classmate

Date 10·9·2019

Page 1

SYNTAX EEN PHUNKSIONS

```
lucky :: (Integral a) => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you are out of luck, pal!"

sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"

factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)

charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

λ charName 'a'

"Albert"

λ charName 'b'

"Broseph"

λ charName 'h' X

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)

addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z

λ let xs = [(1, 3), (4, 3), (2, 4), (5, 3), (5, 6), (3, 1)]
λ [a+b | (a, b) <- xs]
[4, 7, 6, 8, 11, 4]

[1, 2, 3] = 1 : 2 : 3 : []   <-- use for pattern matching
x : xs  -- pattern matching head, tail.
x : y : z : zs

head' :: [a] -> a
head' [] = error "Can't call head on an empty list, dummy!"
head' (x : _) = x
```

```
λ head' [4, 5, 6]
4
λ head' "Hello"
'H'


tell :: (show a) => [a] -> String
tell [] = "The list is empty"
tell (x:[]) = "The list has one element:" ++ show x
tell (x:y:[]) = "The list has two elements :" ++ show x
                ++ " and " ++ show y
tell (x:y:_) = "The list is long. The first two elements are:"
               ++ show x ++ " and " ++ show y


length' :: (Num b) => [a] -> b
length' [] = 0
length' (_:xs) = 1 + length' xs


sum' :: (Num a) => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum'


xs@(x:y:ys)
  ↳ full thing (no need to reppeat x:y:ys)


(xs ++ ys) X
(xs ++ [x, y, z]) X
(xs ++ [x]) X
```

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

```
λ capital "Dracula"
"The first letter of Dracula is D"
```

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = "You are underweight, you emo, you!"
    | bmi <= 25.0 = "You are supposedly normal. Pfffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
```

```
otherwise = True
```

no guard matches => fall through to next pattern

```
λ bmiTell 85 1.90
"You're supposedly normal. Pfffft, I bet you're ugly!"
```

```
max' :: (Ord a) => a -> a -> a
max' a b
    | a > b = a
    | otherwise = b
```

```
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
    | a > b  = GT
    | a == b = EQ
    | otherwise = LT
```

```
λ 3 `myCompare` 2
GT

bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
  | weight / height ^ 2 == 18.5 = "You're underweight, you..."
  | weight / height ^ 2 <= 25.0 = "You're supposedly normal..."
  | weight / height ^2  <= 30.0 = "You're fat! Lose some..."
  | otherwise                   = "You're a whale,..."

bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
  | bmi <= 18.5 = "You're underweight, you..."
  | bmi <= 25.0 = "You're supposedly normal..."
  | bmi <= 30.0 = "You're fat! Lose some..."
  | otherwise   = "You're a whale,..."
  where bmi = weight / height ^2

bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
  | bmi <= skinny = "You're underweight, you..."
  | bmi <= normal = "You're supposedly normal..."
  | bmi <= fat    = "You're fat! Lose some..."
  | otherwise     = "You're a whale,..."
  where bmi = weight / height ^2
        skinny = 18.5
        normal = 25.0
        fat    = 30.0
```

align necessary for haskell

...

where bmi = weight / height ^ 2

(skinny, normal, fat) = (18.5, 25.0, 30.0)

initials :: String -> String -> String

initials firstname lastname = [f] ++ ". " ++ [l] ++ "."

where (f:_) = firstname
(l:_) = lastname

} pattern match

calcBmis :: (RealFloat a) => [(a, a)] -> [a]

calcBmis xs = [bmi w h | (w, h) <- xs]

where bmi weight height = weight / height ^ 2

where bindings can also be nested:.

let is a local expression, and it can be used anywhere

(where applies to the entire function body)

cylinder :: (RealFloat a) => a -> a -> a

cylinder r h =

also supports pattern matching

let sideArea = 2 * pi * r * h

topArea = pi * r ^ 2

in sideArea + 2 * topArea } — expression

λ [if 5 > 3 then "Woo" else "Boo",

if 'a' > 'b' then "Foo" else "Bar"]

["Woo", "Bar"]

λ 4 * (if 10 > 5 then 10 else 0) + 2

42

λ 4 * (let a = 9 in a + 1) + 2

42

λ [let square x = x * x in (square 5, square 3, square 2)]

[(25, 9, 4)]

λ (let a = 100; b = 200; c = 300 in a * b * c,
let foo = "Hey "; bar = "there!" in foo ++ bar)

(6000000, "Hey there!")

λ (let (a, b, c) = (1, 2, 3) in a + b + c) * 100

600

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]
```

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs,
               let bmi = w / h ^ 2, bmi >= 25.0]
```

can use after let and in output

λ let zoot x y z = x * y + z

λ zoot 3 9 2

29

λ let boot x y z = x * y + z in boot 3 4 2

14

λ boot x

```
head' :: [a] -> a
head' [] = error "No head for empty lists!"
head' (x:_) = x

head' :: [a] -> a
head' xs = case xs of
    []  -> error "No head for empty lists!"
    (x:_) -> x
```

} exactly similar

```
describeList :: [a] -> String
describeList xs = "The list is " ++
    case xs of
        []  -> "empty."
        [x] -> "a singleton list."
        xs  -> "a longer list."

describeList :: [a] -> String
describeList xs = "The list is " ++ what xs
  where what [] = "empty."
        what [x] = "a singleton list."
        what xs = "a longer list."
```