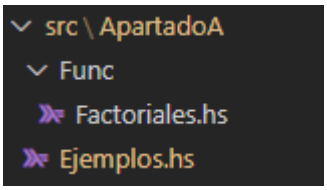
Práctica 1a
Salvador Martín Barcia
Borja Pérez Bernardos

a)

Hemos seguido el siguiente esquema:



En Factoriales.hs están las funciones *fact7* y *fact8* definidas y estas son llamadas desde el fichero Ejemplos.sh donde se define una función *main* que muestra por pantalla los resultados de invocaciones a las dos funciones anteriores.

b)

Para ejecutar *"Ejemplos.hs"* abrimos una Terminal en la carpeta *src* y ejecutamos el comando ghci para abrir el intérprete de Haskell. Seguidamente ejecutamos el comando: *':l "ApartadoA/Ejemplos.hs" '* para compilar el módulo y sus dependencias y ejecutamos la función *"main"* para que muestre los resultados.

- En esta imagen nos muestra un error que nos daba por no importar el módulo *"ApartadoA.Func.Factoriales"* en "*Ejemplos.hs"* donde teníamos las funciones de factorial.

```
Prelude> :r
[1 of 1] Compiling ApartadoA.Ejemplos ( ApartadoA\Ejemplos.hs, interpreted )

ApartadoA\Ejemplos.hs:7:9: error:
    Variable not in scope: fact7 :: Integer -> t

ApartadoA\Ejemplos.hs:8:9: error:
    Variable not in scope: fact8 :: Integer -> t
Failed, modules loaded: none.
Prelude>
```

- En la siguiente imagen, nos mostraba un error de compilación ya que nos faltaba un signo '+' para concatenar texto dentro del 'print'.

```
*ApartadoA.Func.Factoriales> :r
[2 of 2] Compiling ApartadoA.Ejemplos ( ApartadoA\Ejemplos.hs, interpreted )

ApartadoA\Ejemplos.hs:11:16: error:
    * No instance for (Num [Char]) arising from a use of `+'
    * In the first argument of `print', namely
        `("Factorial 5 (fact7) = " + show a)'
      In a stmt of a 'do' block:
        print ("Factorial 5 (fact7) = " + show a)
      In the expression:
        do { print ("Factorial 5 (fact7) = " + show a);
             print ("Factorial 6 (fact8) = " + show b) }
Failed, modules loaded: ApartadoA.Func.Factoriales.
```

- En la siguiente imagen, nos muestra un error por intentar imprimir con *print* texto de tipo *[Char]* e Integers de las variables *a* y *b*. Se solucionaba poniendo la función show delante de estas variables para transformarlas a *[Char]*.

```
*ApartadoA.Func.Factoriales> :r
[2 of 2] Compiling ApartadoA.Ejemplos ( ApartadoA\Ejemplos.hs, interpreted )

ApartadoA\Ejemplos.hs:12:52: error:
    * Couldn't match expected type `[Char]' with actual type `Integer'
    * In the second argument of `(++)', namely `a'
      In the first argument of `print', namely
        `("Factorial 5 (fact7) = " ++ a)'
      In a stmt of a 'do' block: print ("Factorial 5 (fact7) = " ++ a)

ApartadoA\Ejemplos.hs:13:52: error:
    * Couldn't match expected type `[Char]' with actual type `Integer'
    * In the second argument of `(++)', namely `b'
      In the first argument of `print', namely
        `("Factorial 6 (fact8) = " ++ b)'
      In a stmt of a 'do' block: print ("Factorial 6 (fact8) = " ++ b)
Failed, modules loaded: ApartadoA.Func.Factoriales.
*ApartadoA.Func.Factoriales>
```

c)

