Informatique MP
TP
Piles LIFO, Files FIFO.

Antoine MOTEAU
antoine.moteau@wanadoo.fr

. . . /Piles-C.tex (2003)
. . . /Piles-C.tex Compilé le samedi 10 mars 2018 à 14h 44m 50s avec LaTeX.
Compilable avec LaTeX, PDFLaTeX, LuaLaTeX, XeLaTeX.

Reprise sur document antérieur : (2000-2001 écrit en Word)

Ordre du TP : début d'année (premier TP)

Eléments utilisés :

- listes
- vecteurs
- chaînes de caractères
- flux (notions élémentaires)
- récursivité
- ressources graphiques
- ressources de format (printf)

Documents relatifs au TP :

- Texte du TP : .tex, .dvi, .pdf
- Squelette de programme Caml : Piles-S.ml
- Programme corrigé Caml : Piles-C.ml

Antoine MOTEAU – Lycée POTHIER – Option Informatique MP
TP n^o

PILES LIFO, FILES FIFO.

Une pile est une structure de données, destinée à stocker de façon provisoire des informations : un ou plusieurs processus font entrer (instruction "empiler" ou "push") des données dans la pile en vue de leur récupération ultérieure (instruction "dépiler" ou "pop") par eux mêmes ou par d'autres processus.
Bien que les instructions d'empilement et de dépilement interviennent de façon irrégulière, chacun des processus doit retrouver dans la pile les informations qui lui sont destinées.

Nous nous intéresserons ici à deux types de piles simples, à structure de données linéaire (ou quasi-linéaire) : les piles LIFO (Last In First Out) et les piles (ou files) FIFO (First In First Out).

Les ressources pour ce TP sont (peut-être) dans le dossier **. . . /MPx/.../TP-Info/Piles/** et après avoir copié (tout) le dossier **/Pile/** dans votre dossier personnel, vous y trouverez les fichiers :

- `Piles-S.pdf` : ce texte,
- `Piles-S.ml` : fichier Caml à compléter (squelette de programme).

1 Description

1.1 Piles LIFO (Last In First Out, dernier entré premier sorti)

1.1.1 Concept

Le concept des piles LIFO est illustré simplement par l'exemple des piles d'assiettes : l'assiette que l'on dépile est toujours celle qui a été empilée en dernier.
Bien que les piles LIFO utilisent une structure de données rudimentaire, cette notion est fondamentale en informatique : c'est souvent par ce moyen que les processus communiquent entre eux, sauvegardent les informations destinées à assurer le retour des sous-processus, transmettent les données et récupèrent les résultats, parfois au prix de quelques acrobaties : lorsque la donnée dont on a besoin n'est pas au sommet de la pile, il faut dépiler les données qui la précèdent, lire ou dépiler la donnée dont on a besoin, et rempiler dans le bon ordre ce qui avait été dépilé au préalable (exemple d'une pile d'assiettes où l'on a mélangé les assiettes creuses et les assiettes plates).

1.1.2 Objectif

Nous allons ici illustrer le fonctionnement d'une pile LIFO dans le cas de programmes récursifs simples (factorielle, suite de Fibonacci), en simulant une pile LIFO où l'on empilera et/ou dépilera des données lors des appels et des retours. Cette simulation nous permettra de suivre pas à pas l'évolution du contenu de la pile.

1.1.3 Implantation d'une pile LIFO homogène

On utilisera les ressources (minimales) suivantes :

- Structure de donnée : liste Une structure de liste est tout à fait adaptée à la représentation d'une pile LIFO, l'élément de tête de liste, seul élément naturellement accessible, étant l'élément du sommet de la pile. La taille d'une pile LIFO pourra croître indéfiniment (dans la mesure de la capacité mémoire . . .).
- Méthodes :
  - création d'une pile vide `new`
  - empilement (au sommet) `push`
  - dépilement (du sommet) `pop`
  - prédicats (état, taille) `empty`, `full`, `length`
  - destruction d'une pile (vide) . . .
- Méthodes temporaires, pour la simulation
  - édition du contenu voir . . .

Remarques.

- Le fonctionnement d'une pile LIFO est rarement sécurisé : si un processus fait une erreur en manipulant la pile, tous les processus qui utilisent cette pile seront perturbés.
- Les erreurs communes, hors erreurs d'interprétation, sont
  - le débordement de pile ("stackoverflow"), du au dépassement de la capacité de la mémoire,
  - la tentative de dépilement d'une pile vide ("stackunderflow").

1.2 Files FIFO (First In First Out, premier entré premier sorti)

1.2.1 Concept

Le concept des piles (ou files) FIFO est illustré simplement par l'exemple des distributeurs automatiques de café (le gobelet que l'on dépile est toujours celui du dessous, qui a été empilé en premier) ou encore par la file d'attente à un guichet, matérialisé par un "Serre file" (premier arrivé, premier servi).
La structure d'une file FIFO est plus complexe que celle d'une pile LIFO et on peut la concevoir

- comme une structure à capacité limitée (ce sera le plus simple)
- comme une structure dont la capacité évolue en fonction des besoins (c'est plus compliqué)

Les files FIFO sont largement utilisées dans tous les domaines où l'ordre, la chronologie ont de l'importance.
Par exemple, c'est à l'aide d'une file FIFO que l'on gère la saisie des caractères au clavier : les caractères saisis sont "enfilés" dans une file d'attente (de capacité limitée, ce qui explique le "beep" lorsque la file est pleine), en attendant d'être pris, dans l'ordre où ils sont entrés.

1.2.2 Objectif

Nous illustrerons le concept de file FIFO (à capacité limitée) en simulant le fonctionnement de la réception des caractères frappés sur un clavier d'ordinateur et en éditant le contenu de la file, pour suivre pas à pas son évolution.

