

PRÁCTICA 1A

Aitor Melero Picón, Arturo Morcillo Penares
UAM Ampliación de Programación

PRÁCTICA 1a:

a) A partir de las 2 definiciones de `fact7` y `fact8` de la diapositiva 14 del fichero 02-introduccion-pf.ppt, crear el módulo “`ApartadoA.Func.Factoriales`”, donde se implementen. A continuación, en el módulo “`ApartadoA.Ejemplos`” implementar varios ejemplos que muestren en pantalla los resultados de invocaciones a las 2 funciones. Utilizar el editor `notepad++` para este apartado.

Para el módulo creado “`ApartadoA.Func.Factoriales`” simplemente, como pide el enunciado, hemos copiado las implementaciones de las funciones “`fact7`” y “`fact8`”. Ambas funciones calculan el factorial de un número.

Aquí el código del fichero donde se implementa:

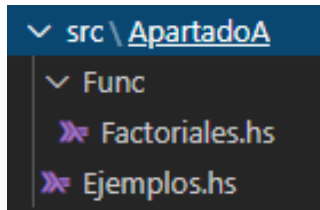
```
Archivo  Editar  Buscar  Vista  Codificación  Lenguaje  Configuración  Herramientas  Macro  Ejecutar  Plugins  Ventana  ?
Factores.hs
1  {--
2      Fichero: Factoriales.hs
3      Autores: Aitor Melero, Arturo Morcillo
4      Fecha: 24/09/2020
5      Funcionalidad: Modulo en el que se definen diferentes
6                      maneras de implementar el factorial.
7  --}
8
9  module ApartadoA.Func.Factoriales where
10
11  -- funcion de pliegue
12  fact7 :: Integer -> Integer
13  fact7 n =
14      foldr (*) 1 [1..n]
15  -- composicion de funciones
16  fact8 = product . enumFromTo 1
```

Para el módulo creado “`ApartadoA.Ejemplos`”, hemos importado el módulo anterior (para poder usar las funciones “`fact7`” y “`fact8`”) y hemos creado una función “`main`” que imprime por pantalla los resultados a varias llamadas de ambas funciones.

Aquí el código del fichero donde se implementa:

```
C:\Users\melero\OneDrive\Escritorio\Curso_20-21\Primer_Cuatrimestre\APROD\Practicas\Entrega_Tercer_ApartadoA\Ejemplos.hs - Notepad++
Factores.hs
1  {--
2      Fichero: Ejemplos.hs
3      Autores: Aitor Melero, Arturo Morcillo
4      Fecha: 24/09/2020
5      Funcionalidad: Modulo que llama a las diferentes
6                      funciones de Factoriales.hs.
7  --}
8
9  module ApartadoA.Ejemplos where
10
11  -- importamos las funciones de factoriales
12  import ApartadoA.Func.Factoriales
13
14  -- imprimos ejemplos de factoriales
15  main :: IO ()
16  main = do
17      print ("El resultado de (fact7 4) es: " ++ (show (fact7 4)))
18      print ("El resultado de (fact7 5) es: " ++ (show (fact7 5)))
19      print ("El resultado de (fact7 6) es: " ++ (show (fact7 6)))
20      print ("El resultado de (fact8 4) es: " ++ (show (fact8 4)))
21      print ("El resultado de (fact8 5) es: " ++ (show (fact8 5)))
22      print ("El resultado de (fact8 6) es: " ++ (show (fact8 6)))
23
24  {--
25      Resultado esperado:
26
27      "El resultado de (fact7) 4 es: 24"
28      "El resultado de (fact7) 5 es: 120"
29      "El resultado de (fact7) 6 es: 720"
30      "El resultado de (fact8) 4 es: 24"
31      "El resultado de (fact8) 5 es: 120"
32      "El resultado de (fact8) 6 es: 720"
33  --}
34  --}
```

Cabe destacar que hemos creado una carpeta “src” aparte del proyecto stack creado más adelante. La estructura de dicha carpeta es la siguiente:



b) Compilar y ejecutar el código del apartado a) usando el comando ghci en una consola, explicando en detalle algunos errores de compilación y los resultados de ejecución para varios casos. Es conveniente ejecutar el comando ghci en una carpeta src.

Usando una consola de comandos, nos hemos dirigido a la carpeta src personal que hemos creado y donde hemos guardado el código del apartado anterior. Una vez estábamos en dicha carpeta, hemos usado el comando “ghci Ejemplos.hs Func\Factoriales.hs” para compilar el código. Aquí el resultado obtenido:

```
C:\Users\mele9\OneDrive\Escritorio\Curso_20-21\Primer_Cuatrimestre\APROG\Practicas\Entrega_1\src\ApartadoA>ghci Ejemplos.hs Func\Factoriales.hs
GHCi, version 8.0.2: http://www.haskell.org/ghc/ :? for help
[1 of 2] Compiling ApartadoA.Func.Factoriales ( Func\Factoriales.hs, interpreted )
Func\Factoriales.hs:14:1: warning: [-Wtabs]
  Tab character found here, and in one further location.
  Please use spaces instead.
[2 of 2] Compiling ApartadoA.Ejemplos ( Ejemplos.hs, interpreted )
Ejemplos.hs:17:1: warning: [-Wtabs]
  Tab character found here, and in five further locations.
  Please use spaces instead.
Ok, modules loaded: ApartadoA.Ejemplos, ApartadoA.Func.Factoriales.
*ApartadoA.Ejemplos>
```

Como podemos observar en la imagen anterior, el código se ha compilado correctamente. Podemos ver que han saltado 2 warnings, ambos por tabulaciones en el código. No les hemos dado importancia a estos warnings, sobre todo al de “Factoriales.hs”, ya que hemos copiado la función “fact7” tal cual aparecía en el pdf nombrado en el enunciado del apartado anterior y no queríamos modificar nada de dicha función. Aún así, estos warnings son de fácil arreglo.

