PLP - Autoevaluacion 2c2020 - Ej 1 - Programación funcional

No se permite usar recursión explícita.

Se quiere representar digrafos en Haskell mediante una lista de nodos y una func de adyacencia,

```
type Nodo = Integer
data Grafo = G [Nodo] (Nodo -> [Nodo])
```

Donde, `G ns f` es un grafo cuyos nodos son los elementos de ns, y dado n $\in$ ns, `f n` es la lista de vecinos de n. Los arcos que salen de n.

Invariante:

- `ns` y las listas que devuelve `f` son finitas sin repetidos.
- Todos los elementos de las listas en la imagen de f pertenecen a ns. Pero podria devolver cualquier cosa si se la aplica a un nodo que no pertenece a ns.

1.a

El grafo que contiene los nodos numerados del 1 al 9, cuya func de adyacencia conecta a cada numero par a todos los nodos, y los impares no tienen conexiones de salida.

```
ejGrafo :: Grafo
ejGrafo = G ns ady
    where ns = [1..9]
          ady n = if n `mod` 2 == 0
                  then ns   -- si es par, todos los nodos.
                  else []   -- sino, ninguno.
```

1.b

Dado un grafo y un nodo, indica si pertenece a su propia lista de vecinos

```
nodoReflex ejGrafo 6 -> True
nodoReflex ejGrafo 1 -> False
```

```
nodoReflex :: Grafo -> Nodo -> Bool
nodoReflex (G ns f) n = n `elem` f n
```

1.c

Dado un grafo y un camino valido dentro del grafo de long mayor o igual a 1, devuelve los caminos resultantes de extender a c con un unico salto en el grafo. Sug: usar `last`

Ej:

```
extenderCamino ejGrafo [2, 4, 6] -> [[2, 4, 6, 1], ..., [2, 4, 6, 9]]
```

```
extenderCamino :: Grafo -> [Nodo] -> [[Nodo]]
-- Para cada adyacente al ultimo nodo, devuelvo un nuevo camino que lo
-- agregue al final
extenderCamino (G ns f) c = map (\n -> c ++ [n]) (f (last c))
```

1.d

Dado un grafo y un nat k, enumera los caminos que pasan por k arcos. Los que pasan por 0 arcos son las listas de un solo nodo. Sug: `foldNat`.

```
-- sacado del taller 1
foldNat :: a->(a->a)->Integer->a
foldNat caso0 casoSuc n
    | n == 0 = caso0
    | n > 0 = casoSuc (foldNat caso0 casoSuc (n-1))
```

Borrador

```
-- en el caso base, devuelvo todos los nodos para iniciar de alli los caminos
caminosDeLong (G ns f) 0 = ns
-- en cada paso, extiendo los caminos agregando todos los adyacentes al ultimo nodo.
caminosDeLong (G ns f) k = map (\c -> extenderCamino (G ns f) c) (caminosDeLong (k - 1))
```

Con `foldNat`

```
caminosDeLong :: Grafo -> Int -> [[Nodo]]
caminosDeLong g k = foldNat ns (\cs -> map (extenderCamino g) cs) k
```

1.e

Indica si existe un camino que pase por todos los nodos del grafo, sin repetirlos.

Si existe, entonces tiene que tener longitud igual a la cantidad de nodos - 1, porque sino repetiria alguno.

```
hamiltoniano :: Grafo -> Bool
-- tendra un camino que pase por todos los nodos del grafo si de todos los
-- caminos de longitud (length ns) hay alguno que no tenga repetidos.
hamiltoniano (G ns f) = (length caminosSinRepetidos) != 0
    where caminosSinRepetidos = filter noTieneRepetidos (caminosDeLong (length ns))
          noTieneRepetidos c = length c == length (nub c)
    -- nub remueve los duplicados de una lista. Entonces si su longitud es igual
    -- que luego de removerlos, no tenia duplicados
```

1.f

Devuelve `True` sii el grafo dado tiene al menos un ciclo.

Si el grafo tiene un ciclo, entonces tiene que tener un ciclo simple de longitud <= n (con n la cantidad de vertices). Si tiene uno de long mayor, no seria simple, y eso es porque en algun momento repite nodos. Se pueden sacar esas repeticiones y conseguir un ciclo simple.

```
-- Veo todos los caminos de longitudes 1...k, y para cada uno me fijo si tiene
-- un ciclo.
tieneCiclo :: Grafo -> Bool
tieneCiclo g = True `elem` (filter (\k -> hayCiclos (caminosDeLong g k) [1..length ns])
    -- tiene ciclos si al menos un camino es un ciclo
    where hayCiclos cs = (length (filter esCiclo cs)) != 0
          -- es un ciclo si arranca y termina en el mismo
          esCiclo c = head c == last c
```

1.g

Enumera todos los caminos del grafo.

```
caminos :: Grafo -> [[Nodo]]
-- todos los caminos van a ser los caminos de long1, unidos con los de long2, ...
caminos g = concatMap caminosDeLong g [1..]
```