1.
(a)
```
hof :: a -> (a->b->a) -> [b] -> a
hof x fx [] = x
hof x fx (y:ys) = hof (fx x y) fx ys
```

(b)
```
f1 = hof s (*)
f2 = hof t (||)
f3 = hof a f3x
   where f3x x y = 2*x + y
f4 = hof xs f4x
   where f4x x y = y++x
f5 = hof m (-)
```

(c)
Yes, it is called foldl.

2.
(a)
beval fails in “beval _ (B True) = ((3 ‘div’ 0) == 0)” because division of zero creates a runtime error that is not due to pattern-matching.

(b)
beval fails when beval _ (B False) is used because the current beval _ (B True) doesn’t handle ‘False’ input and that creates a runtime pattern-matching error.

beval fails when Not is used as it has not been implemented and that creates a runtime pattern-matching error.

(c)
```
beval :: Monad m => Dict Id Bool -> Prop -> m Bool

beval d (And p1 p2)
   = do x <- beval d p1
        y <- beval d p2
        return (x && y)

beval d (P s)= return getBool $ lkp s d

beval d (Let v p1 p2)
   = do b <- beval d p1
        return beval (ins v b d) p2

beval _ (B b) = return b

beval d (Not p)
   = do x <- beval d p
        return (not x)

getBool Nothing = False
getBool (Just x) = x
```

3.
(a)

```
len [] = 0                  -- len.1
len (x:xs) = 1 + len xs     -- len.2
rev [] = []                 -- rev.1
rev (x:xs) = rev xs ++ [x]  -- rev.2
```

len [x]
= “complexify”
len [x:[]]
= “len.2, left2right, at 1st occurrence of len”
1 + len []
= “len.1, left2right, at 1st occurrence of len”
1 + 0
= “arithmetic”
1

(b)

```
len [] = 0                      -- len.1
len (x:xs) = 1 + len xs         -- len.2
len (xs++ys) = len xs + len ys  -- len.3
len [x] = 1                     -- len.4
rev [] = []                     -- rev.1
rev (x:xs) = rev xs ++ [x]      -- rev.2
```

P(xs) = len (rev xs) = len xs

P([])
= “expand P”
len (rev []) = len []
= “rev.1”
len [] = len []
= “reflexivity of =”
True

Assume P(xs),
i.e. len (rev xs) = len xs
Show P(x:xs):

P(x:xs)
= “expand P”
len (rev (x : xs)) = len (x : xs)
= “rev.2”
len (rev xs ++ [x]) = len (x : xs)
= “len.3”
len (rev xs) + len [x] = len (x : xs)
= “len.2”
len (rev xs) + len [x] = 1 + len xs
= “len.4”
len (rev xs) + 1 = 1 + len xs
= “arithmetic”
len (rev xs) = len xs
= “by ind. hypothesis”
True

(c)

```
fInterleaves :: FilePath -> FilePath -> FilePath
fInterleaves inputF1 inputF2 outputF3
   =  do f1 <- openFile inputF1 ReadMode
         f2 <- openFile inputF2 ReadMode
         f3 <- openFile outputF3 WriteMode
         l1 <- getLine f1
         l2 <- getLine f2
         writeFile f3 l1
         writeFile f3 l2
         -- repeat the above four lines until all of the lines have been written to f3
         closeFile f1
         closeFile f2
         closeFile f3
```