Paradigmas de Programación

Práctica 4

Ejercicios:

1. Al igual que hizo en el ejercicio 3 de la práctica 1 y en el ejercicio 1 de la práctica 2, analice la serie de frases en OCaml incluidas en el fichero `frases4.pdf` que se proporciona junto con este mismo enunciado, intente predecir la información que proporcionaría el compilador interactivo de OCaml (`ocaml`) sobre su compilación y ejecución, y escríbala como comentario en la línea siguiente, procurando usar la misma notación que `ocaml`. Realice esta tarea en un fichero `frases4.ml`. Este fichero debe compilar sin errores con la orden `ocamlc -c frases4.ml`.

2. Redefina en un fichero `ej42.ml` las funciones `min`, `max`, `fst` y `snd` del módulo `Stdlib`, sin utilizar dichas funciones. El fichero `ej42.ml` debe compilar sin errores con la orden `ocamlc -c ej42.mli ej42.ml`.

3. Realice las siguientes tareas en un fichero de texto `ej43.ml`:

   - Sin utilizar el tipo de dato `string`, defina (recursivamente) una función `reverse : int -> int` tal que, para cualquier valor `n:int`, `n >= 0`, `reverse n` sea el entero que se obtiene al invertir el orden de las cifras de la representación decimal de `n`. Por ejemplo, debe cumplirse:

     ```
     reverse 3 = 3
     reverse 12 = 21
     reverse 20 = 2
     reverse 10247 = 74201
     reverse 0b101 = 5
     reverse 0b100000 = 23
     ```

   - Dado `s`, un valor de tipo `string`, la expresión `s.[i]` representa el `char` que ocupa la posición `i` dentro de `s` (el primer `char` ocupa la posición `0` y el último ocupa la posición `String.length s - 1`).

     Defina (recursivamente) una función `palindromo : string -> bool` que indique si el `string` sobre el que se aplica se lee igual de izquierda a derecha que de derecha a izquierda. Por ejemplo, debe cumplirse:

     ```
     palindromo "" = true
     palindromo "a" = true
     palindromo "elle" = true
     palindromo "reconocer" = true
     palindromo "recoger" = false
     palindromo "Ana" = false
     ```

   - Defina (recursivamente) una función `mcd : int * int -> int`, tal que, para cualesquiera `x:int`, `y:int`, `x >= 0`, `y >= 0`, `(x <> 0 || y <> 0)`, `mcd (x,y)` sea el máximo común divisor de `x` e `y`.

     Defina dicha función basándose en las siguientes propiedades:

     $$\begin{aligned} mcd(x, y) &= mcd(y, x) \\ mcd(x, y) &= mcd(x \; mod \; y, y), \text{si } y > 0 \end{aligned}$$

   El fichero `ej43.ml` debe compilar sin errores con la orden `ocamlc -c ej43.mli ej43.ml`.

4. Si x es un número entero cualquiera e y es un número entero mayor que 0, se cumple la siguiente propiedad:

$$x^y = x \times x^{y-1}$$

Utilizando directamente esta propiedad, defina (recursivamente) una función `power : int -> int -> int` tal que, para cualesquiera `x:int`, `y:int`, `y >= 0`, `power x y` tenga el valor de x^y. Por ejemplo, debe cumplirse: `power 2 10 = 1024`.

También son ciertas las siguientes propiedades (razónelas desde el punto de vista matemático):

$$\begin{aligned} x^y &= (x \times x)^{y/2}, \quad \text{si } y \text{ es par} \\ x^y &= x \times (x \times x)^{y/2}, \quad \text{si } y \text{ es impar} \end{aligned}$$

Utilice directamente estas dos propiedades para definir (recursivamente) una función `power' : int -> int -> int`, que sea una versión mejorada (en términos de eficiencia) de la definición anterior.

Explique por qué `power'` deberá ser mejor que `power` en términos de eficiencia y razone si realmente merece la pena la ganancia obtenida al estar operando en `int` (y no en $\mathbb{Z}$).

Todo lo anterior será igualmente válido para potencias de base real y exponente natural. Defina (recursivamente) una función `powerf : float -> int -> float`, tal que, para cualesquiera `x:float`, `n:int`, `n >= 0`, `powerf x n` tenga el valor de x^n,

Realice todas las implementaciones de este ejercicio en un fichero de texto `power.ml`. Las explicaciones que se piden inclúyalas dentro de este mismo fichero como comentarios. Este fichero debe compilar sin errores con la orden `ocamlc -c power.mli power.ml`.

5. (Ejercicio opcional) Observe la figura 1. Cada punto de esa parrilla infinita (mostrada sólo parcialmente en la figura) representa un valor de $\mathbb{N}^+ \times \mathbb{N}^+$.