Una vez compilado el código, probamos directamente la función “main” (“Ejemplos.hs”) para ver si los resultados son los esperados:

```
*ApartadoA.Ejemplos> main
"El resultado de (fact7 4) es: 24"
"El resultado de (fact7 5) es: 120"
"El resultado de (fact7 6) es: 720"
"El resultado de (fact8 4) es: 24"
"El resultado de (fact8 5) es: 120"
"El resultado de (fact8 6) es: 720"
```

Como podemos apreciar, los resultados son los esperados para nuestros ejemplos (se puede ver en el comentario final de “Ejemplos.hs”). Las funciones se ejecutan sin problemas.

Hay que mencionar, que antes de probar la función “main”, compilamos solo el código de “Factoriales.hs” para probar la función “fact7” y “fact8” directamente desde consola. Probamos los ejemplos anteriores por separado y funcionaron a la primera.

En conclusión, no tuvimos problemas con la compilación y comprobación del código ya que era sencillo de implementar y ya habíamos practicado el compilado y la ejecución de código Haskell con los ejercicios de tipos realizados en clase.

c) Repetir el apartado anterior usando WinGHCi, explicando las diferencias observadas.

Una vez abierto WinGHCi, nos posicionamos en la carpeta src correspondiente usando el comando “:cd <directorio-src>”. Una vez posicionados en dicho directorio, simplemente cargamos con el comando “:load” los dos módulos implementados. Este fue el resultado:

```
Prelude> :load Ejemplos.hs Func\Factoriales.hs

Func\Factoriales.hs:14:1: warning: [-Wtabs]
  Tab character found here, and in one further location.
  Please use spaces instead.

Ejemplos.hs:17:1: warning: [-Wtabs]
  Tab character found here, and in five further locations.
  Please use spaces instead.
[1 of 2] Compiling ApartadoA.Func.Factoriales ( Func\Factoriales.hs, interpreted )
[2 of 2] Compiling ApartadoA.Ejemplos ( Ejemplos.hs, interpreted )
Ok, modules loaded: ApartadoA.Ejemplos, ApartadoA.Func.Factoriales.
```

No se observan diferencias destacables con respecto a los resultados de la compilación anterior realizada en la consola de comandos del sistema. Aunque si es cierto, que en el apartado anterior hemos usado el comando “ghci” seguido de los dos ficheros con código y en este apartado, una vez posicionados en la carpeta correspondiente, hemos usado el comando “load” para compilar, es decir, hemos cargado los módulos implementados.

Después de compilar el código, hemos llamado a la función “fact7”, “fact8” y “main” para probar la ejecución del código. Estos han sido los resultados:

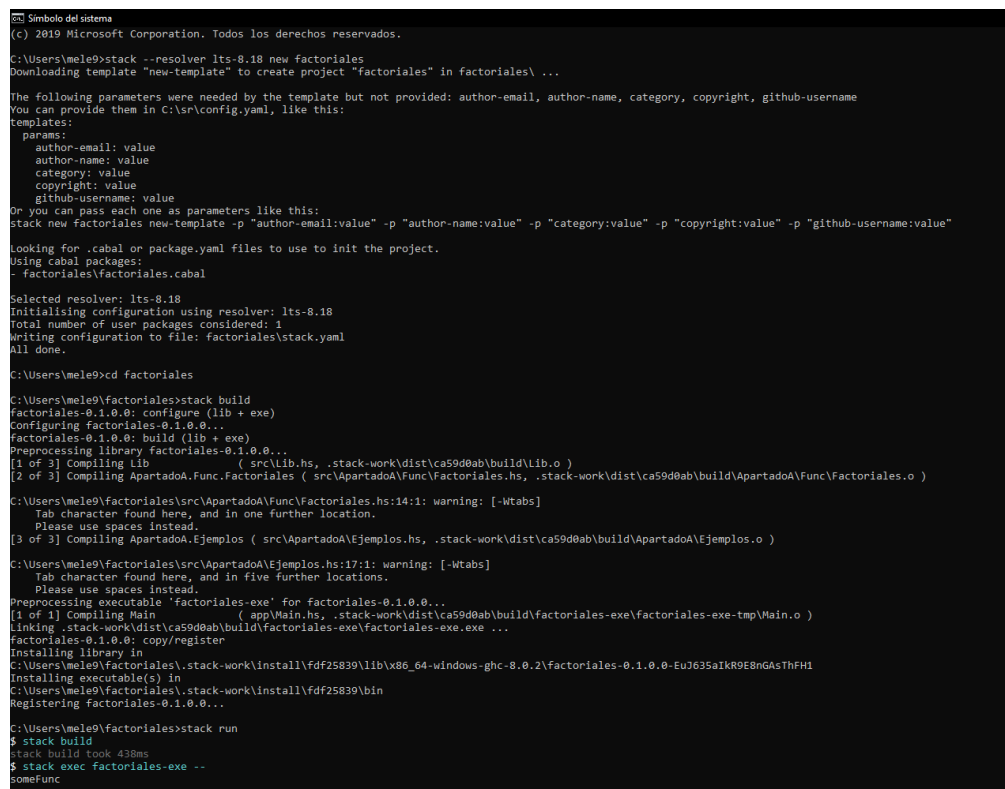
```
Prelude> fact7 4
24
Prelude> fact7 5
120
Prelude> fact7 6
720
Prelude> fact8 4
24
Prelude> fact8 5
120
Prelude> fact8 6
720
Prelude> main
"El resultado de (fact7 4) es: 24"
"El resultado de (fact7 5) es: 120"
"El resultado de (fact7 6) es: 720"
"El resultado de (fact8 4) es: 24"
"El resultado de (fact8 5) es: 120"
"El resultado de (fact8 6) es: 720"
```

Los resultados son correctos. Desde “Prelude>” podemos ejecutar las funciones “fact7”, “fact8” y “main” porque tenemos cargados los dos módulos en la propia consola. En el ejercicio anterior, al usar ghci con los dos ficheros de código, nos hemos posicionado directamente en el módulo “ApartadoA.Ejemplos”, aunque también podíamos ejecutar “fact7” y “fact8” aparte de “main” por el “import ApartadoA.Func.Factoriales” situado dentro del módulo “ApartadoA.Ejemplos”.

d) Repetir el apartado b) usando una consola y stack, y creando el proyecto factoriales basado en el resolver lts-8.18. El proyecto se crea mediante la instrucción “stack --resolver lts-8.18 new factoriales”, se compila mediante “stack build” y se ejecuta mediante “stack run”, ejecutándolos en la carpeta factoriales.

