Les piles

1 Généralités

La notion de pile est une notion fondamentale en informatique. Tout processeur utilise une pile. Une pile informatique est tout à fait comparable à une pile d'assiettes. Quand on range une assiette dans une pile, on la pose sur le haut de la pile et quand on veut une assiette dans une pile, on prend celle qui est sur le dessus. Dans une pile informatique, c'est le même principe, on a accès qu'au sommet de la pile, les autres éléments étant invisibles.

Pour les piles, il existe deux opérations : la fonction d'empilement et la fonction de dépilement.

- Dans la fonction d'empilement (on dit aussi empilage et *push* en anglais) on ajoute un élément au sommet de la pile.
- Dans la fonction de dépilage⁽¹⁾ (*pop* en anglais), on retourne l'élément qui est au sommet de la pile et on le supprime de la pile.

À ces deux opérateurs, il faut ajouter une fonction de création de pile. Dans la littérature anglo-saxonne, les piles se nomment *stack* ainsi que les files que nous verrons un peu plus tard. Dans une pile, le dernier élément rentré est le premier qui sort ce qui donne en anglais : «Last In, First Out», on parle de façon abrégée de pile **LIFO**.

1.1 Mise en œuvre informatique.

1.1.1 En utilisant des listes référencées.

La description que nous venons de faire des piles nous fait penser à un type de données que nous avons déjà traité avec Caml qui le type «liste référencée». De la même façon qu'une pile d'assiettes, notre liste référencée grandit ou diminue au fur et à mesure des empilements et des dépilages d'éléments.

La mise en œuvre des fonctions de création, d'empilement et de dépilement peut se traduire sous la forme suivante :

```

1  #type 'a pile == 'a list ref;;
2  Le type pile est défini.
   #let cree_pile () =
4    (ref [] : 'a pile);;
   cree_pile : unit -> 'a pile = <fun>
6  #let ajoute (p : 'a pile) a =
   p := a::!p;;
8  ajoute : 'a pile -> 'a -> unit = <fun>
   #let depile (p : 'a pile) =
10   let a = hd !p in
   p := tl !p; a;;
12  depile : 'a pile -> 'a = <fun>

```

1. Dans le dictionnaire Larousse, le dépilage est l'action de dépiler les peaux pour le tannage c'est-à-dire le fait d'enlever les poils, mais pour nous ce sera l'action d'enlever un élément dans une pile.

En ligne 1, on renomme le type de base `'a list ref` comme étant `'a pile`. Dans les fonctions travaillant sur les piles ainsi implémentées, nous sommes obligés de forcer le typage des fonctions en précisant que nous travaillons avec des `'a piles`.

On peut ajouter autant d'éléments que l'on veut dans ce type de représentation, c'est Caml qui alloue dynamiquement la mémoire nécessaire jusqu'à ce que la mémoire qui lui est allouée soit toute utilisée.

Voici une dernière version, en utilisant la reconnaissance de motif et la gestion des erreurs :

```
    let depile (p : 'a pile) =
14  match !p with
      [] -> failwith "pile vide"
16  | x::xs -> p:=xs; x;;
```

1.1.2 En utilisant des listes dans des enregistrements «mutables».

On peut aussi définir les piles à l'aide d'une liste dans un enregistrement ce qui évite d'avoir à forcer le typage des fonctions

```
    #type 'a pile = {mutable p : 'a list};;
18  Le type pile est défini.

20  #let cree_pile () = {p = []};;
    cree_pile : unit -> 'a pile = <fun>
22
    #let empile t a =
24  t.p <- a::t.p;;
    empile : 'a pile -> 'a -> unit = <fun>
26
    #let depile t =
28  match t.p with
      [] -> failwith "pile vide"
30  | x::xs -> (t.p <- xs; x);;
    depile : 'a pile -> 'a = <fun>
```

1.1.3 En utilisant des tableaux.

Les deux représentations précédentes ne sont pas des représentations fidèles de ce que sont physiquement les piles au cœur des ordinateurs. Au niveau du processeur, la ou les piles utilisées sont des zones de mémoire contiguës. La structure Caml qui s'en rapprocherait le plus est plutôt le tableau. Dans le cas des listes, quand on ajoute un élément, il n'est pas nécessairement mis en mémoire à côté du dernier élément de la liste, en fait il est stocké dans un endroit où il y a de la place et il est relié à l'élément précédent à l'aide d'un lien plus communément appelé *pointeur*. Cette opération fait appel à un procédé nommé «allocation dynamique de la mémoire» géré par Caml.

