University of Toronto Scarborough

CSCC24 Winter 2018 Midterm Test

Duration - 1 hour 30 minutes
Aid: 1 crib sheet (letter size, double-sided).

Last Name: POON

First Name: KEEGAN

Student Number: 1002423727

UTORid: poonkeeg

1:	7 /10
2:	8/10
3:	15/20
4:	10/10
5:	20 /20
Total:	60/70

1. [10 marks] The *takeWhile* function takes two parameters: a predicate p, a list lst. It "takes elements from the list as long as the predicate holds"; formally, it returns the longest prefix of lst such that every element in the returned list satisfies the predicate. Example:

$$\begin{aligned} &\mathit{takeWhile}\ \mathit{even}\ [2, 6, 4, 1, 4, 8] \\ = &\ [2, 6, 4] \end{aligned}$$

(a) [3 marks] Implement your own version in Scheme. Use your own recursion. Avoid using other list functions.

2

```
(define (take-while p lst)
  (match lst
    ['()          '()  ]
    [(cons hd tl)  (if (p hd) (cons hd (take-while p tl))
                              (take-while p tl)) ]
                                                      ]))
```

(b) [3 marks] Implement your own version in Haskell. Use your own recursion. Avoid using other list functions.

2

```
takeWhile p [] = []

takeWhile p (hd : tl) | p hd = hd : (takeWhile p tl)
                      | otherwise = takeWhile p tl
```

(c) [4 marks] Implement as a *foldr*. Do not use your own recursion. Write in Scheme or Haskell.

3

```
(define (take-while p lst)

  (foldr                                         lst))
```

OR Assuming foldr :: (a->b->b) -> b -> [a] -> b

p :: a -> Bool

```
takeWhile p lst =

  foldr binop [] lst
  where binop y x | p y  y : x
                  | otherwise  x
```

8

2. [10 marks] Standard lists in Haskell and Scheme take $\Theta(n)$ time to compute lengths. Someone then comes up with the idea of a user-defined list type that stores lengths at list nodes, so that asking for lengths take $O(1)$ time. In Haskell and Scheme:

```
data MyList a = End | More Int a (MyList a)
(struct More (len head tail))
; Still use '() for the empty list in the Scheme version.
```

Example in Haskell: `More 3 x (More 2 y (More 1 z End))`
Example in Scheme: `(More 3 x (More 2 y (More 1 z '())))`

Clearly, we would not require users to use `More` directly. We would provide a decent API such as:

(a) [5 marks] *mycons* adds an element to the beginning of a list. Implement in both Haskell and Scheme.

5

```
mycons :: a -> MyList a -> MyList a
mycons a End = More 1 a (End)
mycons a lst@(More n hd tl) = More (n+1) a lst

(define (mycons a lst)
  (match lst
    ['()              (More 1 a `())                    ]
    [(More n hd tl)   (More (+ n 1) a lst)]))
```

(b) [5 marks] *myappend* concatenates two lists. Use your own recursion. Maximize re-use of *mycons*. Implement in both Haskell and Scheme.

3

```
myappend :: MyList a -> MyList a -> MyList a
myappend End lst2 = lst2
myappend (More n hd tl) lst2 = mycons hd (myappend tl lst2)

(define (myappend lst1 lst2)
  (match lst1
    ['()                                    ]
    [(More n hd tl)
                                            ]))
```

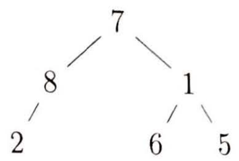
?

3. [20 marks] A type of binary trees (not necessarily binary search trees) that store elements at internal nodes has been defined in Haskell and Scheme:

```
data BT a = Nil | Node (BT a) a (BT a)
(struct nil ())
(struct node (left val right))
```

Example in code and in picture:

```
exBT = Node (Node (Node Nil 2 Nil) 8 Nil)
            7
            (Node (Node Nil 6 Nil) 1 (Node Nil 5 Nil))
(define exBT
  (node (node (node (nil) 2 (nil)) 8 (nil))
        7
        (node (node (nil) 6 (nil)) 1 (node (nil) 5 (nil)))))
```



(a) [5 marks] Implement the *btFoldl* function. It takes 3 parameters: a binary operator, a value, a binary tree. Its return value is as though you applied list *foldl* to the in-order traversal. **Important**: This only specifies external correctness, not internal implementation strategy. Example:

$$btFoldl\ (-)\ 9\ exBT$$
$$= (((((9-2)-8)-7)-6)-1)-5$$

(btfoldl leftchild , val)

binop

btFoldl binop init rchild

The function type in Haskell would be
`(b -> a -> b) -> b -> BT a -> b`

5

Your implementation shall do it directly with recursion and avoid intermediate data structures and mutable variables. You may choose Scheme or Haskell.

