Piles

C. Charignon

Table des matières

I Cours **2**

1 Intro : pile des appels **2**

2 Définition du type pile **3**

3 Pile persistante **3**

4 Pile mutable **3**

5 Notation polonaise inversée **5**
5.1 Présentation du principe 5
5.2 Version persistante 5
5.3 Version impérative 6

6 Récursivité terminale **6**
6.1 Application : lecture d'un gros fichier 7

7 Files d'attente **8**
7.1 Présentation 8
7.2 Application : parcours en largeur d'un arbre 8

II Exercices **8**

1 Écriture polonaise postfixée **1**

2 Autre utilisation des piles **1**

Première partie
Cours

1 Intro : pile des appels

Voyons grossièrement la manière dont sont enregistrés les prochains calculs à faire lors de l'exécution d'une fonction récursive. Prenons l'exemple du tri fusion, et ne représentons pour simplifier que les appels à la fonction principale. Dans les schémas ci-dessous, je note tous les calculs restant à faire, en mettant le premier à faire au dessus.

Imaginons que nous voulions trier la liste `[9;5;6;4;3;5]`, notre premier appel est alors `tri [9;5;6;4;3;5]`. En mémoire, nous avons alors :

```
tri [9;5;6;4;3;5]
```

Ensuite, nous découpons la liste en `[9;6;3]` et `[5;4;5]`, qu'il faudra trier récursivement. En supposant que `[9;6;3]` sera triée en première, nous aurons en mémoire :

```
tri [9;6;3]
tri [5;4;5]
```

L'ordinateur attaque `tri [9;6;3]`. La liste est séparée en `[9;3]` et `[6]`, d'où :

```
 tri [9;3]
  tri [6]
tri [5;4;5]
```

On décompose alors `tri [9;3]` :

```
  tri [9]
  tri [3]
  tri [6]
tri [5;4;5]
```

Lors de l'évaluation de `tri [9]`, le résultat est renvoyé directement car il s'agit d'un cas d'arrêt. Aucun nouvel appel n'est rajouté en mémoire :

```
  tri [3]
  tri [6]
tri [5;4;5]
```

De même pour `tri [3]` et `tri [6]`.

```
tri [5;4;5]
```

À ce stade, le calcul de `tri [9;6;3]` est fini, et l'ordinateur peut passer à `tri [5;4;5]`

Autre exemple : calcul d'une expression algébrique. Imaginons qu'il faille calculer $(2+3)\times(5+2)$. On pourrait représenter la mémoire ainsi :

$$\begin{array}{|c|}\hline 2+3 \\ 5+2 \\ \times \\ \hline \end{array}$$

Remarque : On n'a pas expliqué comment l'ordinateur fait pour transmettre le résultat d'une fonction à une autre fonction...

Le point à remarquer est que les nouveaux éléments sont posés en haut, et c'est aussi en haut que sont extraits les éléments dont on a besoin. Ainsi lorsqu'on extrait un élément, c'est le plus récent qui sort. Les anglais disent qu'il s'agit d'une structure «Last In First Out», abrégé en LIFO. Une telle structure s'appelle une «pile». Et la pile décrite ci-dessus est la «pile des appels» de l'ordinateur.

2 Définition du type pile

Une pile est une structure de données toute simple : elle permet essentiellement deux opérations élémentaires :

- ajouter un élément au sommet de la pile («push») ;
- retirer l'élément du sommet («pop»).

Le point important est que l'élément qu'on peut récupérer est le dernier qu'on a mis.
Imaginer tout simplement une pile d'assiette : celle qu'on peut récupérer facilement est celle du sommet, la dernière qu'on a mise.