1.2.3 Implantation d'une file FIFO homogène, à capacité limitée

On utilisera les ressources (minimales) suivantes :

- Structure de donnée : vecteur de taille n fixe (à indices allant de 0 à $n-1$),
  - interprété de manière "circulaire" : les indices sont calculés modulo n (le suivant de $n-1$ est 0),
  - accompagné de deux indices de pointage :
    - indice du premier entré "`first`" (donc premier à sortir),
    - indice du premier à entrer "`next`",

    le contenu actif de la file allant de l'indice "`next`"+1 à l'indice "`first`" (modulo n).
    Remarque. On fera attention à ne pas confondre une file vide avec une file pleine !
- Méthodes :
  - création d'une file vide `new`
  - prédicat file vide `empty` `(first = next)`
  - prédicat file pleine `full` `(first = next+1)`
  - empilement `push`
  - dépilement `pop`
  - destruction d'une file (vide) ...
- Méthodes temporaires, pour la simulation
  - édition du contenu `voir` ...

Remarque. En général, les files FIFO ne sont pas partagées entre plusieurs processus et le fonctionnement est sécurisé grace aux méthodes "`empty`" et "`full`" (prédicats).

2 Quelques rappels et compléments Caml (listes et vecteurs)

2.1 Listes en Caml

2.1.1 Instruction `do_list` (il ne faut pas l'utiliser au concours !)

1. Extrait de l'aide Caml

   `value do_list : ('a -> 'b) -> 'a list -> unit`
   do_list f [a1 ; ... ; an] applies function f in turn to a1 ; ... ; an, discarding all the results. It is equivalent to `begin` f a1 ; f a2 ; ... ; f an ; () `end`.
2. Exemples
   (a) Edition d'une liste d'entiers, avec un format :

   ```
   let print_int_list L = do_list (fun x -> printf " %5d" x ) L ;;
   ```

   ou, en omettant les arguments

   ```
   let print_int_list = do_list (printf " %5d") ;;
   (* print_int_list : int list -> unit = <fun> *)
   ```

(b) Edition d'une liste de couples d'entiers, sous forme "(x,y)", avec séparation par un espace

```
let print_int_int_list = do_list f
   where f = fun (x,y) -> printf " (%3d,%3d)" x y
;;
(* print_int_int_list : (int * int) list -> unit = <fun> *)
```

3. Exercice (à terminer avec un test)

(a) Sans utiliser `do_list`, écrire

- `print2_int_list : int list -> unit = <fun>`
  qui donne le même résultat que `print_int_list`
- `print2_int_int_list : (int * int) list -> unit = <fun>`
  qui donne le même résultat que `print_int_int_list`

(b) Expliciter la complexité de `print2_int_list` et de `print2_int_int_list`

2.1.2 Instruction `it_list` (il ne faut pas l'utiliser au concours !)

1. Extrait de l'aide Caml

```
value it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
it_list f a [b1 ; ... ; bn] is f (... (f (f a b1) b2) ...) bn.
```

2. Exemple : Calcul de la somme et du produit des éléments d'une liste d'entiers :

```
let L = [2; 4; 50; 105; 23; 44; 55; 12; -17; 25; 26];;
let S = it_list (function x -> function y -> x+y) 0 L;;
let P = it_list (function x -> function y -> x*y) 1 L;;
```

3. Exercice (à terminer avec un test)

(a) Sans utiliser `it_list`, écrire

- `sum_int_list : int list -> int = <fun>`
  qui calcule la somme des éléments d'une liste d'entiers
- `prod_int_list : int list -> int = <fun>`
  qui calcule le produit des éléments d'une liste d'entiers

(b) Expliciter la complexité de `sum_int_list` et de `prod_int_list`

2.2 Vecteurs en Caml

2.2.1 Instruction `do_vect` (il ne faut pas l'utiliser au concours !)

1. Extrait de l'aide Caml

```
value do_vect : ('a -> 'b) -> 'a vect -> unit
do_vect f v applies function f in turn to all the elements of v, discarding all the results :
f v.(0) ; f v.(1) ; ... ; f v.(vect_length v - 1) ; ().
```

2. Exemple : Calcul de la somme et du produit des éléments d'un vecteur d'entiers :
(utilisation de `do_vect` avec un effet de bord)

```
let V = [|2; 4; 50; 105; 23; 44; 55; 12; -17; 25; 26|];;
let S = let u = ref 0 in do_vect (fun x -> u := !u+x) V; !u;;
let P = let m = ref 1 in do_vect (fun x -> m := !m*x) V; !m;;
```

3. Exercices (à terminer avec un test)

(a) Sans utiliser `do_vect`, écrire

- `sum_float_vect : float list -> float = <fun>`
  qui calcule la somme des éléments d'un vecteur de float
- `prod_float_vect : float list -> float = <fun>`
  qui calcule le produit des éléments d'un vecteur de float

(b) Expliciter la complexité de `sum_float_vect` et de `prod_float_vect`

3 Simulation, visualisation d'une pile LIFO, lors d'appels récursifs

3.1 Calcul de factorielles (récurrence simple)

3.1.1 Rappel : écriture récursive d'une fonction de calcul de factorielle

```
let rec Factorielle = function  | n when n < 0 -> failwith "Erreur"
     | 0            -> 1
     | n            -> n * (Factorielle (n-1));;
(* Factorielle : int -> int = <fun> *)
```

Cette écriture utilise la pile du système pour { stocker les résultats intermédiaires ; gérer les adresses d'exécution (adresses de retour) }

3.1.2 Simulation et visualisation

On récrit ce calcul sans le renvoi de résultat, en se contentant d'empiler et de dépiler uniquement des valeurs dans une pile LIFO qui est une liste (variable globale), la pile du système n'étant plus utilisée (par la récursivité) que pour gérer les adresses d'exécution.

```
let LIFO = ref [];;                    (* pile LIFO d'entiers == int list *)
let rec Factopile = function | n when n < 0 -> failwith "Erreur"
    | n            -> ... entrée ...                  (* A ECRIRE (voir ci dessous) *)
                      begin match n with
                              | 0 -> ...                (* A ECRIRE (voir ci dessous) *)
                              | n -> ...                (* A ECRIRE (voir ci dessous) *)
                      end;
                      ... sortie ...                    (* A ECRIRE (voir ci dessous) *)
;;
(* Factopile : int -> unit = <fun> *)  (* ne renvoie pas de résultat, mais gère la pile *)

let Facto2 =
  LIFO := [ ];   (* initialise la pile LIFO    *)
  Factopile n;   (* calculs, dans la pile LIFO *)
  hd !LIFO       (* résultat = dernier calcul et seule valeur contenue dans la pile LIFO *)
;;
(* Facto2 : int -> int = <fun> *)
```

