Grupo 11:
Roberto Omaña 06-39990
Leopoldo Pimentel 06-40095

Investigación

1.

- :
  - Función
    - Retorna el último comando

- :browse
  - Sintaxis
    - `:browse`[`!`] [[*]<*`module`*>] ...
  - Función
    - Muestra los identificadores exportados por el módulo *`module`*, el cual debe ser igualmente cargado en GCHi o ser un miembro de un paquete. Si *`module`* es omitido, el módulo más recientemente cargado es tomado.

- :info
  - Sintaxis
    - `:info` <*`name`*> ...
  - Función
    - Muestra información sobre el(los) nombre(s) dado(s). Por ejemplo, si *`name`* es una clase, entonces los métodos de la clase y sus tipos deben ser impresos; si *`name`* es un constructor de tipo, entonces su definición debe ser impresa; si *`name`* es una función, entonces su tipo debe ser impreso. Si name ha sido cargado desde un archivo fuente, entonces GHCi también mostrará la localización de su definición.

- :module
  - Sintaxis
    - `:module [+|-] [*]`mod_1 `... [*]`mod_n, `import` *`mod`*
  - Función
    - Establece o modifica el contexto actual para las instrucciones escritas en el intérprete. La forma `import` *`mod`* es equivalente a `:module +`*`mod`*.

- :load
  - Sintaxis
    - `:load` [*]<*`module`*> ...
  - Función
    - Carga recursivamente los *`module`*s especificados, y todos los modelos de quienes ellos dependen. Cada *`module`* debe ser un nombre de un módulo o un archivo, pero no el nombre de un módulo en un paquete.

      Todos los módulos previamente cargados, excepto módulos de paquetes, son ignorados. El nuevo conjunto de módulos es conocido como *target set*. Note que :load puede ser usado sin algún argumento para descargar todos los módulos actualmente cargados y sus vínculos.

      Normalmente código pre-compilado para un módulo será cargado si está disponible, o sino el módulo será compilado en código byte. Usando el prefijo * se forza a un módulo a ser cargado como código byte.

- :reload
  - Sintaxis
    - `:reload`
  - Función
    - Intenta recargar el *target set* actual si alguno de los módulos en el conjunto, o algún modulo dependiente ha cambiado. Note que esto puede que implique la carga de módulos nuevos, o dejar ir módulos que ya no son indirectamente requeridos por el *target set*.

- :type
  - Sintaxis
    - `:type` <*`expression`*>
  - Función
    - Infiere e imprime el tipo de *`expression`*, incluyendo cuantificadores explícitos forall para tipos polimórficos. La restricción del monomorfismo no está aplicada a la expresión durante la inferencia de tipo.

2.

- curry – Definido en 'Data.Tuple'
  - Firma
    - curry :: ((a, b) → c) → a → b → c
  - Utilidad
    - Convierte una función no currificada en una función currificada.
      Currificación es el proceso de transformar una función que toma múltiples argumentos en una función que toma sólo un argumento y devuelve otra función si algún otro argumento es aún necesario.

      ```
      f :: a -> b -> c
      ```
      es la **forma currificada** de
      ```
      g :: (a, b) -> c
      ```

      Ambas formas son igualmente expresivas, sin embargo la forma currificada es usualmente más conveniente ya que permite **aplicación parcial**.
      Aplicación parcial en Haskell involucra pasar menos que el número total de argumentos a una función que toma múltiples argumentos.

      Por ejemplo:

      ```
      add     :: Int -> Int -> Int
      add x y = x + y

      addOne = add 1
      ```

      En este ejemplo, addOne es el resultado de aplicar parcialmente add. Es una nueva función que toma un entero, le suma 1 y lo devuelve como resultado.

- uncurry – Definido en 'Data.Tuple'
  - Firma
    - uncurry :: (a → b → c)→ ((a, b) → c)
  - Utilidad
    - Convierte una función currificada a una función en pares.

