HW5

1/9 due

Turn in your code for the red-starred (sub)problems.

1 [Scheme's runtime stack]

a) Given

(define f (let ((x 0)) (lambda () (set! x (+ x 1)) x)))

Draw the run-time stack during the evaluation of

(+ (f) (f))

Highlight the portion of the runtime stack that becomes garbage after the evaluation. (5%)

b) Repeat a), but this time uses the following definition (5%)

(define f (lambda () (let ((x 0)) (set! x (+ x 1)) x)))

2 [Scheme's runtime stack]

Given the Scheme functions

(define t (lambda (f) (lambda (x) (f (f x)))))

(define s (lambda (x) (* x x)))

Draw the run-time stack during the evaluation of

((t s) 2)

Highlight the portion of the runtime stack that becomes garbage after the evaluation. (10%)

3 [Call by name vs call by need]

In Haskell, the infinite data structure that represents the infinite sequence 1, 2, 3, … is easily defined by

ints = 1 : map (1+) ints

We may simulate this infinite data structure in Scheme, as follows.

```
(define-syntax cons-stream
   (syntax-rules ()
      ((cons-stream x y) (cons (delay x) (delay y)))))
(define head (lambda (s) (force (car s))))
(define tail (lambda (s) (force (cdr s))))
(define map-stream
   (lambda (f s) (cons-stream (f (head s)) (map-stream f (tail s)))))
(define ints (cons-stream 1 (map-stream 1+ ints)))
```

Let’s also define the take function in Scheme

```
(define take
   (lambda (n s) (if (= n 0) '() (cons (head s) (take (1- n) (tail s))))))
```

a) How many times will the function 1+ be executed when evaluating (5%)
   (append (take 9 ints) (take 9 ints)) ⇒ (1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9)
   and why?

b) Redo a), but this time assumes that call-by-name is used to implement the infinite data structure, i.e. the call-by-need delay and force are replaced by the call-by-name freeze and thaw given in the lecture. (5%)

Hint: Insert code to print out the number of times the function 1+ is executed.

4 [Haskell’s graph reduction]

Given the Haskell functions

t f x = f (f x)

s x = x*x

a) Draw the graphs for functions t and s. (5%)

b) Draw the graphs step-by-step during the reduction of

   t s 2

   You may ignore the portions of the graphs that become garbage during the reduction. (10%)

5* [list comprehensions] (20%)

Use *list comprehensions* to define the following functions

a) interleave x xs

   returns a list of all possible ways of inserting x into the list xs.

   For example,

   interleave 1 [2,3,4] = [[1,2,3,4],[2,1,3,4],[2,3,1,4],[2,3,4,1]]

   Hint: you may use the built-in functions take, drop and length.

   take 2 [1,2,3,4,5] = [1,2]

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

   length [1,2,3,4,5] = 5

b) permutation xs

   returns a list of all permutations of the list xs.

   For example,

   permutation [1,2,3] = [[1,2,3],[2,1,3],[2,3,1],[1,2,3],[2,1,3],[2,3,1]]

   Hint: Use recursion + interleave

c) nondecreasing xs

returns True if the elements in the list xs are in nondecreasing order, and False, otherwise.

For example,

nondecreasing [1,2,2,3,3,3,4,5] = True

nondecreasing [1,2,2,3,2,4,5] = False

Hint: You may use the built-in functions zip *and* and

zip [1,2,3][4,5,6] = [(1,4),(2,5),(3,6)]

and [1<2,2==2,2<3] = True

and [1<2,2<2,2<3] = False

d) sort xs

returns a list of elements of xs in nondecreasing order by examining all permutations of xs and choosing the first permutation that is sorted in nondecreasing order.

For example,

sort [3,2,4,1,2,3,5,2] = [1,2,2,2,3,3,4,5]

Hint: Use permutation + nondecreasing

6 [Lazy data structure] (25%)

Consider the following infinite sequence

pow2s = [2^0, 2^1, 2^2, 2^3, 2^4, 2^5, …]

a) Let's define pow2s by

pow2s = [2^x | x <-[0..]]

or, equivalently,

pow2s = map (2^) (enumFrom 0)

Explain why this definition is inefficient.

b)* Give an efficient definition that defines pow2s by *generation*.

You shall define it in two ways: one uses list comprehension, and the other doesn't. (Just like the two definitions given in part a).

c) (Continuing b)

Draw the data structure that represents pow2s after the evaluation

Hugs> take 4 pow2s

[1,2,4,8] :: [Integer]

d)* Define pow2s as a cyclic data structure.

Again, you shall define it in two ways: one uses list comprehension, and the other doesn't.

e) (Continuing d)
Draw the cyclic data structure that represents pow2s after the evaluation
Hugs> take 4 pow2s
[1,2,4,8] :: [Integer]

7 [Lazy data structure] (25%)
The Hamming sequence $hamming$ is a sequence of distinct integers in ascending order defined by

1 $1 \in hamming$
2 If $x \in hamming$, then $2x, 3x, 5x \in hamming$
3 Nothing else is in $hamming$

Thus, the Hamming sequence begins with the integers:
1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36,…

a)* Define a function
merge :: [Integer] -> [Integer] -> [Integer]
to merge two ascending sequences into one with duplicates removed, e.g.
merge [1,2,3,4] [2,3,4,5] = [1,2,3,4,5]

b)* Define $hamming$ by generation.
Hint: Use merge to define a generation function
ham $a\ b\ c$
that generates a Hamming-like sequence that contains 1 and ax, bx, cx, for any x in the sequence.

c) (Continuing b)
Draw the data structure that represents *hamming* after the first 4 elements have been printed.

d)* Define $hamming$ as a cyclic data structure.
Hint: Use merge

e) (Continuing d)
Draw the cyclic data structures that represent *hamming* after the first 4 elements have been printed.

8 The function foldl is predefined in Haskell as (15%)

```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

a) Consider the definition

```
sum = foldl (+) 0
```

Explain why this definition of sum is inefficient.

b)* The predefined function foldl' is a strict version of foldl that satisfies

foldl' f ⊥ xs = ⊥

Define foldl' by yourself.

c) Now, define

```
sum = foldl' (+) 0
```

Explain why this version of sum is more efficient than that in part a).