Trabajo Práctico # 3

Programación Funcional, Universidad Nacional de Quilmes

8 de abril de 2018

Aclaraciones:

- *Los ejercicios fueron pensados para ser resueltos en el orden en que son presentados. No se saltee ejercicios sin consultar antes a un docente.*
- *Recuerde que puede aprovechar en todo momento las funciones que ha definido, tanto las de esta misma práctica como las de prácticas anteriores.*
- *Pruebe todas sus implementaciones, al menos en una consola interactiva.*
- *Es sumamente aconsejable resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en clase, dado que los exámenes de la materia evaluación principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*

1. Reducción

1. Reducir las siguientes expresiones hasta una forma normal:

   a) `id id`

   b) `id id x`

   c) `(*2) . (+2) $ 0`

   d) `flip (-) 2 3`

   e) `all id $ map (const True) [1..5]`

   f) `map (map (+1)) [[1..5], [6..10]]`

   g) `map ((*2) . (+1)) [1..5]`

   h) `maybe 0 (const 1) $ Just 1`

2. Reducir hasta que las siguientes expresiones den `True`

   a) `map (+1) [1,2,3] == [2] ++ [3] ++ [4]`

   b) `twice id 5 == (id . id . id) 5`

   c) `maybe 0 (const 2) (Just Nothing) == head (map (+1) [1,2,3])`

   d) `factorial 3 == product [1,2,3]`

   e) `(iterate (1:) []) !! 3 == replicate 3 1`

   f) `takeWhile (<3) [1,2,3] == map (+1) [0,1]`

   g) `(curry . uncurry) (+) 1 2 == sum (filter (>=1)) [0,0,1,2]`

2. Demostraciones simples

Demostrar las siguientes equivalencias entre funciones (pueden ser falsas):

1. `last [x] = head [x]`

2. `(\xs -> null x || not (null xs)) = const True`
3. `or [x] == x || not x`
4. `swap . swap = id`
5. `twice id = id . id`
6. `applyN 2 = twice`
7. `twice twice = applyN 4`
8. `(\x -> maybe x id Nothing) = head . (:[])`
9. `curry (uncurry f) = f`
10. `uncurry (curry f') = f'`
11. `maybe Nothing (Just . const 1) = const (Just 1)`
12. `apply = id`

3. Demostraciones por inducción

Demostrar las siguientes equivalencias. Deben utilizarse las definiciones por recursión explícita de cada función.

1. `factorial x = product (countFrom x)`

   ```
   countFrom :: Int -> Int -> [Int]
   countFrom 0 = []
   countFrom n = n : countFrom (n-1)
   ```

2. `length = sum . map (const 1)`
3. `elem e = any (==e)`
4. `all f = and . (map f)`
5. `(map f) . (map g) = map (f . g)`
6. `length = length . reverse`
7. `length = length . map f`
8. `flip (curry f) = curry (f . swap)`
9. `mirrorT . mirrorT = id`
10. `sumT . mapT (const 1) = sizeT`
11. `sizeT = sizeT . mirrorT`
12. `allT f = andT . (mapT f)`
13. `elemT e = anyT (==e)`
14. `countBy p = length . filter p`
15. `concatMap f = concat . map f`
16. `map f (xs ++ ys) = map f xs ++ map f ys`

17. `map f . concat = concat . map (map f)`
18. `reverse (xs ++ ys) = reverse ys ++ reverse xs`
19. `length (zipWith f xs ys) = min (length xs) (length ys)`
    (tener en cuenta la demostración anterior)
20. `sum (xs ++ ys) = sum (zipWith (+) xs ys)` (dar contraejemplo)
21. `filter p (xs ++ ys) = filter p xs ++ filter p ys`
22. `filter p (filter q xs) = filter (\y -> p y && q y) xs`
23. `filter p . map f = map f . filter (p . f)`
24. `takewhile p xs ++ dropwhile p xs = xs`
25. `applyN n (applyN m f) = applyN (n+m) f`
    por inducción en n
26. `applyN n (applyN m) = applyN (n*m)`
    por inducción en n