On utilisera la pile LIFO pour stocker deux informations { la valeur de n (repérage du numéro de l'appelant) ; le calcul de factorielle (état du calcul) }

Dans `Factopile n`, on doit réaliser (dans cet ordre, pour chaque n) les actions :

1. entrée : { introduire le numéro n dans la pile (push) : repérage de l'appelant ; éditer le contenu de la pile (visualisation de l'état de la pile lors des entrées), sur une ligne }
2. calcul et gestion du résultat dans la pile (avec dépilages/empilages) :
   - Pour $n = 0$, remplacer le sommet de la pile (le numéro) par la valeur 1 (pop, push)
   - Pour $n > 0$,
     (a) appeler `Factopile (n-1)`, qui se termine en laissant la valeur $(n-1)!$ au sommet de la pile
     (b) enlever du sommet de la pile la valeur V $(= (n-1)!)$ puis le numéro (soit n) (pop et pop)
     (c) calculer $n * V$ et introduire ce résultat au sommet de la pile (push)
3. sortie : éditer le contenu de la pile (visualisation de l'état de la pile lors des retours), sur une ligne

Opération "push n" :	`LIFO := n:: !LIFO;`
Opération valeur "pop" :	`begin match !LIFO with a::q -> LIFO := q; a end;`
"pop (a) + pop + push $n \times a$" :	`begin match !LIFO with a::b::q -> LIFO := (n*a)::q end;`
Edition du contenu de la pile :	`print_int_list !LIFO; print_newline ();`

1. Ecrire `Factopile`
2. Visualiser l'évolution de la pile en calculant par exemple `Facto2 10`
3. Quelle est la complexité du calcul de "Factorielle n" (en nombre de multiplications) ?

3.2 Suite de Fibonacci (récurrence linéaire double)

La suite de Fibonacci est définie par la relation de récurrence linéaire double : $\begin{cases} u_0 = 1 \; ; \; u_1 = 1 \\ \forall n \in \mathbb{N}, \; u_{n+2} = u_{n+1} + u_n \end{cases}$

3.2.1 Suite de Fibonacci : programmation déplorable ...

1. Ecriture initiale :

```
let rec Fibonacci_déplorable = function | n when n < 0      -> failwith "Erreur"
    | (0 | 1)           -> 1
    | n                 -> Fibonacci_déplorable (n-1) + Fibonacci_déplorable (n-2)
;;
(* Fibonacci_déplorable : int -> int = <fun> *)
```

Cette (mauvaise) écriture utilise la pile du système pour $\begin{cases} \text{stocker les résultats intermédiaires} \\ \text{gérer les adresses d'exécution (adresses de retour)} \end{cases}$

> Quelle est la complexité de `Fibonacci_déplorable` ?

2. Simulation du calcul récursif de `Fibonacci_déplorable n`, à l'aide d'une pile LIFO qui est une liste :

```
let LIFO = ref [];;                         (* pile LIFO d'entiers == int list *)
let rec Fibopile_dép = function | n when n < 0 -> failwith "Erreur"
    | n             -> ... entrée ...                   A ECRIRE
                       begin match n with
                             | (0 | 1) -> ...          A ECRIRE
                             | n       -> ...          A ECRIRE
                       end;
                       ... sortie ...                   A ECRIRE
;;
(* Fibopile_dép : int -> unit = <fun> *)   (* ne renvoie pas de résultat,
                                              mais gère la pile *)

let Fibo_dép =
  LIFO := [];           (* initialise la pile   *)
  Fibopile_dép n;       (* calculs dans la pile *)
  hd !LIFO              (* résultat = seule valeur contenue dans la pile *)
;;
(* Fibo_dép : int -> int = <fun> *)
```

On utilise exactement le même principe que celui mis en place pour la factorielle, ici avec les appels successifs de `Fibopile_dép (n-2)` puis `Fibopile_dép (n-1);`.

ATTENTION ! , il y a récursivité double, donc au retour du double appel :
`Fibopile_dép (n-2); Fibopile_dép (n-1);` (exécutés dans cet ordre),
la pile contient, dans l'ordre, les valeurs u_{n-1}, u_{n-2}, n, qu'il faut "dépiler".

> (a) Ecrire `Fibopile_dép`
> (b) Visualiser et commenter l'évolution de la pile en calculant par exemple `Fibo_dép 8`
> (Au delà, ce n'est même pas la peine d'essayer !)

3.2.2 Rappels de Sup ...

1. Programmation itérative de la suite de Fibonacci

```
let fiboit n =  let u = ref 1 and v = ref 1 in
   for i = 2 to n
   do let tampon = !v in
      v := !u + !v; u := tampon
   done;
   !v
;;
(* fiboit : int -> int = <fun> *)
```

> - Apporter la preuve de cet algorithme
> - Quelle en est la complexité ?

2. Programmation fonctionnelle (suites de Fibonacci)

```
let fibof = fiboluxe 1 1
  where rec fiboluxe a b = function
   |0 -> a
   |1 -> b
   |n -> fiboluxe b (a+b) (n-1)
;;
(* fibof : int -> int = <fun> *)
```

En utilisant l'espace vectoriel des suites définies par :

$$\begin{cases} u_0 = a;\ u_1 = b \quad \text{avec } (a,b) \in \mathbb{R}^2 \\ \forall n \in \mathbb{N},\ u_{n+2} = u_{n+1} + u_n \end{cases}$$

- Où est l'argument de `fibof` ?
- Apporter la preuve de cet algorithme
- Quelle en est la complexité ?

3.2.3 Suite de Fibonacci : récurrence vectorielle d'ordre 1

La récurrence peut s'écrire sous forme vectorielle : $U_{n+1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} U_n$ avec $U_n = \begin{pmatrix} u_{n+1} \\ u_n \end{pmatrix}$

1. Ecriture initiale :

```
let FibonacciV n = snd (FiboV n)
  where rec FiboV = function | n when n < 0 -> failwith "Erreur"
    | 0            -> 1,1
    | n            -> let u, v = FiboV (n-1) in (u+v, u)
;;
(* FibonacciV : int -> int = <fun> *)
```

Cette (bonne) écriture utilise la pile du système pour $\begin{cases} \text{stocker les résultats intermédiaires} \\ \text{gérer les adresses d'exécution (adresses de retour)} \end{cases}$

Quelle est la complexité de `FibonacciV` ?