Tenemos que decir que la primera vez que creamos un proyecto stack (con el comando “stack –resolver” del enunciado) no tuvimos problemas hasta la hora de compilarlo con “stack build”. El fallo simplemente era que la carpeta donde habíamos creado el proyecto stack tenía un path demasiado largo, algo que no tiene nada que ver con el objetivo de la práctica pero que si nos quitó mucho tiempo.

Creamos un nuevo proyecto stack en una carpeta, con un path más corto, tal como indicaba el enunciado, copiamos el código del apartado a en la carpeta src, compilamos con “stack build” dentro de la carpeta “factoriales” y lo ejecutamos, también dentro de la carpeta “factoriales”, con el comando “stack run”:



```
(C) Símbolo del sistema
(C) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Users\mele9>stack --resolver lts-8.18 new factoriales
Downloading template "new-template" to create project "factoriales" in factoriales\...

The following parameters were needed by the template but not provided: author-email, author-name, category, copyright, github-username
You can provide them in C:\sr\config.yaml, like this:
templates:
  params:
    author-email: value
    author-name: value
    category: value
    copyright: value
    github-username: value
Or you can pass each one as parameters like this:
stack new factoriales new-template -p "author-email:value" -p "author-name:value" -p "category:value" -p "copyright:value" -p "github-username:value"

Looking for .cabal or package.yaml files to use to init the project.
Using cabal packages:
- factoriales\factoriales.cabal

Selected resolver: lts-8.18
Initialising configuration using resolver: lts-8.18
Total number of user packages considered: 1
Writing configuration to file: factoriales\stack.yaml
All done.

C:\Users\mele9>cd factoriales
C:\Users\mele9\factoriales>stack build
factoriales-0.1.0.0: configure (lib + exe)
Configuring factoriales-0.1.0.0...
factoriales-0.1.0.0: build (lib + exe)
Preprocessing library factoriales-0.1.0.0...
[1 of 3] Compiling Lib (src\Lib.hs, .stack-work\dist\ca59d0ab\build\Lib.o)
[2 of 3] Compiling ApartadoA.Func.Factoriales (src\ApartadoA\Func\Factoriales.hs, .stack-work\dist\ca59d0ab\build\ApartadoA\Func\Factoriales.o)
C:\Users\mele9\factoriales\src\ApartadoA\Func\Factoriales.hs:14:1: warning: [-Wtabs]
    Tab character found here, and in one further location.
    Please use spaces instead.
[3 of 3] Compiling ApartadoA.Ejemplos (src\ApartadoA\Ejemplos.hs, .stack-work\dist\ca59d0ab\build\ApartadoA\Ejemplos.o)
C:\Users\mele9\factoriales\src\ApartadoA\Ejemplos.hs:17:1: warning: [-Wtabs]
    Tab character found here, and in five further locations.
    Please use spaces instead.
Preprocessing executable 'factoriales-exe' for factoriales-0.1.0.0...
[1 of 1] Compiling Main (app\Main.hs, .stack-work\dist\ca59d0ab\build\factoriales-exe\factoriales-exe-tmp\Main.o)
Linking .stack-work\dist\ca59d0ab\build\factoriales-exe\factoriales-exe.exe ...
factoriales-0.1.0.0: copy/register
Installing library in
C:\Users\mele9\factoriales\stack-work\install\fd25839\lib\x86_64-windows-ghc-8.0.2\factoriales-0.1.0.0-Eu7635aIkR9E8nGAsThF1
Installing executable(s) in
C:\Users\mele9\factoriales\stack-work\install\fd25839\bin
Registering factoriales-0.1.0.0...

C:\Users\mele9\factoriales>stack run
$ stack build
stack build took 438ms
$ stack exec factoriales-exe --
someFunc
```

Como se puede ver en la imagen anterior, no tuvimos problemas con la creación del proyecto stack y, después de copiar el código en la carpeta src, no tuvimos errores con la compilación y ejecución del código. No tener errores con la compilación y ejecución del código no quiere decir que estuviese correcto lo que hicimos, ya que como se puede ver, la ejecución del programa simplemente imprimía por pantalla “someFunc” en lugar de nuestros ejemplos de “Ejemplos.hs”.

Nos pusimos a investigar y observamos que el proyecto stack estaba compuesto por una carpeta app (situada a la misma altura que la carpeta src) la cual tenía un fichero llamado “Main.hs”. Abrimos el código y vimos que había una función “main” que llamaba a una función “someFunc” importada desde el módulo “Lib”. El módulo “Lib” se encontraba en el fichero “Lib.hs” dentro

