

Haskell: Let, Where, Guards

Principles of Programming Languages

Colorado School of Mines

<https://lambda.mines.edu>

Learning Group Activity

Review the LGA with your group.

- 1 Describe your implementation to your group.
- 2 Group members: how might you have implemented differently?

LGA: Euclid's GCD

The GCD of a and b is:

- a if $b = 0$
- $\gcd(b, a \bmod b)$ otherwise

More info about why this is so can be found at

https://en.wikipedia.org/wiki/Euclidean_algorithm



LGA: Euclid's GCD

The GCD of a and b is:

- a if $b = 0$
- $\gcd(b, a \bmod b)$ otherwise

More info about why this is so can be found at
https://en.wikipedia.org/wiki/Euclidean_algorithm



Implementation in Haskell

```
gcd' :: (Integral a) => a -> a -> a
gcd' a 0 = a
gcd' a b = gcd' b (a `mod` b)
```

`filter`: takes a function `f` and a list, and gives the list for which `f` returns `True` on the element:

```
GHCi> filter odd [1..10]
[1,3,5,7,9]
```

filter: takes a function `f` and a list, and gives the list for which `f` returns `True` on the element:

```
GHCi> filter odd [1..10]
[1,3,5,7,9]
```

Implementation in Haskell

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' _ [] = []
filter' f (x:xs) = if f x
                    then x : filter' f xs
                    else filter' f xs
```

LGA: Split Words

Problem: have list of word lengths and a string without spaces, separate to a list of words:

```
GHCi> splitWords [5,4,3,3] "greeneggsandham"  
["green", "eggs", "and", "ham"]
```

LGA: Split Words

Problem: have list of word lengths and a string without spaces, separate to a list of words:

```
GHCi> splitWords [5,4,3,3] "greeneggsandham"  
["green","eggs","and","ham"]
```

Implementation in Haskell

```
splitWords :: [Int] -> String -> [String]  
splitWords [] _ = []  
splitWords (x:xs) st = (take x st) : (splitWords xs  
                                     (drop x st))
```


Let Expression

let is an **expression** in Haskell to bind a variable to a value within the expression:

```
let v1 = expr1; ... in expr
```

Let Expression

let is an **expression** in Haskell to bind a variable to a value within the expression:

```
let v1 = expr1; ... in expr
```

Example

```
filter' f (x:xs) = let r = filter' f xs in  
                  if f x then x : r  
                  else r
```

Where

where is a **syntactic construct** in Haskell to bind a variable to a value:

```
expr where v1 = expr1
```

Where

where is a **syntactic construct** in Haskell to bind a variable to a value:

```
expr where v1 = expr1
```

Example

```
filter' f (x:xs) = if f x then x : r else r  
                  where r = filter' f xs
```

Where

where is a **syntactic construct** in Haskell to bind a variable to a value:

```
expr where v1 = expr1
```

Example

```
filter' f (x:xs) = if f x then x : r else r
                  where r = filter' f xs
```

Unlike `let`, `where` is *whitespace sensitive*. More on this later.

Where & Pattern Matching

One advantage of where is the ability to use pattern matching in cases:

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

Let, Where, and Functions

You can define *locally bound* functions in a `let` or `where`:

```
-- using let
```

```
doubleList :: (Num a) => [a] -> [a]
doubleList xs = let double = x * 2 in
                 map double xs
```

```
-- using where
```

```
doubleList :: (Num a) => [a] -> [a]
doubleList xs = map double xs
                 where double = x * 2
```

Case Expression

Haskell has a case *expression*:

```
case expr of  
  pattern1 -> result1  
  pattern2 -> result2  
  ....    ->    ...  
  patternN -> resultN
```


Case Expression

Haskell has a case *expression*:

```
case expr of  
  pattern1 -> result1  
  pattern2 -> result2  
  ....    ->    ...  
  patternN -> resultN
```

Example

```
take' n xs = case (n,xs) of  
  (0,-)    -> []  
  (-,[])   -> []  
  (m,y:ys) -> y : take' (m - 1) ys
```

Definition of if Expression

Haskell's if expression can be defined using case:

-- The following two expressions are equivalent

```
if cond  
  then result1  
  else result2
```

```
case cond of  
  True -> result1  
  False -> result2
```

Guards



Guards provide a convenient way to define piecewise functions:

```
func arg1 arg2 ... | cond1      = result1  
                   | cond2      = result2  
                   | ...        = ...  
                   | condN      = resultN  
                   | otherwise = result0otherwise
```

Guards: Example

```
sign :: (Ord a, Num a) => a -> String
sign n | n < 0      = "Negative"
      | n > 0      = "Positive"
      | otherwise = "Zero"
```

Guards: Example

```
sign :: (Ord a, Num a) => a -> String
sign n | n < 0      = "Negative"
      | n > 0      = "Positive"
      | otherwise  = "Zero"
```

Lining up the vertical bars is *mandatory*. For this reason, it is recommended to disable hard tabs in your text editor.

- 1 With your learning group, reimplement Euclid's GCD using guards (*no pattern matching!*)

$$\text{gcd}(a, b) = \begin{cases} a & \text{if } b = 0 \\ \text{gcd}(b, a \bmod b) & \text{otherwise} \end{cases}$$

- 2 With your learning group, reimplement the sign function from the previous slide *without using guards*.
- 3 **Discuss:** why do we have both guards and pattern matching? When might one be more expressive than another?

Guards & Where

A where can be added to the end of guards:

```
bmiScore :: (RealFloat a) => a -> String
bmiScore kg m | bmi <= 18.5 = "underweight"
               | bmi <= 25.0 = "normal"
               | bmi <= 30.0 = "overweight"
               | otherwise  = "obese"
where bmi = kg / m ^ 2
```