3 Pile persistante

Voici le type des fonctions nécessaires pour une pile persistante :

- Une constante représentant une pile vide : `pileVide: 'a pile`.
- Pour renvoyer la pile à laquelle on a rajouté un élément `empile: 'a pile -> 'a -> 'a pile`.
- Pour renvoyer la pile à laquelle on a supprimé le sommet : `depile : 'a pile -> 'a pile`. Mais en fait il sera plus pratique de renvoyer à la fois le sommet et la pile privée de ce sommet : `depile : 'a pile -> ('a * 'a pile)`.
- Pour renvoyer le sommet de la pile (sans modifier la pile puisqu'elle est persistante. « peek » ou « top » en anglais.) `sommet: 'a pile -> 'a`.
  Remarque : Cette fonction n'est pas indispensable car on peut la programmer à l'aide des deux précédentes.

Là inutile de se fatiguer : une liste permet déjà les opérations élémentaires des piles. Ainsi, on peut tout simplement utiliser une liste comme une pile ! On peut définir les fonctions ci-dessus de la manière suivante :

```
type 'a pile = 'a list;;

let empile x p =
  (* Renvoie la pile obtenue en rajoutant x sur p *)
  x::p;;

let depile p=
  (* Renvoie le couple (prochain élément à sortir, reste de la pile) *)
  match p with
  |[] -> failwith "pile vide"
  | t::q -> (t, q)
;;

let pile_vide = [] ;;
```

Cependant c'est en pratique inutile : manipulez les listes comme vous avez l'habitude de le faire.

4 Pile mutable

Voici fonctions élémentaires et leurs types pour une pile mutable :

- Pour créer une pile vide : `pileVide: unit -> 'a pile`
- Empiler : `empile : 'a pile -> 'a -> unit`
- Dépiler : `depile : 'a pile -> 'a` Cette fonction va d'une part retirer le sommet de la pile par effet de bord, mais aussi renvoyer cet élément. C'est pourquoi son résultat n'est pas de type unit.
- Tester si la pile est vide : `estVide : 'a pile -> bool`

Voici une implémentation possible sous Caml. On utilise un tableau, et un entier qui indique le dernier élément de la pile (on fait comme si les autres cases du tableau n'existaient pas ; elles sont là pour permettre de rajouter des éléments par la suite.) Le défaut de ceci est qu'il y a une taille maximale pour la pile. Lorsqu'on veut rajouter un élément à une pile ayant déjà atteint ce nombre maximal, on renverra la célèbre erreur « Stack overflow ».

```
type 'a pile = { donnees : 'a array; longueur : int};;
type 'a pile = { donnees : 'a array; mutable fin : int};; (* fin est l'indice du sommet de la
    ↪ pile *)

let exemple = { donnees = [|3;4;5;5;6;5;4|]; fin =2} ;; (* Ici, le sommet vaut 5 *)

let empile x p=
  p.donnees.( p.fin + 1 )   <- x;
  p.fin <- p.fin +1
;;

let depile p=
  if p.fin=0 then failwith "pile vide"
  else begin
      p.fin <- p.fin-1;
      p.donnees.( p.fin+1 )
    end
;;
```

Une autre méthode est de se baser sur une liste : pour obtenir un type mutable, il suffira d'utiliser une référence.

```
type 'a pile = 'a list ref;;

type 'a pile = 'a list ref ;;

let exemple = ref [3;5;4];; (* Ici, le sommet est 3 *)

let empile x p =
  p := x :: !p
;;

let depile p=
  match !p with
  |[] -> failwith "pile vide"
  |t::q ->
    p:= q; (* On modifie p *)
    t (* Et on renvoie le premier élément *)
;;
```

Remarque : On voit qu'une pile peut être implémentée de différentes manières. On dit parfois qu'il s'agit d'un type "abstrait". L'intérêt est qu'on peut concevoir des algorithmes sans se préoccuper de la manière dont la pile est implémentée (enfin, il faut quand même savoir si elle est mutable ou persistante) ; l'algorithme sera traduisible directement dans n'importe quel langage dès qu'un type pile et ses fonctions élémentaires y auront été programmées.

