Paradigmas de Programación

Práctica 5

Ejercicios:

1. 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 `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 `fibto`, de modo que su ejecución con un argumento `n` (mayor que 0) muestre por la salida estándar (y seguidos de una coma y un espacio) todos los términos de la sucesión de Fibonacci menores que `n` (desde el 1). Después del último término no deben añadirse ni la coma ni el espacio, sino un salto de línea. 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)`) contiene el nombre del programa invocado (en este caso,

`"fibto"`) y el elemento 1 (`Sys.argv.(1)`) contiene, 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 lo que queremos es que se evalúe la expresión `print_endline "hola"`, podemos escribir una definición como `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 tres formas siguientes:

```
(* 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.

Así pues, **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 la línea que debe escribir este programa, se sugiere definir una función que, para cada valor de `n`, devuelva un *string* que contenga todos los caracteres 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 `int`s 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. Compruebe, por ejemplo, cuánto tarda en su equipo para 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 se obtiene una mejora de rendimiento (y en qué medida) respecto al compilador *bytecode* (`ocamlc`). Podría obtener unos resultados similares a los siguientes:

```
$ ocamlc -o fibto fibto.ml

$ time ./fibto 100_000_000
1, 1, 2, 3, 5, 8, 13, 21, ..., 39088169, 63245986

real 0m4,364s   user 0m4,359s   sys 0m0,002s

$ ocamlopt -o fibtO fibto.ml

$ time ./fibtO 100_000_000
1, 1, 2, 3, 5, 8, 13, 21, ..., 39088169, 63245986

real 0m0,659s   user 0m0,655s   sys 0m0,001s
```

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

   Compile y compruebe la eficiencia de esta nueva implementación. Debería obtener unos resultados similares a los siguientes:

   ```
   $ ocamlopt -o fastfibto fastfibto.ml

   $ time ./fastfibto 2_880_067_194_370_816_120
   1, 1, 2, 3, 5, 8, 13, 21, ..., 1100087778366101931, 1779979416004714189

   real 0m0,005s   user 0m0,001s   sys 0m0,001s
   ```

   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 dos términos consecutivos de la sucesión de Fibonacci.

3. Ejercicio opcional. En OCaml, los valores de tipo `string` (cadenas de caracteres) son como *arrays* (secuencias indexadas) de caracteres (valores de tipo `char`). El índice del primer carácter es 0 y el del último es $(n-1)$ si la cadena tiene longitud n (i.e. contiene n caracteres).

   Si `<s>` es una expresión de tipo `string`, `<s>.[i]` representa el `i`-ésimo carácter de la cadena. La función `String.length` sirve para obtener el número de caracteres de un *string*.

```
# let s = "abcd";;
val s : string = "abcd"
# String.length s;;
- : int = 4
# s.[1];;
- : char = 'b'
# ("aei" ^ "ou").[3];;
- : char = 'o'
```

   Consideraremos que un *string* "representa un número natural en base 3" si no es vacío y contiene sólo los caracteres `'0'`, `'1'` y `'2'`.

   Escriba en un archivo `strg.ml` las definiciones en OCaml que se piden en el resto de este enunciado. Utilice recursividad y manténgase dentro del paradigma funcional.

   Defina una función `str3_of_int : int -> string` tal que, para cualquier int `n >= 0`, `str3_of_int n` sea la representación en base 3 de `n`. Así, por ejemplo, se debería tener:

```
# str3_of_int 0;;
- : string = "0"
# str3_of_int 5;;
- : string = "12"
# str3_of_int 316;;
- : string = "102201"
# str3_of_int 11589501;;
- : string = "210210210210210"
```

4. Ejercicio opcional. Defina también una función `int_of_str3 : string -> int` de modo que si `s` representa un número natural en base 3, `int_of_str3 s` sea el valor de tipo `int` que le corresponde. En Ocaml, los valores de tipo `int` se representan con `Sys.int_size` bits (lo más probable es que en su equipo `Sys.int_size` sea 63). Puesto que un *string* puede tener muchos más que estos caracteres, la función que se pide debe devolver su "valor" módulo $2^{\texttt{Sys.int\_size}}$.

```
# int_of_str3 "0";;
- : int = 0
# int_of_str3 "12";;
- : int = 5
# int_of_str3 "102201";;
- : int = 316
# int_of_str3 "210210210210210";;
- : int = 11589501
```

```
# int_of_str3 (str3_of_int 123456789);;
- : int = 123456789
# int_of_str3 (str3_of_int max_int) = max_int;;
- : bool = true
```

5. Ejercicio opcional. Defina dos nuevas funciones `strg_of_int : int -> int -> string` e `int_of_strg : int -> string -> int` que generalicen las funciones de los dos ejercicios anteriores para cualquier base entre 2 y 9, siendo "el primer parámetro" de estas nuevas funciones la base en cuestión. Por ejemplo:

```
# strg_of_int 2 255;;
- : string = "11111111"
# strg_of_int 3 255;;
- : string = "100110"
# strg_of_int 4 255;;
- : string = "3333"
# strg_of_int 5 255;;
- : string = "2010"
# strg_of_int 6 255;;
- : string = "1103"
# strg_of_int 7 255;;
- : string = "513"
# strg_of_int 8 255;;
- : string = "377"
# strg_of_int 9 255;;
- : string = "313"
# int_of_strg 2 (strg_of_int 2 255);;
- : int = 255
# int_of_strg 3 (strg_of_int 3 255);;
- : int = 255
# int_of_strg 4 (strg_of_int 4 255);;
- : int = 255
# int_of_strg 5 (strg_of_int 5 255);;
- : int = 255
# int_of_strg 6 (strg_of_int 6 255);;
- : int = 255
# int_of_strg 7 (strg_of_int 7 255);;
- : int = 255
# int_of_strg 8 (strg_of_int 8 255);;
- : int = 255
# int_of_strg 9 (strg_of_int 9 255);;
- : int = 255
# int_of_strg 9 (strg_of_int 9 max_int) = max_int;;
- : bool = true
```