TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE OAXACA

CARRERA: INGENIERÍA EN SISTEMAS COMPUTACIONALES

PROGRAMACION LOGICA Y FUNCIONAL

Ejercicios de Lógica Funcional

Alumna:

Santiago Jimenez Galilea

DOCENTE: MATADAMAS ORTIZ IDARH CLAUDIO

GRUPO 7SA

OAXACA DE JUÁREZ, 14 DE OCTUBRE DE 2025

1.1 Factorial

- Factorial con condicionales **fact1**

```
fact1 :: Integer -> Integer
fact1 n = if n == 0 then 1
          else n * fact1 (n - 1)
```

Salida:

```
ghci> fact1 3
6
```

Explicación:

fact1 se llama a sí misma una y otra vez, cada vez con un número más pequeño, hasta que llega a 0. En ese punto deja de llamarse y empieza a multiplicar todos los números que guardó en el camino

- Factorial con guardas **fact2**

```
fact2 :: Integer -> Integer
fact2 n
    | n == 0    = 1
    | otherwise = n * fact2 (n - 1)
```

Salida:

```
ghci> fact2 4
24
```

Explicación:

La función fact2 calcula el factorial de un número usando **guardas** que son la líneas que comienzan con |. Si n es 0 devuelve 1, si no se cumple pasa a la siguiente guarda **otherwise** que significa "en cualquier otro caso", multiplica n por el factorial de n - 1.

- Factorial con patrones fact3

```
fact3 :: Integer -> Integer
fact3 0 = 1
fact3 n = n * fact3 (n - 1)
```

Salida:

```
ghci> fact3 5
120
```

Explicación:

fact3 calcula el factorial de un número usando recursión, pero esta vez usa coincidencia de patrones (*pattern matching*) en lugar de if o guardas. Si n es 0 devuelve 1, Si no coincide con el primer patrón, pasa al segundo multiplica n por el resultado de fact3 (n - 1).

- Restricción de dominio mediante guardas fact4

```
fact4 :: Integer -> Integer

fact4 n
    | n == 0    = 1
    | n >=1     =n * fact4 (n - 1)
```

Explicación:

Calcula el factorial usando **guardas** y además **restringe el dominio**: solo permite valores n >= 0. Si n es 0 devuelve 1, y si es mayor, multiplica n por el factorial de n - 1.

Salida:

```
ghci> fact4 3
6
```

- Restricción del dominio mediante patrones fact5

```
fact5 :: Integer -> Integer

fact5 0      =1

fact5 (n+1)  = (n+1) * fact5 n
```

Explicación: (n+1) es el número actual del que queremos el factorial, fact5 n llama a la función con el número anterior. Luego se multiplica (n+1) por el factorial del anterior. Esto sigue hasta que llega al caso base fact5 0.

```
ghci> fact5 5
120
```

- Mediante predefinidas fact6

```
fact6 :: Integer -> Integer

fact6 n = product [1..n]
```

Explicación:

[1..n] para generar todos los números desde 1 hasta n. product para multiplicar todos esos números

```
ghci> fact6 7
5040
```

- Mediante plegado fact7

```
fact7 :: Integer -> Integer

fact7 n = foldr (*) 1 [1..n]
```

Explicación:

Fold: es una manera de "plegar" o combinar todos los elementos de una lista usando una función binaria (aquí *) y un valor inicial (aquí 1).Genera la lista [1..n]. y "Pliega" la lista multiplicando todos sus elementos, empezando desde 1.

Salida:

```
ghci> fact7 3
6
```

- **Comprobación de las definiciones**

```
prop_equivalencia :: Integer -> Property

prop_equivalencia x =

  x >= 0 ==>

    fact2 x == fact1 x &&

    fact3 x == fact1 x &&

    fact4 x == fact1 x &&

    fact5 x == fact1 x &&

    fact6 x == fact1 x &&

    fact7 x == fact1 x
```

Explicación:

prop_equivalencia es una propiedad de QuickCheck que verifica que todas tus funciones de factorial (fact2 a fact7) sean equivalentes a fact1>=0

