Paradigmas de Programación

Práctica 4

Ejercicios:

1. 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 aplicamos repetidamente esta función seguiremos un camino que llegará inexorablemente al 1.

   La función definida a continuación puede usarse para comprobar si un número en particular verifica esta conjetura, en el sentido de que si, al aplicar esta función a dicho número, el cálculo termina, la conjetura se corrobora en ese número.

   ```
   let rec check n =
     n = 1 || check (f n)
   ```

   Así, el siguiente ejemplo comprueba la veracidad de la conjetura en el número 871:

   ```
   # check 871;;
   - : bool = true
   ```

   En un archivo con nombre `collatz.ml`, defina una función `check_to : int -> bool`, de modo que `check_to n` devuelva `true` después de comprobar la conjetura en todos los naturales hasta el `n` (incluido). No debería llevar mucho tiempo comprobar con esta función que la conjetura se cumple para, digamos, el primer millón de números naturales[2].

   ```
   # check_to;;
   - : int -> bool = <fun>
   # check_to 1_000_000;;
   - : bool = true
   ```

   Llamaremos "órbita" de un número al camino que se obtiene al aplicar repetidamente, desde ese número, la función `f` hasta llegar al 1. Así, por ejemplo, la órbita del 13 sería:

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

   Añada al mismo archivo la definición de una función recursiva `orbit : int -> string` tal que, cuando se aplique esta función a cualquier `n > 0`, devuelva un `string` con la representación de su órbita siguiendo exactamente el formato de los siguientes ejemplos (los valores deben ir separados por una coma y un espacio en blanco; no hay coma ni espacio en blanco después del último valor):

[1] Una conjetura es una afirmación que se supone cierta (basándose sobre todo en la observación y en que no se conocen contra-ejemplos), pero para la que no se dispone de una demostración formal. Lothar Collatz enunció su famosa conjetura en 1937, dos años después de doctorarse, y aún no se ha podido probar su veracidad o falsedad.

[2] La conjetura de Collatz ha sido comprobada por ordenador para todos los números hasta, al menos, 2^{68}; lo cual no implica, en absoluto, que no puedan existir contra-ejemplos mayores.

```
# orbit;;
- : int -> string = <fun>
# orbit 13;;
- : string = "13, 40, 20, 10, 5, 16, 8, 4, 2, 1"
# orbit 1;;
- : string = "1"
```

Añada al mismo archivo la definición de una función recursiva `length : int -> int` tal que, para cualquier `n > 0`, `length n` sea el número de valores que contiene la órbita de `n`. Así, por ejemplo, `length 13` debe ser `10`.

```
# length;;
- : int -> int = <fun>
# length 13;;
- : int = 10
# length 27;;
- : int = 112
```

Añada al mismo archivo la definición de una función recursiva `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
```

Puede comprobar que tiene definidas todas las funciones solicitadas con los tipos adecuados utilizando el archivo de interfaz `collatz.mli`.

2. Ejercicio opcional. Defina directamente (de modo recursivo, sin usar las funciones `length` y `top`) una función `length_and_top : int -> int * int` tal que, para cada entero `n`, devuelva un par de enteros indicando la longitud de su órbita y su valor máximo. Así, por ejemplo, `length_and_top 13` debería ser el par `(10, 40)`. Se trata de que al aplicar esta definición "no se recorra dos veces la órbita en cuestión". Esta definición debe incorporarse a un archivo de nombre `collatz_plus.ml`.

```
# length_and_top;;
- : int -> int * int = <fun>
# length_and_top 13;;
- : int * int = (10, 40)
# length_and_top 27;;
- : int * int = (112, 9232)
```

Puede comprobar que tiene definida la función solicitada con el tipo adecuado utilizando el archivo de interfaz `collatz_plus.mli`.

3. Euclides fue un matemático griego que vivió entre los siglos IV y III antes de Cristo. Su obra "Elementos", en 13 volúmenes, es probablemente el mayor *best-seller* de la historia de las matemáticas.

   En esta obra describió por primera vez el algoritmo, conocido hoy como **Algoritmo de Euclides**, que permite calcular, de modo sencillo, el máximo común divisor (MCD) de dos números naturales. Se trata, sin duda, de uno de los algoritmos más antiguos que, aun hoy en día, se sigue utilizando (se usa, por ejemplo, en la simplificación de fracciones y en algoritmos relacionados con la criptografía).

   Este algoritmo se basa en el hecho de que el MCD de dos números no varía si se reemplaza el mayor de ellos por su diferencia con el otro. Repitiendo este proceso, se reduce en cada paso el mayor de ambos números, de modo que, en algún momento, necesariamente, uno llegará a ser 0; en ese momento, el otro es el MCD de los dos números originales.

   Utilice directamente la versión original de Euclides para implementar, de modo recursivo, en OCaml una función `mcd : int * int -> int` tal que (para cualesquiera `x` ≥ 0 e `y` ≥ 0) `mcd (x, y)` coincida con el MCD de `x` e `y` (siempre que al menos uno de ellos sea estrictamente mayor que 0). Intente, como de costumbre, que la definición sea lo más sencilla posible; pero, como se ha dicho antes, utilice directamente la versión del algoritmo que acabamos de describir.

   Si la definición de `mcd` es correcta, deberían obtenerse resultados como los siguientes:

