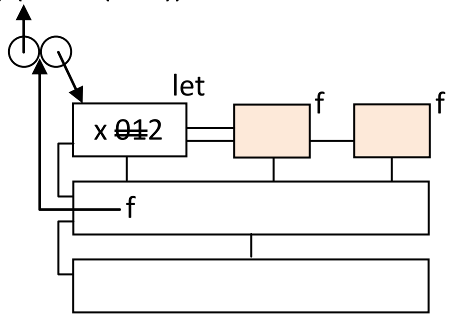
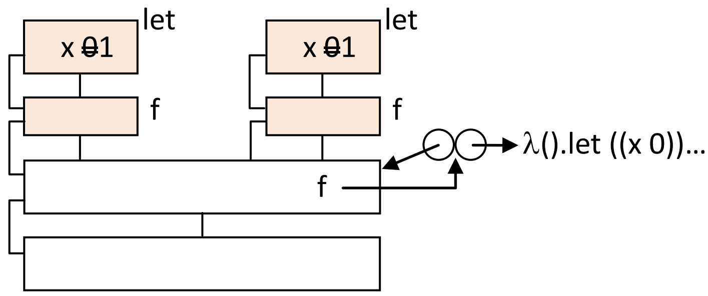
HW5 solution

1 a) λ().(set! x (+ x1)) x



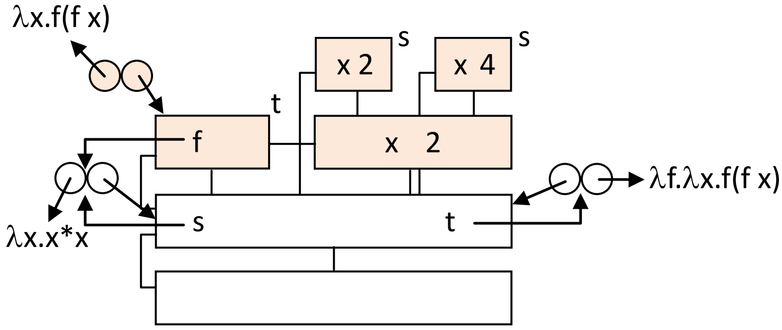
Note: The variable x is similar to C/C++'s local static variable.

b)



Note: The variable x is similar to C/C++'s local auto variable.

2



3 a) 8

(take 9 ints) forces 1, 2, 3, …, 9 to be generated.

Clearly, 2 is computed by evaluating 1+1. Since lazy evaluation memoizes the forced value, 2 is memoized and used to obtain 3. That is, 3 is computed by evaluating 1+2. Similarly, 4, 5, …, and 9 are computed by evaluating 1+3, 1+4, …, and 1+8, respectively.

Thus, the first time (take 9 ints) is evaluated, 1+ is executed 8 times.

Because of lazy evaluation, 2, 3, …, and 9 are all momoized. Therefore, the second time (take 9 ints) is evaluated, 1+ will not be executed any more.

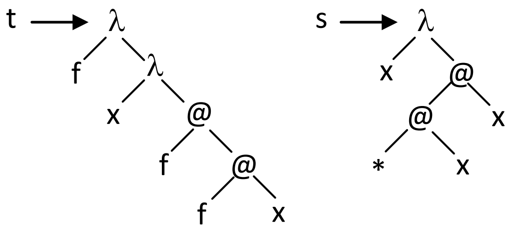
b) 72

Again, 2 is computed by evaluating 1+1. Since normal order evaluation (i.e. call by name) doesn't memoize the thawed value, 2 isn't memoized. So, to compute 3, 2 must be regenerated again. That is, 3 is computed by evaluating 1+1+1. Similarly, 4, 5, …, and 9 are computed by evaluating 1+1+1+1, 1+1+1+1+1, …, and 1+1+1+1+1+1+1+1+1, respectively.