Realizamos la comprobación con

```
Main>quickCheck prop_equivalencia
```

```
ghci> let x=10
ghci> x >= 0 && fact1 x == fact2 x && fact1 x == fact3 x && fact1 x == fact4 x && fact1 x == fact5 x && fact1 x == fact6 x && fact1 x == fact7 x
True
```

```
ghci> :l Factorial.hs
[1 of 1] Compiling Factoriales      ( Factorial.hs, interpreted )
Ok, one module loaded.
ghci> quickCheck prop_equivalencia
+++ OK, passed 100 tests; 98 discarded.
```

1.2 Número de combinaciones

```
{-# LANGUAGE  NPlusKPatterns #-}

fact5 :: Integer -> Integer

fact5 0      =1

fact5 (n+1)  = (n+1) * fact5 n


comb :: Integer -> Integer -> Integer

comb n k = div (fact5 n) (fact5 k * fact5 (n - k))
```

Explicación: NPlusKPatterns permite escribir patrones como (n+1) en la definición de funciones, que facilitan la recursión.

Combinatorio (comb): calcula cuántas formas hay de escoger k elementos de n.

La función Divide el factorial de n entre el producto del factorial de k y el factorial de n-k para obtener el número de formas de elegir k elementos de n".

Salida:

```
PS C:\Haskell_plf\Ejercicios_programacion_haskell> ghci
Loaded package environment from C:\Users\USUARIO\AppData\Roaming\ghc\x86_64-mingw32-9.6.7\environments\default
GHCi, version 9.6.7: https://www.haskell.org/ghc/  :? for help
ghci> :l NumeroCombinaciones.hs
[1 of 2] Compiling Main             ( NumeroCombinaciones.hs, interpreted )
Ok, one module loaded.
ghci> comb 10 2
45
ghci>
```

1.3 Comprobación de número impar

- Usando la predefinida odd impar1

```
impar1 :: Integer -> Bool

impar1 = odd
```

Explicación:

Aquí no escribimos la lógica de impar nosotros mismos, sino que usamos la función predefinida odd. impar1 es una función que verifica si un número entero es impar usando la función predefinida odd.

Devuelve True si el número es impar y False si es par.

Salida:

```
ghci> :l Impares.hs
[1 of 1] Compiling Impares          ( Impares.hs, interpreted )
Ok, one module loaded.
ghci> impar1 7
True
```

- Usando las predefinidas not y even

```
impar2 :: Integer -> Bool
impar2 x = not (even x)
```

Explicación: even comprueba si un número es par y not: invierte un booleano. Comprueba si el número no es par (not (even x)), devolviendo True si es impar y False si es par

Salida:

```
ghci> impar2 6
False
```

- Usando las predefinidas not, even y (.)

```
impar3 :: Integer -> Bool
impar3 = not . even
```

Explicación: El operador . es composición de funciones:

(f . g) x = f (g x)

En (not . even) x = not (even x)

Asi que aplica primero even al número y luego invierte el resultado con not, devolviendo True si el número es impar y False si es par".

Salida

```
ghci> impar3 3
True
```

- Por recursión

```
impar4 :: Integer -> Bool
impar4 x
  | x > 0     = impar4_aux x
  | otherwise = impar4_aux (-x)
  where
    impar4_aux 0 = False
    impar4_aux 1 = True
    impar4_aux n = impar4_aux (n - 2)
```

Explicación:
Guardas (|): permiten evaluar condiciones y ejecutar diferentes acciones según la condición.

Función auxiliar (where): define funciones internas que solo se usan dentro de otra función.

Valor absoluto: se usa (-x) para que la función funcione con números negativos.

