```
elem e = foldr (\x rec -> x == e ||rec) False
```

Práctica 2

Programación Funcional (parte 2)

Paradigmas de (lenguajes de) programación

```
sumaAlternada ::Num a => [a] -> a
sumaAlternada = foldr (-) 0
```

3 de septiembre de 2024

Recursión primitiva:

```
recr:: (a -> [a] -> b -> b) -> b -> [a] -> b
recr f z [] = z
recr f z (x : xs) = f x xs (recr f z xs)
```

Recursión estructural:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x : xs) = f x (foldr f z xs)
```

Recursión iterativa:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f ac [] = ac
foldl f ac (x : xs) = foldl f (f ac x) xs
```


Recursión sobre listas

```
take = flip takeAux
takeAux :: [a] -> Int -> [a]
takeAux = foldr (\x rec n -> if n == 0 then [] else x : rec (n - 1)) (const [])
```

Implementar las siguientes funciones utilizando esquemas de recursión

1. `elem :: Eq a => a -> [a] -> Bool` que indica si un elemento pertence o no a la lista.
2. `sumaAlt`, que realiza la suma alternada de los elementos de una lista. Es decir, da como resultado: el primer elemento, menos el segundo, más el tercero, menos el cuarto, etc.
3. **take** `:: Int -> [a]-> [a]` utilizando `foldr`.
4. `sacarUna:: Eq a => a -> [a] -> [a]` que elimina la primera aparición de un elemento en la lista.

¿Qué otros esquemas de recursión conocen?

```
sacarUna e = recr (\x xs rec -> if e == x then xs else x : rec) []
```

Un breve repaso

¿Qué tipo de recursión tiene cada una de las siguientes funciones? (Estructural, primitiva, global).

```
take' :: [a] -> Int -> [a]
take' [] _ = []
take' (x:xs) n = if n==0 then [] else x : take' xs (n-1)
```

Recursión estructural

```
listasQueSuman :: Int -> [[a]]
listasQueSuman 0 = [[]]
listasQueSuman n | n > 0 =
                 [x : xs | x <- [1..], xs <- listasQueSuman (n-x)]
```

Recursión global

```
fact :: Int -> Int
fact 0 = 1
fact n | n > 0 = n * fact (n-1)
```

Recursión primitiva

```
fibonacci :: Int -> Int
fibonacci 0 = 1
fibonacci 1 = 1
fibonacci n | n > 1 = fibonacci (n-1) + fibonacci (n-2)
```

Recursión global

Generación Infinita

Definir:

`pares :: [(Int, Int)]`, una lista (infinita) que contenga todos los pares de números naturales (sin repetir).

```
pares = [(x, y) | s <- [0..], x <- [0..s], y <- [0..s], x + y == s]
```


Paradigmas de (lenguajes de) programación
Programación Funcional (parte 2)

Folds sobre estructuras nuevas

Sea el siguiente tipo:
`data AEB a = Hoja a | Bin (AEB a) a (AEB a)`
Ejemplo: `miÁrbol = Bin (Hoja 3) 5 (Bin (Hoja 7) 8 (Hoja 1))`

Definir el esquema de recursión estructural (*fold*) para árboles estrictamente binarios, y dar su tipo.

El esquema debe permitir definir las funciones `altura`, `ramas`, `cantNodos`, `cantHojas`, `espejo`, etc.

¿Cómo hacemos?

Recordemos el tipo de `foldr`, el esquema de recursión estructural para listas.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

¿Por qué tiene ese tipo?
(Pista: pensar en cuáles son los constructores del tipo `[a]`).

Un esquema de recursión estructural espera recibir **un argumento por cada constructor** (para saber qué devolver en cada caso), y además **la estructura que va a recorrer**.

El tipo de cada argumento va a depender de lo que reciba el constructor correspondiente. (¡Y todos van a devolver lo mismo!)

Si el constructor es recursivo, el argumento correspondiente del fold va a recibir el resultado de cada llamada recursiva.

¿Cómo hacemos? (Continúa)

```
altura :: AEB a -> Int
altura = foldAEB (const 1) (\izq _ der -> 1 + max izq der)
```

Miremos bien la estructura del tipo.

data AEB a = Hoja a | Bin (AEB a) a (AEB a)

Estamos ante un tipo inductivo con un constructor *no recursivo* y un constructor *recursivo*.

¿Cuál va a ser el tipo de nuestro `fold`?

¿Y la implementación?

```
cantNodos :: AEB a -> Int
cantNodos = foldAEB (const 1) (\izq _ der ->1+ izq + der)
```


Solución

```
ramas :: AEB a -> [[a]]
ramas = foldAEB (\n -> [[n]]) (\izq n der -> map (n :) (izq ++ der))
```

```
foldAEB :: (a -> b) -> (b -> a -> b -> b) -> AEB a -> b
foldAEB fHoja fBin t   =   case t of
      Hoja n           ->  fHoja n
      Bin t1 n t2      ->  fBin (rec t1) n (rec t2)
                            where rec = foldAEB fHoja fBin
```

```
espejo :: AEB a -> AEB a
espejo = foldAEB Hoja (\izq n der -> Bin der n izq)
```

Ejercicio para ustedes: definir las funciones `altura`, `ramas`, `cantNodos`, `cantHojas` y `espejo` usando `foldAEB`.
Si quieren podemos hacer alguna en el pizarrón.

```
cantHojas :: AEB a -> Int
cantHojas = foldAEB (const 1) (\izq _ der -> izq + der)
```

Funciones sobre árboles

Recursión estructural

Dado el tipo de datos:

```
data AB a = Nil | Bin (AB a) a (AB a)
```

¿Qué tipo de recursión tiene cada una de las siguientes funciones? (Estructural, primitiva, global).

