

# Trabajo Práctico # 3

Programación Funcional, Universidad Nacional de Quilmes

8 de mayo 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..2]`
- f) `map (map (+1)) [[1], [2]]`
- g) `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. `swap . swap = id`

3. `twice id = id`
4. `applyN 2 = twice`
5. `twice twice = applyN 4`
6. `(\x -> maybe x id Nothing) = head . (:[])`
7. `curry (uncurry f) = f`
8. `uncurry (curry f') = f'`
9. `maybe Nothing (Just . const 1) = const (Just 1)`
10. `curry snd = curry (fst . swap)`
11. `(\xs -> null xs || not (null xs)) = const True`
12. `flip (curry f) = curry (f . swap)`
13. `fst = uncurry const`
14. `snd = uncurry (flip const)`
15. `swap = uncurry (flip (,))`
16. `or [x] = x || not x`
17. `not (x && y) = not x || not y`
18. `not (x || y) = not x && not y`

### 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]`  
`countFrom 0 = []`  
`countFrom n = n : countFrom (n-1)`
2. `length = sum . map (const 1)`
3. `elem = any . (==)`  
  
`elem :: Eq a => a -> [a] -> Bool`  
`elem y [] = False`  
`elem y (x:xs) = y == x || elem y xs`
4. `all f = and . map f`
5. `any p = or . map p`
6. `(map f) . (map g) = map (f . g)`
7. `length (xs ++ ys) = length xs + length ys`
8. `length = length . reverse`

```
9. length = length . map f

10. subset xs ys = all ('elem' ys) xs

    subset :: Eq a => [a] -> [a] -> Bool
    subset [] ys = True
    subset (x:xs) ys = elem x ys && subset xs ys

11. concat = concatMap id

    concat :: [[a]] -> [a]
    concat [] = []
    concat (x:xs) = x ++ concat xs

12. concatMap f = concat . map f

13. replicate n x = applyN n (x:) []

14. snoc xs y = xs ++ [y]

    snoc :: [a] -> a -> [a]
    snoc [] y = [y]
    snoc (x:xs) y = x : snoc xs y

15. notElem = (.) not . elem

    notElem :: Eq a => a -> [a] -> Bool
    notElem y [] = True
    notElem y (x:xs) = y /= x && notElem y xs

16. mirrorT . mirrorT = id

17. sumT . mapT (const 1) = sizeT

18. sizeT = sizeT . mirrorT

19. allT f = andT . (mapT f)

20. elemT e = anyT (==e)

21. countBy p = length . filter p

22. map f (xs ++ ys) = map f xs ++ map f ys

23. map f . concat = concat . map (map f)

24. reverse (xs ++ ys) = reverse ys ++ reverse xs

25. last = head . reverse

26. sum (xs ++ ys) = sum (zipWith (+) xs ys) (dar contraejemplo)

27. filter p (xs ++ ys) = filter p xs ++ filter p ys

28. filter p (filter q xs) = filter (\y -> p y && q y) xs

29. filter p . map f = map f . filter (p . f)

30. takeWhile p xs ++ dropWhile p xs = xs

31. applyN n f . applyN m f = applyN (n+m) f
```

- 32. `applyN n (applyN m f) = applyN (n*m) f`
- 33. `applyN n (applyN m) = applyN (m^n)`
- 34. `applyN n f x = iterate f x !! n`
- 35. `(!!) n = head . drop n`
- 36. `lookup x = maybe Nothing (Just . snd) . find (\(k,v) -> x == k)`