


``xs !! n``: Obtiene el elemento en la posición ``n`` de la lista ``xs``.

• Ejemplo:

haskell


 Copy code

```
let lista = [1, 3, 5, 7, 9]
print (lista !! 2) -- Salida: 5
```

``f . g``: Compone dos funciones, aplicando ``g`` primero y luego ``f``.

• Ejemplo:

haskell


 Copy code

```
let f x = x * 2
let g x = x + 3
let h = f . g
print (h 4) -- Salida: 14 (f(g(4)))
```

``and xs``: Devuelve ``True`` si todos los elementos de ``xs`` son ``True``.

• Ejemplo:

haskell


 Copy code

```
let lista1 = [True, False, True]
let lista2 = [True, True, True]
print (and lista1) -- Salida: False
print (and lista2) -- Salida: True
```

``ceiling x``: Devuelve el entero más pequeño que es mayor o igual a ``x``.

• Ejemplo:

haskell


 Copy code

```
print (ceiling 4.5) -- Salida: 5
```

``chr n``: Convierte un número ``n`` en su caracter correspondiente en Unicode.

• Ejemplo:

haskell


 Copy code

```
print (chr 65) -- Salida: 'A'
```

``const x y``: Siempre devuelve ``x``, ignorando ``y``.

• Ejemplo:

haskell


 Copy code

```
let f = const 42
print (f "Ignorado") -- Salida: 42
```

`curry f`: Convierte una función de dos argumentos en una función que toma un solo argumento y devuelve otra función.

• Ejemplo:

haskell


 Copy code

```
let f x y = x + y
let g = curry f
let h = g 2 3
print h -- Salida: 5
```

`dropWhile p xs`: Elimina elementos de la lista `xs` mientras el predicado `p` se cumpla.

• Ejemplo:

haskell


 Copy code

```
let lista = [1, 3, 5, 7, 8, 9]
print (dropWhile (< 6) lista) -- Salida: [7,8,9]
```

`flip f x y`: Intercambia el orden de los argumentos en la función `f`.

• Ejemplo:

haskell


 Copy code

```
let f x y = x / y
let g = flip f
print (g 2 8) -- Salida: 4.0 (8 / 2)
```

`floor x`: Devuelve el entero más grande que es menor o igual a `x`.

• Ejemplo:

haskell


 Copy code

```
print (floor 4.8) -- Salida: 4
```

`foldl f e xs`: Aplica la función `f` a cada elemento de la lista `xs` de izquierda a derecha, utilizando un valor inicial `e`.

• Ejemplo:

haskell


 Copy code

```
let lista = [1, 2, 3, 4, 5]
let resultado = foldl (\acc x -> acc + x) 0 lista
print resultado -- Salida: 15 (suma de la lista)
```

`foldr f e xs`: Aplica la función `f` a cada elemento de la lista `xs` de derecha a izquierda, utilizando un valor inicial `e`.

• Ejemplo:

haskell


 Copy code

```
let lista = [1, 2, 3, 4, 5]
let resultado = foldr (\x acc -> acc + x) 0 lista
print resultado -- Salida: 15 (suma de la lista)
```

`fromIntegral x`: Convierte un número entero en un tipo numérico más general.

• Ejemplo:

haskell


 Copy code

```
let num = fromIntegral (5 :: Int)
print (num + 3.5) -- Salida: 8.5
```

`fst p`: Devuelve el primer elemento de una tupla.

• Ejemplo:

haskell


 Copy code

```
let tupla = (7, "Hola")
print (fst tupla) -- Salida: 7
```

`iterate f x`: Genera una lista infinita aplicando repetidamente la función `f` a `x`.

• Ejemplo:

haskell


 Copy code

```
let lista = take 5 $ iterate (* 2) 1
print lista -- Salida: [1, 2, 4, 8, 16]
```

`scanr f e xs`: Similar a `foldr`, pero devuelve una lista de valores acumulados.

• Ejemplo:

haskell


 Copy code

```
let lista = scanr (+) 0 [1, 2, 3, 4, 5]
print lista -- Salida: [15,14,12,9,5,0]
```

`takeWhile p xs`: Toma elementos de la lista `xs` mientras el predicado `p` se cumpla.

• Ejemplo:

haskell


 Copy code

```
let lista = takeWhile (< 5) [1, 2, 3, 4, 5, 6, 7]
print lista -- Salida: [1,2,3,4]
```

`uncurry f`: Convierte una función que toma tuplas en una que toma dos argumentos separados.

• Ejemplo:

haskell


 Copy code

```
let f (x, y) = x + y
let g = uncurry f
print (g 3 4) -- Salida: 7
```

`until p f x`: Aplica repetidamente la función `f` a `x` hasta que `p` sea verdadero.

• Ejemplo:

haskell


 Copy code

```
let condicion x = x > 10
let funcion x = x * 2
let resultado = until condicion funcion 2
print resultado -- Salida: 16
```

`zip xs ys`: Combina dos listas en una lista de tuplas.

• Ejemplo:

haskell


 Copy code

```
let lista1 = [1, 2, 3]
let lista2 = ["a", "b", "c"]
let lista_combinada = zip lista1 lista2
print lista_combinada -- Salida: [(1,"a"),(2,"b"),(3,"c")]
```

`zipWith f xs ys`: Combina dos listas aplicando la función `f` a los elementos correspondientes.

• Ejemplo:

haskell


 Copy code

```
let suma x y = x + y
let lista1 = [1, 2, 3]
let lista2 = [4, 5, 6]
let lista_sumada = zipWith suma lista1 lista2
print lista_sumada -- Salida: [5,7,9]
```

`all`: Esta función devuelve `True` si todos los elementos de la lista cumplen con el predicado proporcionado, y `False` en caso contrario.

• Ejemplo:

haskell

 Copy code


```
let lista1 = [2, 4, 6, 8, 10]
let lista2 = [2, 4, 6, 7, 8]

print (all (\x -> x `mod` 2 == 0) lista1) -- Salida: True (todos son pares)
print (all (\x -> x `mod` 2 == 0) lista2) -- Salida: False (hay un número impar)
```

`any`: Esta función devuelve `True` si al menos un elemento de la lista cumple con el predicado, y `False` en caso contrario.

• Ejemplo:

haskell

 Copy code


```
let lista1 = [1, 3, 5, 7, 8]
let lista2 = [1, 3, 5, 7, 9]

print (any (\x -> x `mod` 2 == 0) lista1) -- Salida: True (al menos un número par)
print (any (\x -> x `mod` 2 == 0) lista2) -- Salida: False (ningún número par)
```

`foldr1`: Aplica una función binaria a los elementos de una lista de derecha a izquierda, utilizando el primer elemento como el valor inicial.

• **Ejemplo:**

haskell


 Copy code

```
let lista = [1, 2, 3, 4, 5]
let resultado = foldr1 (\x y -> x + y) lista
print resultado -- Salida: 15 (1 + (2 + (3 + (4 + 5))))
```

`foldl1`: Aplica una función binaria a los elementos de una lista de izquierda a derecha, utilizando el primer elemento como el valor inicial.

• **Ejemplo:**

haskell

 Copy code

```
let lista = [1, 2, 3, 4, 5]
let resultado = foldl1 (\x y -> x + y) lista
print resultado -- Salida: 15 (((1 + 2) + 3) + 4) + 5
```