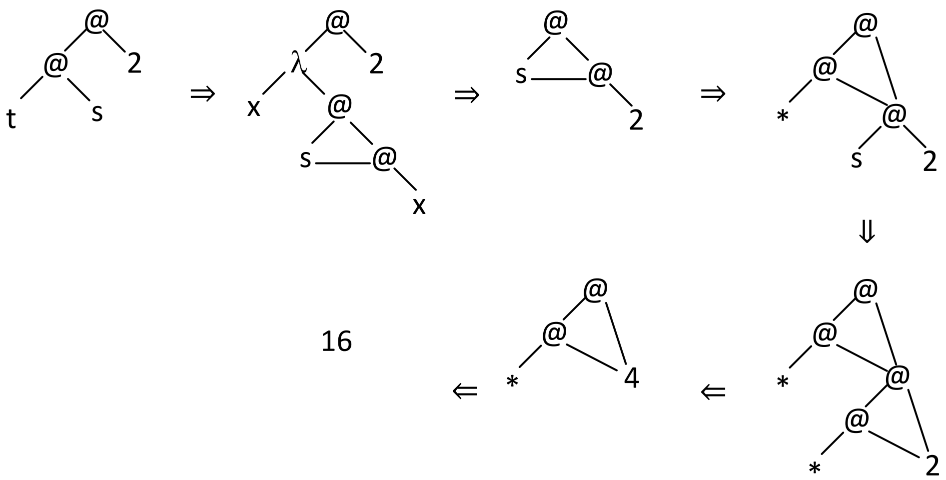
Thus, the first time (take 9 ints) is evaluated, 1+ is executed $\sum_{i=1}^{8} i = 36$ times.

Because of normal order evaluation, 2, 3, …, and 9 aren't memoized. Therefore, the second time (take 9 ints) is evaluated, 1+ has to be executed 36 times again.

4 a)



b)



5 a) interleave x xs = [take i xs++[x]++drop i xs | i<-[0..length xs]]

b) permutation [] = [[]]

permutation (x:xs) = [zs | ys<-permutation xs,zs<-interleave x ys]

c) nondecreasing xs = and [x<=y | (x,y)<-zip xs (tail xs)]

d) sort xs = head [ys | ys<-permutation xs,nondecreasing ys]

6 a) With these definitions, the element 2^x is generated by evaluating 2^x, which takes $O(\log x)$ time by the fast exponentiation algorithm.

The definitions of part b) take the advantage of already-generated element 2^{x-1} to generate 2^x by evaluating $2 \times 2^{x-1}$ in $O(1)$ time.

N.B.

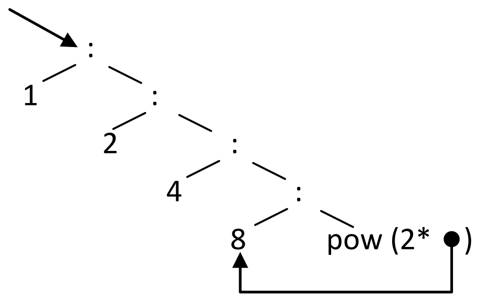
The sample run below illustrates the difference in time and space between a) and b).

Hugs> take 10 pow2s where pow2s = map (2^) (enumFrom 0)
[1,2,4,8,16,32,64,128,256,512] :: [Integer]
(1575 reductions, 2639 cells)
Hugs> take 10 pow2s where pow2s = pow 1 where pow n = n : pow (2*n)
[1,2,4,8,16,32,64,128,256,512] :: [Integer]
(261 reductions, 440 cells)

b) pow2s = pow 1 where pow n = n : pow (2*n)

pow2s = [x | x <- pow 1] where pow n = n : pow (2*n)

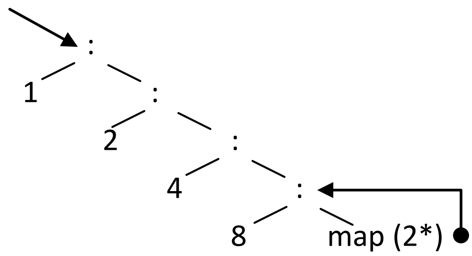
c) pow2s



d) pow2s = 1 : [2*x | x<-pow2s]