Dans le cas des piles du processeur, la place mémoire est définie au départ et l'ajout d'un nouvel élément à une pile se fait juste à coté du dernier élément stocké, il n'y qu'à gérer la position du premier emplacement libre dans l'espace mémoire qui est alloué à la pile. Il est assez naturel de définir une pile à l'aide d'un enregistrement contenant un tableau, sa longueur ainsi qu'un indicateur de la position courante (indice du premier emplacement libre du tableau) si l'on veut tenir compte des remarques précédentes, ce qui nous donne :

```

1 #type 'a pile = {p : 'a vect; long : int; mutable pos : int};;
2 Le type pile est défini.
  #let cree_pile n x =
4   {p = make_vect n x; long = n; pos = 0};;
  cree_pile : int -> 'a -> 'a pile = <fun>
6
  #let depile t =
8   if t.pos = 0 then failwith "pile vide"
      else (t.pos <- t.pos-1; t.p.(t.pos));;
10 depile : 'a pile -> 'a = <fun>
12
  #let empile t a =
      if t.pos = t.long then failwith "pile pleine"
14      else (t.p.(t.pos) <- a; t.pos <- t.pos+1);;
  empile : 'a pile -> 'a -> unit = <fun>
16
  #let pp = cree_pile 10 0;;
18 pp : int pile = {p = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]; long = 10; pos = 0}
  #empile pp 1;;
20 - : unit = ()
  #empile pp 2;;
22 - : unit = ()
  #empile pp 3;;
24 - : unit = ()
  #pp;;
26 - : int pile = {p = [|1; 2; 3; 0; 0; 0; 0; 0; 0; 0|]; long = 10; pos = 3}
  #depile pp;;
28 - : int = 3
  #depile pp;;
30 - : int = 2
  #pp;;
32 - : int pile = {p = [|1; 2; 3; 0; 0; 0; 0; 0; 0; 0|]; long = 10; pos = 1}
  #empile pp 5;;
34 - : unit = ()
  #pp;;
36 - : int pile = {p = [|1; 5; 3; 0; 0; 0; 0; 0; 0; 0|]; long = 10; pos = 2}

```

remarque : Contrairement aux deux mises en œuvre précédentes, lorsque l'on crée une pile, elle est nécessairement typée alors que dans les cas précédents le typage de la pile ne se fait pas au moment de la création de la pile mais au moment du premier empilement.

1.1.4 En utilisant les bibliothèques Caml.

Les piles étant régulièrement utilisées en informatique, on dispose d'une bibliothèque de fonctions permettant de créer et de gérer des piles (*stack* en anglais) avec Caml.

```
1  ##open "stack";;
2  #push;;
   - : 'a -> 'a t -> unit = <fun>
4  #pop;;
   - : 'a t -> 'a = <fun>
6  #new;;
   - : unit -> 'a t = <fun>
```

1.1.5 À titre culturel, une programmation par encapsulation des opérations.

```
1  #type 'a t = {
2    push : 'a -> unit ;
3    pop  : unit -> 'a ;
4    clear : unit -> unit ;
5    length : unit -> int ;
6    iter : ('a -> unit) -> unit};;
   Le type t est défini.
8  #let new () =
   let pile = ref [] in {
10  push = (function x -> pile := x::!pile);
11  pop  = (function () -> if !pile = [] then failwith "Pile vide";
12                        let a = hd(!pile) in
13                          pile := tl(!pile);
14                          a);
15  clear = (function () -> pile := []);
16  length = (function () -> list_length(!pile));
17  iter = (function f -> do_list f !pile)};;
18  new : unit -> 'a t = <fun>

20  #let pile = new ();;
   pile : 'a t =
22  {push = <fun>; pop = <fun>; clear = <fun>; length = <fun>; iter = <fun>}

24  #pile.push 5;;
   - : unit = ()
26  #pile.push 2;;
   - : unit = ()
28  #pile.pop ();;
   - : int = 2
30  #pile.pop ();;
   - : int = 5
32  #pile.pop ();;
   Exception non rattrapée: Failure "Pile vide"
```