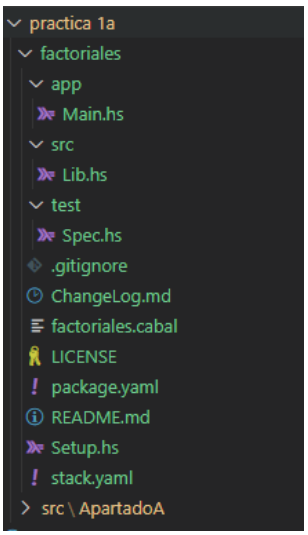
Hemos tenido problemas porque en WinGhci al cargar *“Ejemplos.sh”*, automáticamente se metía en el directorio *“ApartadoA”* y hacía el load desde ahí en vez de desde el directorio *“src”*. Lo hemos solucionado ejecutando el “:load” desde el directorio *“src”* como se ve en la siguiente imagen.

```
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
Prelude> :cd C:\Users\Borja\Desktop\APROG\prácticas\Practica 1\practica 1a\src\ApartadoA
Prelude> :load "Ejemplos.hs"
[1 of 1] Compiling ApartadoA.Ejemplos ( Ejemplos.hs, interpreted )

Ejemplos.hs:5:5: error:
    Failed to load interface for ‘ApartadoA.Func.Factoriales’
    Use -v to see a list of the files searched for.
Failed, modules loaded: none.
Prelude> :cd ..
Warning: changing directory causes all loaded modules to be unloaded,
because the search path has changed.
Prelude> :load "ApartadoA/Ejemplos"
[1 of 2] Compiling ApartadoA.Func.Factoriales ( ApartadoA\Func\Factoriales.hs, interpreted )
[2 of 2] Compiling ApartadoA.Ejemplos ( ApartadoA\Ejemplos.hs, interpreted )
Ok, modules loaded: ApartadoA.Ejemplos, ApartadoA.Func.Factoriales.
*ApartadoA.Ejemplos> main
"Factorial 5 (fact7) = 120"
"Factorial 6 (fact8) = 720"
"Factorial 7 (fact7) = 5040"
"Factorial 8 (fact8) = 40320"
*ApartadoA.Ejemplos>
```

d,e)

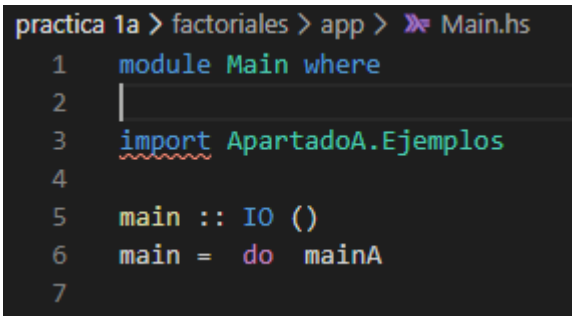
Ejecutando el comando *“stack --resolver lts-8.18 new factoriales”* en la carpeta que contiene al directorio *“src”*, se crea sin problemas el proyecto factoriales como se ve en la siguiente imagen.

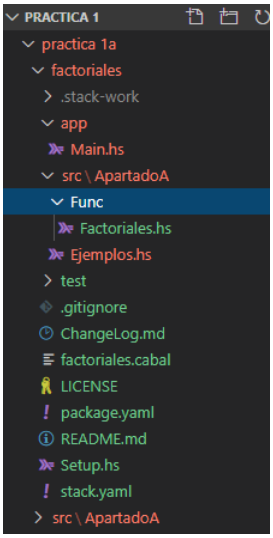
```
PS C:\Users\Borja\Desktop\APROG\prácticas\Practica 1\practica 1a> stack --resolver lts-8.18 new factoriales
Downloading template "new-template" to create project "factoriales" in factoriales\ ...

The following parameters were needed by the template but not provided: author-email, author-name, category, copyright, github-userna
me
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 "githu
b-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.
```



Después compilamos con *"stack build"* pero tuvimos un error que se solucionó quitando una tilde del nombre de una carpeta.
Después ejecutamos con *"stack run"* para comprobar que el proyecto se había generado correctamente.
Seguidamente, en la carpeta *src* del proyecto, pegamos la carpeta *ApartadoA* que contiene las funciones fact7 y fact8 en *Func.Factoriales* y la función *mainA* en *Ejemplos.hs*.

Para terminar, en *app/Main.hs* importamos el módulo *ApartadoA.Ejemplos* y cambiamos el código para que la función *main* ejecutará la función *mainA* de *Ejemplos.sh*.





f)

```
:type <expr>                    show the type of <expr>
```

```
:info[!] [<name> ...]       display information about the given names
```

Aquí tenemos unos ejemplos de los comandos :type e :info, también hemos añadido ejemplos de los comandos :set con distintos parámetros porque creemos que son importantes, para poder ver la eficiencia de diferentes implementaciones.

```
*ApartadoA.Ejemplos> a
120
*ApartadoA.Ejemplos> :set +r
*ApartadoA.Ejemplos> a
120
*ApartadoA.Ejemplos> :type a
a :: Integer
*ApartadoA.Ejemplos> :info b
b :: Integer      -- Defined at ApartadoA\Ejemplos.hs:10:5
*ApartadoA.Ejemplos> :set +s
*ApartadoA.Ejemplos> a
120
(0.00 secs, 29,744 bytes)
*ApartadoA.Ejemplos> :set +t
*ApartadoA.Ejemplos> a
120
it :: Integer
(0.00 secs, 29,744 bytes)
*ApartadoA.Ejemplos>
```

