Práctica 1

Grupo 2

Oscar García de Lara

Jorge Cifuentes

Práctica 1A

Apartado A:

La expresión

```
print ((coger 2 (repetir 3)) ++ [5,5])
```

Coge los 2 primeros elementos de una lista de 3 (evaluación perezosa), y lo concatena a [5,5].

Algunos errores de compilación:

```
print ((coger 2 repetir 3) ++ [5,5])
```

```
    · Couldn't match expected type ‘Integer -> [Integer]’
                  with actual type ‘[a0]’
    · The function ‘coger’ is applied to three arguments,
```

Al eliminar el paréntesis que rodea a repetir 3, el compilador interpreta la palabra repetir como un argumento de coger.

Si cambiamos en la definición de coger:

```
coger n x:xs = x : coger (n-1) xs
```

```
Funciones.hs:9:1: error: Parse error in pattern: coger
```

La lista x:xs necesita estar entre paréntesis en el patrón.

Si escribimos:

```
    print coger 5 [1,2,3,4]
```

```
Principal.hs:8:5: error:
    · Couldn't match expected type ‘Integer -> [Integer] -> IO b’
                  with actual type ‘IO ()’
    · The function ‘print’ is applied to three arguments,
      but its type ‘(Integer -> [a0] -> [a0]) -> IO ()’ has only one
      In a stmt of a 'do' block: print coger 5 [1, 2, 3, 4]
      In the expression:
        do { print ((coger 2 (repetir 3)) ++ [5, 5]);
             print (coger 5 ([1, 2, ....] ++ repetir 5));
             print coger 5 [1, 2, ....] }
```

```
    · Relevant bindings include
        main :: IO b (bound at Principal.hs:4:1)
```

Ya que necesitamos poner entre paréntesis la llamada a coger, ya que en caso contrario , interpreta la llamada a print sobre tres parámetros (por su asociatividad).

Apartado B:

Hemos leído la documentación que proporciona Haskell.org, a partir de ahí hemos seleccionado los comandos que hemos considerado mas importante.

- **:type** *exp* sirve para mostrar el tipo/clase, que tiene una expresión.

:type "hola"

- **:info** *class* sirve para mostrar la información de una clase.

:info Integer

- **:load** *modulo* se usa para cargar los módulos, *.hs, que hemos creado, para poder compilarlo y ejecutarlo.

:load main.hs

- **:main** *arg1..argn* se utiliza cuando necesitas pasar por argumento a un programa que has cargado anteriormente.

:main 1 "hola"

- **:reload** sirve para recargar todos los módulos, es útil cuando se han modificado varios a la vez.

```
> :i repetir

repetir :: a -> [a]     -- Defined at Funciones.hs:4:1

> :t repetir

repetir :: a -> [a]

> :i coger

coger :: Integer -> [a] -> [a]      -- Defined at Funciones.hs:7:1

> :t coger

coger :: Integer -> [a] -> [a]
```

Como se puede observar la diferencia es que con :info se muestra donde está definida lo que puede ser muy útil cuando trabajas con varios ficheros.

Apartado C:

Definición de take en prelude:

```
take   :: Int -> [a] -> [a]
take n _    | n <= 0 =  []
take _ []  =  []
take n (x:xs)  =  x : take (n-1) xs
```

Definición de coger en Funciones.hs:

```
coger :: Integer -> [a] -> [a]
coger n _ | n <= 0 = []
coger _ [] = []
coger n (x:xs) = x : coger (n-1) xs
```

La única diferencia es que take recibe un Int y coger un Integer, el funcionamiento en cada caso es igual para ambas, con los casos base para listas vacías y el caso recursivo.

Definición de repeat en prelude:

```
repeat :: a -> [a]
repeat x = xs where xs = x : xs
```

Definición de repetir en Funciones.hs:

```
repetir :: a -> [a]
repetir x = x:(repetir x)
```

La función repetir consigue la lista infinita para un caso 'x', creando una lista formada por ese 'x' concatenado de una llamada recursiva a repetir x, mientras que la función repeat, de forma parecida, para el caso 'x' devuelve una lista 'xs' formada por x concatenado a xs.

Apartado D:

Se podría definir coger y repetir a partir de take y repeat haciendo que la llamada recursiva de las primeras invoque a las segundas, pero no al revés, ya que no disponemos el código de esas funciones.

- FromInteger: Sirve para transformar un Integer a Num

```
i::Integer

i=2
```

- toInteger: Sirve para transformar un Int a Integer

```
i::Int

i=2

coger (toInteger i) [2,3,45]
```

- maxBound: Indica el valor maximo, en este caso, del tipo Integral

`maxBoound::Int`

- Integral: Clase que engloba a Int e Integer

- Bounded: Clase que sirve para poner limites a los tipos.

Apartado E:

Hemos creado una función alternativa coger' para probar la clausula error que lance un mensaje de error cuando se intenta coger mas elementos de los que tiene la lista.