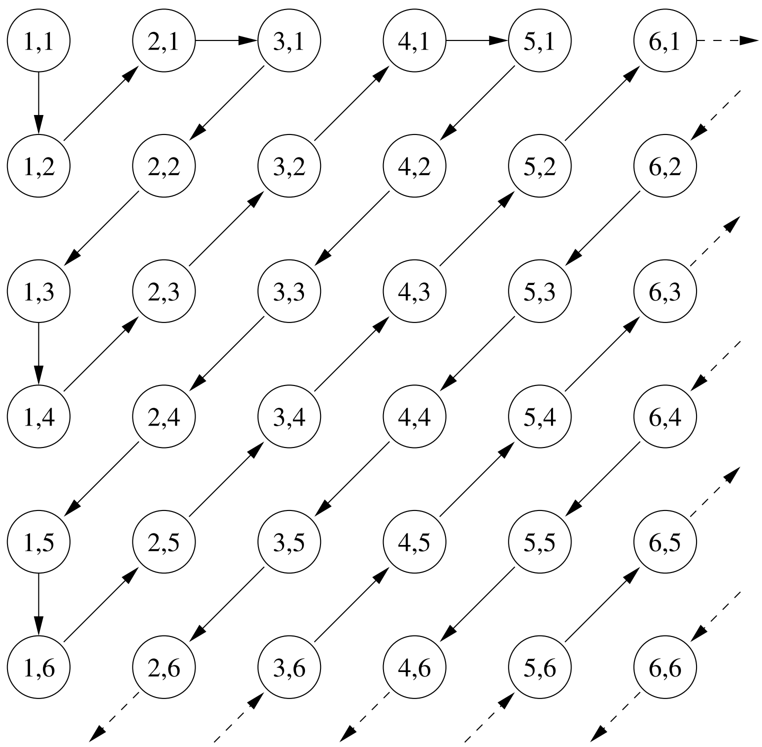

Figura 1: Una posible enumeración de los elementos de $\mathbb{N}^+ \times \mathbb{N}^+$

El recorrido (infinito) que marcan las flechas propone una manera de numerar todos los elementos de $\mathbb{N}^+ \times \mathbb{N}^+$, demostrando que se trata de un conjunto numerable; es decir que existe una aplicación $pair : \mathbb{N}^+ \to \mathbb{N}^+ \times \mathbb{N}^+$, y su inversa $pair\_i : \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}^+$. Podemos hablar así, de la posición que ocupa cada valor de $\mathbb{N}^+ \times \mathbb{N}^+$ en este recorrido. De este modo, por ejemplo, el par que ocupa la primera posición es el $(1, 1)$: $pair(1) = (1, 1)$, y el par $(3, 2)$ ocupa la novena posición: $pair\_i(3, 2) = 9$.

Defina una función `next : int * int -> int * int`, de modo que, para cada valor `(x,y) : int * int` que corresponda a un valor de $\mathbb{N}^+ \times \mathbb{N}^+$, `next (x,y)` corresponda al siguiente valor de $\mathbb{N}^+ \times \mathbb{N}^+$ en el recorrido descrito (siempre, claro, que ese valor “entre” todavía en el tipo `int * int`). (Sugerencia: observe que pueden numerarse de izquierda a derecha las diagonales del recorrido mostrado, de modo que el recorrido es descendente en las diagonales impares y ascendente en las pares, y el par (x, y) se encuentra siempre en la diagonal $x + y - 1$).

Defina una función `steps_from : int * int -> int -> int * int`, de modo que, para cualesquiera valores `x > 0`, `y > 0`, `n >= 0`, `steps_from (x,y) n` devuelva el par al que se llega después de avanzar `n` pasos del recorrido partiendo del par `(x,y)`.

Defina una función `pair : int -> int * int` para representar (en la medida de lo posible) la función $pair : \mathbb{N}^+ \to \mathbb{N}^+ \times \mathbb{N}^+$ mencionada previamente.

Incluya estas tres definiciones en un archivo con nombre `pairs.ml`. Este archivo debe compilar sin errores con la orden `ocamlc -c pairs.mli pairs.ml`.

Puede comprobar el comportamiento de su implementación comparándolo con el siguiente ejemplo de uso:

```
$ ledit ocaml
OCaml version 4.14.0
Enter #help;; for help.
# #load "pairs.cmo";;
# open Pairs;;
# next (2,2);;
- : int * int = (1, 3)
# next (1,3);;
- : int * int = (1, 4)
# next (1,4);;
- : int * int = (2, 3)
# next (2,3);;
- : int * int = (3, 2)
# next (3,2);;
- : int * int = (4, 1)
# next (4,1);;
- : int * int = (5, 1)
# next (5,1);;
- : int * int = (4, 2)
# steps_from (2,3) 0;;
- : int * int = (2, 3)
# steps_from (2,3) 10;;
- : int * int = (3, 4)
# pair 1;;
- : int * int = (1, 1)
# pair 9;;
- : int * int = (3, 2)
```

```
# pair 20;;
- : int * int = (5, 2)
# pair 1001;;
- : int * int = (35, 11)
# pair 99999;;
- : int * int = (130, 318)
```

6. (Ejercicio opcional) Analice y compruebe el funcionamiento de la siguiente implementación de la función *pair_i* mencionada en el enunciado del ejercicio anterior:

```
let pair_i p =
    let rec find i =
        if pair i = p then i
        else find (i+1)
    in find 1;;
```

   Observe cuánto tarda el cálculo de `pair_i (12,130)` y el de `pair_i (100,101)`. Justifique estos tiempos y trate de implementar una versión más eficiente `pair_i' : int * int -> int`.

   Copie el archivo `pairs.ml` del ejercicio anterior a un archivo `pairs2.ml` y añada al final de este último su definición de la función `pair_i'`. Incluya en este archivo como comentario la explicación de por qué era tan lenta la implementación proporcionada y cómo se ha mejorado.

   Este archivo debe compilar sin errores con la orden `ocamlc -c pairs2.mli pairs2.ml`.