

Paradigmas de Programación

Práctica 3

Ejercicios:

1. Recuerde que una expresión de la forma

```
if <b> then <e1> else <e2>
```

es siempre equivalente a

```
(function true -> <e1> | false -> <e2>) <b>
```

Utilizando esta relación, reescriba el siguiente fragmento de código OCaml sin utilizar frases `if-then-else`:

```
let f1 n = if n mod 2 = 0 then n / 2 else 2 * n;;

let f2 n = if n mod 2 = 0 then "es par" else "es impar";;

let f3 n = if n mod 2 = 0 then "múltiplo de 2"
           else if n mod 3 = 0 then "múltiplo de 3"
           else "impar";;
```

Realice esta tarea en el fichero de texto `ej31.ml`. Este fichero debe compilar sin errores con la orden `ocamlc -c ej31.mli ej31.ml`.

2. En el lenguaje OCaml, las funciones `(&&) : bool -> bool -> bool` y `(||) : bool -> bool -> bool` implementan la conjunción y la disyunción booleanas.

A diferencia del resto de funciones en OCaml, la aplicación de estas funciones sigue una estrategia *lazy* (sólo se evalúa el “segundo” argumento si es necesario).

Es por ello, que es preferible ver las expresiones de la forma `<b1> || <b2>` como una abreviatura de la expresión `if <b1> then true else <b2>` (en vez de verlas como aplicación de funciones).

De modo análogo, las expresiones de la forma `<b1> && <b2>` deben ser vistas como una abreviatura de `if <b1> then <b2> else false`.

Al igual que hizo en el ejercicio 2 de la práctica 1, prediga y compruebe el resultado de compilar y ejecutar las siguientes frases en OCaml, escribiendo todo ello en el fichero de texto `condis.ml` (es decir, tanto las frases en sí mismas, como el resultado de su compilación y ejecución entre comentarios):

```
false && (2 / 0 > 0);;
```

```
true && (2 / 0 > 0);;
```

```
true || (2 / 0 > 0);;
```

```
false || (2 / 0 > 0);;
```

```

let con b1 b2 = b1 && b2;;

let dis b1 b2 = b1 || b2;;

con (1 < 0) (2 / 0 > 0);;

(1 < 0) && (2 / 0 > 0);;

dis (1 > 0) (2 / 0 > 0);;

(1 > 0) || (2 / 0 > 0);;

```

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

3. Considere la siguiente definición:

```

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

```

En un fichero de texto `ej33.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 `ej33.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 `ej33.ml` debe compilar sin errores con la orden `ocamlc -c ej33.mli ej33.ml`.

4. 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 `fib` que muestre por la salida estándar (seguido de un salto de línea) el valor de cada uno de los términos de la serie de Fibonacci, desde 0 hasta el número que se pase como argumento al invocar el programa.

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

```

$ ./fib 10
0
1
1
2
3
5
8
13
21
34
55
$

```

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 `fib.ml`. Puede compilarlo con la orden `ocamlc -o fib fib.ml`.

Atención: valores superiores a 40 como argumento de entrada de este programa podrían provocar tiempos de ejecución elevados.

5. 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 `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`.

Nota Importante:

Cuando se solicite la entrega de esta práctica, cada alumno deberá subir a su repositorio de prácticas (del cual se indicará su ubicación más adelante) un directorio `p3` cuyo contenido debe ser únicamente los ficheros `ej31.ml`, `condis.ml`, `ej33.ml`, `fib.ml` y `prime.ml`.

Sea muy cuidadoso a la hora de crear el directorio y los ficheros, y **respete los nombres indicados**. En particular, fíjese que todos estos nombres sólo contienen letras en minúsculas, números y puntos.

Además, **todos los ficheros deben compilar sin errores** con las órdenes anteriormente citadas.