~~btFoldl binop init~~

btFoldl _ init NIL _ = init

btFoldl binop init (Node lchild val rchild) = btFoldl binop (binop (btfoldl binop init lchild) val) rchild

(b) [5 marks] Implement the *btFoldr* function. It takes 3 parameters: a binary operator, a value, a binary tree. Its return value is as though you applied list *foldr* to the in-order traversal. **Important**: This only specifies external correctness, not internal implementation strategy. Example:

$$btFoldr\ (-)\ 9\ exBT$$
$$= 2 - (8 - (7 - (6 - (1 - (5 - 9)))))$$

The function type in Haskell would be
`(a -> b -> b) -> b -> BT a -> b`

1

Your implementation shall do it directly with recursion and avoid intermediate data structures and mutable variables. You may choose Scheme or Haskell.

```
btFoldr _ init NIL _ = init
btFoldr binop init (Node lchild val rchild) = btFoldr binop leftside rchild
    where leftside = binop val (btFoldr binop init lchild)
```

X

(c) [5 marks] Use *btFoldr* to convert a binary tree to the list of its elements in in-order. You may choose Scheme or Haskell. Avoid writing more than minimum code.

4

```
(define (toList tree) (btFoldr                tree))
```

OR

```
toList :: BT a -> [a]
toList tree = btFoldr (\x y -> x:y) [] tree
```

(d) [5 marks] Use *btFoldl* to count the number of elements in a binary tree. You may choose Scheme or Haskell. Avoid writing more than minimum code.

5

```
(define (size tree)
  (btFoldl                                   tree))
```

OR

(b->a->b) ->b->BT a ->b

```
size :: BT a -> Integer
size tree = btFoldl (\x y -> x+1)  0          tree
```

4. [10 marks] Implement in Haskell this function

```
mwalk :: [Maybe a] -> Maybe [a]
```

If the input list contains one or more `Nothing`, the answer is `Nothing`; otherwise, the answer factors out `Just` from the input list, e.g.,

$$mwalk\,[Just\,1, Just\,2] = Just\,[1, 2]$$

5

(a) [5 marks] Use your own recursion and pattern matching on both list and Maybe. In the empty list case, do the one thing that makes the recursive case the simplest.

```
mwalk []  = Just []
mwalk (Nothing : _) = Nothing
mwalk (Just a : tl) = case (mwalk tl) of
        Nothing -> Nothing
        Justlist -> Just (a : list)
```

5

(b) [5 marks] Use your own recursion and pattern matching on list. But not pattern matching on Maybe—recall Maybe is an instance of Applicative with:

```
fmap f Nothing  = Nothing
fmap f (Just a) = Just (f a)

pure a              = Just a
Just f <*> Just a   = Just (f a)
_      <*> _        = Nothing
```

So use `fmap`, `pure`, and `<*>` instead.

```
mwalk [] = Just []
mwalk (hd : tl) =
    fmap (\x -> \y -> x : y) hd <*> mwalk tl
```

(This means `mwalk` is generalizable from Maybe to all Applicative instances:
`Applicative f => [f a] -> f [a]`
)

5. [20 marks] In this question, a row vector is represented by a non-empty list of numbers, and a matrix is represented by a non-empty list of row vectors. Example:

$$\begin{pmatrix} 4 & 1 & 6 \\ 9 & 0 & 5 \\ 6 & 4 & 7 \end{pmatrix} \text{ is represented as } [[4, 1, 6], [9, 0, 5], [6, 4, 7]]$$

In this question, it will be most useful to recall:

- In Scheme, *map* can apply a function to the elements of a list:
  `(map abs '(-1 -2))` = `'(1 2)`
  and can apply a binary operator to the respective elements of two lists:
  `(map + '(1 2) '(10 20))` = `'(11 22)`

  Summing a list can be done by `(apply + lst)`.