de la carpeta src del proyecto. La función “someFunc” que se llamaba desde el módulo “Main” simplemente imprimía la cadena “someFunc” que era justo lo que veíamos por la consola al ejecutar el proyecto. Se nos ocurrió entonces cambiar el nombre de nuestra función “main” del módulo de “ApartadoA.Ejemplos” a “ejemplos” (para tener solo un main y por lo tanto tener un código más limpio) e importar el módulo “ApartadoA.Ejemplos” desde el módulo “Lib” para que la función “someFunc” en lugar de imprimir por pantalla una cadena, llamase a nuestra función encargada de imprimir los ejemplos de factoriales. Todo este código se puede ver aquí:

```
1 module Main where
2
3 import Lib
4
5 main :: IO ()
6 main = someFunc
```

```
1 module Lib
2   ( someFunc
3   ) where
4
5 import ApartadoA.Ejemplos
6
7 someFunc :: IO ()
8 --someFunc = putStrLn "someFunc"
9 someFunc = ejemplos
```

```
1 {--
2   Fichero: Ejemplos.hs
3   Autores: Aitor Melero, Arturo Morcillo
4   Fecha: 24/09/2020
5   Funcionalidad: Modulo que llama a las diferentes
6                 funciones de Factoriales.hs.
7 --}
8
9 module ApartadoA.Ejemplos where
10
11 -- importamos las funciones de factoriales
12 import ApartadoA.Func.Factoriales
13
14 -- imprimos ejemplos de factoriales
15 ejemplos :: IO ()
16 ejemplos = do
17   print ("El resultado de (fact7 4) es: " ++ (show (fact7 4)))
18   print ("El resultado de (fact7 5) es: " ++ (show (fact7 5)))
19   print ("El resultado de (fact7 6) es: " ++ (show (fact7 6)))
20   print ("El resultado de (fact8 4) es: " ++ (show (fact8 4)))
21   print ("El resultado de (fact8 5) es: " ++ (show (fact8 5)))
22   print ("El resultado de (fact8 6) es: " ++ (show (fact8 6)))
23
24 {--
25   Resultado esperado:
26
27   "El resultado de (fact7) 4 es: 24"
28   "El resultado de (fact7) 5 es: 120"
29   "El resultado de (fact7) 6 es: 720"
30   "El resultado de (fact8) 4 es: 24"
31   "El resultado de (fact8) 5 es: 120"
32   "El resultado de (fact8) 6 es: 720"
33 --}
```

Una vez realizado los cambios recientemente mencionados, compilamos de nuevo el código con el comando “stack build” y lo ejecutamos con el comando “stack run”. Esta vez el resultado de la ejecución fue el esperado:

```
C:\Users\mele9\APROG\Practicas_GITHUB\APROG\Practica_1\stack\factoriales>stack build
factoriales-0.1.0.0: build (lib + exe)
Preprocessing library factoriales-0.1.0.0...
[3 of 3] Compiling Lib      ( src\lib.hs, .stack-work\dist\ca59d0ab\build\lib.o )
Preprocessing executable 'factoriales-exe' for factoriales-0.1.0.0...
[1 of 1] Compiling Main      ( app\Main.hs, .stack-work\dist\ca59d0ab\build\factoriales-exe\factoriales-exe-tmp\Main.o )
Linking .stack-work\dist\ca59d0ab\build\factoriales-exe\factoriales-exe.exe ...
factoriales-0.1.0.0: copy/register
Installing library in
C:\Users\mele9\APROG\Practicas_GITHUB\APROG\Practica_1\stack\factoriales\stack-work\install\fd25839\lib\x86_64-windows-ghc-8.0.2\factoriales-0.1.0.0-EuJ635aIkR9E8nGAsThFH1
Installing executable(s) in
C:\Users\mele9\APROG\Practicas_GITHUB\APROG\Practica_1\stack\factoriales\stack-work\install\fd25839\bin
Registering factoriales-0.1.0.0...
C:\Users\mele9\APROG\Practicas_GITHUB\APROG\Practica_1\stack\factoriales>stack run
$ stack build
stack build took 729ms
$ stack exec factoriales-exe --
"El resultado de (fact7 4) es: 24"
"El resultado de (fact7 5) es: 120"
"El resultado de (fact7 6) es: 720"
"El resultado de (fact8 4) es: 24"
"El resultado de (fact8 5) es: 120"
"El resultado de (fact8 6) es: 720"
```

e) Repetir el apartado b) usando el editor Visual Studio Code (VSC) y el proyecto stack del apartado anterior. Se comienza abriendo el proyecto mediante “File/Open Folder...”.

Para este apartado, abrimos la carpeta con el proyecto stack desde *Visual Studio Code* y pulsamos en el botón de Stack Build y de Stack Run de la parte inferior para compilar y ejecutar el código respectivamente. Poco que destacar ya que los resultados son los mismos que los del apartado anterior por el hecho de que estamos haciendo lo mismo, la diferencia simplemente es el hacerlo más rápido con *Visual Studio Code*:

The first screenshot shows the Visual Studio Code terminal window with the following output:

```
PS C:\Users\mele9\APROG\Practicas_GITHUB> cd "C:\Users\mele9\APROG\Practicas_GITHUB\APROG\Practica_1\stack\factoriales/"
PS C:\Users\mele9\APROG\Practicas_GITHUB\APROG\Practica_1\stack\factoriales> stack build --fast
factoriales-0.1.0.0: unregistering (local file changes: app\Main.hs)
Configuring factoriales-0.1.0.0...
factoriales-0.1.0.0: build (lib + exe)
Preprocessing library factoriales-0.1.0.0...
Preprocessing executable 'factoriales-exe' for factoriales-0.1.0.0...
[1 of 1] Compiling Main      ( app\Main.hs, .stack-work\dist\ca59d0ab\build\factoriales-exe\factoriales-exe-tmp\Main.o )
Linking .stack-work\dist\ca59d0ab\build\factoriales-exe\factoriales-exe.exe ...
factoriales-0.1.0.0: copy/register
Installing library in
C:\Users\mele9\APROG\Practicas_GITHUB\APROG\Practica_1\stack\factoriales\stack-work\install\fd25839\lib\x86_64-windows-ghc-8.0.2\factoriales-0.1.0.0-EuJ635aIkR9E8nGAsThFH1
Installing executable(s) in
C:\Users\mele9\APROG\Practicas_GITHUB\APROG\Practica_1\stack\factoriales\stack-work\install\fd25839\bin
Registering factoriales-0.1.0.0...
PS C:\Users\mele9\APROG\Practicas_GITHUB\APROG\Practica_1\stack\factoriales>

The second screenshot shows the Windows PowerShell terminal window with the following output:
```

f) Examinar los diferentes comandos disponibles de GHCi (:type, :info, ...) tanto en la documentación de GHCi como los mostrables mediante el comando :?. Poner ejemplos representativos de uso de algunos de los comandos y parámetros más importantes. Además, mostrar el uso de “:type” o “:info” para diferentes identificadores usados en el código del apartado a), o relacionados con ellos, explicando su significado.

