Programming In Haskell Chapter 12

CS 1JC3

Let's Play Rock-Paper-Scissors!

- Suppose we wish to create a Rock-Paper-Scissors game, with an AI that's capable of implementing different Strategies
- We'll start by modeling the data we wish to work on

```
data Move = Rock | Paper | Scissors
    deriving (Show,Eq)

type Strategy = [Move] -> Move
```

- The intuition to our Strategy type is to take a list of Moves previously made by our opponent and return a counter Move

Let's Play Rock-Paper-Scissors!

Given our model, lets define some sample Strategies we could use

```
copyCat :: Strategy
copyCat []            = Rock
copyCat (latest:_) = latest

cycleS :: Strategy
cycleS moves = case (length moves) `mod` 3 of
                 0 -> Rock
                 1 -> Paper
                 2 -> Scissors

alwaysRock :: Strategy
alwaysRock _ = Rock     -- Rock Always Wins!!!
```

Let's Play Rock-Paper-Scissors

We can now define a main program for playing our game, for example

```
playGame :: Strategy -> [Move] -> IO ()
playGame strategy moves =
  do { putStr "Enter Move: ";
       inp <- getLine;
       putStrLn $ "AI Plays: " ++ (show $ strategy moves);
       case inp of
          "Rock"      -> playGame strategy (Rock:moves)
          "Paper"     -> playGame strategy (Paper:moves)
          "Scissors" -> playGame strategy (Scissors:moves)
          _           -> return () }

main :: IO ()
main = do playGame alwaysRock []
          putStrLn "Game Over!"
```

Let's Play Rock-Paper-Scissors!

To build more complicated Strategies, we can define functions that combine Strategies into new ones

```
alternate :: Strategy -> Strategy -> Strategy
alternate str1 str1 moves =
        case (length moves) `mod` 2 of
            0 -> str1 moves
            1 -> str2 moves

switchUp :: Strategy -> Strategy
switchUp str moves = case str moves of
                        Rock -> Paper
                        Paper -> Scissors
                        Scissors -> Rock

switchDown :: Strategy -> Strategy
switchDown str moves = switchUp (switchUp str) moves
```

Combinators

- The functions on the previous slide are known as combinators
- These are Higher Order Functions that can combine together endlessly to express infinite variations of computation
- For example: consider the more complicated Strategy

```
complexStrategy :: Strategy
complexStrategy =
    (switchUp copyCat) `alternate` (switchDown cycleS)
```

Exercises: Simple Recursion

Use recursion to redefine the following Prelude functions

- **drop** removes a given number of elements from the start of a list

```
drop :: Int -> [a] -> [a]
    drop 3 [1,2,3,4,5] = [4,5]
```

- **take** returns a given number of elements from the start of a list

```
take :: Int -> [a] -> [a]
    take 3 [1,2,3,4,5] = [1,2,3]
```

- **length** returns the length of a list

```
length :: [a] -> Int
    length [2,5,3] = 3
```

Solution

```haskell
drop :: Int -> [a] -> [a]
drop 0 xs       = xs
drop n (_:xs) = drop (n-1) xs

take :: Int -> [a] -> [a]
take 0 _    = []
take n (x:xs) = x : take (n-1) xs

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

Exercises: Simple Recursion

Use recursion to redefine the following Prelude functions

- **!!** returns the element at a given index **n** in a list

```
(!!) :: [a] -> Int -> a
    [1,2,3,4,5] !! 2 = 3
```

- **dropWhile** takes a boolean function and removes elements from a list while that function is satisfied

```
dropWhile :: (a -> Bool) -> [a] -> [a]
    dropWhile even [2,4,6,7,2,5] = [7,2,5]
```

- **takeWhile** takes a boolean function and takes elements from a list while that function is satisfied

```
takeWhile :: (a -> Bool) -> [a] -> [a]
    takeWhile even [2,4,6,7,2,5] = [2,4,6]
```

Solution

```
(!!) :: [a] -> Int -> a
(x:xs) !! 0 = x
(x:xs) !! n = xs !! (n-1)

dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p []  = []
dropWhile p (x:xs)
    | p x       = dropWhile p xs
    | otherwise = x:xs

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

Exercises: List Comprehensions

List Comprehensions are a way of easily generating a list. They take the form

```
[ expression | generators (x <- xs) , guards (x == x) ]
```

Some examples of lists comprehensions are

```
[ y | y <- [1..100], even y]

[ x*x | x <- [1..1000], x <= 50]

[ z | z <- [1..100],
      y <- [1..100],
      x <- [1..100], z*z = x*x + y*y]
```

Exercises: List Comprehensions

Use List Comprehensions to redefine the following

- **filter** takes a boolean function and a list and returns a list of all the elements that satisfy that function

```
filter :: (a -> Bool) -> [a] -> [a]
filter (<5) [9,2,6,3,10] = [2,3]
```

- **map** takes a function and a list and applies that function to every element in the list

```
map  :: (a -> b) -> [a] -> [b]
map (+1) [1,2,3,4,5] = [2,3,4,5,6]
```

- **unzip** takes a list of 2D tuples and returns a 2D tuples of lists

```
unzip  :: [(a,b)] -> ([a],[b])
unzip [('a',1), ('b',2), ('c',3)] = ([1,2,3],"abc")
```