- In Haskell, the corresponding examples become:
  `map abs [-1, -2]`
  `zipWith (+) [1,2] [10,20]`

  Summing a list can be done by `sum lst`.

(a) [4 marks] Implement the function *absMat* that applies *abs* to every number in a matrix, e.g.,

$$absMat\ [[-1, -2], [-4, -3]] = [[1, 2], [4, 3]]$$

Use the least code; do not write your own recursion. Use Haskell.

4 absMat = map (map abs) mat

mat

(b) [4 marks] The dot product of two row vectors is defined as

$$dot\ [a_1, \ldots, a_k]\ [b_1, \ldots, b_k] = a_1 \times b_1 + \cdots + a_k \times b_k$$

Implement this function with the least code; do not write your own recursion. You may assume that the two lists have the same length. You may use Scheme or Haskell.

dot a b = sum (zipWith (x) a b)

4

$\begin{bmatrix} 9 & 0 & 5 \\ 6 & 4 & 7 \end{bmatrix}^T = \begin{bmatrix} 9 & 6 \\ 0 & 4 \\ 5 & 7 \end{bmatrix}$

(c) [6 marks] The transpose of a matrix switches the roles between rows and columns. Formally, given a matrix M, the ith row of M becomes the ith column of the transpose of M. Examples:

$\begin{bmatrix} 4 & 1 & 6 \\ 9 & 0 & 5 \\ 6 & 4 & 7 \end{bmatrix}^T = \begin{bmatrix} 4 & 9 & 6 \\ 1 & 0 & 4 \\ 6 & 5 & 7 \end{bmatrix}$

6

$$
\begin{aligned}
\mathit{transpose}\,[[9,0,5],[6,4,7]] &= [[9,6],[0,4],[5,7]] \\
\mathit{transpose}\,[[4,1,6],[9,0,5],[6,4,7]] &= [[4,9,6],[1,0,4],[6,5,7]] \\
&= [4:[9,6], 1:[0,4], 6:[5,7]]
\end{aligned}
$$

That last equation looks really like applying (:) as a binary operator to the respective elements of two lists: [4, 1, 6] and [[9, 6], [0, 4], [5, 7]]. Wait, those two lists look familiar...

Implement this function. Do not assume square matrices. You may assume that all rows have the same length. You may use Scheme or Haskell.

```
(define (transpose matrix)
  (match matrix
   [(cons row1 '())
                                                    ]
   [(cons row1 more)
     (map cons
                                                    ]))
```

OR

```
transpose (row1 : []) = map (\x-> x:[]) row1          3

transpose (row1 : more) =
    zipWith (:) row1 (transpose more)                 3
```

map dot A   A·B   A T   ~~zipWith dot A (transpose B~~)

(d) [6 marks] Matrix multiplication of two matrices A and B is defined as:

$$[[\mathit{dot}\,(A\text{'s row }1)\,(T\text{'s row }1), \ldots, \mathit{dot}\,(A\text{'s row }1)\,(T\text{'s row }n)]$$
$$\ldots,$$
$$[\mathit{dot}\,(A\text{'s row }m)\,(T\text{'s row }1), \ldots, \mathit{dot}\,(A\text{'s row }m)\,(T\text{'s row }n)]$$

6

where $T = \mathit{transpose}\ B$.

The Prof has implemented this in a super-slick Haskell one-liner that doesn't need its own recursion but just takes advantage of *map*, *dot*, *transpose*, and a lambda. The Prof is about to present it in a lecture, but then...

Our most esteemed guest the much awaited Code Mangler finally enters! He deletes all parentheses, sorts the words on the RHS, and if a word occurs twice on the RHS, he deletes the second occurrence. (Fortunately, every word on the RHS appears at most twice in the Prof's correct code.) The code is mangled to:

```
mul matA matB = -> \ arowi dot map matA matB transpose
```

Help the Prof restore the correct one-liner. Remember: Some words should occur twice on the RHS, and you have to put back parentheses.

```
mul matA matB = map (\row -> map (dot row) (transpose matB)) matA
```

(End of questions.)

Blank page for sketch work.

[dot row 1 , dot row2 . . .]

\x ->

map(\row -> map (row `dot`) ~~$~~ (transpose mat B)) matA

$$\begin{bmatrix} \text{map (dot A's row 1) (transpose mat B)} \\ \text{map (dot A's row2) (transpose mat B)} \\ \\ \\ \end{bmatrix}$$