

TP13 Piles

February 20, 2022

On redonne l'interface des piles en OCaml. On utilisera directement ces fonctions dans la suite.

```
[15]: let pile_creer () = ref []  
      let empile p x = p := x :: !p  
      let depile p = match !p with  
        | [] -> failwith "pile vide"  
        | t::q -> p := q; t  
      let est_vide p = !p = []
```

```
[15]: val pile_creer : unit -> 'a list ref = <fun>
```

```
[15]: val empile : 'a list ref -> 'a -> unit = <fun>
```

```
[15]: val depile : 'a list ref -> 'a = <fun>
```

```
[15]: val est_vide : 'a list ref -> bool = <fun>
```

Comme on va être amenés à depiler en ignorant la valeur renvoyée, OCaml va faire un warning car on jette la valeur alors qu'elle n'est pas de type `unit`. Pour éviter ce warning, on utilise la fonction `ignore` : `ignore (depile p)` pourra ainsi être appelé pour retirer une valeur de la pile en l'ignorant.

1 Exercice

Dans son épicerie, Gérard vend de nombreux produits alimentaires et doit porter une attention particulière aux dates de péremption de certains d'entre-eux. Il ne voudrait surtout pas que ses clients tombent malades après avoir consommé ses conserves. Gérard n'a pas beaucoup de place, mais beaucoup de produits à vendre. Il entasse donc les produits d'un même type les uns sur les autres. Lorsqu'un client achète un produit, il prend toujours celui du haut de la pile. Quand Gérard renouvelle son stock, il n'a en général pas beaucoup de temps et place les nouveaux produits juste au dessus de ceux qui sont déjà en rayon.

Avec ce système, les produits du bas de la pile risquent cependant d'y rester très (trop) longtemps. Gérard doit donc régulièrement faire le tour de son magasin, pour vérifier si les produits en bas des piles n'y sont pas depuis trop longtemps, puis les placer sur le dessus, ou les jeter s'ils sont périmés

Votre ami vous demande de l'aider à éviter cette tâche fastidieuse, en lui écrivant un programme qui détecte automatiquement, pour une pile de produits donnée, la date de péremption la plus ancienne parmi les produits de la pile.

Il vous fournit pour cela une liste des opérations effectuées sur cette pile (qui est vide au départ) : les achats et ventes de produits de cette pile, accompagnés des dates d'expiration. Cette liste est composée de couples d'entiers.

Le premier entier est la quantité de produits concernés par l'opération. Cette quantité est un entier positif lorsqu'il s'agit d'un achat par Gérard (ajout sur la pile), et négatif lorsqu'il s'agit d'une vente (retrait).

Le deuxième entier vaut 0 si l'opération est une vente (retrait). S'il s'agit d'un achat, cet entier représente la date de péremption du produit. L'entier correspond à la concaténation de l'année sur quatre chiffres, du mois sur deux chiffres et du jour sur deux chiffres.

Par exemple, si la liste d'opérations est :

```
[ (3, 20040810); (-1, 0); (-1, 0); (3, 20040920);  
  (-1, 0); (4, 20040916); (-3, 0); (-2, 0) ]
```

cela correspond à :

- 3 produits sont ajoutés, date de péremption le 10 août 2004
- 1 produit est retiré (date : 10 août 2004)
- 1 produit est retiré (date : 10 août 2004)
- 3 produits sont ajoutés (date : 20 septembre 2004)
- 1 produit est retiré (date : 20 septembre 2004)
- 4 produits sont ajoutés (date : 16 septembre 2004)
- 3 produits sont retirés (date : 16 septembre 2004)
- 2 produits sont retirés (dates : 16 et 20 septembre 2004)

Après ces opérations, la pile contient deux produits. De haut en bas :

- Un produit ayant pour date de péremption le 20 septembre 2004.
- un produit ayant pour date de péremption le 10 août 2004.

La date de péremption la plus ancienne parmi les produits restants est donc le 10 août 2004.

