Paradigmas de Programación

Práctica 5

Ejercicios:

1. Considere la siguiente definición:

   ```
   let g n = (n >= 0 && n mod 2 = 0) || n mod 2 = -1;;
   ```

   En un fichero de texto `ej31.ml`, defina en un valor `g1` el resultado de reescribir la definición anterior sin utilizar la conjunción `(&&)` ni la disyunción `(||)` booleanas (es decir, con frases `if-then-else`).

   En el mismo fichero de texto `ej31.ml`, defina en un valor `g2` el resultado de reescribir la definición anterior sin utilizar la conjunción `(&&)` ni la disyunción `(||)` booleanas, ni tampoco frases `if-then-else`.

   El fichero `ej31.ml` debe compilar sin errores con la orden `ocamlc -c ej31.mli ej31.ml`.

2. 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 check_from 2;;
   ```

   En un fichero `prime.ml`, defina una función `next_prime : int -> int`, tal que (para cualquier `n > 1`) `next_prime n` sea el primer número primo mayor que `n`. Así, por ejemplo, tendríamos `next_prime 11 = 13` y `next_prime 12 = 13`. Defina una función `last_prime_to : int -> int`, tal que (para cualquier `n > 1`) `last_prime_to n` sea el mayor primo menor o igual que `n`. Así, por ejemplo, tendríamos `last_prime_to 11 = 11` y `last_prime_to 12 = 11`.

   En el mismo fichero de texto `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`.

   El fichero `prime.ml` debe compilar sin errores con la orden `ocamlc -c prime.mli prime.ml`.

3. Estudie la siguiente definición escrita en OCaml:

   ```
   let rec fib n =
     if n <= 1 then n
     else fib (n-1) + fib (n-2)
   ```

   Compile esta definición en el *toplevel* (compilador interactivo) `ocaml` y compruebe su funcionamiento.

   Utilizando esta función construya un programa ejecutable `fibto`, de modo que `fibto n` muestre por la salida estándar (y seguidos de un salto de línea) todos los términos de la serie de Fibonacci menores o iguales que `n` (desde el 0).

   De este modo, su ejecución podría verse como se indica a continuación:

```
$ ./fibto 0
0

$ ./fibto 10
0
1
1
2
3
5
8

$ ./fibto 55
0
1
1
2
3
5
8
13
21
34
55

$ ./fibto
fibto: Invalid number of arguments
```

Para acceder desde el programa OCaml al argumento de la línea de comandos, puede utilizar el array de strings `Sys.argv` (que contiene las palabras usadas en la línea de comandos al invocar el programa). El elemento 0 de ese vector (`Sys.argv.(0)`) debe contener el nombre del programa invocado (en este caso, `fibto`) y el elemento 1 (`Sys.argv.(1)`) debe contener el primer argumento (en los casos del ejemplo, `0`, `10` y `55`). Para comprobar que el número de argumentos empleado al invocar el programa es correcto, puede aplicar la función `Array.length` al vector `Sys.argv` (en este caso debería devolver el valor 2, pues la línea de comando debería contener exactamente 2 palabras). Si no es así, el programa debe escribir el mensaje de error mostrado en el ejemplo y seguidamente terminar.

En este ejercicio se trata de salirse lo menos posible del paradigma funcional, implementando la repetición mediante la aplicación de funciones recursivas. Está prohibido, por tanto, el uso de palabras reservadas como `while` y `for`.

Guarde el código fuente del programa en un archivo con nombre `fibto.ml`. Puede compilarlo con la orden `ocamlc -o fibto fibto.ml`.

Atención: valores superiores a 102334155 (término 40 de la serie de Fibonacci) como argumento de entrada de este programa podrían provocar tiempos de ejecución elevados.

4. Considere la función f de $\mathbb{N} \to \mathbb{N}$, que podría aproximarse en OCaml con la siguiente definición para `f : int -> int`

```
let f n = if n mod 2 = 0 then n / 2 else 3 * n + 1
```

Según la Conjetura de Collatz[1], si partimos de cualquier número positivo y vamos aplicando repetidamente esta función, seguiremos un camino que llegará inexorablemente al 1. Por ejemplo, partiendo del 13, tendríamos el siguiente camino:

$$13, 40, 20, 10, 5, 16, 8, 4, 2, 1$$

Realice las siguientes tareas en un fichero de texto `collatz.ml`:

- Llamaremos "órbita" de un número al camino que se sigue de esta manera desde ese número hasta el 1 (ambos incluidos). Por ejemplo, la que hemos escrito más arriba es la órbita del 13.

  Defina (de modo recursivo) una función `orbit : int -> string` tal que, cuando se aplique esta función a cualquier $n > 0$ se devuelva un *string* con la órbita de n, siguiendo el formato del ejemplo (los distintos valores por los que pasa la órbita deben estar separados por una coma y un espacio; no deben aparecer espacios ni al principio ni al final del *string*).

```
# orbit;;
- : int -> string = <fun>

# orbit 13;;
- : string = "13, 40, 20, 10, 5, 16, 8, 4, 2, 1"

# orbit 1;;
- : string = "1"
```

- Defina (de modo recursivo) una función `length : int -> int`, tal que, para cualquier $n > 0$, `length n` sea el número de pasos necesarios (en la órbita de n) para llegar hasta el 1. Así, por ejemplo, `length 13` debe ser `9`.

```
# length;;
- : int -> int = <fun>

# length 13;;
- : int = 9

# length 27;;
- : int = 111
```

- Defina (de modo recursivo) una función `top : int -> int`, tal que, para cada $n > 0$, `top n` sea el valor más alto alcanzado en la órbita de n. Así, por ejemplo, `top 13` debe ser `40`.

```
# top;;
- : int -> int = <fun>

# top 13;;
- : int = 40

# top 27;;
- : int = 9232
```

[1] `https://es.wikipedia.org/wiki/Conjetura_de_Collatz`

- Defina directamente (de modo recursivo, y sin usar las funciones `length` y `top`) una función `length'n'top : int -> int * int`, tal que, para cada entero n, devuelva un par de enteros indicando la longitud de su órbita y su altura máxima. Así, por ejemplo, `length'n'top 13` debería ser el par `(9, 40)`. Se trata de que al aplicar esta definición "no se recorra dos veces la órbita en cuestión".

```
# length'n'top;;
- : int -> int * int = <fun>

# length'n'top 13;;
- : int * int = (9, 40)

# length'n'top 27;;
- : int * int = (111, 9232)
```

- Defina (de modo recursivo) una función `longest_in : int -> int -> int * int` tal que `longest_in m n` devuelva el menor valor del intervalo $[m, n]$ cuya órbita tenga longitud maximal en ese intervalo, acompañado de dicha logitud maximal. De esta forma, por ejemplo, `longest_in 86 87` debería ser el par `(86, 30)`.

```
# longest_in;;
- : int -> int -> int * int = <fun>

# longest_in 1 1000;;
- : int * int = (871, 178)
```

- Defina (de modo recursivo) una función `highest_in : int -> int -> int * int` tal que `highest_in m n` devuelva el menor valor del intervalo $[m, n]$ cuya órbita tenga altura maximal en ese intervalo, acompañado de dicha altura maximal. De esta forma, por ejemplo, `highest_in 86 87` debería ser el par `(87, 592)`.

```
# highest_in;;
- : int -> int -> int * int = <fun>

# highest_in 1 1000;;
- : int * int = (703, 250504)
```

El fichero `collatz.ml` debe compilar sin errores con la orden `ocamlc -c collatz.mli collatz.ml`.