Paradigmas de Programación

Práctica 13

1. Para representar expresiones de la lógica proposicional con variables, un equipo de programación ha diseñado el siguiente tipo de dato:

```
type log_exp =
    Const of bool
  | Var of string
  | Neg of log_exp
  | Disj of log_exp * log_exp
  | Conj of log_exp * log_exp
  | Cond of log_exp * log_exp
  | BiCond of log_exp * log_exp;;
```

   De este modo, la proposición $(p \to q) \Leftrightarrow (\neg p \vee q)$ estaría representada por el valor:

```
BiCond (Cond (Var "p", Var "q"), Disj (Neg (Var "p"), Var "q"))
```

   Para calcular el valor de cualquiera de estas proposiciones en un determinado contexto o lista de pares (variable, valor-booleano), han implementado la siguiente función:

```
let rec eval ctx = function
    Const b -> b
  | Var s -> List.assoc s ctx
  | Neg e -> not (eval ctx e)
  | Disj (e1, e2) -> (eval ctx e1) || (eval ctx e2)
  | Conj (e1, e2) -> (eval ctx e1) && (eval ctx e2)
  | Cond (e1, e2) -> (not (eval ctx e1)) || (eval ctx e2)
  | BiCond (e1, e2) -> (eval ctx e1) = (eval ctx e2);;
```

   Posteriormente, otro equipo de programación ha pensado que podía resultar más adecuado representar estas expresiones de la siguiente forma:

```
type oper = Not;;

type biOper = Or | And | If | Iff;;

type prop =
    C of bool
  | V of string
  | Op of oper * prop
  | BiOp of biOper * prop * prop;;
```

   De este modo, la expresión del ejemplo anterior estaría representada por el valor:

```
BiOp (Iff, BiOp (If, V "p", V "q"), BiOp (Or, Op (Not, V "p"), V "q"))
```

   Como pretenden aprovechar parte del trabajo realizado por el primer equipo, necesitan funciones que transformen las expresiones del tipo antiguo en las equivalentes del nuevo tipo, y viceversa. Así pues:

(a) Para realizar ese cometido, defina las siguientes funciones:

```
prop_of_log_exp : log_exp -> prop
log_exp_of_prop : prop -> log_exp
```

(b) Las funciones `opval` y `biopval`, esbozadas a continuación, proporcionan el valor de cada una de las conectivas u operadores lógicos:

```
let opval = function
    Not -> ...;;

let biopval = function
    Or -> (||)
  | And -> ...
  | If -> ...
  | Iff -> ...;;
```

Complete la definición de estas funciones y, a partir de ellas, implemente una función

```
peval : (string * bool) list -> prop -> bool
```

que calcule el valor de cada proposición lógica del nuevo tipo.

(c) Implemente, para el nuevo tipo `prop`, una función

```
is_tau: prop -> bool
```

de forma que `is_tau p` indique si la proposición lógica `p` es o no una tautología.

Nota: El tipo `log_exp` sólo debe ser utilizado en el apartado (a), pues en el futuro se pretende trabajar sólo con el nuevo tipo de dato `prop`.

Nota: Realice todas las implementaciones en un fichero con nombre `logic.ml`. Este fichero debe compilar sin errores con la orden `ocamlc -c logic.mli logic.ml` (el fichero `logic.mli` se proporciona junto con el presente enunciado).

2. (Ejercicio opcional) Defina una función

```
asort : ('a -> 'a -> bool) -> 'a array -> unit
```

que implemente el método de ordenación por fusión para vectores polimórficos.

Compare el rendimiento con la ordenación por fusión para listas implementada en la práctica 10, y también con las funciones predefinidas List.sort y Array.sort.

Nota: Se trata de un ejercicio relacionado con el paradigma de programación imperativa. Por lo tanto, no se permite realizar conversiones entre arrays y listas. Y tal y como puede deducirse del tipo de la función (procedimiento) `asort`, el resultado de la ordenación debe quedar sobre el mismo vector que se le pasa como argumento.

Nota: Realice la implementación en un fichero con nombre `asort.ml`. Este fichero debe compilar sin errores con la orden `ocamlc -c asort.mli asort.ml` (el fichero `asort.mli` se proporciona junto con el presente enunciado).