2. Simulation du calcul récursif de `FibonacciV` à l'aide d'une pile LIFO :
   Rigoureusement identique à ce qui a été fait pour Factorielle, avec une pile qui est une liste, sauf que
   - les données de la pile (`LIFO2`) sont des couples d'entiers.
   - pour respecter l'homogénéité, le numéro n (d'appelant) est empilé sous la forme du couple $(n,0)$
   - l'édition de la pile se fait avec `print_int_int_list !LIFO2; print_newline ();`

```
let LIFO2 = ref [];;
let rec Fibopile_V = function  | n when n < 0 -> failwith "Erreur"
  | n -> LIFO2 := (n,0)::!LIFO2;      (* ne sert qu'à identifier l'appelant *)
         print_int_int_list !LIFO2; print_newline ();  (* pile en croissance *)
         begin match n with
                | 0 -> begin ...                (* A ECRIRE 2 à 3 lignes *)
                       end
                | n -> ...                      (* A ECRIRE 3 à 4 lignes *)
         end;
         print_int_int_list !LIFO2; print_newline ()  (* pile en décroissance)
;;
(* Fibopile_V : int -> unit = <fun> *)

let FiboV n = LIFO2 := []; Fibopile_V n; snd (hd !LIFO2)
;;
(* FiboV : int -> int = <fun> *)
```

(a) Ecrire, selon le même principe que précédemment, les fonctions
   - `Fibopile_V : int -> unit = <fun>`
   - `FiboV : int -> int = <fun>`

(b) Visualiser l'évolution de la pile en calculant par exemple `FiboV 8`

(c) Commenter ...

4 Simulation, visualisation d'une pile (file) FIFO : buffer de clavier

On ne peut pas à la fois gérer le flux en provenance du clavier et utiliser les touches du clavier pour commander la simulation de ce flux ! On gérera donc un pseudo-clavier :

- les caractères alimentant le buffer seront pris dans un FLUX préparé à l'avance (flux d'entrée),
- les caractères extraits du buffer seront stockés dans une chaîne de caractères initialement vide.

Le vrai clavier servira à commander (de façon un peu simpliste) la simulation :

- "12 <return>" fera prendre au plus 12 caractères de la tête du flux pour les introduire dans le buffer,
- "- 7 <return>" fera sortir au plus 7 caractères du buffer vers le bout de la chaîne de réception.

Les états de la simulation seront édités dans la fenêtre graphique de Caml, ce qui évitera de mélanger ce qui concerne la simulation du clavier (pseudo-clavier) et l'écho des commandes saisies au vrai-clavier.

4.1 Information rapide sur les flux (stream)

Un flux (d'entrée) est une structure linéaire de données, dont les éléments peuvent être de types différents.

- Un flux est lu grâce à un dispositif de filtrage (destiné à reconnaître la nature de la donnée qui se trouve en tête du flux avant de l'extraire du flux).
- La lecture d'un flux est destructive (l'élément de tête, un fois lu, est extrait du flux et c'est le suivant qui devient l'élément de tête).

Remarque. Contrairement à ce qui se passe pour les listes, le contenu n'est pas forcément homogène et on ne peut pas copier l'élément de tête du flux sans l'extraire du flux !

De même, on aura des flux en sortie ...

4.2 Exemple de flux, syntaxe

1. Lecture d'un flux de char (de petite taille) pour le cumuler dans une chaîne de caractères :

```
let rec string_from_char_stream m = function
   | [<'c ; (string_from_char_stream "") m1 >] -> m ^ (string_of_char c) ^ m1
   | [< >] -> m
;;
(* string_from_char_stream : string -> char stream -> string = <fun> *)
```

   Remarque. "(string_from_char_stream u) m1" signifie : "string_from_char_stream u" appliqué au reste du flux et dont le résultat est désigné par "m1"

2. Filtrage d'un flux de char pour le transformer en un autre flux (échange Majuscules et minuscule)

```
let rec transforme = function
 | [< '('A'..'Z' as c); transforme r >] -> [< 'char_of_int ((int_of_char c) +32); r >]
 | [< '('a'..'z' as c); transforme r >] -> [< 'char_of_int ((int_of_char c) -32); r >]
 | [< 'c ; transforme r >]              -> [< 'c ; r >]
 | [<>]                                 -> [< >]     (* sinon, renvoie un flux vide *)
;;
(* transforme : char stream -> char stream = <fun> *)
```

3. Exemple

```
let the_flux = stream_of_string
   "Il était Un PetiT naVire, qui NaviGait sur les FLOTS, HO hé hO hE MateLOT 123";;
string_from_char_stream "" (transforme the_flux);;
```

4.3 Simulation d'un buffer de clavier, avec visualisation des états

4.3.1 Ressources globales

```
#open "graphics";;              (* ressource de son et de visualisation *)
let N = 17;;                        (* taille de la file FIFO (buffer de 16 char utiles) *)
let Buffer = make_string N '_';;    (* file FIFO *)
let p = ref 0;;         (* indice du caractère prêt à sortir, compris entre 0 et N *)
let q = ref 0;;         (* indice où doit entrer le prochain caractère, compris entre 0 et N *)
let ssout = ref  "";;   (* chaîne de réception, à remplir par extraction du buffer *)
```

4.3.2 Entrée et sortie de caractères

```
let rec entree_buffer the_flux n =
  if n > 0
  then if (!q+1) mod N = !p then sound 1000 10   (* "beep" *)
       else match the_flux with
              | [< 'c >] -> Buffer.[!q] <- c; q := (!q+1) mod N; entree_buffer the_flux (n-1)
              | [< >]    -> ()
;;
(* entree_buffer : char stream -> int -> unit = <fun> *)
```

`entree_buffer F n` lit jusqu'à n caractères du flux F vers le buffer (file FIFO de 16 char, "circulaire"), en faisant évoluer l'indice d'entrée q. Lorsque le buffer est plein (q+1 = p (mod N)), entree_buffer émet un "beep" et se termine.

```
let rec sortie_buffer n =                                  A ECRIRE (6 lignes)
;;
(* sortie_buffer : int -> unit = <fun> *)
```

`sortie_buffer n` extrait jusqu'à n caractères du buffer (file FIFO) vers le bout de la chaîne ssout, en faisant évoluer l'indice de sortie p (...`ssout := !ssout ^ (string_of_char Buffer.[!p] );`...).
Lorsque le buffer est vide (p = q (mod N)), sortie_buffer émet un "beep" (`sound 1000 10;`) et se termine.