Salida

```
ghci> impar4 248
False
```

- **Comprobación de las definiciones**

```
prop_equivalencia :: Integer -> Bool

prop_equivalencia x =

    impar2 x == impar1 x &&

    impar3 x == impar1 x &&

    impar4 x == impar1 x
```

Explicación: Para cualquier número entero, impar2, impar3 e impar4 deben dar el mismo resultado que impar1

Salida:

```
ghci> quickCheck prop_equivalencia
+++ OK, passed 100 tests.
```

```
ghci> let x = 10
ghci> x>=0 && impar1 x== impar2 x && impar1 x== impar3 x && impar1 x== impar4 x
True
```

1.4 Cuadrado

- Mediante (*)

```
cuadrado1 :: Num a => a -> a

cuadrado1 x = x*x
```

Salida:

```
ghci> cuadrado1 2
4
```

- Mediante (^)

```
cuadrado2 :: Num a => a -> a

cuadrado2 x = x ^ 2
```

Salida:

```
ghci> cuadrado2 4
16
```

- Mediante secciones

```
cuadrado3 :: Num a => a -> a
cuadrado3 = (^2)
```

Explicación: **Point-free style**: escribir funciones sin nombrar explícitamente los argumentos.

- cuadrado3 = (^2) es equivalente a cuadrado3 x = x ^ 2.

Salida:

```
ghci> cuadrado3 16
256
```

- **Comprobación de las definiciones**

```
prop_equivalencia :: Int -> Bool

prop_equivalencia x =

    cuadrado1 x == cuadrado2 x &&

    cuadrado1 x == cuadrado3 x
```

Salida:

```
ghci> prop_equivalencia 16
True
```

```
ghci> quickCheck prop_equivalencia
+++ OK, passed 100 tests.
```

1.5 Suma de cuadrados

- Suma Cuadrados con Sum, map y cuadrado

```
suma_de_cuadrados_1 :: [Integer] -> Integer

suma_de_cuadrados_1 l = sum (map cuadrado l)
```

Explicación: usamos . **Listas [Integer]** ya que la función usara una secuencias de elementos del mismo tipo. **map** aplica una función a cada elemento de una lista, en este caso map eleva al cuadado los elementos de la lista. **sum** suma todos los elementos de una lista.

Salida:

```
ghci> suma_de_cuadrados_1 [5, 10]
125
```

- Suma cuadrados con Sum y listas intensionales:

```
suma_de_cuadrados_2 :: [Integer] -> Integer

suma_de_cuadrados_2 l = sum [x*x | x <- l]
```

> Explicación: **Listas por comprensión**: una forma elegante de crear listas aplicando una operación a cada elemento de otra lista.
>
> - Sintaxis: [expresion | variable <- lista]. [x*x | x <- l] hace lo mismo que **map cuadrado l.**

Salida:

```
ghci> suma_de_cuadrados_2 [10, 10]
200
```

- Suma de cuadrados con sum, map y lamda

```
suma_de_cuadrados_3 :: [Integer] -> Integer

suma_de_cuadrados_3 l = sum (map (\x -> x*x) l)
```

> Explicación: **Lambda (\x -> ...)** es una función sin nombre que se define directamente donde se necesita. Aplica a cada elemento de la lista una función que eleva al cuadrado, y luego suma todos los resultados

Salida:

```
ghci> suma_de_cuadrados_3 [40, 5]
1625
```

- Suma de cuadrados por recursión

```
suma_de_cuadrados_4 :: [Integer] -> Integer

suma_de_cuadrados_4 [] = 0

suma_de_cuadrados_4 (x:xs) = x*x + suma_de_cuadrados_4 xs
```

Salida:

```
ghci> suma_de_cuadrados_4 [3, 5]
34
```

Explicación: **Patrones de listas (x:xs)**: permite "descomponer" la lista en cabeza (x) y cola (xs). Si la lista está vacía, devuelve 0; si no, suma el cuadrado del primer elemento con la suma de los cuadrados del resto de la lista

- **Comprobación de las definiciones**

```
prop_equivalencia_suma :: [Integer] -> Bool
prop_equivalencia_suma xs =
    suma_de_cuadrados_1 xs == suma_de_cuadrados_2 xs &&
    suma_de_cuadrados_1 xs == suma_de_cuadrados_3 xs &&
    suma de cuadrados 1 xs == suma de cuadrados 4 xs
```

Salida:

```
ghci> quickCheck prop_equivalencia_suma
+++ OK, passed 100 tests.
ghci> prop_equivalencia_suma[10, 3]
True
```

1.6 Suma de cuadrados

- Raíz definición directa

```
raices_1 :: Double -> Double -> Double -> [Double]
raices_1 a b c = [ (-b+sqrt(b*b-4*a*c))/(2*a),
                   (-b-sqrt(b*b-4*a*c))/(2*a) ]
```

Explicación: Aplica la fórmula general (-b ± sqrt(b^2 - 4ac)) / (2a). Si el discriminante es negativo, no hay raíces reales y Haskell devuelve NaN de lo contrario da las raíces.

