Programming In Haskell

Week 2 - Polymorphism and Functional Programming

Last Week

- Basic types
  - Int, Integer, Double, Bool, Char, String.
- Functions
  - Function Declarations
  - Multiple Variables
  - Pattern Matching
  - Guards
  - Recursion

```
isEven :: Integer -> Bool
isEven n
  | n `mod` 2 == 0 = True
  | otherwise      = False

f :: Int -> Int -> Int -> Int
f x y z = x + y + z

intListLength :: [Integer] -> Integer
intListLength []     = 0
intListLength (x:xs) = 1 + intListLength xs
```

This Week

- Polymorphism
- Total and Partial Functions
- Recursion Patterns

Additional Syntax

- Let Expressions
  - Allows definition of local variables
  - Starts with a “let” ends with an “in”

```
strLength :: String -> Int
strLength []     = 0
strLength (_:xs) = let len_rest = strLength xs in
                   len_rest + 1
```

Additional Syntax

- Where expression
  - Also allows definition of local variables
  - Begins with “where”

```
frob :: String -> Char
frob []  = 'a'   -- len is NOT in scope here
frob str
  | len > 5   = 'x'
  | len < 3   = 'y'
  | otherwise = 'z'
  where
    len = strLength str
```

Polymorphism

- So far we've seen

```
isEven :: Integer -> Bool
isOdd :: Integer -> Bool
sumListInt :: [Int] -> Int
firstLetter :: [Char] -> Char
```

- But what about this?

```
head :: [a] -> a
```

Polymorphism

```
head :: [a] -> a
```

- The “a” represents a type variable
- In this case it represents any type without restriction
- The “a” is resolved to a type dependant on how the function is called.
- Thus head **must** be able to handle any type.

Total and Partial Function

- What happens if you give head an empty list?
- Partial Functions have inputs which cause them to crash
- Total Functions are well-defined on all possible inputs

```
tail  :: [a] -> [a] - Get  all but the first element of a list
init  :: [a] -> [a] - Get all but the last element of a list
last  :: [a] -> a - Get the last element of a list
(!!)  :: [a] -> Int -> a - Get the element at the Int passed from the list
```

Recursion Patterns

- Perform an operation on every element in the list
- Keep only some elements of the list based on a test
- “Summarize” the elements of the list somehow

Recursion Patterns - Map

- Perform an operation on every element in the list

```
addOneToAll :: [Int] -> [Int]
addOneToAll []     = []
addOneToAll (x:xs) = x + 1 : addOneToAll xs

absAll :: [Int] -> [Int]
absAll []     = []
absAll (x:xs) = abs x : absAll xs

squareAll :: [Int] -> [Int]
squareAll []     = []
squareAll (x:xs) = x^2 : squareAll xs
```

Recursion Patterns - Map

```
map :: (a -> b) -> [a] -> [b]
```

- Takes
  - Function with a type signature of a -> b
  - List of “a”s
- Returns
  - List of “b”s

Recursion Patterns - Map

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

- Allows you to “Perform an operation on every element in the list”
- Constructs a new list of each element in the list with the function “f” applied to it.

Recursion Patterns - Filter

- Keep only some elements of the list based on a test

```
keepOnlyPositive :: [Int] -> [Int]
keepOnlyPositive [] = []
keepOnlyPositive (x:xs)
  | x > 0     = x : keepOnlyPositive xs
  | otherwise = keepOnlyPositive xs

keepOnlyEven :: [Int] -> [Int]
keepOnlyEven [] = []
keepOnlyEven (x:xs)
  | even x    = x : keepOnlyEven xs
  | otherwise = keepOnlyEven xs
```

Recursion Patterns - Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

- Takes
  - Function with a type signature of a -> Bool
  - List of “a”s
- Returns
  - List of “a”s

Recursion Patterns - Filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
  | p x       = x : filter p xs
  | otherwise = filter p xs
```

- Allows you to “Keep only some elements of the list based on a test”
- Constructs a new list of elements which satisfy the condition “p”

Recursion Patterns - Fold

- “Summarize” the elements of the list somehow

```
sum' :: [Int] -> Int
sum' []     = 0
sum' (x:xs) = x + sum' xs

product' :: [Int] -> Int
product' [] = 1
product' (x:xs) = x * product' xs

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

Recursion Patterns - Fold

```
fold :: (a -> b -> b) -> b  -> [a] -> b
```

- Takes
  - Function with a type signature of a -> b -> b
  - A single element of type b
  - List of “a”s
- Returns
  - A single element of type b

Recursion Patterns - Fold

```
fold :: (a -> b -> b) -> b  -> [a] -> b
fold f z []     = z
fold f z (x:xs) = f x (fold f z xs)
```

- Allows you to ‘“Summarize” the elements of the list somehow’
- Constructs a single item based on the values in the list