Paradigmas de Programación

Práctica 3

Ejercicio 1

Recuerde que la evaluación de la aplicación de funciones en OCaml es “eager” y analice con cuidado la evaluación de las siguientes expresiones

```
let pi = 2. *. asin 1.0 in pi *. pi

(function pi -> pi *. pi) (2. *. asin 1.0)
```

Debería llegar a la conclusión de que ambas se evalúan exactamente igual (y, por tanto, producen exactamente el mismo resultado; es decir, son equivalentes). Puede comprobarlo con ayuda del compilador interactivo.

En general la frase
`let <id> = <e2> in <e1>`
será equivalente a
`(function <id> -> <e1>) <e2>`

De modo que podemos afirmar que (si quisiéramos) podríamos prescindir de las definiciones locales (en el código OCaml).

Utilice esta relación para eliminar todas las definiciones locales en el código OCaml del ejercicio 1 de la Práctica 2. Para ello, copie el archivo “frases.ml” escrito durante la realización de ese ejercicio en un archivo “frases2.ml” y, a continuación, substituya (en este nuevo archivo) cada una de las expresiones que contenga una definición local, siguiendo el esquema anterior.

Conviene que se asegure que el único código OCaml contenido en ambos archivos es el que figura en el enunciado de la Práctica 2 (en el caso del archivo “frases2.ml”, con las substituciones que acabamos de indicar). Es importante que aquellas frases que provocan errores de compilación o ejecución (no “warnings”) figuren como comentarios (en ambos archivos).

Si el ejercicio se ha realizado correctamente, el código contenido en el archivo “frases2.ml” no debería contener la palabra reservada “in” y ambos deberían provocar exactamente la misma salida si se ejecutan los comandos:

```
ocaml < frases.ml
ocaml < frases2.ml
```

Ejercicio 2

Si `<b>` es una expresión correcta de tipo bool en OCaml y `<e1>` y `<e2>` son también dos expresiones correctas en OCaml (ambas del mismo tipo), entonces:

```
if <b> then <e1> else <e2>
```

es una expresión correcta en OCaml, del mismo tipo que `<e1>` y `<e2>`, que se evalúa igual que

```
(function true -> <e1> | false -> <e2>) <b>
```

Utilice esta equivalencia para reescribir el siguiente código OCaml sin utilizar frases if...then…else (las palabras reservadas, if, then y else están prohibidas en este ejercicio).

```
if x > y then "first is greater" else "second is greater";;
if x > 0 then x else -x;;
if x > 0 then x else if y > 0 then y else 0;;
if x > y then if x > z then x else z else if y > z then y else z;;
```

Escriba el código equivalente en un archivo con nombre “if_then_else.ml”.

Puede “chequear” este código con ayuda del compilador interactivo si define antes valores para los nombres involucrados (no incluya estas definiciones en el archivo “if_then_else.ml”, o hágalo como comentario).

Ejercicio 3

Las definiciones de función en las que se utiliza una expresión lambda con una sola regla, pueden abreviarse siguiendo el siguiente esquema:

En vez de escribir
```
let <f> = function <x> -> <e>
```

podemos escribir
```
let <f> <x> = <e>
```

Asi, por ejemplo, podemos escribir
```
let doble x = 2 * x
```

en lugar de
```
let doble = function x -> 2 * x
```

Reescriba, en un archivo con nombre “def2_a.ml”, las definiciones del ejercicio 3 de la Práctica 2 utilizando esta abreviatura. En este ejercicio, por tanto, está prohibida la palabra reservada “function”.

Ejercicio 4

Añadiendo la palabra reservada “rec” a continuación de “let” es posible escribir en OCaml definiciones recursivas de funciones.

Así, por ejemplo, podemos escribir

```
let rec factorial = function 0 -> 1 | n -> n * fact (n-1)
```

Intente predecir y luego compruebe con el compilador interactivo qué sucede al compilar y ejecutar las siguientes frases:

```
let rec factorial = function 0 -> 1 | n -> n * factorial (n-1);;
factorial 0 + factorial 1 + factorial 2;;
factorial 10;;
factorial 100;;
factorial (-1);;
```

En un archivo con nombre “funciones.ml” defina en OCaml las siguientes funciones

sumto: int -> int, de modo que ***sumto n***, devuelva la suma de los n primeros naturales

exp10: int -> int, de modo que, para cualquier n >= 0, ***exp10 n*** devuelva el valor de 10^n

num_cifras: int -> int, de modo que ***num_cifras n*** devuelva el número de cifras de la representación decimal de n (el signo no cuenta como cifra)

sum_cifras: int -> int, de modo que, ***sum_cifras n*** devuelva la suma de las cifras correspondientes a la representación decimal de n.

El siguiente ejemplo muestra cómo deberían comportarse estas funciones una vez definidas

```
# sum_to 0;;
- : int = 0
# sum_to 10;;
- : int = 55
# sum_to 999;;
- : int = 499500
# exp10 0;;
- : int = 1
# exp10 9;;
- : int = 1000000000
# num_cifras 129;;
- : int = 3
# num_cifras 0;;
- : int = 1
# num_cifras (-1999);;
- : int = 4
```

```
# num_cifras 0b101;;
- : int = 1
# num_cifras 0b100000;;
- : int = 2
# sum_cifras 129;;
- : int = 12
# sum_cifras 0;;
- : int = 0
# sum_cifras (-111);;
- : int = 3
# sum_cifras 0b100000;;
- : int = 5
#
```