Solution

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]

map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]

unzip :: [(a,b)] -> ([a],[b])
unzip xs = ([a | (a,b) <- xs],[b | (a,b) <- xs])
```

Exercises List Comprehensions

Define the following functions

- **sorted** takes a list and returns True if the list is sorted (use the zip function)

```
sorted :: (Ord a) => [a] -> Bool
sorted [1,2,3,4,5] = True
```

- **find** takes a list of 2D tuples and a **key** and returns a list of corresponding elements

```
find :: (Eq a) => a -> [(a,b)] -> [b]
find 'a' [('b',1),('a',2),('c',3),('a',4)] = [2,4]
```

- **concat** takes a list of lists and put the elements in a single list

```
concat :: [[a]] -> [a]
concat [[1,2,3],[4,5,6]] = [1,2,3,4,5,6]
```

Solution

```haskell
sorted :: (Ord a) => [a] -> Bool
sorted xs = and [a <= b | (a,b) <- zip xs (tail xs)]

find :: (Eq a) => a -> [(a,b)] -> b
find x xs = [z | (y,z) <- xs, y == x]

concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

Exercises: List Comprehensions

Define the following functions

- **position** takes a list and a value and returns the positions of that value

```
position :: (Eq a) => a -> [a] -> [Int]
position 5 [1,2,5,3,4,5] = [3,6]
```

- **perfect** takes an Integer and returns True if its a perfect number. A number is said to be perfect if it is the sum of all its divisors (excluding itself)

```
perfect :: (Integral a) => a -> Bool
    perfect 6 = True
```

  Note: $1 + 2 + 3 = 6$

Solution

```
position :: (Eq a) => a -> [a] -> [Int]
position x xs = [z | (y,z) <- zip xs [1..], y == x]

perfect :: (Integral a) => a -> Bool
perfect n = n == sum [x | x <- [1..n-1], n `mod` x == 0]
```

Exercises: Data Types

Haskell provides two main ways of defining your own types.

- The **type** decleration used for making type synonyms

```
type String = [Char]
type Pos    = (Int, Int)
type Pair a = (a,a)
```

- And the more powerful **data** decleration

```
data Bool = True | False
     deriving (Show,Eq)

data Tree = Leaf Int | Node Tree Tree Int
     deriving Show

data Tree a = Leaf a | Node (Tree a) (Tree a) a
     deriving Show
```

Exercises: Data Types

- Define a data type capable of encapsulating a ancestral tree (i.e it should be able to hold your name and the names of all your direct ancestors, and end at Unknown)
- Create a function that takes that ancestral tree type and returns a list of names of all female ancestors (i.e their mom, both grandmothers, all great-grandmothers, etc)

Solution

```haskell
data Person = Person { name   :: String
                     , mother :: Person
                     , father :: Person }
               | Unknown
    deriving Show

maternal :: Person -> [Person]
maternal (Person _ Unknown Unknown) = []
maternal (Person _ Unknown dad)     = maternal dad
maternal (Person _ mom Unknown)
         = [name mom] ++ maternal mom
maternal (Person _ mom dad)
         = [name mom] ++ maternal mom ++ maternal dad
```

More Exercises

- Define the library function elem which returns True if a given element is in a given list. For example

```
elem 5 [1,2,3] = False
elem 'a' "abcd" = True
```

- Define a function remove that removes a given element from a list. For example

```
remove 5 [2,5,3,5] = [2,3]
remove 'a' "bcd" = "bcd"
```

- Define a function push that pushes the first element to the back while it is bigger than the next. For example

```
push [4,1,2,3,6] = [1,2,3,4,6]
push "bca"       = "bca"
```

Solutions

```
elem :: (Eq a) => a -> [a] -> Bool
elem y []       = False
elem y (x:xs) = y == x || elem y xs

remove :: (Eq a) => a -> [a] -> [a]
remove y []      = []
remove y (x:xs)
    | y == x     = remove xs
    | otherwise = x : remove y xs

push :: (Ord a) => [a] -> [a]
push (x:y:xs)
  | x > y       = y : push (x:xs)
  | otherwise = x:y:xs
push x          = x
```

And More Exercises

- Define the library function iterate that takes a function and an initial value and returns an infinite list of iterations.

```
iterate (+1) 1 = [1,2,3,4,5,.....]
iterate reverse "abc" = ["abc","cba","abc",....]
```

- Define the library function div that preforms integer division

```
5 `div` 2 = 2
9 `div` 3 = 3
```

- Define a function distinct that returns True if all the elements in a list are distinct (no two elements are the same).

```
distinct [1,2,3,4] = True
distinct "pop"     = False
```

Solutions

```haskell
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

div  :: (Integral a) => a -> a -> a
div x y = if x-y >= 0
            then 1 + div (x-y) y
            else 0

distinct :: (Eq a) => [a] -> Bool
distinct []     = True
distinct (x:xs) = and (map (/=x) xs) && distinct xs
```

One More!!!

- Define a function cut that removes removes repeats of elements from a list. For example

```
cut [1,2,1,4] = [1,2,4]
cut "abc"     = "abc"
```

One More!!!

- Define a function cut that removes removes repeats of elements from a list. For example

```
cut [1,2,1,4] = [1,2,4]
cut "abc"     = "abc"
```

- Solution:

```
cut :: (Eq a) => [a] -> [a]
cut []     = []
cut (x:xs) = x : cut [y | y <- xs, y /= x]
```