Salida:

```
ghci> raices_1 1 2 1
[-1.0,-1.0]
```

```
ghci> raices_2 20 1 4
[NaN,NaN]
```

- Con entornos locales

```
raices_2 ::Double -> Double -> Double -> [Double]

raices_2 a b c =

    [(-b+d)/n, (-b-d)/n]

    where d = sqrt(b*b-4*a*c)

          n = 2*a
```

Explicación: Discriminante: d = sqrt(b*b-4*a*c) Calcula el discriminante y el denominador en variables locales d y n, y luego aplica la fórmula (-b ± d)/n. Si el discriminante es negativo, Haskell devuelve NaN

Salida:

- **Comprobación de legibilidad y eficiencia**

```
:set +s

raices_1 1 3 2

raices_2 1 3 2
```

Explicación:

:set +s

Activa la impresión de tiempo y memoria de cada comando que ejecutes.

Salida:

1.7 Valor absoluto

- Valor absoluto con condicionales

```
n_abs_1 :: (Num a, Ord a) => a -> a
n_abs_1 x = if x>0 then x else (-x)
```

Explicación:

Ord a → a se puede comparar (>, <, >=, <=)

Si el número es positivo o cero devuelve el mismo número; si es negativo, devuelve su opuesto para que siempre sea no negativo

Salida:

```
ghci> n_abs_1 (-10)
10
```

- Valor absoluto con guardas

```
n_abs_2 :: (Num a, Ord a) => a -> a
n_abs_2 x | x>0       = x
          | otherwise = -x
```

Salida:

```
ghci> n_abs_2 (-10)
10
```

- **Comprobación que las definiciones son equivalentes**

```
prop_equivalencia_abs :: Integer -> Bool
prop_equivalencia_abs x =
    n_abs_1 x == abs x &&
    n_abs_2 x == abs x
```

Salida:

```
ghci> :l Valorabsoluto.hs
[1 of 2] Compiling Main             ( Valorabsoluto.hs, interprete
d )
Ok, one module loaded.
(0.04 secs,)
ghci> quickCheck prop_equivalencia_abs
+++ OK, passed 100 tests.
(2.92 secs, 920,552 bytes)
```

```
ghci> prop_equivalencia_abs 10
True
```

1.8 Signo

```
n_signum x | x > 0      = 1
           | x == 0     = 0
           | otherwise = -1
```

Explicación: Esta función Si el número es positivo devuelve 1, si es cero devuelve 0, y si es negativo devuelve -1

Salida:

```
ghci> :l signo.hs
[1 of 2] Compiling Main             ( signo.hs, interpreted )
Ok, one module loaded.
(0.02 secs,)
ghci> n_signum 5
1
(0.00 secs, 51,448 bytes)
ghci> n_signum 7
1
```

- **Comprobación que las definiciones son equivalentes**

```
prop_equivalencia_signum :: Integer -> Bool
prop_equivalencia_signum x =
    n_signum x == signum x
```

```
ghci> quickCheck prop_equivalencia_signum
+++ OK, passed 100 tests.
(0.00 secs, 904,536 bytes)
```

1.9 Conjunción

```
(&&&) :: Bool -> Bool -> Bool

False &&& x  = False

True  &&& x  = x
```

Explicación:

Es una función que realiza la conjunción lógica (AND) entre dos booleanos:

“Si el primer valor es False, devuelve False. Si el primero es True, devuelve el segundo valor

