Haskell: Higher Order Functions (Part II)

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu
The $ Function

Haskell has an infix function: $.

Here is how it’s defined:

\[ (\cdot \cdot) :\to (a \to b) \to a \to b \]

\[ f \cdot \cdot x = f \cdot x \]
Haskell has an infix function: \$. Here is how it’s defined:

\$(\mathbb{a} \to \mathbb{b}) \to \mathbb{a} \to \mathbb{b}

\( f \ $ x = f \ x \)

**What the heck is this worthless function?** It’s a function applicator: it takes a function on the left and an argument on the right, and applies the function to the argument.
Haskell has an infix function: $ . Here is how it’s defined:

\[(\$) :: (a \to b) \to a \to b\]

\[f \; $ \; x = f \; x\]

**What the heck is this worthless function?** It’s a function applicator: it takes a function on the left and an argument on the right, and applies the function to the argument.

**So it’s still worthless, right?** What if I told you that it has the lowest precedence and is right-associative?
Function application using spaces is left-associative and high precedence, so \( f \ a \ b \ c \) is equivalent to \(((f \ a) \ b) \ c\).

What if \( a \) and \( b \) were functions and we wanted \( f (a (b \ c)) \) instead? We had to add lots of parentheses and it gets messy fast.
Function application using spaces is left-associative and high precedence, so \( f \ a \ b \ c \) is equivalent to (((f a) b) c).

What if \( a \) and \( b \) were functions and we wanted \( f \ (a \ (b \ c)) \) instead? We had to add lots of parentheses and it gets messy fast.

Let’s use $ to fix this:

\[
\text{-- The following two expressions are equivalent}
\]
\[
\begin{align*}
  & f \ (a \ (b \ c)) \\
  & f \ $ \ a \ $ \ b \ c
\end{align*}
\]
Reducing Parentheses: More Examples

- length (filter odd [1..10])
Reducing Parentheses: More Examples

- \text{length} (\text{filter odd} \ [1..10])
- \text{length} \ $ \text{filter odd} \ [1..10]
Reducing Parentheses: More Examples

- \text{length (filter odd [1..10])}
- \text{length $ filter \ odd \ [1..10]$}

- \text{sum (map \ sqrt \ (filter \ even \ [1..100]))}
Reducing Parentheses: More Examples

- `length (filter odd [1..10])`
- `length $ filter odd [1..10]`

- `sum (map sqrt (filter even [1..100]))`
- `sum $ map sqrt $ filter even [1..100]`
Reducing Parentheses: More Examples

- length (filter odd [1..10])
- length $ filter odd [1..10]

- sum (map sqrt (filter even [1..100]))
- sum $ map sqrt $ filter even [1..100]

More examples:

- What does sqrt 3 + 4 + 5 compute?
- What does sqrt $ 3 + 4 + 5 compute?
-- This function takes a list of functions and applies
-- [1..10] to each
onCountToTen = map ($ [1..10])

-- For example:
-- onCountToTen [filter even, filter odd, map (*2)]
-- [[2,4,6,8,10],[1,3,5,7,9],[2,4,6,8,10,12,14,16,18,20]]
$: What I expect you to know

- How to **interpret** an expression which uses $ 
- How to **use** $ to reduce parentheses 
- How to **use** a partial application of $ to apply an argument to a list of functions
$: What I expect you to know

- How to **interpret** an expression which uses $$
- How to **use** $ to reduce parentheses
- How to **use** a partial application of $ to apply an argument to a list of functions

Understanding the definition of the $ function and its precedence is optional, but I think it’s helpful to figure out the above.
In mathematics, if we have a function $f(x)$ and $g(x)$, we can rewrite $f(g(x))$ as:

$$(f \circ g)(x)$$
In mathematics, if we have a function $f(x)$ and $g(x)$, we can rewrite $f(g(x))$ as:

$$(f \circ g)(x)$$

In Haskell, this $\circ$ can equivalently be written as .:

```haskell
sumOfSquares = sum . (^2)
```
Which do you choose?

-- All of these are equivalent, which would you write?
crazy x y = floor (negate (tan (sin (max x y))))
crazy x y = floor $\neg$ tan $\sin$ max x y
crazy = floor . negate . tan . sin . max
A **reduction function** is a function which takes a list and reduces the elements in the list to a single value. For example, `sum` and `product` are reduction functions:

```
GHCi> sum [1..10]
55
GHCi> product [1..10]
3628800
```
A **reduction function** is a function which takes a list and reduces the elements in the list to a single value. For example, `sum` and `product` are reduction functions:

```
GHCi> sum [1..10]
55
GHCi> product [1..10]
3628800
```

What if we had a generalized reduction function which took a function and applied it across a sequence to obtain a result? Something like this:

```
reduce(f, seq) = f(f(f(seq[0], seq[1]), seq[2]), ...)
```
Haskell has a function called `foldr`, which takes a function, an initial value, and a list, and applies the function to each element in the list, recursively calling `foldr` for the right value.

\[
\text{foldr} \ f \ z \ [] = z \\
\text{foldr} \ f \ z \ (x:xs) = f \ x \ (\text{foldr} \ f \ z \ xs)
\]
Haskell has a function called `foldl`, which takes a function, an initial value, and a list. It recurses immediately, making the new initial value the result of calling the function on the initial value and the current element.

\[
\text{foldl } f z \left[\right] = z \\
\text{foldl } f z \left(x : \text{xs}\right) = \text{foldl } f \left(f z x\right) \text{xs}
\]
Examples: Folding

-- sum using foldl
sum' = foldl (+) 0

-- sum using foldr
sum' = foldr (+) 0

-- product
product' = foldl (*) 1
Quiz Prep Time

With your new learning groups, take some time preparing for the quiz using whatever study mechanism you wish.

Topics covered:

- Pattern Matching and Recursion
- Let, Where, Case, Guards