4.3.3 Simulateur du fonctionnement d'un buffer de clavier

```
let Simulation_Clavier ssin =       (* simulation simpliste *)
  clear_graph ();
  p := 0; q := 0;                                  (* buffer vide *)
  ssout := "";                                     (* chaîne de sortie vide *)
  let flux_d'entrée = stream_of_string ssin in     (* flux d'entrée *)

  let n = ref 1 in while !n <> 0                   (* boucle initialisée *)
  do  n := read_int ();                                (* n = nb char à transférer *)
      if !n > 0 then entree_buffer flux_d'entrée !n;
      if !n < 0 then sortie_buffer (- !n);
      set_color red; moveto 10 300; draw_string Buffer;              (* état buffer  *)
      let w = make_string N ' ' in w.[!p] <- 'p'; w.[!q] <- 'q';
         set_color green; moveto 10 280; draw_string w;              (* état indices *)
      set_color blue; moveto 10 260; draw_string !ssout;             (* état sortie  *)
  done
;;
(* Simulation_Clavier : string -> unit = <fun> *)
```

- initialisations : buffer "vide" (indices p et q nuls), chaîne de réception vide, création du flux d'entrée.
- Boucle initialisée : lit au (vrai) clavier un entier n, par exemple, -12 ou 10 (surtout pas +10)
  - si $n > 0$, tente de faire d'entrer n caractères (pris dans le flux) dans le buffer
  - si $n < 0$, tente de faire sortir $-n$ caractères, du buffer vers la chaîne de réception
  - $n = 0$ sera le signal d'abandon de la simulation (fin de boucle)
- édite (dans la fenêtre graphique) le contenu du buffer, les indices et la chaîne en sortie, positionnés les uns au dessus des autres :
  - buffer, de l'indice 0 à l'indice $N-1$ (en rouge)
  - indices p et q nommés et positionnés (en vert)
  - contenu de la chaîne de sortie (en bleu)

  Remarque. On ne peut pas visualiser le contenu du flux d'entrée puisque sa lecture est destructrice.

1. Compléter les fonctions (parties "A ECRIRE")
2. Exécuter `Simulation_Clavier ssin`, où ssin est une chaîne de caractères assez longue, et visualiser dans la fenêtre graphique le fonctionnement du buffer.

