Chapter 6 Exercises

Recursive Functions

Connor Baker

Compiled on December 21, 2019 at 9:03pm

6.8 Exercises

1. How does the recursive version of the factorial function behave if applied to a negative argument, like -1? Modify the definition to prohibit negative arguments by adding a guard to the recursive case.

   Solution:

```
facRecursive :: Int -> Int
facRecursive n
    | n < 0      = 0
    | n == 0     = 1
    | otherwise = n * facRecursive (n - 1)
```

2. Define a recursive function `sumdown :: Int -> Int` that returns the sum of the non-negative integers from a given value down to zero. For example, `sumdown 3` should return the result $3 + 2 + 1 + 0 = 6$.

   Solution:

```
sumdown :: Int -> Int
sumdown 0 = 0
sumdown n = n + sumdown (n - 1)
```

3. Define the exponentiation operator `^` for non-negative integers using the same pattern of recursion as the multiplication operator `*` and show how the expression `2 ^ 3` is evaluated using your definition.

   Solution:

```
(^) :: Int -> Int -> Int
0 ^ _ = 0
_ ^ 0 = 1
n ^ m = n * (n ^ (m - 1))
```

4. Define a recursive function `euclid :: Int -> Int -> Int` that implements Euclid's algorithm for calculating the greatest common divisor of two non-negative integers: if the two numbers are equal, this number is the result; otherwise, the smaller number is subtracted from the larger, and the same process is then repeated. For example:

```
> euclid 6 27
3
```

   Solution:

```
euclid :: Int -> Int -> Int
euclid n m
    | n == m = n
    | n >  m = euclid m (n - m)
    | n <  m = euclid n (m - n)
```

5. Using the recursive definitions given in this chapter, show how `length [1,2,3]`, `drop 3 [1,2,3,4,5]`, and `init [1,2,3]` are evaluated.

   For length:

```
length [1,2,3]
={ applying length }
1 + length [2,3]
={ applying length }
1 + (1 + length [3])
={ applying length }
1 + (1 + (1 + length []))
={ applying length }
1 + (1 + (1 + (0)))
={ applying + }
3
```

For drop:

```
drop 3 [1,2,3,4,5]
={ applying drop }
drop 2 [2,3,4,5]
={ applying drop }
drop 1 [3,4,5]
={ applying drop }
drop 0 [3,4,5]
=[3,4,5]
```

For init (the definition is provided below for reference):

```
init :: [a] -> [a]
init [_]    = []
init (x:xs) = x : init xs

init [1,2,3]
={ applying init }
1 : init [2,3]
={ applying init }
1 : 2 : init [3]
={ applying init }
1 : 2 : []
={ list notation }
[1,2]
```

6. Without looking at the definitions from the standard prelude, define the following library functions on lists using recursion:

   - Decide if all logical values in a list are true:
     - `and :: [Bool] -> Bool`
   - Concatenate a list of lists:
     - `concat :: [[a]] -> [a]`
   - Produce a list with n identical elements:
     - `replicate :: Int -> a -> [a]`
   - Select the n^{th} element of a list:

    - `(!!):: [a] -> Int -> a`

- Decide if a value is an element of a list:
    - `elem :: Eq a => a -> [a] -> Bool`

Note: most of these functions are defined in the prelude using other library functions rather than using explicit recursion, and are generic functions rather than being specific to the type of lists.

Solution:

```
-- Part a
and :: [Bool] -> Bool
and []          = True
and (False:_) = False
and (_:xs)      = and xs

-- Part b
concat :: [[a]] -> [a]
concat []        = []
concat (xs:xss) = xs ++ concat xss

-- Part c
replicate :: Int -> a -> [a]
replicate 0 _ = []
replicate n a = [a] ++ replicate (n - 1) a

-- Part d
-- For simplicity, assume that the index is valid
-- and that the array is non-empty
(!!) :: [a] -> Int -> a
(x:_)  !! 0 = x
(_:xs) !! n = xs !! (n - 1)

-- Part e
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
-- Note: ys can be [] which is why (y:ys) matches singletons
elem x (y:ys)
    | x == y = True
    | x /= y = elem x ys
```

7. Define a recursive function `merge :: Ord a => [a] -> [a] -> [a]` that merges two sorted lists to give a single sorted list.

   For example:

```
> merge [2,5,6] [1,3,4]
[1,2,3,4,5,6]
```

   Note: your definition should not use other functions on sorted lists such as insert or isort, but should be defined using explicit recursion.

   Solution:

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] y'      = y'
merge x' []      = x'
merge x'@(x:xs) y'@(y:ys)
    | (x <= y)  = x : merge xs y'
    | otherwise = y : merge x' ys
```

8. Using merge, define a function `msort :: Ord a => [a] -> [a]` that implements merge sort, in which the empty list and singleton lists are already sorted, and any other list is sorted by merging together the two lists that result from sorting the two halves of the list separately. Hint: first define a function `halve :: [a] -> ↪ ([a],[a])` that splits a list into two halves whose lengths differ by at most one.

   Solution:

```
halve :: [a] -> ([a],[a])
halve xs = splitAt (div (length xs) 2) xs

msort :: Ord a => [a] -> [a]
msort [] = []
msort [x] = [x]
msort xs = merge (msort a) (msort b)
    where a = fst $ halve xs
          b = snd $ halve xs
```

9. Using the five-step process, construct the library function that:

   - Calculates the sum of a list of numbers;
   - takes a given number of elements from the start of a list;
   - Selects the last element of a non-empty list.

   Solution:

```
-- Part a
sumOfList :: Num a => [a] -> a
sumOfList []     = 0
sumOfList (x:xs) = x + (sumOfList xs)

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

-- Part c
lastElem :: [a] -> a
lastElem [x]    = x
lastElem (_:xs) = lastElem xs
lastElem []     = error "List is empty!"
```