Utilizando el comando :?, hemos obtenido información sobre cómo usar el comando :set y sus diferentes flags que hemos probado en la anterior imagen.

```
Options for ':set' and ':unset':

  +m            allow multiline commands
  +r            revert top-level expressions after each evaluation
  +s            print timing/memory stats after each evaluation
  +t            print type after evaluation
  +c            collect type/location info after loading modules
  -<flags>      most GHC command line flags can also be set here
                       (eg. -v2, -XFlexibleInstances, etc.)
                  for GHCi-specific flags, see User's Guide,
                  Flag reference, Interactive-mode options
```

g)

```
*ApartadoA.Ejemplos> :type enumFrom
enumFrom :: Enum a => a -> [a]
*ApartadoA.Ejemplos> :type enumFromThen
enumFromThen :: Enum a => a -> a -> [a]
*ApartadoA.Ejemplos> :type enumFromThenTo
enumFromThenTo :: Enum a => a -> a -> a -> [a]
*ApartadoA.Ejemplos> :type enumFromTo
enumFromTo :: Enum a => a -> a -> [a]
*ApartadoA.Ejemplos>
```

- **enumFrom** enumera todos los números desde el número pasado por argumento hasta que se interrumpe, si se pasa un número decimal irá contando de uno en uno desde ese número

```
*ApartadoA.Ejemplos> enumFrom 5
[5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,
```

- **enumFromTo** enumera todos los números desde el número pasado por primer argumento hasta igualar o superar al pasado por segundo argumento, si se pasa un número decimal irá contando de uno en uno desde ese número

```
*ApartadoA.Ejemplos> enumFromTo 2 10
[2,3,4,5,6,7,8,9,10]
it :: (Num a, Enum a) => [a]
(0.00 secs, 38,004 bytes)
*ApartadoA.Ejemplos>
```

- **enumFromThen** enumera el primer y el segundo número pasado por argumento y después va enumerando los números sumando la diferencia entre el segundo y el primero hasta interrumpirlo.

```
*ApartadoA.Ejemplos> enumFromThen 2 8
[2,8,14,20,26,32,38,44,50,56,62,68,74,80,86,92,
```

- **enumFromThenTo** enumera siguiendo la secuencia que se marca con los dos primeros argumentos hasta el tercer argumento. Pero si alguno de los argumentos es decimal, si llega al tercer argumento, este se va a pasar del límite por el decimal que se le haya introducido. Esto último también pasa con enumFromTo.

```
*ApartadoA.Ejemplos> enumFromThenTo 2 8 100
[2,8,14,20,26,32,38,44,50,56,62,68,74,80,86,92,98]
it :: (Num a, Enum a) => [a]
(0.00 secs, 46,100 bytes)
*ApartadoA.Ejemplos>
```

h)

En la siguiente imagen vemos que el método product recibe un parámetro de tipo Foldable y devuelve un objeto de tipo Num.

```
Prelude> :type product
product :: (Num a, Foldable t) => t a -> a
Prelude> product [1,2,3,4]
24
Prelude> product [1,2,3,4,5,6]
720
Prelude>
```

Código nativo:

```
-- | The 'product' function computes the product of the numbers of a
-- structure.
--
-- @since 4.8.0.0
product :: Num a => t a -> a
product = getProduct #. foldMap Product
```

Es un método de la clase Foldable.

i)

practica_1a > factoriales > src > ApartadoI > Func > Factoriales.hs

```
1
2   module ApartadoI.Func.Factoriales where
3
4       -- función de pliegue
5       fact7 :: Integer -> Integer
6       fact7 = fact7'
7
8       fact7' :: Integer -> Integer
9       fact7' n = foldr (*) 1 [1..n]
10
```

```
"MAIN I"
"Factorial 5 (fact7) = 120"
"Factorial 7 (fact7) = 5040"
```

La última imagen es la ejecución de *Ejemplos.hs* con fact7 sin recibir argumentos.