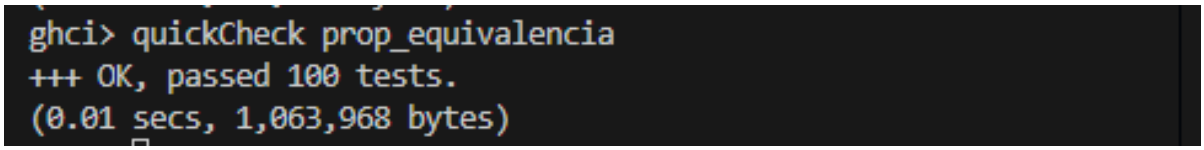
Salida:

- **Las definiciones son equivalentes**

```
prop_equivalencia x y =

    (x &&& y) == (x && y)
```

Salida:



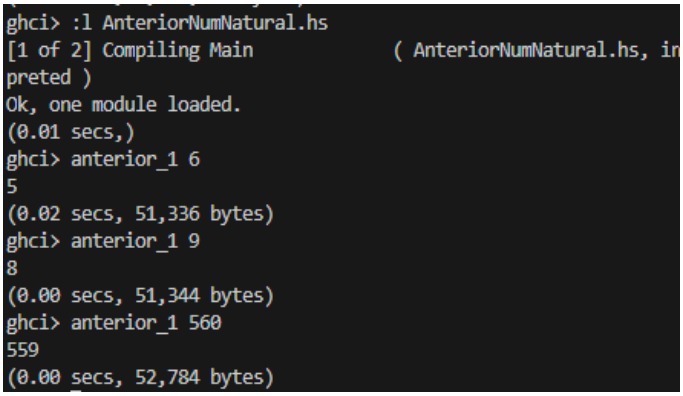
1.10 Anterior l número natural

- Con patrones

```
anterior_1 :: Int -> Int

anterior_1 (n+1) = n
```

Salida:



Explicación:

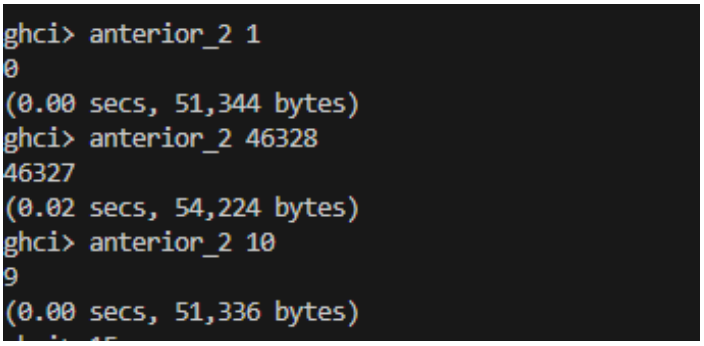
Patrones N+1 (n+1): Permiten **descomponer un número natural** en un "anterior" (n) más uno. Así que Recibe un número x, lo representa como n+1 y devuelve n, es decir, el número anterior a x **N+1** necesita la extensión NPlusKPatterns en Haskell

- Con guardas

```
anterior_2 :: Int -> Int

anterior_2 n | n>0 = n-1
```

Salida:



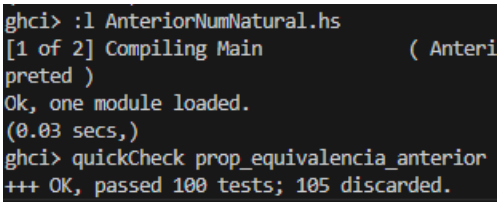
Explicación: Si el número es mayor que 0, devuelve n-1; de lo contrario, no está definido

- **Definiciones equivalentes sobre los números naturales**

```
prop_equivalencia_anterior :: int -> property

prop_equivalencia_anterior n =

  n > 0 ==> anterior_1 n == anterior_2 n
```



1.11 Potencia

Explicación: Si el exponente es 0, devuelve 1; si es mayor, multiplica el número base por la potencia del número elevado al exponente anterior

