Paradigmas de Programación

Práctica 6

Ejercicios:

1. Analice hasta su total comprensión el siguiente fragmento de código OCaml:

```
let rec hanoi n ori des =
  (* n número de discos, 1 <= ori <= 3, 1 <= dest <= 3, ori <> des *)
  if n = 0 then ""
  else
    let otro = otro ori des in
    hanoi (n-1) ori otro ^ mueve (ori, des) ^ hanoi (n-1) otro des
```

Con esta definición, `hanoi n ori des` devuelve, codificados en un *string*, los movimientos necesarios para resolver el **problema de las "Torres de Hanoi"** con `n` discos, partiendo del poste `ori` y teniendo que dejarlos todos en el poste `des` (los postes se suponen numerados del 1 al 3):

```
# hanoi 1 1 3;;
- : string = "1---2-->3\n"

# hanoi 2 3 1;;
- : string = "1   2<--3\n1<--2---3\n1<--2   3\n"
```

Se dispone, además, de una redefinición de esta función que permite contemplar el caso de que el poste de origen sea el mismo que el de destino (en ese caso, el problema está ya resuelto sin necesidad de realizar ningún movimiento):

```
let hanoi n ori des =
  if n = 0 || ori = des then ""
  else hanoi n ori des
```

Adicionalmente, también está definida una función `print_hanoi` que muestra por la salida estándar una representación de los movimientos, después de haber comprobado la validez de los argumentos:

```
let print_hanoi n ori des =
  if n < 0 || ori < 1 || ori > 3 || des < 1 || des > 3
  then print_endline "**ERROR**\n"
  else print_endline ("=========\n" ^
                      hanoi n ori des ^
                      "=========")
```

Observe los siguientes ejemplos de uso:

```
# print_hanoi 3 1 4;;
**ERROR**
- : unit = ()
```

```
# print_hanoi 3 1 1;;
=========
=========
- : unit = ()

# print_hanoi 0 2 3;;
=========
=========
- : unit = ()

# print_hanoi 3 3 2;;
=========
1   2<--3
1<--2---3
1<--2   3
1   2<--3
1---2-->3
1-->2   3
1   2<--3
=========
- : unit = ()
```

Lamentablemente, se han perdido las definiciones de las funciones `mueve` y `otro`. Reconstruya estas definiciones, de modo que las funciones descritas anteriormente se comporten exactamente como en los ejemplos mostrados. Para ello, complete el archivo `hanoi.ml` suministrado junto a este enunciado. **No está permitido modificar de manera alguna las definiciones que se conservan completas.**

2. Como seguramente sabrá, el número de movimientos necesario para resolver las "Torres de Hanoi" con n discos (si el poste inicial y final son distintos) es exactamente $2^n - 1$.

   Añada al archivo `hanoi.ml` la definición de una función

   ```
   n_hanoi_mov : int -> int -> int -> int -> int * int
   ```

   de modo que `n_hanoi_mov n nd ori des` devuelva el `n`-ésimo movimiento de la secuencia de movimientos necesarios para resolver el problema con `nd` discos desde el poste `ori` al poste `des`.

   Dado que el mayor `int` es $2^{\texttt{Sys.int\_size - 1}} - 1$, sólo se pretende que funcione correctamente si `0 <= nd < Sys.int_size` y `ori <> des`.

   Si el resultado es el par `(i, j)`, esto indicará que el `n`-ésimo movimiento consiste en llevar el disco de la cima del poste `i` a la cima del poste `j`. Por ejemplo:

   ```
   # n_hanoi_mov 1 2 3 1;;
   - : int * int = (3, 2)

   # n_hanoi_mov 2 2 3 1;;
   - : int * int = (3, 1)

   # n_hanoi_mov 3 2 3 1;;
   - : int * int = (2, 1)
   ```

```
# n_hanoi_mov 16 6 2 3;;
- : int * int = (2, 1)

# n_hanoi_mov 32 6 2 3;;
- : int * int = (2, 3)

# n_hanoi_mov 15 6 2 3;;
- : int * int = (1, 3)

# n_hanoi_mov 63 6 2 3;;
- : int * int = (1, 3)

# n_hanoi_mov max_int (Sys.int_size - 1) 3 2;;
- : int * int = (1, 2)
```

Ya que el número de movimientos para resolver "Torres de Hanoi" con n discos es 2^{n-1}, la complejidad de un algoritmo que enumere esos movimientos ha de ser $\mathcal{O}(2^n)$. Por ello, debemos esperar que el tiempo empleado para calcular estos movimientos con la función `hanoi` se duplique cada vez que se incrementa en 1 el número de discos.

