Paradigmas de Programación

Práctica 3

1. **Conjunción y disyunción**. En OCaml, las funciones `(&&) : bool -> bool -> bool` y `(||) : bool -> bool -> bool` implementan la cojunción 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`.

   Reescriba el siguiente código OCaml sin utilizar la conjunción `(&&)` ni la disyunción `(||)` booleanas (es decir, con frases `if-then-else`):

```
let f = function x -> function y -> function z ->
  (z > y) || ((x <> y) && (z / (x - y) > y));;
```

   Seguidamente, prediga y compruebe (como en las prácticas 1 y 2) el resultado de compilar y ejecutar las siguientes frases en OCaml:

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

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

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

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

let con = (&&);;

let dis = (||);;

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

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

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

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

2. **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 en OCaml 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 las prácticas 1 y 2) 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;;
```

3. **Composición**. Defina en OCaml 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 las prácticas 1 y 2) el resultado de compilar y ejecutar las siguientes frases en OCaml:

```
let f2 = let square x = x * x in comp square ((+) 1);;

f2 1, f2 2, f2 3;;
```

4. (Ejercicio opcional) Como sabemos, una expresión que contenga una definición local, de la forma

```
let <x> = <eL> in <eG>
```

puede siempre reescribirse, sin definiciones locales, utilizando la aplicación de funciones, como la expresión equivalente

```
(function <x> -> <eG>) <eL>
```

Reescriba el siguiente fragmento de código OCaml, de modo que no se empleen definiciones locales:

```
let e1 =
  let pi = 2. *. asin 1. in pi *. (pi +. 1.);;

let e2 =
  let lg2 = log 2. in
  let log2 = function x -> log x /. lg2
  in log2 (float (1024 * 1024));;

let e3 =
  let pi_2 = 4. *. asin 1. in
  function r -> pi_2 *. r;;

let e4 =
  let sqr = function x -> x *. x in
  let pi = 2. *. asin 1. in
  function r -> pi *. sqr r;;
```

NOTA:

- Realice todas las tareas involucradas en los ejercicios 1, 2 y 3 en un fichero de texto `p3.ml` compilable con el compilador `ocamlc`.
- Realice todas las tareas involucradas en el ejercicio opcional 4 en un fichero de texto `ej34.ml` compilable con el compilador `ocamlc`.