Características más relevantes del debugging en Haskell:

Uso de Stack Trace: se puede investigar el trace cuando se lanza una excepción. La librería Debug.Trace contiene funciones útiles para manipular las dichas trazas.

`trace ("llamada a la funcion f " ++ show x) (f x)` muestra el mensaje y después ejecuta la función.

Debugger de GHCi: se pueden depurar módulos interpretados de esta manera, poniendo breakpoints, inspeccionando el estado actual del programa.

`Funcion> :break 3` pone un breakpoint en la linea 3

Comando importantes: :list (muestra el código cercano al breakpoint actual), :step (para ejecutar el código paso a paso).

Análisis offline de trazas: Haskell Tracer HAT y Hoed

Otras librerías útiles: Test.QuickCheck para automatizar tests, librerías Safe (sin excepciones).

Apartado F:

Hemos evaluado dos versiones online http://tryhaskell.org/ y http://codepad.org/. Las ventajas es que pueden solucionarte alguna duda básica rápidamente, sobretodo cabeceras de funciones sin tener que instalar nada. Las desventajas es, como todo lo online, que necesitas tener acceso, también pierdes mucha potencia del interprete como los comandos o el hecho de cargar módulos. Las ventajas de WinGHCI son las dichas antes como desventajas y la única desventaja es que necesita descarga e instalación.

Apartado G:

Hemos creado la función comprobar sobre una lista de Num y un predicado de Num, de manera que compruebe si todos los elementos cumplen el predicado (usando la función all).

En Principal.hs hemos usado la función sobre la lista fija ['a'..'z'].

Apartado H:

```
cogerNoCurry :: forall a. (Int, [a]) -> [a]
cogerNoCurry = uncurry take
```

```
>cogerNoCurry (2,[3,4,5,6])
< [3,4]
```

Co se ve en el ejemplo hemos descurrificado la función take creado otra propia que permite pasarle los argumentos como tuplas. Y con curry la currificamos de nuevo.

```
cogerCurry :: forall a. Int -> [a] -> [a]
cogerCurry = curry cogerNoCurry
```

```
> cogerCurry 2 [3,4,5]
< [3,4]
```

Apartado I:

Hemos definido las funciones curry y uncurry para tríos de manera similar a como están definidas las originales en Data.Tuples, sirviéndonos de funciones fst', snd' y trd' auxiliares. Después, para currificar y descurrificar una función prueba, hacemos una llamada parcializada a curryTuplas/uncurryTuplas.

Apartado J:

Hemos creado tres funciones: listaParejas que devuelve todas las tuplas (x,y) con x menor que y (y empieza en valor x), listaTrios que para una tupla devuelve una lista de trios (x, y, z) con el valor de z comenzando en y, y la propia función triángulos que filtra la dicha condición usando funciones de orden superior.

Práctica 1B

Apartado A:

En este apartado hemos implementado el problema de manera monomórfica (con data) y con una partida como una tupla (con type, suponiendo que si es una bebida única, la cantidad es 1). Después hemos definido las funciones de acuerdo a dichos tipos Double e Integer, usando en casi todas recursividad (ver código), y finalmente un main de pruebas.

Apartado B:

Hemos cambiado la definición de Partida de una tupla a un Data con dos constructores ParUnit y ParMult, y las funciones de acceso bebida y cantidad para ambos (ya que los dos campos van a ser usados, igual que en el apartado A). Los cambios en el código han sido mínimos, solo en funciones que accedan a los campos de Partida, como precioPartida. También hemos añadido el operador (+).

Apartado C:

En este apartado no hemos cambiado las definiciones de data y type ya que no era necesario. Hemos cambiado todas las funciones (ver código) para que usen funciones de orden superior y listas por comprensión (sum, filter, map, etc...)

También hemos hecho que Bebida y Partida sean instancias de Ord para así poder ordenarlas, algo necesario en la función de eliminar repeticiones para poder usar un sort.

Apartado D:.

Para conseguir que sea polimórfico hemos tenido que añadir restricciones de clase (Integral y Fractional) a los data de Bebida y Partida, así como hacer un type Factura sobre tres parámetros polimórficos a b c.

Una vez hecho esto hemos añadido estas restricciones a las funciones de acceso, y después a las instanciaciones y a las funciones variadas (ver código).

Apartado E:

Lo primero que hemos hecho ha sido definir una clase Preciable con una única cabecera de función precio , que dado un elemento Preciable aplicado sobre un Fractional b, devuelve un Fractional b. Después hemos instanciado Bebida sobre un tipo polimórfico y Partida sobre dos, haciendo que la función "Precio" de Preciable llame a precioBebida o precioPartida respectivamente.

Apartado F:

Primero hemos definido una función factura tal y como se pide de manera que dada una factura polimórfica y una bebida devuelva la cantidad de esa bebida (un integral del tipo a).