```
# mcd (1, 1);;
- : int = 1
# mcd (12, 4);;
- : int = 4
# mcd (12, 20);;
- : int = 4
# mcd (1716, 105);;
- : int = 3
# mcd (31, 30);;
- : int = 1
# mcd (6589923, 167745);;
- : int = 3
# mcd (101, 101);;
- : int = 101
# mcd (0, 7);;
- : int = 7
```

   Esta primera versión del algoritmo puede requerir muchos pasos para llegar al resultado. Por ejemplo, el cálculo de `mcd (1, 200_000_000)` llevaría 200 millones de pasos. A pesar de la enorme velocidad de los procesadores actuales que pueden realizar miles de millones de operaciones básicas por segundo, en mi ordenador, este cálculo lleva unos 2 segundos. Compruebe cuánto tiempo consume ese mismo cálculo en su equipo.

   A esa velocidad, un cálculo extremo como `mcd (1, max_int)` (con `int`s de 63 bits) llevaría ¡más de 1.000 años! Una versión mucho más eficiente del algoritmo consistiría en reemplazar, en cada paso, el mayor de ambos números por el resto de su división por el menor. Esto equivaldría a restarle al mayor tantas veces el menor como sea posible, pero en un solo paso.

En 1844, Gabriel Lamé publicó un teorema[3] donde demostraba que, de esta manera, el número necesario de pasos para completar el algoritmo de Euclides es menor que 5 veces el número de cifras (en decimal) del menor de ambos números. Según esto, el MCD de 1 y cualquier otro número se terminaría inmediatamente (lo cual es obvio). Un algoritmo implementado siguiendo esta mejora, si trabajamos con un tipo `int` de 63 bits, terminaría, por tanto, siempre en menos de 95 pasos (ya que `max_int`, con `int`s de 63 bits, tiene 19 cifras decimales); lo cual quiere decir que, a nuestra vista, el resultado sería prácticamente instantáneo.

Defina en OCaml una función `mcd' : int * int -> int` aprovechando esta mejora del algoritmo y compruebe, luego, que ahora todas sus aplicaciones son prácticamente instantáneas.

Las definiciones de `mcd` y `mcd'` deben incluirse en una archivo con nombre `mcd.ml`.

4. Ejercicio opcional. Defina una función `mcd_pasos : int * int -> int * int` que calcule el MCD según el algoritmo de Euclides optimizado y devuelva un par de enteros: en la primera componente el valor del MCD y en la segunda el número de pasos necesarios para calcularlo (esto es, el número de veces que se aplicó la función).

   La definición de `mcd_pasos` debe incluirse en una archivo con nombre `mcd_plus.ml`.

   Esta función debería devolver (paso arriba, paso abajo) valores como los siguientes:

```
# mcd_pasos (1, 1);;
- : int * int = (1, 2)
# mcd_pasos (12, 4);;
- : int * int = (4, 2)
# mcd_pasos (12, 20);;
- : int * int = (4, 4)
# mcd_pasos (1716, 105);;
- : int * int = (3, 5)
# mcd_pasos (1, max_int);;
- : int * int = (1, 2)
# mcd_pasos (927, 573);;
- : int * int = (3, 11)
# mcd_pasos (902685, 557890);;
- : int * int = (5, 23)
# mcd_pasos (99012591, 61695194);;
- : int * int = (1, 32)
# mcd_pasos (64251823723465, 37728299599179);;
- : int * int = (7, 48)
# mcd_pasos (10610209857723, 17167680177565);;
- : int * int = (1, 64)
# mcd_pasos (1779979416004714189, 2880067194370816120);;
- : int * int = (1, 89)
```

[3]Esta demostración establece el comienzo del estudio de la Teoría de la Complejidad Computacional.