La plupart des langages proposent le type pile dans une bibliothèque. En Caml, c'est la bibliothèque `stack`, qui donne accès à une pile mutable. Les commandes basiques sont les suivantes :

- pile vide : `Stack.create ()`
- empiler x sur la pile p : `Stack.push x p`
- dépiler la pile p : `Stack.pop p`
- Tester si une pile est vide : `Stack.is_empty`

Remarque : Pour s'épargner les "`Stack.`" au début de chaque commande, on peut charger le module `stack` complètement par la commande `#open "stack"`. Il faut cependant être sûr que les noms `create`, `push`, `pop`, ... ne seront pas ambigüs. Éviter donc de charger de la sorte plusieurs modules.

Remarque : La plupart du temps, les piles sont prises mutables dans la littérature. C'est logique au sens ou la notion de pile persistante est inutile puisque ça revient à une liste.

5 Notation polonaise inversée

5.1 Présentation du principe

Les premières calculatrices utilisaient souvent des piles (les HP ont continué très longtemps). En effet, il y a un moyen très pratique de noter des calculs qui ne nécessite pas de parenthèses, et qui pourra être traduit très facilement pour un ordinateur, c'est la notation polonaise inversée. Elle consiste à noter un opérateur après ses arguments. Par exemple $1 + 2$ sera noté $12+$. On dit que l'opérateur est alors "postfixe" (vous vous souvenez des opérateur "infixe" ou "préfixe" ?).

Plus compliqué : que signifient les calculs suivants ?

- $123 \times +$
- $12 + 3\times$
- $123 - +$
- $1234 - + + \times$

Remarque : Le symbole $-$ peut avoir plusieurs sens : il importe de se mettre d'accord. Voulons-nous d'un opérateur binaire ou unaire ?

Une fois qu'on a convaincu l'utilisateur d'utiliser la notation polonaise inversée, il devient facile [1] de programmer la calculette.

On part d'une pile vide, et on lit les instructions tapées par l'utilisateur dans l'ordre. Quand on lit un nombre, on l'empile. Quand on lit un opérateur unaire (qui prend un seul argument : par exemple une fonction cos sin, etc.. ou le signe $-$) on dépile le nombre au sommet de la pile, on lui applique l'opérateur, et on le rempile. Enfin, si on rencontre un opérateur binaire ($+, \times$ etc...) on dépile les deux nombres au sommet de la pile, on leur applique l'opérateur, on rempile le résultat.

Si l'utilisateur ne s'est pas trompé, à la fin la pile contiendra un seul élément : le résultat voulu.

Notons que ceci ne demande qu'une seule lecture de la suite d'instruction : le temps d'exécution est linéaire (à condition que les opérateurs utilisées s'exécutent tous en temps borné bien sûr).

Programmons ceci. Pour commencer, on définit un type lexème :

```
type 'a lexeme = OpBin of ('a -> 'a-> 'a) (* opérateur binaire *)
               | Nb of 'a (* nombre *)
;;
```

Remarque : Si vous voulez prendre en compte les opérateurs unaire, rajoutez le constructeur `OpUn of ('a -> 'a)`.

5.2 Version persistante

Plaçons-nous en mode récursif.

- Une formule sera une liste de lexèmes. On la parcourra au moyen d'une fonction récursive.
- La pile sera une simple liste.

```

let applique o pile =
  (* Entrée : un opérateur binaire o
              une pile pile (en fait de type list)
     Sortie  : la pile obtenue en remplaçant les deux éléments x,y au sommet de la pile par (o x y)
   *)
  match pile with
  | x::y::en_dessous -> (o x y) :: en_dessous
  | _ -> failwith "erreur syntaxe"
;;


let evalue_rec f =
  (* Entrée : f, formule, de type lexeme list
```