El comando “:type” nos permite conocer el tipo de una expresión. Así, si ejecutamos en ghci “:type 4” nos dice que 4 está definido como “Num t => t”.

```
Prelude> :type 4
4 :: Num t => t
```

Por otro lado, el comando “:info” nos da información sobre el nombre introducido. Si es una clase nos da sus métodos y tipos, si es un constructor su definición y si es una función su tipo.

Ahora podemos utilizar estos dos comandos para analizar el código del apartado a).

```
Please use spaces instead.
Ok, modules loaded: ApartadoA.Func.Factoriales.
*ApartadoA.Func.Factoriales> :type fact7
fact7 :: Integer -> Integer
*ApartadoA.Func.Factoriales> :type fact8
fact8 :: Integer -> Integer
*ApartadoA.Func.Factoriales> :info fact7
fact7 :: Integer -> Integer    -- Defined at Factoriales.hs:13:1
*ApartadoA.Func.Factoriales> :info fact8
fact8 :: Integer -> Integer    -- Defined at Factoriales.hs:16:1
*ApartadoA.Func.Factoriales> 
```

Al haberlo utilizado en funciones obtenemos resultados similares, la diferencia es que “info” nos muestra donde está definida la función y “type” no nos lo indica.

Otro ejemplo de uso es si realizamos un “:info” de “Integer”, donde podemos ver como obtenemos todas sus definiciones.

```
*ApartadoA.Func.Factoriales> :info Integer
data Integer
  = Integer-gmp-1.0.0.1:GHC.Integer.Type.S# !GHC.Prim.Int#
  | Integer-gmp-1.0.0.1:GHC.Integer.Type.Jp# {-# UNPACK #-}!Integer-gmp-1.0.0.1:GHC.Integer.Type.BigNat
  | Integer-gmp-1.0.0.1:GHC.Integer.Type.Jm# {-# UNPACK #-}!Integer-gmp-1.0.0.1:GHC.Integer.Type.BigNat
  -- Defined in 'integer-gmp-1.0.0.1:GHC.Integer.Type'
instance Enum Integer -- Defined in 'GHC.Enum'
instance Eq Integer
  -- Defined in 'integer-gmp-1.0.0.1:GHC.Integer.Type'
instance Integral Integer -- Defined in 'GHC.Real'
instance Num Integer -- Defined in 'GHC.Num'
instance Ord Integer
  -- Defined in 'integer-gmp-1.0.0.1:GHC.Integer.Type'
instance Read Integer -- Defined in 'GHC.Read'
instance Real Integer -- Defined in 'GHC.Real'
instance Show Integer -- Defined in 'GHC.Show'
*ApartadoA.Func.Factoriales> 
```

Por último, destacar ciertas funcionalidades de ghci que hemos encontrado al utilizar el comando “:?” y nos parecen de real utilidad.

- Los comandos para hacer debug entre los que destacamos “:break”, “:continue” y “:delete”.

```
-- Commands for debugging:

:abandon          at a breakpoint, abandon current computation
:back [<n>]        go back in the history N steps (after :trace)
:break [<mod>] [<l>] [<col>] set a breakpoint at the specified location
:break <name>      set a breakpoint on the specified function
:continue         resume after a breakpoint
:delete <number>   delete the specified breakpoint
:delete *         delete all breakpoints
:force <expr>      print <expr>, forcing unevaluated parts
:forward [<n>]     go forward in the history N step s(after :back)
:history [<n>]     after :trace, show the execution history
:list            show the source code around current breakpoint
:list <identifier> show the source code for <identifier>
:list [<module>] [<line>] show the source code around line number <line>
:print [<name> ...] show a value without forcing its computation
:sprint [<name> ...] simplified version of :print
:step            single-step after stopping at a breakpoint
:step <expr>      single-step into <expr>
:steplocal        single-step within the current top-level binding
:stepmodule       single-step restricted to the current module
:trace           trace after stopping at a breakpoint
:trace <expr>     evaluate <expr> with tracing on (see :history)
```


- “:set” para fijar opciones. Por ejemplo “:set -fwarn-missing-signatures”.
- “:main” para iniciar la función “main”.
- “:def” para definir comandos. Por ejemplo: “:def date date”. Al ejecutar “:date” se muestra la fecha.

g) Existen funciones nativas semejantes en cuanto a su propósito, con los siguientes nombres `enumFrom`, `enumFromThen`, `enumFromThenTo` y `enumFromTo`. Explicar en detalle el sentido de sus tipos y, con ejemplos, el comportamiento de dichas funciones y la relación que existen entre ellas, particularizándolas para algunos tipos ejemplo de la clase `Enum`. Explicar también qué se obtendría al realizar parcializaciones sobre las mismas con varios ejemplos.

Para entender estas funciones primero tenemos que comprender la clase `Enum`:

```
Prelude> :info Enum
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
  {-# MINIMAL toEnum, fromEnum #-}
  -- Defined in `GHC.Enum'

instance Enum Word -- Defined in `GHC.Enum'
instance Enum Ordering -- Defined in `GHC.Enum'
instance Enum Integer -- Defined in `GHC.Enum'
instance Enum Int -- Defined in `GHC.Enum'
instance Enum Char -- Defined in `GHC.Enum'
instance Enum Bool -- Defined in `GHC.Enum'
instance Enum () -- Defined in `GHC.Enum'
instance Enum Float -- Defined in `GHC.Float'
instance Enum Double -- Defined in `GHC.Float'
Prelude>
```

Con el comando `info` podemos ver que se trata de una clase que define operaciones en tipos ordenados.