Recursión primitiva

```
insertarABB :: Ord a => a -> AB a -> AB a
insertarABB x Nil = Bin Nil x Nil
insertarABB x (Bin i r d) = if x < r
          then Bin (insertarABB x i) r d
          else Bin i r (insertarABB x d)
```

Recursión estructural

```
truncar :: AB a -> Int -> AB a
truncar Nil _ = Nil
truncar (Bin i r d) n = if n == 0 then Nil else
          Bin (truncar i (n-1)) r (truncar d (n-1))
```

Recursión primitiva:

```
recAB :: b -> (AB a -> b -> a -> AB a -> b -> b) -> AB a -> b
recAB cNil cBin Nil = cNil
recAB cNil cBin (Bin i r d) = cBin i (recAB cNil cBin i) r d (recAB cNil cBin d)
```

```
truncar t n = foldAB (\_ -> Nil) (\i r d n -> if n == 0 then Nil else Bin (i (n - 1)) r (d (n - 1))) t n
```


Folds sobre otras estructuras

```
foldAB :: b -> (b -> a -> b -> b) -> AB a -> b
foldAB cNil cBin Nil = cNil
foldAB cNil cBin (Bin i r d) = cBin (foldAB cNil cBin i) r (foldAB cNil cBin d)
```

Dado el siguiente tipo que representa polinomios:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

```
insertarABB x = recAB (Bin Nil x Nil) (\i ri r d rd ->if x < r then Bin ri r d else Bin i r rd)
```

- Definir la función `evaluar :: Num a => a -> Polinomio a -> a`
- Definir el esquema de recursión estructural `foldPoli` para polinomios (y dar su tipo).
- Redefinir `evaluar` usando `foldPoli`.

En la última página

Una estructura más compleja

Dado el tipo de datos
`data RoseTree a = Rose a [RoseTree a]`
de árboles donde cada nodo tiene una cantidad indeterminada de hijos.

1. Escribir el esquema de recursión estructural para `RoseTree`.
2. Usando el esquema definido, escribir las siguientes funciones:
   - `hojas`, que dado un `RoseTree`, devuelva una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
   - `ramas`, que dado un `RoseTree`, devuelva los caminos de su raíz a cada una de sus hojas.
   - `tamaño`, que devuelve la cantidad de nodos de un `RoseTree`.
   - `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.

Funciones como estructuras de datos

Representando conjuntos con funciones

Se cuenta con la siguiente representación de conjuntos `type Conj a = (a->Bool)` caracterizados por su función de pertenencia. De este modo, si c es un conjunto y e un elemento, la expresión $c\ e$ devuelve `True` si e pertenece a c y `False` en caso contrario.

1. Definir la constante `vacío :: Conj a`, y la función `agregar :: Eq a => a -> Conj a -> Conj a`.
2. Escribir las funciones `intersección`, `unión` y `diferencia` (todas de tipo `Conj a -> Conj a-> Conj a`).

```
evaluar x pol = case pol of
                X -> x
                Cte c -> c
                Suma p q -> evaluar x p + evaluar x q
                Prod p q -> evaluar x p * evaluar x q

foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) ->Polinomio a -> b
foldPoli fX fCte fSuma fProd pol = case pol of
                                   X -> fX
                                    Cte c -> fCte c
                                    Suma p q -> fSuma (rec p) (rec q)
                                    Prod p q -> fProd (rec p) (rec q)
                                    where rec = foldPoli fX fCte fSuma fProd

evaluar x = foldPoli x id (+) (*)
```

```
foldRT :: (a -> [b] -> b) -> RoseTree a -> b
foldRT fRose (Rose n hijos) = fRose n (map rec hijos)
                              where rec = foldRT fRose

recRT :: (a -> [RoseTree a] -> [b] -> b) -> RoseTree a -> b
recRT fRose (Rose n hijos) = fRose n hijos (map rec hijos)
                             where rec = recRT fRose

hojas :: RoseTree a -> [a]
hojas = foldRT (\n rec -> if null rec then [n] else concat rec)

ramas :: RoseTree a -> [[a]]
ramas = foldRT (\n ns -> if null ns then [[n]] else map (n :) (concat ns))

tamaño :: RoseTree a -> Int
tamaño = foldRT (\n rhijos -> 1 + sum rhijos)

altura :: RoseTree a -> Int
altura = foldRT (\_ rec -> if null rec then 1 else 1 + maximum rec)
```

```
vacio = const False

agregar e c = \x -> x == e || c x

interseccion c1 c2 = \e -> c1 e && c2 e

union c1 c2 = \e -> c1 e || c2 e

diferencia c1 c2 = \e -> c1 e && not (c2 e)
```