
    where (m,cnt) = maxOccurs a b
```

5.18 `doubleAll` function

Give a definition of the function

```
doubleAll :: [Integer] -> [Integer]
```

which doubles all the elements of a list of integers.

5.18 doubleAll function

Give a definition of the function

```
doubleAll :: [Integer] -> [Integer]
```

which doubles all the elements of a list of integers.

```
doubleAll :: [Integer] -> [Integer]
doubleAll xs = [2*x | x <- xs]
```

5.21 `matches` function

Define the function

```
matches :: Integer -> [Integer] -> [Integer]
```

which picks out all occurrences of an integer `n` in a list. For instance,
`matches 1 [1,2,1,4,5,1]` ⇝ `[1,1,1]`
`matches 1 [2,3,4,6]` ⇝ `[]`

Next, use it to implement the function `isElementOf n xs` which returns `True` if `n` occurs in the list `xs`, and `False` otherwise.

5.21 matches function

Define the function

```
matches :: Integer -> [Integer] -> [Integer]
```

which picks out all occurrences of an integer `n` in a list. For instance,
`matches 1 [1,2,1,4,5,1]` $\rightsquigarrow$ `[1,1,1]`
`matches 1 [2,3,4,6]` $\rightsquigarrow$ `[]`

Next, use it to implement the function `isElementOf n xs` which returns `True` if `n` occurs in the list `xs`, and `False` otherwise.

```
matches :: Integer -> [Integer] -> [Integer]
matches n xs = [n | x<-xs, x==n]
```

5.21 matches function

Define the function

```
matches :: Integer -> [Integer] -> [Integer]
```

which picks out all occurrences of an integer `n` in a list. For instance,
`matches 1 [1,2,1,4,5,1]` ⇝ `[1,1,1]`
`matches 1 [2,3,4,6]` ⇝ `[]`

Next, use it to implement the function `isElementOf n xs` which returns `True` if `n` occurs in the list `xs`, and `False` otherwise.

```
matches :: Integer -> [Integer] -> [Integer]
matches n xs = [n | x<-xs, x==n]

isElementOf :: Integer -> [Integer] -> Bool
isElementOf n xs = not(null (matches n xs))
```