- flip – Definido en 'GHC.Base'
  - Firma
    - flip :: (a → b → c) → b → a → c
  - Utilidad
    - flip f toma los (primeros) dos argumentos en el orden inverso de f.

      Por ejemplo:

      ```
      flip f x y  =  f y x
      ```

- null – Definido en 'GHC.List'
  - Firma
    - null :: [a] → Bool
  - Utilidad
    - Chequea si una lista es vacía. Devuelve True si lo es, False si no lo es.

- (!!) – Definido en 'GHC.List'
  - Firma
    - (!!) :: [a] → Int → a
  - Utilidad
    - Devuelve el elemento de la lista que se encuentra en la posición dada.

      Por ejemplo:
      (x:xs) (!!) 0 = x

- take – Definido en 'GHC.List'
  - Firma
    - take :: Int → [a] → [a]
  - Utilidad
    - take n, aplicado a una lista xs, devuelve el prefijo de xs de longitud n (primeros n elementos de xs), – o xs si n > longitud xs.

      Por ejemplo:
      take 2 [3,4,7] = [3,4]

- drop – Definido en 'GHC.List'
  - Firma
    - drop :: Int → [a] → [a]
  - Utilidad
    - drop n xs devuelve el sufijo de xs –el resto de la lista luego de los primeros n elementos– , o [] si n > longitud xs.

      Por ejemplo:
      drop 3 [1,2,3,4] = [4]

- elem – Definido en 'GHC.List'
  - Firma
    - elem :: (Eq a) => a → [a] → Bool
  - Utilidad
    - elem es el predicado de membresía a una lista, indica si el primer argumento es miembro de la lista dada.

      Por ejemplo:

      elem 4 [1,2,3] = False

- sqrt – Definido en 'GHC.Float'
  - Firma
    - sqrt :: Floating a => a -> a
  - Utilidad
    - Devuelve la raíz cuadrada del argumento.

      Por ejemplo:

      sqrt 25 = 5.0

Implementación

1.
```
nPrimos :: Int -> Int -> [Int]
nPrimos n m = [ x | x<-[n..m], esPrimo x]

esPrimo :: Int -> Bool
esPrimo n = [ x | x<-[1..n], mod n x == 0] == [1,n]
```

2.
```
emparejar :: [a] -> [b] -> [(a,b)]
emparejar = emparejarCon (,)

emparejarCon :: (a->b->c) -> [a]->[b]->[c]
emparejarCon z (x:xs) (y:ys) = z x y : emparejarCon z xs ys
emparejarCon _ _ _ =  []
```

3.
```
desemparejar :: [(a,b)] -> ([a],[b])
desemparejar []  =  ([], [])
desemparejar xs = (,) (desemparejarAux fst xs) (desemparejarAux snd xs)

desemparejarAux :: (a->b)->[a]->[b]
desemparejarAux _ [] = []
desemparejarAux f (x:xs) = f x : desemparejarAux f xs
```

4.

zip y unzip, respectivamente.

5.
```
separarMitad :: [a] -> ([a],[a])
separarMitad xs
    | mod longitud 2 == 0 = (take (div longitud 2) xs, drop (longitud - div longitud 2) xs)
    | otherwise = (take (div longitud 2) xs, drop (longitud - div longitud 2 - 1) xs)
        where longitud = length xs
```

6.
```
mergeSort :: Ord a => [a] -> [a]
mergeSort []    =  []
mergeSort [x]  =  [x]
mergeSort xs   =  mezcla (mergeSort ladoizquierdo) (mergeSort ladoderecho)
  where
   (ladoizquierdo, ladoderecho) = separarMitad xs
   mezcla [] xs = xs
   mezcla xs [] = xs
   mezcla (x:xs) (y:ys)
     | x <= y      = x : mezcla xs (y:ys)
     | otherwise = y : mezcla (x:xs) ys
```