< FIN >

5 Squelette du programme (à compléter), Non éxécutable !

```
#open "graphics";;      (* bibliothèque Caml : ressources graphiques *)
#open "printf";;        (* bibliothèque Caml : éditions formatées *)

let A_ECRIRE s = print_string s; print_newline ();;       (* Avertissement interne *)
(* A_ECRIRE : string -> unit = <fun> *)

(* === 1. Descriptions  === *)
(* === 2. Listes  et Vecteurs === *)
(* === 2.1. Listes *)
let LI1 = [1;2;3;4;5;6;7] and LI2 = [1,-1;2,-2;3,-3;4,-4;5,-5;6,-6;7,-7];;

(* === 2.1.1. do_list *)
let print_int_list = do_list (printf " %5d");;
(* print_int_list : int list -> unit = <fun> *)

let print_int_int_list = do_list f where f = fun (x,y) -> printf " (%3d,%3d)" x y;;
(* print_int_int_list : (int * int) list -> unit = <fun> *)
(*=================================================================*)
(* >>>>>>>>>>>>>>>>  NON CORRECT A PARTIR D'ICI <<<<<<<<<<<<<<<<<< *)
(*=================================================================*)
let rec print2_int_list = function x -> A_ECRIRE                    "Ecrire 2 lignes"
;;
(* print2_int_list : int list -> unit = <fun> *)

let rec print2_int_int_list = function x -> A_ECRIRE                "Ecrire 2 à 3 lignes"
;;
(* print2_int_int_list : (int * int) list -> unit = <fun> *)

print_int_list LI1;;            print2_int_list LI1;;
print_int_int_list LI2;;        print2_int_int_list LI2;;

(* === 2.1.2. it_list *)
let S = it_list (function x -> function y -> x+y) 0 LI1;;
let P = it_list (function x -> function y -> x*y) 1 LI1;;

let rec sum_int_list = function x -> A_ECRIRE                       "Ecrire 2 lignes"
;;
(* sum_int_list : int list -> int = <fun> *)

let rec prod_int_list = function x -> A_ECRIRE                      "Ecrire 2 lignes"
;;
(* prod_int_list : int list -> int = <fun> *)

sum_int_list LI1;;        prod_int_list LI1;;

(* === 2.1. Vecteurs *)
(* === 2.1.1. do_vect *)
let V = [|2; 4; 50; 105; 23; 44; 55; 12; -17; 25; 26|];;
let S = let u = ref 0 in do_vect (fun x -> u := !u+x) V; !u;;
let P = let m = ref 1 in do_vect (fun x -> m := !m*x) V; !m;;

let sum_float_vect v = function x -> A_ECRIRE                       "Ecrire 3 lignes"
;;
(* sum_float_vect : float vect -> float = <fun> *)

let prod_float_vect v = function x -> A_ECRIRE                      "Ecrire 3 lignes"
;;
(* prod_float_vect : float vect -> float = <fun> *)

let VF1 = [| 1.2; 2.3; 5.4; -5.3; 2.7; 6.3 |];;
sum_float_vect VF1;;  prod_float_vect VF1;;
```

```
(* === 3. Pile LIFO  === *)
(* === 3.1. Factorielle *)
let rec Factorielle = function | n when n < 0 -> failwith "Erreur"
  | 0 -> 1
  | n -> n * Factorielle (n-1)
;;
(* Factorielle : int -> int = <fun> *)
Factorielle 5;;

let LIFO = ref [];;
let rec Factopile = function | n when n < 0 -> failwith "Erreur"
  | n -> A_ECRIRE                                                    "Ecrire 2 lignes"
         begin match n  with
                | 0 -> A_ECRIRE                                     "Ecrire 3 à 4 lignes"
                | n -> A_ECRIRE                                     "Ecrire 4 lignes"
         end;
         A_ECRIRE                                                   "Ecrire 1 ligne"
;;
(* Factopile : int -> unit = <fun> *)
let Facto2 n = LIFO := []; Factopile n; hd !LIFO
;;
(* Facto2 : int -> int = <fun> *)

Facto2 8;;      !LIFO;;

(* === 3.2. Nombres de Fibonacci *)
(* === 3.2.1. Récurrence double *)
let rec Fibonacci_déplorable = function  | n when n < 0 -> failwith "Erreur"
  | (0|1) -> 1
  | n     -> Fibonacci_déplorable (n-1) + Fibonacci_déplorable (n-2)
;;
(* Fibonacci_déplorable : int -> int = <fun> *)

Fibonacci_déplorable 8;;        Fibonacci_déplorable 10;;

let rec Fibopile_dép = function  | n when n < 0 -> failwith "Erreur"
  | n ->  A_ECRIRE                                                   "Ecrire 2 lignes"
          begin match n with
                | (0 | 1)  ->  A_ECRIRE                              "Ecrire 4 lignes"
                | n        ->  A_ECRIRE                             "Ecrire 4 lignes"
          end;
          A_ECRIRE                                                  "Ecrire 1 ligne"
;;
(* Fibopile_dép : int -> unit = <fun> *)

let Fibo_dép n = LIFO := []; Fibopile_dép n; hd !LIFO
;;
(* Fibo_dép : int -> int = <fun> *)

Fibo_dép 5;;    Fibo_dép 8;;

(* === 3.2.2. Rappels de Sup *)
(* === 3.2.2.1. Programmation itérative de la suite de Fibonacci *)
let fiboit n =
   let u = ref 1 and v = ref 1 in
   for i = 2 to n do let tampon = !v in v:= !u + !v; u := tampon done;
   !v
;;
(* fiboit : int -> int = <fun> *)
fiboit 8;;
```

```
(* === 3.2.2.1. Programmation fonctionnelle des suites de Fibonacci *)
let fibof = fiboluxe 1 1
  where rec fiboluxe a b = function
   |0 -> a
   |1 -> b
   |n -> fiboluxe b (a+b) (n-1)
;;
(* fibof : int -> int = <fun> *)
fibof 8;;
(* === 3.2.3. Récurrence vectorielle simple *)
let FibonacciV n = snd (FiboV n)
  where rec FiboV = function
    | n when n < 0 -> failwith "Erreur"
    | 0 -> 1,1
    | n -> let u, v = FiboV (n-1) in (u+v,u)
;;
(* FibonacciV : int -> int = <fun> *)
FibonacciV 8;;

let LIFO2 = ref [];;
let rec Fibopile_V = function | n when n < 0 -> failwith "Erreur"
  | n -> LIFO2 := (n,0)::!LIFO2;
         print_int_int_list !LIFO2; print_newline ();
         begin match n with
                | 0 ->  A_ECRIRE                                    "Ecrire 4 lignes"
                | n ->  A_ECRIRE                                    "Ecrire 4 lignes"
         end;
         print_int_int_list !LIFO2; print_newline ()
;;
(* Fibopile_V : int -> unit = <fun> *)

let FiboV n = LIFO2 := []; Fibopile_V n; snd (hd !LIFO2)
;;
(* FiboV : int -> int = <fun> *)

FiboV 5;;       FiboV 8;;       !LIFO2;;

(* === 4. Pile FIFO (clavier) === *)
(* === 4.1. Information sur les flux *)
(* === 4.2. Exemple simple de flux *)

(* lecture d'un flux de char (de petite taille) produisant une chaîne de caractères *)
let rec string_from_char_stream m = function
   | [<'c ; (string_from_char_stream "") m1 >] -> m ^ (string_of_char c) ^ m1
   | [< >] -> m
;;
(* string_from_char_stream : string -> char stream -> string = <fun> *)

(* filtrage d'un flux de char (flux d'entrée) pour produire un flux de char (flux de sortie)
   (avec échange des Majuscules et minuscules) *)
let rec transforme = function
   | [< '('A'..'Z' as c) ; transforme r >] -> [< 'char_of_int ( (int_of_char c) + 32 ) ; r  >]
   | [< '('a'..'z' as c) ; transforme r >] -> [< 'char_of_int ( (int_of_char c) - 32 ) ; r  >]
   | [< 'c   ; transforme r >]  -> [< 'c ; r >]
   | [<>]                       -> [< >]      (* sinon, renvoie un flux vide *)
;;
(* transforme : char stream -> char stream = <fun> *)

(* exemple *)
let the_flux = stream_of_string
       "Il était Un PetiT naVire, qui NaviGait sur les FLOTS, hé hO hE HO MateLOT 123454321";;
string_from_char_stream "" (transforme the_flux);;
```

```
(* === 4.3. Simulation d'un buffer de clavier *)

#open "graphics";;
(*----------------- variables globales *)
let N = 17;;                    (* taille de la file FIFO (buffer de 16 char utiles) *)
let Buffer = make_string N `_`;; (* file FIFO *)
let p = ref 0;;                 (* indice du caractère prêt à sortir, 0 <= p <= N *)
let q = ref 0;;                 (* indice où doit entrer le prochain caractère, 0 <= q <= N *)
let ssout = ref "";;            (* chaîne à remplir *)

let rec entree_buffer the_flux n =
  if n > 0
  then if (!q+1) mod N = !p
       then sound 1000 10
       else match the_flux with
              | [< 'c >] -> Buffer.[!q] <- c; q := (!q+1) mod N; entree_buffer the_flux (n-1)
              | [< >]    -> ()
;;
(* entree_buffer : char stream -> int -> unit = <fun> *)

let rec sortie_buffer n = function A_ECRIRE                          "Ecrire 6 lignes"
;;
(* sortie_buffer : int -> unit = <fun> *)

let Simulation_Clavier ssin =       (* similation simpliste *)
  clear_graph ();
  p := 0; q := 0;                                 (* buffer vide *)
  ssout := "";                                    (* chaîne de sortie *)
  let flux_d'entrée = stream_of_string ssin in    (* flux d'entrée *)

  let n = ref 1 in
  while !n <> 0
  do  n := read_int ();
      if !n > 0 then entree_buffer flux_d'entrée !n;
      if !n < 0 then sortie_buffer (- !n);

      set_color red; moveto 10 300; draw_string Buffer;          (* état* )

      let w = make_string N ` ` in w.[!p] <- `p`; w.[!q] <- `q`;  (* état *)
      set_color green; moveto 10 280; draw_string w;

      set_color blue; moveto 10 260; draw_string !ssout;         (* état *)
  done
;;
(* Simulation_Clavier : string -> unit = <fun> *)

(* exemple d'entrée et exécution *)
let ssin = "Ceci n'est qu'un vague baratin mais je ne dirai rien d'autre, même sous la torture.";;
Simulation_Clavier ssin;;

(************************************)
(* Fin de Piles-S.ml   (Squelette)   *)
(************************************)
```