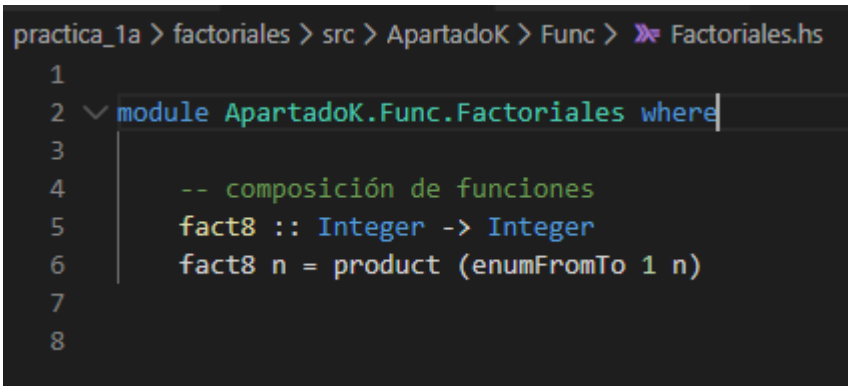
j)

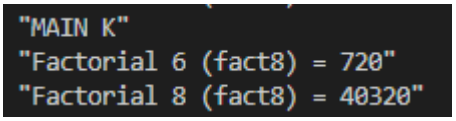
```
fact8 = product . enumFromTo 1
```

fact8 devuelve el resultado de la función *product* que multiplica todos los elementos de la lista que devuelve *enumFromTo.* Esta función devuelve una lista desde el 1 hasta el número que se haya puesto después. El "." es composición de funciones e impide que por defecto haskell ponga los paréntesis por la izquierda.

k)

fact8 recibe un argumento tipo integer y devuelve el valor correspondiente a su factorial.





La última imagen es la ejecución de *Ejemplos.hs* con fact8 recibiendo un argumento.

l)

Para hacer este apartado hemos utilizado if/then/else y guardas ya que ahora puede devolver un número o Nothing.

practica_1a > factoriales > src > ApartadoL > Func > Factoriales.hs

```haskell

module ApartadoL.Func.Factoriales where

    -- función de pliegue
    fact7 :: Integer -> Maybe Integer
    fact7 n = if n >= 0 then Just (foldr (*) 1 [1..n])
                        else Nothing


    -- composición de funciones
    fact8 :: Integer -> Maybe Integer
    fact8 x
        | x >= 0 = Just (product (enumFromTo 1 x))
        | otherwise = Nothing
```

En el resultado de la ejecución hemos añadido números negativos para hacer una comprobación de todos los casos.

```
"MAIN L"
"Factorial -1 (fact7) = Nothing"
"Factorial -2 (fact8) = Nothing"
"Factorial 7 (fact7) = Just 5040"
"Factorial 8 (fact8) = Just 40320"
```

m)

Hacemos que las funciones sean polimórficas para el tipo más general.

```
factoriales > src > ApartadoM > Func > Factoriales.hs
1
2  module ApartadoM.Func.Factoriales where
3
4      -- función de pliegue
5      fact7 :: (Enum a, Num a, Ord a) => a -> Maybe a
6      fact7 n = if n >= 0 then Just (foldr (*) 1 [1..n])
7                          else Nothing
8
9
10     -- composición de funciones
11     fact8 :: (Num a, Enum a, Ord a) => a -> Maybe a
12     fact8 x
13         | x >= 0 = Just (product (enumFromTo 1 x))
14         | otherwise = Nothing
15
```

```
"MAIN M"
"Factorial 5 (fact7) = Just 120"
"Factorial 6 (fact8) = Just 720"
"Factorial 7 (fact7) = Just 5040"
"Factorial 8 (fact8) = Just 40320"
```

n)

Para hacer este apartado hemos tenido que añadir la dependencia QuickCheck en el fichero *package.yaml* .

```
22  dependencies:
23  - base >= 4.7 && < 5
24  - QuickCheck
```

Después, hemos creado un fichero llamado *Funciones.hs*, que contiene 4 definiciones distintas de factoriales para poder hacer el Quickcheck, el cual hemos implementado en el fichero *QuickCheck.hs* y comprueba si todas las definiciones devuelven lo mismo.
Solo comprobamos 3 igualdades ya que si uno de los fact es igual que un fact' todos los demás lo serían entre ellos.

practica_1a > factoriales > src > ApartadoN > Quickcheck.hs

```haskell
module ApartadoN.Quickcheck where

    import ApartadoN.Funciones
    import Test.QuickCheck

    comprobarPrueba :: Integer -> Bool
    comprobarPrueba n = fact7 n == fact8 n &&
                        fact7' n == fact8' n &&
                        fact8 n == fact7' n

    comprobar = quickCheck comprobarPrueba

    mainQC = do     print("QuickCheck:")
                    comprobar
```

Resultado de la ejecución:

```
"QuickCheck:"
+++ OK, passed 100 tests.
```