La sucesión de Fibonacci

El matemático italiano Leonardo de Pisa, más conocido como Fibonacci, dio a conocer en Europa, en el siglo XIII, la sucesión infinita de números naturales en la que, comenzando con dos unos, el resto de los términos se obtiene, cada uno de ellos, sumando los dos anteriores:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, ,…

Estudie la siguiente definición escrita en OCaml:

```
let rec fib n =
    if n <= 2 then 1
    else fib (n-1) + fib (n-2)
```

Con esta definición *fib n* correspondería (dentro del tipo *int*) al el *n-ésimo término* de la sucesión de Fibonacci.

Compile esta definición en el toplevel (compilador interactivo) ocaml y compruebe su funcionamiento.

```
# fib 10;;
- : int = 55
# fib 20;;
- : int = 6765
#
```

Utilizando esta definición de la función *fib*, construya un programa ejecutable ***fib_to***, de modo que su ejecución con un argumento *n* (mayor que 0) muestre por la salida estándar (y seguidos de un salto de línea) todos los términos de la sucesión de Fibonacci **menores** que *n* (desde el 1). Después del último término debe añadirse un salto de línea adicional. Se pretende que el programa funcione correctamente para valores del argumento hasta el que corresponda al mayor término de la sucesión de Fibonacci que se pueda representar en el tipo *int*.

Su ejecución podría verse como se indica a continuación:

```
% ./fibto 10
1
1
2
3
5
8

% ./fibto 55
1
1
2
```

```
3
5
8
13
21
34

% ./fibto
fibto: Invalid number of arguments

%
```

El código fuente de este programa debe escribirse en un archivo de nombre ***fibto.ml***

Para acceder desde un programa OCaml al argumento con el que se invocó su ejecución desde la línea de comandos, puede utilizar el array de strings *Sys.argv* (que contiene las palabras usadas en la línea de comandos al invocar el programa). El elemento *0* de ese vector *(Sys.argv.(0))* debe contener el nombre del programa invocado (en este caso, "fibto") y el elemento 1 *(Sys.argv.(1))* debe contener, si lo hay, el primer argumento (en los casos del ejemplo, "10" y "55"). Para comprobar que el número de argumentos empleado al invocar el programa es correcto, puede aplicar la función *Array.length* al vector *Sys.argv* (en este caso debería devolver el valor *2*, pues la orden debería contener exactamente *2* palabras). Si no es así, el programa debe escribir el mensaje de error mostrado en el ejemplo y seguidamente terminar. **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*.**

El programa debe compilar correctamente con la orden

```
% ocamlc -o fibto fibto.ml
```

Dado que el doble punto y coma (;;) sólo es necesario para indicar la finalización de las frases en el compilador interactivo *(ocaml)* y este programa va a compilarse con el compilador batch *(ocamlc)*, debería ser posible escribir el código completo del programa sin utilizarlo. El problema para esto es que si escribimos dos expresiones seguidas sin ningún separador entre ellas el compilador intentaría interpretarlo ("erróneamente") como una única expresión. Lo mismo sucedería si escribimos una definición seguida de una expresión (pues toda definición termina con una expresión.
Para evitar este problema, el código fuente del programa debería constar de una única expresión o de una simple sucesión de definiciones. Esto último puede ser más sencillo y, si necesitamos que simplemente se evalúe una expresión es fácil conseguirlo: basta con colocarla como parte derecha de una definición. Así, por ejemplo, si queremos que se evalúe la expresión *print_endline "hola"*, podemos escribir la definición *let _ = print_endline "hola"*. Igualmente podría usarse cualquier nombre en lugar del comodín, pero resulta más expresivo lo primero pues resalta el hecho de que el resultado de la expresión no nos interesa. Y ya que la aplicación de la función *print_endline* siempre devuelve el valor unit, también podríamos escribir *let () = print_endline "hola"*.
Así pues, por ejemplo, un programa que calcule y muestre el valor de *pi*, podría escribirse, entre otras, de cualquiera de las siguientes formas:

```
(* como una única expresión *)
let pi = 4. *. atan 1.
in print_endline (string_of_float pi)

(*como sucesión de definiciones *)
let pi = 4. *. atan 1.
let _ = print_endline (string_of_float pi)

let pi = 4. *. atan 1.
let () = print_endline (string_of_float pi)
```

Más adelante, al estudiar los aspectos imperativos del lenguaje veremos el significado del punto y coma (;), y cómo puede utilizarse para secuenciar la evaluación de expresiones (será lo que llamaremos "composición secuencial" de expresiones). Pero, por el momento, trataremos de evitarlo.

Escriba el código del programa *fibto* sin utilizar ni el punto y coma (;) ni el doble punto y coma (;;)

Para gestionar más fácilmente la salida de las diversas líneas que debe realizar este programa, se sugiere definir una función que, para cada valor de *n,* devuelva un *string* que contenga todas la líneas que habría que enviar a la salida estándar para mostrar del modo solicitado los números de Fibonacci menores que *n.*

Aunque el valor más grande de la sucesión de Fibonacci representable con *ints* de 63 bits es el que corresponde al término 90 (2_880_067_194_370_816_120), la implementación dada de la función *fib* es muy ineficiente. Esto hará que la ejecución del programa pueda resultar muy lenta. Comprueba, por ejemplo, cuánto tarda en su equipo para un argumentos como 100_000_000 o mayores.

El compilador ***ocamlc*** genera *bytecode:* código para la máquina virtual de OCaml *(ocamlrun)*. Sin embargo, el compilador ***ocamlopt*** (cuando está disponible) genera *código nativo* para la plataforma en la que se compila; esto produce, en general, ejecutables más rápidos.

Compile el programa ***fibto*** en su máquina con el compilador optimizado y compruebe si (y en qué medida) se obtiene una mejora de rendimiento respecto al compilador *bytecode (ocamlc)*. Podría obtener unos resultados similares a los siguientes:

```
% ocamlc -o fibto fibto.ml

% time ./fibto 100_000_000
1
. . .
39088169
63245986

./fibto 100_000_000  3,55s user 0,01s system 74% cpu 4,760 total
```

```
% ocamlopt -o fibtO fibto.ml

% time ./fibtO 100_000_000
1
. . .
39088169
63245986

./fibtO 100_000_000  0,60s user 0,01s system 34% cpu 1,740 total
```

Intente realizar una definición más eficiente de la función fib y construya con ella un nuevo programa ***fast_fibto*** substituyendo la definición original de fib (en un archivo ***fast_fibto.ml***).

Compile y compruebe la eficiencia de esta nueva implementación.

```
% ocamlopt -o fast_fibto fast_fibto.ml

% time ./fast_fibto 2_880_067_194_370_816_120
1
1
2
3
5
8
13
21
34
55
. . .
99194853094755497
160500643816367088
259695496911122585
420196140727489673
679891637638612258
1100087778366101931
1779979416004714189

./fast_fibto 2_880_067_194_370_816_120  0,00s user 0,00s system 0% cpu 1,108 total
```

Curiosidad: La sucesión de Fibonacci está también relacionada con la complejidad del algoritmo de Euclides para el cálculo del máximo común divisor. Para la versión mejorada en la que se utiliza la resta de la división entera en cada paso, el peor caso se produce cuando aplicamos el algoritmo a 2 términos consecutivos de la sucesión de Fibonacci.