$< \mathcal{FJN} >$ **Piles** (énoncé + squelette).

6 Corrigé du TP LIFO FIFO : fonctions écrites

```
#open "graphics";; (* bibliothèque Caml : ressources graphiques *)
#open "printf";;   (* bibliothèque Caml : éditions formatées *)

(* === 1. Descriptions === *)
(* === 2. Listes  et Vecteurs === *)
(* === 2.1. Listes *)
let LI1 = [1;2;3;4;5;6;7];;  let LI2 = [1,-1;2,-2;3,-3;4,-4;5,-5;6,-6;7,-7];;

(* === 2.1.1. do_list *)
let print_int_list = do_list (fun x -> printf " %3d" x)
and print_int_int_list = do_list f where f = fun (x,y) -> printf " (%3d,%3d)" x y
;;
(* print_int_list : int list -> unit = <fun> *)
(* print_int_int_list : (int * int) list -> unit = <fun> *)

let rec print2_int_list = function
  | x::q -> printf " %3d" x; print2_int_list q
  | _    -> ()
and     print2_int_int_list = function
  | (x,y)::q -> printf " (%3d,%3d)" x y; print2_int_int_list q
  | _    -> ()
;;
(* print2_int_list : int list -> unit = <fun> *)
(* print2_int_int_list : (int * int) list -> unit = <fun> *)

print_int_list LI1;;  print2_int_list LI1;;    print_int_int_list LI2;; print2_int_int_list LI2;;

(* === 2.1.2. it_list *)
let S = it_list (function x -> function y -> x+y) 0 LI1;;
let P = it_list (function x -> function y -> x*y) 1 LI1;;

let rec sum_int_list = function
    | []   -> 0
    | a::q -> a + sum_int_list q
and     prod_int_list = function
    | []   -> 1
    | a::q -> a * prod_int_list q
;;
(* sum_int_list : int list -> int = <fun> *)
(* prod_int_list : int list -> int = <fun> *)

sum_int_list LI1;;  prod_int_list LI1;;

(* === 2.1. Vecteurs *)
(* === 2.1.1. do_vect *)
let V = [|2; 4; 50; 105; 23; 44; 55; 12; -17; 25; 26|];;
let S = let u = ref 0 in do_vect (fun x -> u := !u+x) V; !u;;
let P = let m = ref 1 in do_vect (fun x -> m := !m*x) V; !m;;

let sum_float_vect v =
  let u = ref 0.0 in for i = 0 to vect_length v -1 do u := !u +. v.(i) done;  !u
and prod_float_vect v =
  let u = ref 1.0 in for i = 0 to vect_length v -1 do u := !u *. v.(i) done; !u
;;
(* sum_float_vect : float vect -> float = <fun> *)
(* prod_float_vect : float vect -> float = <fun> *)

let VF1 = [| 1.2; 2.3; 5.4; -5.3; 2.7; 6.3 |];;
sum_float_vect VF1;;    prod_float_vect VF1;;
```

```
(* === 3. Pile LIFO === *)
(* === 3.1. Factorielle *)
let rec Factorielle = function | n when n < 0 -> failwith "Erreur"
  | 0 -> 1  | n -> n * Factorielle (n-1)
;;
(* Factorielle : int -> int = <fun> *)
Factorielle 5;;

let LIFO = ref [];;
let rec Factopile = function | n when n < 0 -> failwith "Erreur"
  | n -> LIFO := n::!LIFO;
         print_int_list !LIFO; print_newline ();
         begin match n  with
                | 0 -> begin match !LIFO with
                               | a::q -> LIFO := 1::q
                               | _    -> ()
                       end
                | n -> Factopile (n-1);
                       match !LIFO with
                         | a::b::q -> LIFO := n*a::q
                         | _       -> ()
         end;
         print_int_list !LIFO; print_newline ()
;;
(* Factopile : int -> unit = <fun> *)

let Facto2 n = LIFO := []; Factopile n; hd !LIFO
;;
(* Facto2 : int -> int = <fun> *)

Facto2 5;;  !LIFO;;

(* === 3.2. Nombres de Fibonacci *)
(* === 3.2.1. Récurrence double *)
let rec Fibonacci_déplorable = function | n when n < 0 -> failwith "Erreur"
  | (0|1) -> 1
  | n     -> Fibonacci_déplorable (n-1) + Fibonacci_déplorable (n-2)
;;
(* Fibonacci_déplorable : int -> int = <fun> *)

Fibonacci_déplorable 8;;  Fibonacci_déplorable 10;;

let rec Fibopile_dép = function | n when n < 0 -> failwith "Erreur"
  | n ->  LIFO := n::!LIFO;
          print_int_list !LIFO; print_newline ();
          begin match n with
                | (0 | 1)  -> begin match !LIFO with
                                      | a::q -> LIFO := 1::q
                                      | _ -> ()
                              end
                | n -> Fibopile_dép (n-2); Fibopile_dép (n-1);
                       match !LIFO with
                         | a::b::c::q -> LIFO := (a+b)::q
                         | _ -> ()
          end;
          print_int_list !LIFO; print_newline ()
;;
(* Fibopile_dép : int -> unit = <fun> *)

let Fibo_dép n = LIFO := []; Fibopile_dép n; hd !LIFO
;;
(* Fibo_dép : int -> int = <fun> *)

Fibo_dép 5;;  Fibo_dép 8;;
```

```
(* === 3.2.2. Rappels de Sup *)
(* === 3.2.2.1. Programmation itérative de la suite de Fibonacci *)
let fiboit n =
  let u = ref 1 and v = ref 1 in for i= 2 to n do let w = !v in v:= !u + !v ; u := w done; !v
;;
(* fiboit : int -> int = <fun> *)
fiboit 8;;

(* === 3.2.2.1. Programmation fonctionnelle des suites de Fibonacci *)
let fibof = fiboluxe 1 1
  where rec fiboluxe a b = function