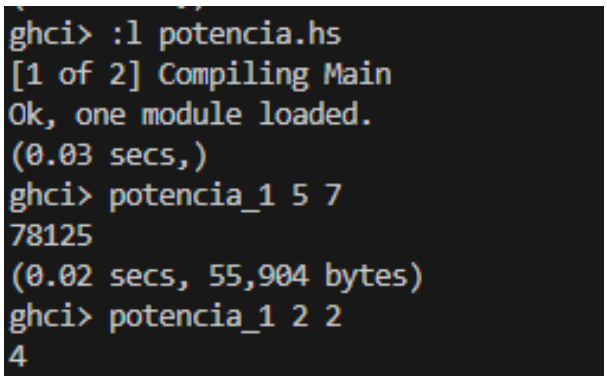
- Por patrones

```
potencia_1 :: Num a => a -> Int -> a

potencia_1 x 0 = 1

potencia_1 x (n+1)= x * (potencia_1 x n)
```

Salida:

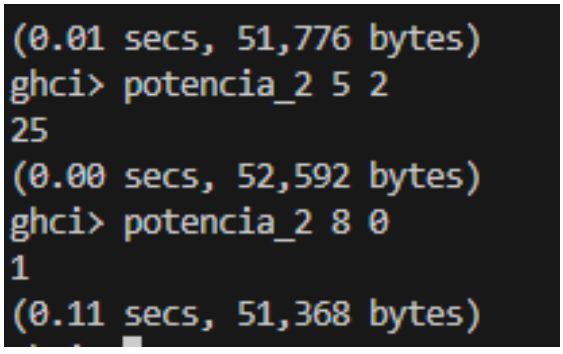


- Por condicionales

```
potencia_2 :: Num a => a -> Int -> a

potencia_2 x n = if n==0 then 1
                 else x * potencia_2 x (n-1)
```

Salida:



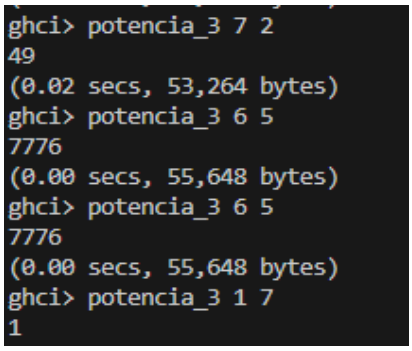
Explicación: Tipado polimórfico (Num a => a): funciona con cualquier tipo numérico (Integer, Double

Potencia_2 calcula x elevado a n de manera recursiva, pero usando una condición if en lugar de patrones N+1:

- Definición eficiente

```
potencia_3 :: Num a => a -> Int -> a
potencia_3 x 0 = 1
potencia_3 x n | n > 0 = f x (n - 1) x
  where
    f _ 0 y = y
    f x n y = g x n
      where
        g x n
          | even n    = g (x * x) (n `quot` 2)
          | otherwise = f x (n - 1) (x * y)
```

Salida:



Si el exponente es 0, devuelve 1. Si es mayor, usa una función auxiliar con acumulador y optimización por exponentes pares para reducir el número de multiplicaciones necesarias

x → la base

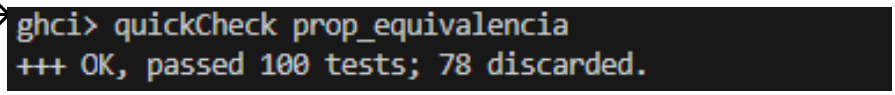
n → exponente restante

y → acumulador que guarda el resultado parcial

- La funciones son equivalentes

```
prop_equivalencia :: Int -> Int -> Property
prop_equivalencia x n=
  n >= 0 ==>
    (potencia_1 x n == x^n &&
     potencia_2 x n == x^n &&
     potencia_3 x n == x^n)
```

Salida →



1.12 Función Identidad

```
n_id :: a-> a

n_id x = x
```

Salida:

```
ghci> n_id 5
5
(0.00 secs, 18,512 bytes)
ghci> n_id "hola"
"hola"
(0.00 secs, 55,248 bytes)
ghci> n_id True
True
(0.00 secs, 20,440 bytes)
```

Explicación: Es una función de tipo polimórfico así que a puede ser cualquier tipo

Recibe un valor de cualquier tipo y lo devuelve tal cual, sin modificarlo