1. tout est relatif

```
15      Sortie : le résultat de l'évaluation de f *)
16
17
18    let rec aux pile a_lire =
19      (* pile : la pile des calculs intermédiaires (en l'occurence, une 'a list
20         a_lire : liste des lexèmes restant à lire *)
21
22      match a_lire with
23      | [] -> (* On a fini la lecture. Normalement, la pile ne contient qu'un élément : le ré
        ↪ sultat. *)
24         begin
25           match pile with
26           |[res] -> res
27           |_ -> failwith "Erreur syntaxe"
28         end
29
30
31      | Nb n :: suite_f -> (* On lit un nombre : on le met dans la pile *)
32         aux (n::pile) suite_f
33
34
35      | OpBin o :: suite_f -> (* On lit un opérateur binaire : on l'applique aux deux éléments
        ↪ au sommet de la pile. *)
36         aux ( applique o pile  )   suite_f
37    in
38
39    aux [] f
40  ;;
```

5.3 Version impérative

À présent, voyons une version impérative.

- Une formule sera un tableau de lexèmes, qu'on parcourra par une boucle for.
- La pile sera mutable, du module `Stack` de OCaml.

Remarque : Cette fonction ne gère pas les erreurs de syntaxe comme la version récursive du paragraphe précédent...

6 Récursivité terminale

Maintenant que nous avons évoqué la notion de pile des appels, étudions une notion supllémentaire concernant la récursivité.

Dans la fonction ci-dessous, le cas d'arrêt a été oublié, elle ne termine donc pas.

```
1 let rec identite n=
2   1+ identite (n-1)
3 ;;
4
5 identite 1;;
```

On obtient l'erreur « Stack overflow during evaluation (looping recursion ?). » signifiant que la pile des appels a débordé. C'est bien ce à quoi on s'attend.

Corrigeons la fonction :

```
1 let rec identite n=
2   if n=0 then 0
3   else 1+ identite (n-1)
4 ;;
```

On constate alors que cette fonction fonctionne jusqu'à environ 200000 (sur l'ordinateur ayant servi à écrire ces lignes). Ainsi la pile des appels peut-elle contenir environ 200000 appels à `identite`.

Remarque : Avec la fonction suivante, on échoue à 150000, ce qui signifie qu'un appel à `identite2` occupe plus d'espace dans la pile des appels, à cause de ses arguments plus nombreux.

```
let rec identite2 a iu i n=
  if n=0 then 0
  else 1+ identite2  a iu i  (n-1)
;;
identite2 "b" "c" "c" 150000;;
```

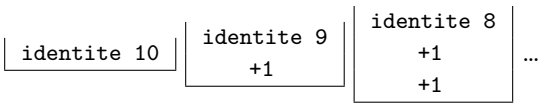
À présent, écrivons une version avec accumulateur et fonction auxiliaire :

```
let identite_terminale_qui_plante n=
  let rec aux accu i=
    aux (accu+1) (i-1)
  in
  aux 0 n
;;
```

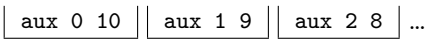
Cette fois, plus de message d'erreur ! La pile des appels n'est jamais remplie !

La grosse différence est que dans `aux`, l'appel récursif est le dernier calcul que doit effectuer l'ordinateur (il est « terminal »). Dès lors l'ordinateur va juste remplacer l'appel actuel par l'appel suivant, et la pile n'a pas augmenté.

Voici le schéma de la pile des appel dans la fonction `identite` non terminale appelée sur l'entier 10.



Et pour l'évaluation de `aux 0 10` :



Remarque : Python ne détecte pas la récursivité terminale.

6.1 Application : lecture d'un gros fichier

Caml ne fournit pas de base une fonction permettant de savois si on est arrivé à la fin d'un fichier. On doit passer par l'exception « `End_of_File` ».

```
let list_of_fichier chemin=
  let entree=open_in chemin in
  let rec aux () =
    try
      let ligne = input_line entree in
       ligne :: aux ()
    with
     |End_of_file -> []
  in
  aux()
;;
```

Remarque : Si l'on remplaçait le contenu du « try » par `input_line entree :: aux ()`, la fonction pourrait planter car on ne sait pas dans quel ordre les arguments du `::` seront évalués. Si Caml commence par évaluer le `aux ()` on n'atteindra jamais le `End_of_File` levé par `input_line`.