   |0 -> a  |1 -> b
   |n -> fiboluxe b (a+b) (n-1)
;;
(* fibof : int -> int = <fun> *)
fibof 8;;

(* === 3.2.3. Récurrence vectorielle simple *)
let FibonacciV n = snd (FiboV n)
  where rec FiboV = function | n when n < 0 -> failwith "Erreur"
    | 0 -> 1,1
    | n -> let u, v = FiboV (n-1) in (u+v,u)
;;
(* FibonacciV : int -> int = <fun> *)

FibonacciV 8;;

let LIFO2 = ref [];;
let rec Fibopile_V = function | n when n < 0 -> failwith "Erreur"
  | n -> LIFO2 := (n,0)::!LIFO2;
         print_int_int_list !LIFO2; print_newline ();
         begin match n with
                | 0 -> begin match !LIFO2 with
                               | (a,b)::q -> LIFO2 := (1,1)::q
                               | _ -> ()
                       end
                | n -> Fibopile_V (n-1);
                       match !LIFO2 with
                          | (a,b)::c::q -> LIFO2 := (a+b,a)::q
                          | _ -> ()
         end;
         print_int_int_list !LIFO2; print_newline ()
;;
(* Fibopile_V : int -> unit = <fun> *)

let FiboV n = LIFO2 := []; Fibopile_V n; snd (hd !LIFO2)
;;
(* FiboV : int -> int = <fun> *)

FiboV 5;;  FiboV 8;;  !LIFO2;;

(* === 4. Pile FIFO (clavier) === *)
(* === 4.1. Information sur les flux *)
(* === 4.2. Exemple simple de flux *)
(* lecture d'un flux de char (de petite taille) produisant une chaîne de caractères *)
let rec string_from_char_stream m = function
   | [<'c ; (string_from_char_stream "") m1 >] -> m ^ (string_of_char c) ^ m1
   | [< >] -> m
;;
(* string_from_char_stream : string -> char stream -> string = <fun> *)

(* REMARQUE IMPORTANTE: ci-dessus, (string_from_char_stream "") m1
     signifie: string_from_char_stream ""  appliqué au reste du flux
                                           et dont le résultat est désigné par m1 *)
```

```
(* filtrage d'un flux de char (flux d'entrée) pour produire un flux de char (flux de sortie)
   (avec échange des Majuscules et minuscules) *)
let rec transforme = function
   | [< '('A'..'Z' as c) ; transforme r >] -> [< 'char_of_int ( (int_of_char c) + 32 ) ; r  >]
   | [< '('a'..'z' as c) ; transforme r >] -> [< 'char_of_int ( (int_of_char c) - 32 ) ; r  >]
   | [< 'c   ; transforme r >]  -> [< 'c ; r >]
   | [<>]                       -> [< >]      (* sinon, renvoie un flux vide *)
;;
(* transforme : char stream -> char stream = <fun> *)

let the_flux = stream_of_string
   "Il était Un PetiT naVire, qui NaviGait sur les FLOTS, HO hé hO hE MateLOT 123454321";;
string_from_char_stream "" (transforme the_flux);;

(* === 4.3. Simulation d'un buffer de clavier *)
let N = 17;;                      (* taille de la file FIFO (buffer de 16 char utiles) *)
let Buffer = make_string N '_';; (* file FIFO *)
let p = ref 0;;                   (* indice du caractère prêt à sortir, 0 <= p <= N *)
let q = ref 0;;                   (* indice où doit entrer le prochain caractère, 0 <= q <= N *)
let ssout = ref "";;              (* chaîne à remplir *)

let rec entree_buffer the_flux n =
  if n > 0
  then if (!q+1) mod N = !p then sound 1000 10
       else match the_flux with
              | [< 'c >] -> Buffer.[!q] <- c; q := (!q+1) mod N; entree_buffer the_flux (n-1)
              | [< >]    -> ()
and     sortie_buffer n =
  if n > 0
  then if !p = !q then (sound 1000 10)
       else begin ssout := !ssout ^ (string_of_char Buffer.[!p] ); p := (!p+1) mod N;
                  sortie_buffer (n-1)
            end;
;;
(* entree_buffer : char stream -> int -> unit = <fun> *)
(* sortie_buffer : int -> unit = <fun> *)

let Simulation_Clavier ssin =       (* similation simpliste *)
  clear_graph ();
  p := 0; q := 0;                                 (* buffer vide *)
  ssout := "";                                    (* chaîne de sortie *)
  let flux_d'entrée = stream_of_string ssin in    (* flux d'entrée *)
  let n = ref 1 in
  while !n <> 0
  do  n := read_int ();
      if !n > 0 then entree_buffer flux_d'entrée !n;
      if !n < 0 then sortie_buffer (- !n);
      set_color red; moveto 10 300; draw_string Buffer;             (* état buffer  *)
      let w = make_string N ' ' in w.[!p] <- 'p'; w.[!q] <- 'q';
         set_color green; moveto 10 280; draw_string w;             (* état indices *)
      set_color blue; moveto 10 260; draw_string !ssout;            (* état sortie  *)
  done
;;
(* Simulation_Clavier : string -> unit = <fun> *)

let ssin = "Ceci n'est qu'un vague baratin mais je ne dirai rien d'autre, même sous la torture.";;
Simulation_Clavier ssin;;

(************************************)
(* Fin de Piles-C.ml   (Corrigé)     *)
(************************************)
```

$< \mathscr{F}\mathscr{I}\mathscr{N} >$ **Piles** (énoncé + squelette + corrigé).