Entendiendo esto, viendo las definiciones y leyendo la documentación podemos saber que:

- **enumFrom** nos permite obtener una lista ascendente de números incrementados de uno en uno. De hecho, si ejecutamos “`enumFrom 1`” en `ghci` nunca terminaría porque se trata de una lista infinita.
- **enumFromThen** acepta dos parámetros, el número donde va a comenzar la enumeración y el incremento en que se va a realizar. De hecho, “`enumFromThen x 1`” es similar a “`enumFrom x`”.
- **enumFromTo** funciona igual que “`enumFrom`” pero el segundo argumento supone un límite. Así “`enumFrom 1 10`” nos devuelve una lista que va del 1 al 10: `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
- **enumFromThenTo** funciona igual que “`enumFromThen`” pero con un límite establecido.

En el caso de “`Enum`” se debe tener un límite o “`bound`”. En el caso de “`enumFrom`” y “`enumFromThen`” existe un límite explícito:

```
enumFrom      x      = enumFromTo      x maxBound
  enumFromThen x y = enumFromThenTo x y bound
  where
    bound | fromEnum y >= fromEnum x = maxBound
          | otherwise                = minBound
```

Entonces podemos concluir que “Enum” permite hacer enumeraciones con límites implícitos y explícitos.

También podemos destacar que es posible realizar listas infinitas gracias a la evaluación perezosa que utiliza haskell. Veamos esto utilizando parcializaciones:

```
Prelude> take 10 (enumFrom 1)
[1,2,3,4,5,6,7,8,9,10]
Prelude> take 5 (enumFromThen 5 10)
[5,10,15,20,25]
Prelude>
```

En el código mostrado hay varias cosas que queremos comentar.

Por un lado, queremos destacar la necesidad de los paréntesis en “enumFrom1” y “enumFromThen 5 10”. Esto es debido a que “take” está definido como “take :: Int -> [a] -> [a]” por lo que primero evalúa el entero y luego la lista y es por ello por lo que tenemos que llamar a la función que devuelve la lista después de analizar el entero.

Por otro lado, resulta de vital importancia entender que “enumFrom” y “enumFromThen” no se ejecutan en su totalidad, sino que por sus definiciones se sabe que cada elemento tiene un elemento anterior y siguiente y no es hasta que “take” se evalúa (para obtener los primeros elementos) que se conoce el valor de estos mismos. Si se empleara una “greedy evaluation” primero se tratarían de obtener las listas y no se terminaría nunca.

En las otras dos funciones funciona de manera similar pero como están acotadas no tiene tanto interés ver su análisis ya que una evaluación voraz obtendría el mismo resultado.

h) Explicar el tipo definido en HS para la función product, y su significado, Poner ejemplos de ejecución para algunos casos de sus tipos parámetro. Buscar en la documentación el código nativo de la función product y también para el caso concreto del producto de una lista de elementos, explicando su significado.

Primero vamos a ver su definición: “product :: (Num a, Foldable t) => t a -> a”

De esto podemos ver que si pasamos una lista de “Num” los multiplica todos:

INPUT: product [1,3,5]

OUTPUT: 15

Una definición que nos acerca al ejemplo anterior es “Product = foldl (*) 1”, donde “foldl” aplica el primer argumento “(*)” al segundo argumento “1” asociándolo por la izquierda teniendo así el mismo resultado que el caso anterior.

Si observamos la definición de fact7 vemos que aplica la multiplicación de 1 al primer elemento, ese resultado al segundo y así hasta el final teniendo una asociatividad por la derecha (se usa “foldr”) con respecto a la lista “[1..n]”, ya que las listas son estructuras que se pueden plegar.

i) Redefinir la función fact7 con un código semejante pero que no contenga argumentos, como le ocurre a la función fact8. Incluir ejemplos de uso.

Obtenemos dos soluciones para este apartado:

```
fact7_noArgs1 = foldr (*) 1 . enumFromTo 1
fact7_noArgs2 = f
  where f n =
    foldr (*) 1 [1..n]
```

En el primer ejemplo aplicamos composición de funciones empleando el “foldr” con “enumFromTo”. Así es similar a “fact8” ya que aplica la composición de “foldr (*) 1” con la lista “[1..n]” y como hemos visto en apartados anteriores es similar utilizar “foldr” que producto.

En el segundo caso la solución es idéntica ya que devolvemos “f”, teniendo “f” la misma estructura que “fact 7” tal y como se nos enseñó que se podía hacer en clase.

En resumen, la cuestión es que “fact7_noArgs1” y “fact7_noArgs” nos devuelve una función en lugar de un tipo “Integer” como ocurría con “fact7”.

Los ejemplos de uso son ejecutados en la función “ejemplo2” (main del apartado I) y aparecen los resultados en los comentarios del propio código.

j) Explicar el significado de la definición de fact8 y de las funciones y operadores que utiliza.

Vamos a omitir las explicaciones de la función “producto” (apartado h) y la función “enumFromTo” (apartado g) ya que han sido explicadas previamente.

Lo que es necesario ahora es entender el concepto de composición de funciones. Vamos a realizarlo como si “f” fuera “producto” y “g” fuera “enumFromTo 1”.

Tenemos “f . g (x) => f(g(x))”, por tanto, aplicamos el “enumFromTo 1” al argumento n y obtenemos la lista del 1 al argumento n. Posteriormente se aplica “f” (producto) y permite obtener el factorial.

Puede resultar complejo pero si entendemos bien como se ponen los paréntesis en haskell por la asociatividad y que la composición de funciones en haskell funciona igual que en las matemáticas, resulta trivial.

k) Redefinir la función fact8 con un código semejante pero que contenga 1 argumento, como le ocurre a la función fact7. Incluir ejemplos de uso.

Para el apartado A eliminamos la composición de funciones y seguimos aplicando producto a la lista de 1 hasta n.

```
fact8_arg :: Integer -> Integer
fact_arg n = product [1..n]
```

Los ejemplos de uso son ejecutados en la función “ejemplo3” (main del apartado K) y aparecen los resultados en los comentarios del propio código.

l) Redefinir las funciones fact7 y fact8 para que obtengan como resultado un valor de tipo “Maybe Integer”. Incluir ejemplos de uso.

