Solutions

CMPT 383 Midterm 1, Summer 2019

Last name *exactly as it appears on student card*									
First name *exactly as it appears on student card*									
SFU Student #									
SFU email *ends with* `sfu.ca`									

This is a **closed book exam**: notes, books, computers, calculators, electronic devices, etc. are **not permitted**. Do not speak to any other students during their exam or look at their work. If you have a question, please remain seated and raise your hand and a proctor will come to you.

	Out of	***Your Mark***
Deep Functions	10	
Folding	10	
Short Answer: Scheme	10	
Currying and Continuations	10	
Haskell	5	
Total	45	

Deep Functions

(10 marks) Write a "deep" function called `(deep-a2b x)` that works as follows:

- If `x` is the symbol `'a`, then `'b` is returned.
- If `x` is a non-list other than `'a`, then `x` is returned.
- If `x` is a list, it returns a new list that is the same as `x`, except all occurrences of the symbol `'a` (even ones deeply nested within sub-lists) have been replaced with the symbol `'b`. The structure of `x` is not changed.

For example:

```
> (deep-a2b 'a)
b

> (deep-a2b 'm)
m

> (deep-a2b '(a 4 (b a c)))
(b 4 (b b c))

> (deep-a2b '((a) (1 (((a)) cat) 2)))
((b) (1 (((b)) cat) 2))
```

Use only basic Scheme functions that are part of standard Scheme and were discussed in the lectures and notes (no loops!).

Make sure to use good Scheme programming style and perfect indentation: make your code easy to read.

```
(define (deep-a2b x)
  (cond ((equal? x 'a)
            'b)
        ((not (list? x))
            x)
        (else
            (map deep-a2b x)
        )
  )
)
```

Readability is especially important in Scheme, and so your answer should use proper indentation and carefully matched brackets.

Folding

a) (5 marks) Give an implementation of the **right fold function**. Call it `foldr` and use only recursion and basic Scheme code in your solution.

```
(define foldr
    (lambda (fn empty-val lst)
        (cond
            ((null? lst)    ;; base case
                empty-val)
             (else          ;; recursive case
               (fn (car lst) (foldr fn empty-val (cdr lst)))
             )
        )
    )
)
```

b) (5 marks) Write a version of Scheme's `map` function called `mymap` in terms of the `foldr` function above. It should use only basic Scheme functions (besides `foldr`) and should *not* use recursion.

```
(define mymap
    (lambda (fn lst)
        (foldr (lambda (a rest) (cons (fn a) rest))
               ()
               lst)
    )
)
```

Short Answer: Scheme

a) (3 marks) In Scheme, implement the `compose` function. It takes two single-input functions as input and returns a new function that is their composition.	`(define (compose f g)`
	`    (lambda (x)`
	`        (f (g x))))`
	
	`;; or`
	
	`(define compose`
	`   (lambda (f g)`
	`      (lambda (x)`
	`         (f (g x)))))`

b) (2 marks) What does this expression evaluate to? `((compose cdr car) '((a b c) (1 2 3)))`	`(b c)`
c) (2 marks) What does this expression evaluate to? `((compose car cdr) '((a b c) (1 2 3)))`	`(1 2 3)`
d) (3 marks) In each of the three boxes on the right, write a different Scheme expression that, when typed into the MIT Scheme interpreter, will make it print exactly this on the screen: `(append (a (b)) (1 2))` Of course, MIT Scheme includes extra information like ";Value 13:" before the expression, but ignore that for this question.	`(list 'append '(a (b)) '(1 2))`
	`(cons 'append '((a (b)) (1 2)))`

	`(cons 'append (list '(a (b))`
	`                     '(1 2)`
	`              )`
	`)`
	Many other expressions are possible!

Currying and Continuations

a) (3 marks) Write a definition for a curried version of the `cons` function called c-cons. **Don't** use any pre-defined currying function (such as `curry2`).

```
(define c-cons
    (lambda (x)
        (lambda (lst)
            (cons x lst)
        )
    )
)
```

b) (3 marks) Suppose `f` is any curried function that takes two inputs. Write a general-purpose function called `(uncurry2 f)` that returns an uncurried version of `f`.

```
(define uncurry2
    (lambda (f)
        (lambda (x y)
            ((f x) y)
        )
    )
)
```

c) (2 marks) Write a continuation-passing style (CPS) version of the `cons` function named `cons-c`.

```
(define (cons-c x lst k) (k (cons x lst)))
```

d) (2 marks) What is the main application of continuation-passing style?

It was designed to be used in Scheme compilers as an intermediate language between Scheme and machine code.

Haskell

(5 marks) Write a function (including its signature) called `count` that takes one `Char` and one `String` as input and returns the number of times the character appears in the string. For example:

```
> count 'a' "tuna"
1

> count 'b' "tuna"
0

> count 'p' "pepper"
3

> count 'e' ""
0
```

To get full marks, *don't* use recursion in your solution.

```
-- There are many ways to write this function. Here are a few …

count1 :: Char -> String -> Int
count1 c s = length $ filter (==c) s

count2 :: Char -> String -> Int
count2 c s = length (filter (==c) s)

count3 :: Char -> String -> Int
count3 c s = length (filter (\a -> a == c) s)

count4 :: Char -> String -> Int
count4 c s = sum (map toNum s)
             where toNum a = if a == c then 1 else 0

-- filtering, but using a list comprehension
count5 :: Char -> String -> Int
count5 c s = length [a | a <- s, a == c]

-- another list comprehension; similar idea to count4
count6 :: Char -> String -> Int
count6 c s = sum [1 | a <- s, a == c]

count7 :: Char -> String -> Int
count7 c s = foldr (\a accum -> if a == c then accum + 1 else accum) 0 s
```