Si le fichier est trop volumineux, la pile des appels va déborder. Il faut donc écrire une version récursive terminale.

```
let list_of_fichier_term chemin=
  let entree=open_in chemin in
  let rec aux accu =
    try
      let ligne = input_line entree in
      aux (ligne::accu)
    with
     |End_of_file -> accu
```

```
  in
  aux []
;;
```

Mais celle-ci ne fonctionne toujours pas. Elle n'est pas terminale à cause du `try ... with`. On va mettre le `try ... with` dans une fonction externe :

```
let prochaine_ligne entree =
  try
    input_line entree
  with
  |End_of_file -> ""
;;

let list_of_fichier_term chemin=
  let entree=open_in chemin in
  let rec aux accu =
      let ligne = prochaine_ligne entree in
      if ligne="" then
        (* une ligne vide dans le fichier contiendrait au moins le retour charriot.*)
        accu
      else
        aux (ligne::accu)
  in
  aux []
;;
```

Une autre solution est d'utiliser une boucle `while true` [2]

```
let lis_fichier_imp chemin =
  let entree=open_in chemin
  and res = ref [] in
  begin
    try
      while true do
        res:= input_line entree :: !res
      done
    with
    |End_of_file -> ()
  end ;
    !res
;;
```

7 Files d'attente

7.1 Présentation

Une file d'attente est une structure symétrique à la pile : elle propose les même opérations de base mais lorsqu'on extrait un élément, c'est le plus ancien qui sort. C'est donc une structure de type FIFO (ou LILO).

On peut imaginer deux manières d'implémenter une file d'attente :

- **Avec deux piles** : une des deux piles sera la pile des entrées, et l'autre la pile des sorties. Seule subtilité : lorsque la pile des sorties est vide, on y met la pile d'entrée retournée.
  - ◇ *Version persistante :* On utilise deux listes
  - ◇ *Version modifiable :* On utilise deux piles modifiables. Une telle version est disponible dans la bibliothèque `Queue` de OCaml.
- **Dans un tableau :** On peut enregistrer les éléments de la file dans un tableau. Il faut alors enregistrer également deux indices : le début et la fin de la pile. Cette méthode est plus efficace (évite le retournement de la pile d'entrée) mais le nombre d'éléments enregistrables est fixé à l'avance.

2. beurk

7.2 Application : parcours en largeur d'un arbre

Deuxième partie
Exercices

MPSI : TP d'informatique

1 Écriture polonaise postfixée

Exercice 1. * Calcul booléen

1. Adapter la fonction qui évalue une expression écrite en notation polonaise inversée pour traiter des booléens, et les opérations "et", "ou", "non", "implique".
2. (**) Puis prendre en compte des tests simples à base de "<", ">" et "=" sur des nombres entiers.
   Par exemple « `2 3 < 5 7 > ou` » sera évalué en « vrai ».

Exercice 2. ** Passage à l'écriture normale

Le but est de transformer une expression écrite en notation polonaise inversée en une écriture classique. Par exemple Le principe est simplement de procéder comme lorsqu'on effectue le calcul, sauf qu'on lieu d'effectuer les opérations, on créera la chaîne de caractère correspondante.

Exercice 3. *! Des chiffres et pas de lettres**

Le but est de prendre une liste l de lexèmes et un nombre m et de trouver un calcul utilisant ces lexèmes pour obtenir ce nombre. Par exemple pour obtenir 2 à l'aide de +, 1 et 1, on peut faire 1 + 1 (c'est-à-dire 1 1 + en notation postfixée).

