Paradigmas de Programación

Práctica 6

Ejercicios:

1. La **Exponenciación Modular** es una operación especialmente útil en Ciencias de la Computación, en particular en el campo de la criptografía (protocolo Diffie-Hellman, RSA, ...)[1]. Por esta razón, muchos lenguajes de programación incluyen funciones predefinidas para realizarla.

   La operación consiste en, dados tres valores enteros no negativos m, b y e (con $m > 0$), calcular el valor de $b^e \ mod \ m$.

   Defina en un fichero `powmod.ml` una función `powmod: int -> int -> int -> int` tal que `powmod m b e` dé este valor para cualesquiera `m > 0`, `b >= 0` y `e >= 0` (al menos para aquellos `m` que en $\mathbb{Z}$ cumplan $(\texttt{m} - 1)^2 <= \texttt{max\_int}$).

   Para ello, podría utilizarse la función `power': int -> int -> int`, implementada en el ejercicio 2 de la Práctica 4, realizando la siguiente definición:

   ```
   let powmod m b e = power' b e mod m
   ```

   Pero con esta definición obtenemos valores incorrectos en $\mathbb{Z}$ si b^e supera el valor `max_int`. Por ejemplo, en mi máquina, `powmod 3 2 62` da `-1` y `powmod 3 2 63` da `0`, cuando, en realidad, deberían dar `1` y `2`, respectivamente.

   Modifique entonces la anterior definición de `powmod`, para que se comporte adecuadamente en las condiciones arriba mencionadas.

   Sugerencia: puede ser de utilidad emplear la relación

   $$(a \times b) \ mod \ m = [(a \ mod \ m) \times (b \ mod \ m)] \ mod \ m.$$

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

2. (Ejercicio opcional) Realice las siguientes tareas en un fichero de texto `ej62.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)
     ```

[1] https://en.wikipedia.org/wiki/Modular_exponentiation

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:
  - $i : {'a} \rightarrow {'a}$
  - $j : {'a} * {'b} \rightarrow {'a}$
  - $k : {'a} * {'b} \rightarrow {'b}$
  - $l : {'a} \rightarrow {'a}\ list$

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

El fichero `ej62.ml` debe compilar sin errores con la orden `ocamlc -c ej62.mli ej62.ml`.