Remarque On vous garantit qu'il y aura toujours un produit restant.

```
[20]: let date_ancienne l : int =  
      (* BEGIN SOLUTION *)  
      let produits = pile_creer() in  
      let rec aux l =  
        match l with  
        | (o, d)::q ->  
          if o < 0  
          then for i = 0 to -o-1 do  
            ignore (depile produits)  
          done else for i = 0 to o-1 do  
            empile produits d  
          done;  
      in
```

```

        aux q
    | [] -> () in
aux l;
let minimum = ref (depile produits) in
while not (est_vide produits) do
    minimum := depile produits
done;
!minimum
(* END SOLUTION *)

```

[20]: val date_ancienne : (int * int) list -> int = <fun>

```

[21]: assert(date_ancienne [ (3, 20040810); (-1, 0); (-1, 0); (3, 20040920); (-1, 0);
    ↪(4, 20040916); (-3, 0); (-2, 0) ] = 20040810) ;
assert(date_ancienne [ (1, 20201008); (1, 20201108); (-2, 0); (1, 20211008) ] =
    ↪20211008)

```

[21]: - : unit = ()

2 Expressions bien parenthésées

2.1 Avec un seul type de parenthèses

On va considérer des chaînes de caractères contenant des parenthèses et des lettres comme "(a(bc)a)" et on va chercher à savoir si elles sont bien parenthésées : c'est-à-dire que toutes les parenthèses ouvertes sont bien fermées. Pour cela, il suffit de compter le nombre qui sont ouvertes et de s'assurer à la fin que ce nombre vaut 0.

Rappel, on accède au ième caractère de la chaîne s avec s.[i] et on obtient sa taille avec String.length.

```

[22]: let bien_parenthesee s : bool =
    (* BEGIN SOLUTION *)
    let ouvertes = ref 0 in
    let bp = ref true in
    for i = 0 to String.length s - 1 do
        let c = s.[i] in
        if c = '('
        then incr ouvertes;
        if c = ')'
        then begin
            if !ouvertes = 0 then bp := false;
            decr ouvertes
        end
    done;

```

```
!ouvertes = 0 && !bp
(* END SOLUTION *)
```

```
[22]: val bien_parenthesee : string -> bool = <fun>
```

```
[23]: assert(bien_parenthesee "(a(bc)a)");
assert(not (bien_parenthesee "(a(bc)a)"));
assert(not (bien_parenthesee "(a)bc)a("));
assert(bien_parenthesee "()((()))((()))())")
```

```
[23]: - : unit = ()
```

2.2 Avec plusieurs types de parenthèses

Maintenant, on va considérer une liste `l` de couples (`ouvrant`, `fermant`) comme `('(', ')')` ou `('{', '}' )` montrant quelles les types de parenthèses autorisées et on va poser la même question du parenthésage. Ainsi avec la liste `let l = [('(', ')'); ('{', '}'); ('[', ']') ]` la chaîne `(a{bc})[a(b)]` est bien parenthésée mais `'(a)'` ne l'est pas (pas le même symbole qui ferme) et `[(b)]` non plus (pas de `']'` fermante).

On ne va donc plus pouvoir utiliser un simple entier. On ne peut pas non plus utiliser un entier par symbole car ça ne permettra pas de détecter des problèmes comme `[]` (*exercice : vous en convaincre*). On va donc utiliser une pile de symbole de fermeture attendu. Au départ, la pile est vide, ensuite, à chaque fois qu'on voit un symbole d'ouverture on empile le symbole de fermeture correspondant, et si on voit un symbole de fermeture on s'assure que ce soit le sommet de la pile.

```
[33]: let bien_parenthesee_l (l:(char * char) list) (s:string) : bool =
  (* BEGIN SOLUTION *)
  let ouvertes = pile_creer () in
  let bp = ref true in
  for i = 0 to String.length s - 1 do
    let c = s.[i] in
    let rec aux l = match l with
      | (o,f)::q ->
          if c = o
          then empile ouvertes f;
          if c = f
          then bp := !bp && (not (est_vide ouvertes) && depile ouvertes =
↳ f);
          aux q
      | [] -> ()
    in aux l
  done;
  est_vide ouvertes && !bp
  (* END SOLUTION *)
```

```
[33]: val bien_parenthesee_l : (char * char) list -> string -> bool = <fun>
```

```
[32]: assert(bien_parenthesee_l [('(', ')')] "(a(bc)a)");
assert(not (bien_parenthesee_l [('(', ')')] "(a(bc)a)"));
assert(not (bien_parenthesee_l [('(', ')')] "(a)bc)a("));
assert(bien_parenthesee_l [('(', ')')] "()((()))((()))()");
assert(bien_parenthesee_l [('(', ')'); ('{', '}'); ('[', ']')]
  ↪ "([]{}){[()]()}");
assert(not (bien_parenthesee_l [('(', ')'); ('{', '}'); ('[', ']')] "([])"));
assert(not (bien_parenthesee_l [('(', ')'); ('{', '}'); ('[', ']')] "([{}])"))
```

```
[32]: - : unit = ()
```

