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?
The GCD of $a$ and $b$ is:

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

More info about why this is so can be found at https://en.wikipedia.org/wiki/Euclidean_algorithm
The GCD of $a$ and $b$ is:

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

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

Implementation in Haskell

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

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:

\[
\text{GHCi}> \text{filter odd [1..10]}
\]

\[
[1,3,5,7,9]
\]

**Implementation in Haskell**

```haskell
filter' :: (a -> Bool) -> [a] -> [a]
filter' _ [] = []
filter' f (x:xs) = if f x
                then x : filter' f xs
                else filter' f xs
```
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"]
```
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 is an **expression** in Haskell to bind a variable to a value within the expression:

```haskell
let v1 = expr1; ... in expr
```
Let is an **expression** in Haskell to bind a variable to a value within the expression:

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

**Example**

```haskell
filter' f (x:xs) = let r = filter' f xs in
                    if f x then x : r
                    else r
```
Where is a **syntatic construct** in Haskell to bind a variable to a value:

```haskell
expr where v1 = expr1
```

Example:

```haskell
filter' f (x:xs) = if f x then x:r else r where r = filter' f xs
```
where is a **syntactic construct** in Haskell to bind a variable to a value:

\[ \text{expr where } v_1 = \text{expr1} \]

**Example**

```haskell
filter' \( f \ (x:xs) = \text{if } f \ x \ \text{then } x : r \ \text{else } r \)
\text{where } r = \text{filter'} \ f \ xs
```
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.
One advantage of where is the ability to use pattern matching in cases:

```haskell
initials :: String -> String -> String
initials first last = [f] ++ ". " ++ [l] ++ "."
  where (f:_)= first
       (l:_)= last
```
You can define *locally bound* functions in a `let` or `where`:

```haskell
-- 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
```
Haskell has a case expression:

```haskell
case expr of
  pattern1 -> result1
  pattern2 -> result2
  ....      -> ...
  patternN -> resultN
```
Haskell has a case expression:

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

Example:

```haskell
take' n xs = case (n,xs) of
  (0,_)  -> []
  (_,[]) -> []
  (m,y:ys) -> y : take' (m - 1) ys
```
Haskell’s if expression can be defined using case:

```haskell
-- The following two expressions are equivalent

if cond
  then result1
  else result2

case cond of
  True -> result1
  False -> result2
```
Guards provide a convenient way to define piecewise functions:

```
func arg1 arg2 ... | cond1 = result1
| cond2 = result2
| ... = ...
| condN = resultN
| otherwise = resultOtherwise
```
Guards: Example

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

```haskell
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!*)

   \[ \text{gcd}(a, b) = \begin{cases} 
   a & \text{if } b = 0 \\ 
   \text{gcd}(b, a \mod b) & \text{otherwise} 
   \end{cases} \]

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?
A where can be added to the end of guards:

\[
\text{bmiScore :: (RealFloat a) => a -> String}
\]

\[
\text{bmiScore kg m | bmi <= 18.5 = "underweight"}
| bmi <= 25.0 = "normal"
| bmi <= 30.0 = "overweight"
| otherwise = "obese"
\]

\[
\text{where bmi = kg / m}^2
\]