pow2s = 1 : map (2*) pow2s

e) pow2s



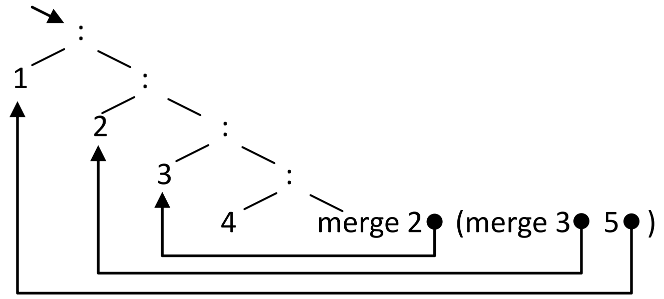
7 a)

```
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) | x==y = x : merge xs ys
                    | x<y  = x : merge xs (y:ys)
                    | x>y  = y : merge (x:xs) ys
```

b)

```
ham a b c = 1 : merge [a*x|x<-ham a b c]
                  (merge [b*x|x<-ham a b c] [c*x|x<-ham a b c])
hamming = ham 2 3 5
```

c) hamming



Comments on the abstract (or simplified) lazy data structure

1 The generator of the 2x subsequence has already consumed 1 and 2 to produce 2 and 4, respectively. Thus, it points to 3, getting ready to produce 6 on demand.

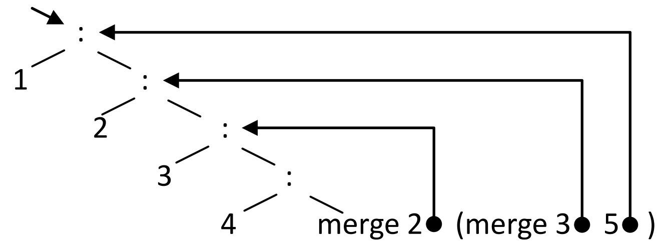
2 The generator of the 3x subsequence has already consumed 1 to produce 3. Thus, it points to 2, getting ready to produce 6 on demand.

3 The generator of the 5x subsequence hasn't consumed any element yet. Thus, it points to 1, getting ready to produce 5 on demand.

d)

```
hamming = 1 : merge [2*x|x<-hamming]
                (merge [3*x|x<-hamming] [5*x|x<-hamming])
```

e) hamming



Comment

The ideas of this abstract (or simplified) cyclic data structure are similar to those of part b).

8 a) Consider

foldl f z (x:xs) = foldl f (f z x) xs

Due to lazy evaluation, the blue-colored expression isn't reduced on calling foldl recursively. As the recursion proceeds, the corresponding graph will grow larger and larger.

For example,

sum [1..10]
= foldl (+) 0 [1..10]
= foldl (+) (0+1) [2..10]
= foldl (+) ((0+1)+2) [3..10]
= ...
= foldl (+)) ((...((0+1)+2)+...)+10) [] // *
= (...((0+1)+2)+...)+10
= 55

b) foldl' f a [] = a
foldl' f a (x:xs) = (foldl' f $! f a x) xs

c) Now, consider

foldl' f a (x:xs) = (foldl' f $! f a x) xs

The strict operator $! forces the blue-colored expression to be reduced on calling foldl' recursively. Thus, as the recursion proceeds, the corresponding graph will NOT grow larger and larger.

For example,

sum [1..10]
= foldl' (+) 0 [1..10]
= (foldl' (+) $! (0+1)) [2..10] // *
= foldl' (+) 1 [2..10] ∵ foldl' (+) $! (0+1) = seq x (foldl' (+) x) where x=0+1
= ... = foldl' (+) x where x=1
= foldl' (+) 55 []
= 55

Comment

As explained above, foldl introduces big unevaluated graphs, but, foldl' doesn't. So, when is foldl useful?

To answer this question, observe from the two starred lines above that foldl' reduces the 1st *argument* of f on recursive call. That is,

1st argument of the pointed-to + is reduced

(...((0+1)+2)+...+10)

1st argument of the pointed-to + is reduced

Thus, foldl' is better if f is strict in its 1st argument. Conversely, foldl is better if f is non-strict in its 1st argument.

For example, define

f _ y = y

Observe that f is non-strict in its 1st argument, i.e. f $\perp$ y = y.

Then, we have

foldl f 0 [1, $\perp$, 3] = 3where 0 is an arbitrary value

but

foldl' f 0 [1, $\perp$, 3] = $\perp$ where 0 is an arbitrary value