En OCaml podemos utilizar la pseudo-función `Sys.time` para consultar el tiempo de CPU consumido por el proceso. Así, la función `crono`, definida a continuación, puede usarse para cronometrar el tiempo que consume la aplicación de cualquier función sobre cualquier argumento:

```
let crono f x =
  let t = Sys.time () in
  let _ = f x in
  Sys.time () -. t
```

Utilice esta función para ver cómo crecen los tiempos de aplicación de hanoi cuando el número de discos aumenta de, digamos, 20 a 25. Si su máquina es muy lenta o muy rápida quizá deba cambiar algo este intervalo. Considere repetir la operación varias veces para conseguir valores más fiables. Añada los resultados al final del archivo `hanoi.ml` (como comentario) y responda a la pregunta de si avalan o no lo esperado teóricamente.

La complejidad de la función `n_hanoi_mov` debe ser, sin embargo, $\mathcal{O}(nd)$ (siendo nd el número de discos). Compruébelo empíricamente (e incluya también su experimento, como comentario, en el archivo `hanoi.ml`), teniendo en cuenta que, al tener que ser nd menor que `Sys.intsize` (seguramente 63 en su equipo), cualquier aplicación válida de la función debería resultar prácticamente instantánea. **Si no es así, debería considerar la redefinición de esta función.**

3. Estudie la siguiente definición (no muy eficiente) para la función `is_prime`, que sirve para determinar si un número positivo es o no primo:

```
let is_prime n =
  let rec check_from i =
    i >= n || (n mod i <> 0 && check_from (i+1))
  in n > 1 && check_from 2
```

En el fichero proporcionado de nombre `prime.ml`, trate de implementar como una función `is_prime2 : int -> bool` una versión más eficiente de la función `is_prime`. Si lo ha conseguido, debería notar una mejora clara en tiempo de ejecución si compara, por ejemplo, `is_prime 1_000_000_007` con `is_prime2 1_000_000_007`.

4. La **conjetura de Goldbach** dice que todo número entero positivo par es la suma de dos números primos. Por ejemplo, $28 = 5 + 23$. Es uno de los hechos más famosos de la teoría de números que no ha sido probado en general, pero que sí ha sido confirmado numéricamente para números muy grandes.

   En el mismo fichero `prime.ml` escriba una función `goldbach : int -> int * int` que, dado un número entero positivo par n, encuentre dos números primos p_1 y p_2 tales que $n = p_1 + p_2$. Sugerencia: quizás le sea útil la función del ejercicio anterior.

5. Ejercicio opcional. Realice las siguientes tareas en un fichero de texto `ej65.ml`:

   - **Curry y Uncurry**. Dada una función $f : X \times Y \to Z$, podemos siempre considerar una función $g : X \to (Y \to Z)$ tal que $f(x, y) = (g\,x)\,y$.

     A esta transformación se le denomina "currificación" (*currying*) y decimos que la función g es la forma "currificada" de la función f (y que la función f es la forma "descurrificada" de la función g). A la transformación inversa se le denomina "descurrificación" (*uncurrying*).

     Defina una función

     ```
     curry : (('a * 'b) -> 'c) -> ('a -> ('b -> 'c))
     ```

     de forma que para cualquier función `f` cuyo origen sea el producto cartesiano de dos tipos, `curry f` sea la forma currificada de `f`.

     Y defina también la función inversa

     ```
     uncurry : ('a -> ('b -> 'c)) -> (('a * 'b) -> 'c)
     ```

     Una vez definidas estas dos funciones, prediga y compruebe (como en la práctica 1) el resultado de compilar y ejecutar las siguientes frases en OCaml:

     ```
     uncurry (+);;

     let sum = (uncurry (+));;

     sum 1;;

     sum (2,1);;

     let g = curry (function p -> 2 * fst p + 3 * snd p);;

     g (2,5);;

     let h = g 2;;

     h 1, h 2, h 3;;
     ```

     Escriba las frases y sus correspondientes respuestas en el mismo fichero (las respuestas deben ir como comentarios).

- **Composición**. Defina la forma currificada de la composición de funciones:

  ```
  comp : ('a -> 'b) -> ('c -> 'a) -> ('c -> 'b)
  ```

  Una vez definida esta función, prediga y compruebe (como en la práctica 1) el resultado de compilar y ejecutar las siguientes frases en OCaml:

  ```
  let f = let square x = x * x in comp square ((+) 1);;

  f 1, f 2, f 3;;
  ```

  Escriba las frases y sus correspondientes respuestas en el mismo fichero (las respuestas deben ir como comentarios).

- **Polimorfismo**. Defina funciones con los siguientes tipos:
  - `i : 'a -> 'a`
  - `j : 'a * 'b -> 'a`
  - `k : 'a * 'b -> 'b`
  - `l : 'a -> 'a list`

  ¿Cuántas funciones se pueden escribir para cada uno de esos tipos? Escriba las respuestas como comentarios en el mismo fichero.