Le but est d'obtenir la liste de toutes les possibilités. Chaque possibilité étant elle-même une liste de lexèmes, représentant le calcul en notation polonaise inversée.

La stratégie sera d'essayer toutes les permutations de l et de garder celles qui correspondent à une expression post-fixée valide, et dont le résultat est m.

1. Récupérer une fonction `calcule` d'évaluation d'expression post-fixée. Il est ici indispensable que cette fonction lève une exception en cas de formule mal construite.
2. Écrire une fonction `essai` prenant en entrée une pile contenant une expression post-fixée et un entier et regardant si le calcul de cet expression donne cet entier.
   Si c'est le cas, on renverra la liste contenant uniquement cette pile.
   Si ce n'est pas le cas, ou si l'expression est mal construite (c'est-à-dire si `calcule` lève une exception), on renverra une liste vide.
   On donne la syntaxe pour gérer les exceptions :

```
try
    à faire si tout va bien
with
    |_ -> à faire en cas d erreur
```

   Remarque : En fait, dans le "with", on peut filtrer selon le type d'erreur. D'où le "`|_ ->`".
3. (***!) Écrire une fonction prenant une liste `l` et renvoyant la liste des permutations de `l`. On pourra utiliser deux fonctions mutuellement récursives. (En fait on est en train de parcourir l'arbre des possibilités...) À chaque appel la fonction preincipale prendra en arguments la liste des élémets de `l` déjà pris et la liste des éléments restant. Quant à la fonction auxiliaire, elle sera chargée de parcourir la liste des éléments restant et de réappeler à chaque fois la fonction principale.
4. Finalement écrire une fonction pour résoudre le problème posé.
5. *bonus* Dans la vraie version du jeu on ne donne qu'une liste d'entier, et on autorise autant d'opérations qu'on veut. Créer une fonction pour résoudre cette version.

2 Autre utilisation des piles

Exercice 4. ** Parenthésage

Dans cet exercice, on demande des fonctions impératives, et on utilisera une pile impérative.
Le but est de prendre en entrée une chaîne de caractères et de déterminer si elle est bien parenthésée.

1. Pour commencer on ne prend en compte que les parenthèses classiques "(" et ")". Il n'y a pas besoin de pile, un simple entier contenant le nombre de parenthèses ouvertes mais pas encore fermées rencontrées suffira.

2. Maintenant, on suppose qu'il y a différent type de parenthèses, mettons les parenthèses simples et les accolades. Par exemple, la chaîne suivante est mal parenthésée : " `{bla (blabla} blu)`".

   On propose alors d'utiliser une pile de caractères. Chaque parenthèse ouvrante sera empilée. Et lorsqu'on rencontre une parenthèse fermante, on dépile et on vérifie que le type de parenthèse ouvrante dépilé est le bon.

Exercice 5. ** Exemple de dérécursivation

1. On reprend le problème des tours de Hanoï. Le but est d'écrire une fonction non récursive pour afficher les opérations à faire pour le résoudre.

   On utilisera une pile `aFaire` contenant les opérations restant à faire. L'opération déplacer n disque de la tige i vers la tige j sera représentée par le triplet (n, i, j). Ainsi, initialement, la pile contiendra $(n, 0, 2)$.

   On utilisera une boucle "tant que la pile est non vide". A chaque étape, on regardera la tâche suivante à effectuer. S'il s'agit de déplacer un disque, on affichera directement le déplacement à effectuer. Sinon, on remplacera dans la pile la tâche de déplacer k disque par une suite de tâches déplaçant $k-1$ disques.
2. (***) De même, dérécursifier le tri fusion. La difficulté supplémentaire est qu'il faut gérer les valeurs renvoyées par les appels intermédiaires. Pour ce, on pourra utiliser une deuxième pile chargée de garder les résultats intermédiaires, tout comme dans l'évaluation d'une expression postfixée.