

CORRIGÉ DE LA FEUILLE D'EXERCICES N°11.

1. Définition d'une «pile» en utilisant un enregistrement contenant une liste «mutable» :

```

2  type 'a pile = {mutable pile : 'a list};;
4  let créepile () = {pile = []};;
6  let empile p x =
    p.pile <- x::p.pile;;
8
    let depile p =
10 match p.pile with
    [] -> failwith "depile : liste vide"
12 | x::xs -> p.pile <- xs; x;;

```

2.

```

    let vide = function p -> p.pile = [];;
14
    type operateur = add | mult | sous | div;;
16 type fonction = moins | Cos | Sin | carré | Rac_carrée;;

18 type expression =
    reel of float
20 | op of operateur
    | fct of fonction;;
22
    let e = [reel(1.);reel(2.);fct(Sin);fct(Cos);op(add)];;

```

3.

```

24 let eval expr =
    let rec reval p = function
26     reel(x)::xs -> empile p x; reval p xs
    | fct(f)::xs -> begin
28         let x = depile p in
            match f with
30         | moins -> empile p (-. x); reval p xs
            | Sin -> empile p (sin x) ; reval p xs
32         | Cos -> empile p (cos x) ; reval p xs
            | carré -> empile p (x *. x); reval p xs
34         | Rac_carrée -> empile p (x *. x); reval p xs
            end
36     | op(o)::xs -> begin
            let y = depile p and x = depile p in
38         match o with
            add -> empile p (x +. y); reval p xs
40         | sous -> empile p (x -. y); reval p xs
            | mult -> empile p (x *. y); reval p xs
42         | div -> empile p (x /. y); reval p xs

```

```

end
44 | [] -> () in
    let p = créepile () in
46   reval p expr;
    let res = depile p in
48   if vide p then res
        else failwith "eval : Expression erronée" ;;
50
    eval e;;

```

4.

```

52 type expression =
    reel of float
54 | var of string
    | op of operateur
56 | fct of fonction;;

```

La fonction «test» retourne le booléen «true» si l'expression donnée par `l` est bien une expression postfixée sinon elle retourne le booléen «false». Elle repose sur la propriété donnée dans l'énoncé qui nous dit qu'une expression est postfixée si son poids est égal à 1 et si les poids de tous ses préfixes non vides sont supérieurs ou égaux à 1.

```

    let test l =
58     (* <<poids>> donne le poids d'un élément simple d'une expression *)
        let poids = function
60         reel(x) -> 1          (* Cas des constantes réelles *)
        | var(x)  -> 1          (* Cas des variables *)
62         fct(f)  -> 0          (* Cas des fonctions *)
        | op(o)   -> -1 in      (* Cas des opérateurs *)
64     let p = ref(poids (hd l)) and l = ref(tl l) in
        while (!l <> []) & (!p >= 1) do (* Si le poids devient < 1 on sort ou
66                                         si on a parcouru toute la liste *)
            p := !p + (poids (hd !l));
68             l := tl(!l)
        done;
70     if !p = 1 then true  (* On a bien une expression postfixée *)
        else false  (* On n'a pas une expression postfixée *);;

```

5. On commence par définir un type pour représenter les expressions préfixées :

```

72  type expr_prefixe =
      reel_pf of float
74  | var_pf of string
      | Plus of expr_prefixe * expr_prefixe
76  | Sous of expr_prefixe * expr_prefixe
      | Mult of expr_prefixe * expr_prefixe
78  | Div of expr_prefixe * expr_prefixe
      | Moins of expr_prefixe
80  | Sinu of expr_prefixe
      | Cosu of expr_prefixe
82  | Carré of expr_prefixe
      | Racine of expr_prefixe;;

```

Pour transformer une expression postfixée en une expression préfixée, on adapte la solution utilisée pour résoudre l'exercice 1, en suivant l'algorithme suivant :

- si on rencontre un réel ou une constante, on les empile sous forme de réel ou constante d'expression préfixée,
- si on rencontre une fonction f , on dépile la dernière expression préfixée x et on empile l'expression préfixée $f(x)$,
- si on rencontre un opérateur op , on dépile les deux dernières expressions préfixées y et x et on empile l'expression préfixée $op(x, y)$,

```

84  let prefixe_de_postfixe expr =
      let rec reval p = function
86      | reel(x)::xs -> empile p (reel_pf x); reval p xs
          | var(x)::xs -> empile p (var_pf x); reval p xs
88      | fct(f)::xs -> begin
                  let x = depile p in
90                  match f with
                      | moins -> empile p (Moins x); reval p xs
92                  | Sin -> empile p (Sinu x) ; reval p xs
                      | Cos -> empile p (Cosu x) ; reval p xs
94                  | carré -> empile p (Carré x); reval p xs
                      | Rac_carrée -> empile p (Racine x); reval p xs
96                  end
          | op(o)::xs -> begin
98                  let y = depile p and x = depile p in
                      match o with
100                 | add -> empile p (Plus (x,y)); reval p xs
                      | sous -> empile p (Sous (x,y)); reval p xs
102                 | mult -> empile p (Mult(x,y)); reval p xs
                      | div -> empile p (Div(x,y)); reval p xs
104                 end
          | [] -> () in
106  let p = créepile () in
      reval p expr;
108  let res = depile p in
      if vide p then res
110  else failwith "eval : Expression éronnée" ;;

```