Paradigmas de Programación

Práctica 3

Ejercicios:

1. Complete el archivo `def.ml`, proporcionado con este enunciado, con el código OCaml necesario para definir los siguientes valores (cuando se trate de funciones, deben utilizarse expresiones lambda con forma `function ... -> ...`):

   - Un valor `pi : float` que sea una buena aproximación del número π.
   - Un valor `e : float` que sea una buena aproximación del número e.
   - Un valor `max_int_f : float` que sea una buena aproximación, como `float`, de `max_int`.
   - Un valor `perimeter : float -> float` que haga corresponder a cada número no negativo el perímetro de la circunferencia que tenga como radio ese número (no importa lo que suceda con los negativos).
   - Un valor `area : float -> float` que haga corresponder a cada número no negativo el área del círculo que tenga como radio ese número (no importa lo que suceda con los negativos).
   - Un valor `next_char : char -> char` que haga corresponder a cada valor de tipo `char` el que ocupa la siguiente posición en la tabla ascii.
   - Un valor `abs_f : float -> float` que haga corresponder a cada número su valor absoluto.
   - Una función `odd : int -> bool` de modo que, al aplicarla a cualquier entero, el resultado que devuelva indique si el entero es impar. Intente que la defínción sea lo más concisa posible (sugerencia: intente evitar el uso de expresiones `if-then-else` innecesarias).
   - Un valor `next_5_mult : int -> int` que haga corresponder a cada entero el menor múltiplo de 5 que sea mayor que él.
   - Un valor `is_letter : char -> bool` que devuelva `true` en los caracteres de la a a la z (tanto mayúsculas como minúsculas) y `false` en los demás. A estos efectos, sólo consideraremos como "letras" los caracteres del alfabeto inglés (es decir, quedan excluidas la ñ, la ç, las letras con tilde, etc.).
   - Y por último, redefina la función `string_of_bool : bool -> string`, de modo que devuelva (adecuadamente) los valores `"verdadero"` o `"falso"` (puede hacerse una definición por casos o utilizarse una expresión `if-then-else`).

   El archivo `def.ml` debe compilar sin errores con el archivo de interfaz `def.mli` proporcionado[1], mediante la orden:

   ```
   ocamlc -c def.mli def.ml
   ```

[1] La interfaz es una declaración de todo lo que debe estar definido en el archivo de implementación `.ml`. Compruebe su contenido para ver la sintaxis que se utiliza en los archivos `.mli`. Si una implementación compila correctamente con su interfaz, podemos asegurar que contiene todas las definiciones especificadas en ella.

2. 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>
```

Así, por ejemplo, podemos escribir

```
let doble x = 2 * x
```

en lugar de

```
let doble = function x -> 2 * x
```

Analice con cuidado esta abreviatura y reescriba, en un archivo con nombre `def_a.ml`, las definiciones del ejercicio 1, utilizando esta abreviatura. En este ejercicio, por tanto, está prohibida la palabra reservada `function`.

Dado que el archivo `def_a.ml` debe contener las mismas definiciones que `def.ml`, si copia la interfaz `def.mli` a otro archivo `def_a.mli`, puede compilar el nuevo archivo con su interfaz para comprobar que contiene todas las definiciones requeridas. En este caso, la orden de compilación sería:

```
ocamlc -c def_a.mli def_a.ml
```

3. 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 * factorial (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 recursivas:

- `sum_to : int -> int`, de modo que `sum_to n` devuelva la suma de todos los naturales desde `0` hasta `n` (incluido).
- `exp_2 : int -> int`, de modo que, para cualquier `n >= 0`, `exp_2 n` devuelva el valor de $2^{\texttt{n}}$.
- `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
# exp_2 0;;
- : int = 1
# exp_2 9;;
- : int = 512
# 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
```

Puede comprobar que el archivo `funciones.ml` contiene todas las definiciones requeridas utilizando la interfaz `funciones.mli` que se proporciona con este enunciado.

4. Este ejercicio es opcional. 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 `frases_2.ml` y, a continuación, substituya (en este nuevo archivo) cada una de las expresiones que contenga una definición local, siguiendo el esquema anterior.

Reescriba también las definiciones que contengan expresiones lambda, utilizando la abreviatura explicada en el ejercicio 2 de esta misma práctica, de modo que, al final, no aparezca la palabra `function` dentro de ninguna definición.

Conviene que se asegure de 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 `frases_2.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 `frases_2.ml` no debería contener la palabra reservada `in` y ambos deberían provocar exactamente la misma salida si se ejecutan los comandos:

```
ocaml -noprompt -w -A < frases.ml
ocaml -noprompt -w -A < frases_2.ml
```

5. Este ejercicio es opcional. 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>
   ```

   Analice esta equivalencia y utilícela 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 "x is greater" else "y 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).