3 Pile pour évaluer des expressions

On va considérer ici des expressions écrites sous forme postfixe.

La présentation traditionnelle des expressions est infixe : le symbole d'opération est au milieu des opérandes, comme dans $2 + 3$. Cette notation est naturelle mais a le désavantage de nécessiter des parenthèses comme dans $(2 + 3) * 5$. Une autre possibilité est d'avoir une notation postfixe $2\ 3\ +$ où le symbole vient après ses opérandes. L'avantage est de permettre de ne pas avoir besoin de parenthèse, en effet, on peut écrire $2\ 3\ +\ 5\ *$ dans l'exemple précédent.

En OCaml, on représentera une telle expression sous la forme d'une liste contenant :

- soit des entier pour les constantes
- soit un caractère parmi '+', '-', '*', '/' pour les opérations

Pour cela on définit le type suivant :

```
[36]: type op = Plus | Minus | Mult | Div
type token = C of int | O of op
```

```
[36]: type op = Plus | Minus | Mult | Div
```

```
[36]: type token = C of int | O of op
```

L'expression précédente correspond alors à la liste `[ C 2; C 3; O Plus; C 5; O Mult ]`.

Pour évaluer une telle expression, on utilise une pile, initialement vide, et on parcourt de gauche à droite l'expression postfixe. Si la valeur lue est

- un entier : on le place sur la pile
- un symbole d'opération : on dépile deux valeurs sur la pile et on effectue l'opération, puis on empile le résultat.

Quand on a fini le parcours, il ne doit il y avoir qu'une seule valeur sur la pile qui est le résultat (sinon, c'est un problème similaire à un mauvais parenthésage). On dit qu'une expression qui permet d'aboutir à une unique valeur à la fin du parcours est **bien formée**.

Pour les opérations non commutatives comme '-' et '/', on adopte la convention que $a \ b$ - calcule $a - b$. Il faudra donc faire attention à respecter cela dans l'évaluation. **On effectue aussi des divisions entières ici et non flottantes.**

On vous demande d'écrire une fonction d'évaluation des expressions.

```
[43]: let evaluate (e : token list) : int =
  (* BEGIN SOLUTION *)
  let p = pile_creer () in
  let rec aux e = match e with
    | C k :: e' -> empile p k; aux e'
    | O o :: e' ->
      let op2 = depile p in
      let op1 = depile p in
      let res = match o with
        | Plus -> op1 + op2
        | Minus -> op1 - op2
        | Mult -> op1 * op2
        | Div -> op1 / op2
      in empile p res; aux e'
    | [] -> ()
  in
  aux e;
  depile p
  (* END SOLUTION *)
```

```
[43]: val evaluate : token list -> int = <fun>
```

```
[44]: assert(evaluate [C 1; C 2; O Plus] = 3);
assert(evaluate [C 1; C 2; O Minus] = -1);
assert(evaluate [C 3; C 1; O Minus; C 5; O Mult] = 10);
assert(evaluate [C 1; C 2; O Plus; C 2; O Div] = 1);
assert(evaluate [C 1; C 2; C 3; C 4; C 5; C 6; O Plus; O Mult; O Div; O Minus; O ↵
↪Plus] = 3)
```

```
[44]: - : unit = ()
```