Vivian Do
CSC 4330
Sep 25 2023

Homework 3

1. (20 points: 5 points for each of "length" and "init", and 10 points for "drop").

Using the recursive definitions given on the slides of chapter 6, show how "length [1,2,3]", "drop 3 [1,2,3,4,5]", and "init [1,2,3]" are evaluated. Here, you need to write out (i.e., show) each step that Haskell takes to evaluate the corresponding three expressions, like on slides 4, 8, etc., of the slides of chapter 6.

Note: that the recursive definitions of "length" and "drop" are on slides 9 and 14 of the slides of chapter 6, respectively, while the recursive definition of "init" is:

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

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

drop 3 [1, 2, 3, 4, 5]
= drop 2 [2, 3, 4, 5]
= drop 1 [3, 4, 5]
= drop 0 [4, 5]
= [4, 5]

init [1, 2, 3]
= 1 : init [2, 3]
= 2 : init [3]
= 3 : []
= [1, 2]
```

2. (20 points: 4 points for each of a.--e.).

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

a. Decide if all logical values in a list are "True":

```
and :: [Bool] -> Bool
```

b. Concatenate a list of lists:

```
concat :: [[a]] -> [a]
```

c. Produce a list with "n" identical elements:

```
replicate :: Int -> a -> [a]
```

d. Select the "n"th element of a list:

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

e. 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.

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

b)
```
concat :: [[a]] -> [a]
concat [] = []
concat (x : xs) = x ++ concat xs
```

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

d)
```
(!!) :: [a] -> Int -> a
(x : xs) !! 0 = x
( _ : xs) !! n = xs !! (n - 1)
_ !! _ = error “Index out of bounds”
```

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

3. (20 points).

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.

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

- if first list is empty, return second list; if second list is empty, return first list
- if “x” is smaller or equal to “y”, put “x” in result & print the rest of “xs”
- if “y” is smaller, put “y” in result & print the rest of “ys”

4. (20 points).

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.

```
msort :: Ord a > [a] -> [a]
msort [] = []                       -- base
msort [x] = [x]                     -- base
msort xs = merge (msort left) (msort right)
        where
                (left, right) = halve xs

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

- takes list “xs”, implements into merge sort algorithm, recursively sorts left/right halves & returns merged list
- “halve” splits list into two halves

5.

Construct recursive versions of the library functions that:

a. calculate the "sum" of a list of numbers (5 points);
b. "take" a given number of elements from the beginning of a list (10 points);
c. select the "last" element of a non-empty list (5 points).

a)
```
sum :: Num a => [a] -> a
sum [] = 0                          -- base
sum (x:xs) = x + sum xs
```

b)
```
take :: Int -> [a] -> [a]
take n _
    | n <= 0 = []                   -- base
take _ [] = []                      -- base
take n (x:xs) = x : take (n - 1) xs
```

c)
```
last :: [a] -> a
last [x] = x                        -- base
last ( _:xs) = last xs
```