La forma de realizarlo ha sido primero comprobar que el resultado se puede obtener ($n \geq 0$). En el caso que no se pueda devolvemos “Nothing” y si se puede devolvemos el factorial.

Tanto “fact7_maybe” como “fact8_maybe” son del tipo “Integer -> Maybe Integer”.

Aquí la implementación de dichas funciones:

```
fact7_maybe :: Integer -> Maybe Integer
fact7_maybe n
  | n < 0 = Nothing
  | otherwise = Just (foldr (*) 1 [1..n])

fact8_maybe = f
  where f n
    | n < 0 = Nothing
    | otherwise = Just (product (enumFromTo 1 n))
```

Hay que destacar que como en el ejemplo proporcionado fact8 no tiene ningún argumento de entrada, es redefinido en la función f.

Los ejemplos de uso son ejecutados en la función “ejemplo4” (main del apartado L) y aparecen los resultados en los comentarios del propio código.

m) Redefinir las funciones fact7 y fact8 del apartado anterior para que sean polimórficas, y no monomórficas. Incluir ejemplos de uso.

La implementación de las dos funciones son las siguientes:

```
module ApartadoM.Func.Factoriales where

fact7_poli :: (Num a, Enum a, Ord a) => a -> Maybe a
fact7_poli n
  | n < 0 = Nothing
  | otherwise = Just (foldr (*) 1 [1..n])

fact8_poli :: (Num a, Enum a, Ord a) => a -> Maybe a
fact8_poli = f
  where f n
    | n < 0 = Nothing
    | otherwise = Just (product (enumFromTo 1 n))
```

En ambas funciones se puede ver que ahora no introducimos un tipo “Integer” ni devolvemos un tipo “Integer”, sino que introducimos un tipo polimórfico “a” para obtener un tipo “Maybe a”. Ese tipo polimórfico “a” puede ser cualquier tipo dentro de la clase “Num” (para poder realizar la operación del factorial correspondiente), de la clase “Enum” (para poder usar la lista correspondiente a la que se va aplicando el producto con asociatividad por la derecha) y de la clase “Ord” (para poder realizar la correspondiente evaluación del argumento para llamar al constructor “Nothing” o “Just”)

Los ejemplos de uso son ejecutados en la función “ejemplo5” (main del apartado M) y aparecen los resultados en los comentarios del propio código.

n) Usando stack y VSC, utilizar la función Test.QuickCheck.quickCheck para la comprobación mediante 100 pruebas que al menos 4 definiciones de factorial distintas obtienen el mismo resultado.

Lo primero que hemos hecho para este apartado es incluir en el fichero “package.yaml” de nuestro proyecto stack, la dependencia “QuickCheck” para poder usar el mismo módulo necesario para las pruebas. Aquí se puede ver dicho cambio:

```
! package.yaml X
Practicas > APROG > AProg > Practica_1a > stack > factoriales > ! package.yaml
12
13 # Metadata used when publishing your package
14 # synopsis:          Short description of your package
15 # category:          Web
16
17 # To avoid duplicated efforts in documentation and dealing with the
18 # complications of embedding Haddock markup inside cabal files, it is
19 # common to point users to the README.md file.
20 description:          Please see the README on GitHub at <https://github.com/githubuser/factoriales#readme>
21
22 dependencies:
23 - base >= 4.7 && < 5
24 - QuickCheck
25
```

Después, como hemos hecho en los apartados anteriores, hemos creado la subcarpeta dentro de “src” para el apartado N y, en dicha subcarpeta, hemos creado el fichero “Pruebas.hs”.

En el fichero “Pruebas.hs”, hemos importado todos los módulos usados en la práctica y el módulo “Test.QuickCheck” para poder usar las funciones de prueba. Una vez importados todos los módulos necesarios, hemos ido creando diferentes funciones que devuelven un tipo “Bool” indicándonos si se cumple o no una condición, es decir, que cada función “prueba” comprobamos las diversas funciones implementadas a lo largo de la práctica. Hay que resaltar que las funciones del apartado a) no se prueban ya que están copiadas de la teoría de la asignatura y sabemos que están bien, de hecho, esas funciones se han usado para comprobar el correcto funcionamiento del resto de funciones que ya si hemos hecho nosotros por nuestra cuenta. Estas son dichas funciones de prueba:

```
23
24 -- apartado i
25 prueba1 :: Integer -> Bool
26 prueba1 n = fact7_noArgs1 n == fact7 n
27 prueba2 :: Integer -> Bool
28 prueba2 n = fact7_noArgs2 n == fact7 n
29
30 -- apartado k
31 prueba3 :: Integer -> Bool
32 prueba3 n = fact8b n == fact8 n
33
34 -- apartado l
35 prueba4 :: Integer -> Bool
36 prueba4 n
37   | n < 0 = fact7_maybe n == Nothing
38   | otherwise = fact7_maybe n == Just (fact7 n)
39 prueba5 :: Integer -> Bool
40 prueba5 n
41   | n < 0 = fact8_maybe n == Nothing
42   | otherwise = fact8_maybe n == Just (fact8 n)
43
44 -- apartado m
45 prueba6 :: Integer -> Bool
46 prueba6 n
47   | n < 0 = fact7_poli n == Nothing
48   | otherwise = fact7_poli n == Just (fact7 n)
49 prueba7 :: Integer -> Bool
50 prueba7 n
51   | n < 0 = fact8_poli n == Nothing
52   | otherwise = fact8_poli n == Just (fact8 n)
53
```

Como se puede observar, dichas funciones reciben un argumento y si los resultados de usar las funciones implementadas con dicho argumento son iguales que los resultados de usar dicho argumento con las dos funciones (“fact7” y “fact8”), entonces devolveremos True, en caso contrario, False.

