Haskell Function Syntax

Evan Misshula

2017-02-14

Outline

1. Pattern Matching
2. Patten matching makes thing easier
3. Projections
4. Pattern on a list
5. Pattern on a list
6. A custom length function
7. A custom sum function
8. All patterns
9. Guards
10. N-variables
11. Compare
12. Where
13. Where and constants
14. Getting initials
15. Let
16. Let in list
17. Let in a list with predicate

When defining a function

- You cand define a whole different function body based on an input

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

When defining a function

- You cand define a whole different function body based on an input

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

- We could use an if statement but let's look at a more complicated pattern

Here we avoid a complicated

- if-then-else tree
- walk through the code

```
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"
```

Pattern matching and recursion

- *Recursion* is when a function calls itself

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

Failed pattern matching

- Pattern matching without an otherwise clause can fail
- try this:

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

Without patterns

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

With patterns

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

We were able to *destructure* 2-tuples

- We used the functions 'fst' and 'snd'
- Verify these will not work 3 tuples
    - (1,2,3) or ('x','y','z')

Make our own projections with patterns

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

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

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

Pattern matching can also be done on a list

```
xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
[a+b | (a,b) <- xs]
```

If we built our own fst and snd

- Can we build our own 'head' function?
- Pattern matching on a list

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

Let's use pattern matching to build a new length function

- we have the base case of an empty list
- each non-base calls length' on a smaller list

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

Let's use pattern matching to build a new sum function

- we have the base case of an empty list
- each non-base calls length' on a smaller list
- it almost exactly like our length function expcept the value of 'x' not just 1

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

In some functions, we need to refer to both

- components and whole
- we can use an all pattern

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

If we need a case statement

- we can use a '|' which is called a guard statement

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

no single variable restriction

- We can modify the function and use both weight and height to determine what to print

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

we can make our own compare function

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

we can add a 'where' clause to eliminate boiler plate code

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

We can even make the program more expressive

- by getting rid of constants

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

Let's make a function that gets initials

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
  (l:_) = lastname
```

For more local computations

- 'let' binding may be appropriate
- can't be used across guards
- more flexible than 'where'
    - 'let' can happen in the middle of an expression

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

Here is a list comprehension using let

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

Here is a let with a predicate

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

Pattern matching is just

- case expression in function definition
- case expressions can be used throughout the function

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

Here is head with case syntax

```
head1' :: [a] -> a
head1' xs = case xs of [] -> error "No head for empty lists!"
      (x:_) -> x
```

We can put the case in a where clause

- Let's talk about the properties of a list

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