Para hacer las 100 pruebas distintas con cada función hemos creado otras 7 funciones “comp” que lo que hacen es usar la función “quickCheck” que usará de entre una lista (en nuestro caso una lista por comprensión) diferentes valores para pasar como argumentos a las funciones “prueba” explicadas en el párrafo anterior. Estas son dichas funciones:

```
Pruebas.hs
55 -- llamadas a pruebas
56 comp1 :: IO ()
57 comp1= quickCheck prueba1
58 |   where valores = [n | n<-[0..150]]
59
60 comp2 :: IO ()
61 comp2= quickCheck prueba2
62 |   where valores = [n | n<-[0..150]]
63
64 comp3 :: IO ()
65 comp3= quickCheck prueba3
66 |   where valores = [n | n<-[0..150]]
67
68 comp4 :: IO ()
69 comp4= quickCheck prueba4
70 |   where valores = [n | n<-[50..100]]
71
72 comp5 :: IO ()
73 comp5= quickCheck prueba5
74 |   where valores = [n | n<-[50..100]]
75
76 comp6 :: IO ()
77 comp6= quickCheck prueba6
78 |   where valores = [n | n<-[50..100]]
79
80 comp7 :: IO ()
81 comp7= quickCheck prueba7
82 |   where valores = [n | n<-[50..100]]
83
84
```

Finalmente, hemos creado una especie de “main” llamado “pruebas” que llama a todas las funciones “comp”.

En resumen, “pruebas” llama a las diferentes funciones “comp” que usan la función “quickCheck” que usará 100 veces cada función “pruebaN” pasándole un argumento de entre la lista de argumentos posibles establecidas en la propias funciones “comp”.

El resultado de ejecutar la función principal “pruebas” es el siguiente:

```
*ApartadoN.Pruebas> pruebas
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

Se puede ver que todas las funciones implementadas a lo largo de la práctica con diferentes argumentos devuelven lo esperado.

Compilación y Ejecución del proyecto Stack al completo:

En este apartado queremos mostrar como compilamos y ejecutamos el proyecto stack al completo, es decir, ver que todo se compila y ejecuta correctamente además de poder ver la salida que tiene que mostrar por pantalla el “main” del proyecto stack.

Este es el resultado de compilar el proyecto stack (después de solo modificar el apartado n)) al completo con el botón “Stack Build”:

```
PS C:\Users\mele9\APROG> cd "C:\Users\mele9\APROG\Practicas\APROG\Practica_1a\stack\factoriales/"
PS C:\Users\mele9\APROG\Practicas\APROG\Practica_1a\stack\factoriales> stack build --fast
factoriales-0.1.0.0: configure (lib + exe)
Configuring factoriales-0.1.0.0...
factoriales-0.1.0.0: build (lib + exe)
Preprocessing library factoriales-0.1.0.0...
[11 of 12] Compiling ApartadoN.Pruebas ( src\ApartadoN\Pruebas.hs, .stack-work\dist\ca59d8ab\build\ApartadoN\Pruebas.o )
Preprocessing executable 'factoriales-exe' for factoriales-0.1.0.0...
factoriales-0.1.0.0: copy/register
Installing library in
C:\Users\mele9\APROG\Practicas\APROG\Practica_1a\stack\factoriales\.stack-work\install\fd25839\lib\x86_64-windows-ghc-8.0.2\factoriales-0.1.0.0-JzgQsiCuxouzIgzYkKSDAN6
Installing executable(s) in
C:\Users\mele9\APROG\Practicas\APROG\Practica_1a\stack\factoriales\.stack-work\install\fd25839\bin
Registering factoriales-0.1.0.0...
PS C:\Users\mele9\APROG\Practicas\APROG\Practica_1a\stack\factoriales>
```

Y este es el resultado de usar el botón “Stack Run” que ejecuta el “main” del proyecto Stack y, por lo tanto, todas nuestras funciones “main” (“ejemploN” en nuestro caso):

```
PS C:\Users\mele9\APROG\Practicas\APROG\AProg\Practica_1a\stack\fa
$ stack build
stack build took 445ms
$ stack exec factoriales-exe --
"El resultado de (fact7 4) es: 24"
"El resultado de (fact7 5) es: 120"
"El resultado de (fact7 6) es: 720"
"El resultado de (fact8 4) es: 24"
"El resultado de (fact8 5) es: 120"
"El resultado de (fact8 6) es: 720"
"El resultado de (fact7_noArgs1 4) es: 24"
"El resultado de (fact7_noArgs1 5) es: 120"
"El resultado de (fact7_noArgs1 6) es: 720"
"El resultado de (fact7_noArgs2 4) es: 24"
"El resultado de (fact7_noArgs2 5) es: 120"
"El resultado de (fact7_noArgs2 6) es: 720"
"El resultado de (fact8b 4) es: 24"
"El resultado de (fact8b 5) es: 120"
"El resultado de (fact8b 6) es: 720"
"El resultado de (fact7_maybe 4) es: Just 24"
"El resultado de (fact7_maybe -1) es: Nothing"
"El resultado de (fact7_maybe 6) es: Just 720"
"El resultado de (fact8_maybe 4) es: Just 24"
"El resultado de (fact8_maybe -1) es: Nothing"
"El resultado de (fact8_maybe 6) es: Just 720"
"El resultado de (fact7_poli -1) es: Nothing"
"El resultado de (fact7_poli 5) es: Just 120"
"El resultado de (fact7_poli 6) es: Just 720"
"El resultado de (fact8_poli -1) es: Nothing"
"El resultado de (fact8_poli 5) es: Just 120"
"El resultado de (fact8_poli 6) es: Just 720"
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

Como se puede ver, todo sale según lo esperado.

Nota: Hemos usado los mismos números para todos los factoriales y poder ver que sale el mismo resultado. Aún así, las pruebas del último apartado ya se encargan de probar las funciones con diferentes parámetros.

También hay que destacar que hemos intentado poner de la manera más clara los resultados, por eso ponemos que función se llama en cada momento.