Listes et piles

Cours d'Informatique MSI

Denis MONASSE

[1]Classes Préparatoires aux Grandes Ecoles MP*
Lycée Louis le Grand, Paris

Février 2008

Nous oublierons volontairement dans cette section que les listes sont une des structures préexistantes de Caml pour mieux comprendre leur structure, leur puissance et leurs limitations.

Résumé

Soit E un ensemble. Définissons une suite d'ensembles $(E_n)_{n\in\mathbb{N}}$ par $E_0 = \{\emptyset\}$ et $E_{n+1} = E \times E_n$.

Définition

On appelle ensemble des listes d'éléments de E l'ensemble $\mathcal{L}(E) = \bigcup_{n\in\mathbb{N}} E_n$.

Autrement dit une liste d'éléments de E est soit rien, soit un élément de E_n du type

$$(a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))$$

où les a_i sont dans E.

Remarque

Ce qui est important à noter, c'est qu'on s'interdit d'utiliser l'associativité du produit cartésien et donc d'identifier E_n à $E^n = E \times \ldots \times E$. En effet cette identification a bien un sens pour chaque n fixé : il suffit d'identifier $(a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots))$ à $(a_1, \ldots, a_n)$. Par contre, elle est beaucoup moins significative sur la réunion des E_n. En particulier la projection $p_i : E^n \to E$, $(a_1, \ldots, a_n) \mapsto a_i$ n'est plus définie sur $\bigcup_{n\in\mathbb{N}} E^n$ tout entier, mais

seulement sur $\bigsqcup E^n$.

Définition

On appelle première projection, ou fonction Tête, l'application $\text{tete} : \mathcal{L}(E) \setminus \{\emptyset\} \to E$ *qui à toute liste non vide d'éléments de E associe son premier élément :*

$$\text{tete} : (a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots)) \mapsto a_1$$

On appelle deuxième projection, ou fonction Queue, l'application $\text{queue} : \mathcal{L}(E) \setminus \{\emptyset\} \to \mathcal{L}(E)$ *qui à toute liste non vide d'éléments de E associe son deuxième élément :*

$$\text{queue} : (a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots)) \mapsto (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots)$$

Résumé

On peut reprendre de manière inductive la définition des listes d'éléments de E. Elle se fonde sur les deux règles suivantes

- il existe une suite n'ayant aucun élément, que nous noterons *nil* ou ()
- si a est un élément de E et ℓ une liste, alors le couple (a, ℓ) est encore une liste

ce que l'on peut encore symboliser sous la forme

$$\textit{liste} = \textit{nil} \text{ ou } (\textit{element}, \textit{liste})$$

ou si l'on préfère par

$$\textit{liste} = \textit{nil} \ + \ \textit{element} \times \textit{liste}$$

Ceci correspond exactement à la définition d'un type construit en Caml, où l'on désigne par le *paramètre de type* `'a` le type des éléments de E.

Caml

```
#type 'a Liste = Nil
               |  Cons of 'a * ('a Liste);;
Type Liste defined.
```

La construction des fonctions Tête et Queue se fait alors par :

Caml

```
#let tete l =
               match l with
                   Nil -> failwith "liste vide"
                |  Cons (a,_) -> a;;
tete : 'a Liste -> 'a = <fun>
#let queue l =
                match l with
                    Nil -> failwith "liste vide"
                |   Cons (_,l) -> l;;
queue : 'a Liste -> 'a Liste = <fun>
```

Quant à l'ajout d'un élément en tête de liste, il peut se faire par la fonction `cons` (à ne pas confondre avec le constructeur `Cons`)

Caml

```
#let cons x l = Cons(x,l);;
cons : 'a -> 'a Liste -> 'a Liste = <fun>
```

Remarque

On constate que dans une liste construite par une application répétée de cette fonction `cons`, le seul élément auquel on peut accéder directement est le dernier élément qui a été ajouté. On dit encore que la liste est une structure de type LIFO : *last in first out* ou encore *dernier entré premier sorti*

En fait Caml contient un type liste déjà défini et que nous utiliserons par la suite. Il est défini exactement comme ci dessus à deux petites différences près qui portent uniquement sur les notations

- la liste vide est désignée par `[]` à la place de `Nil`
- la liste de tête `h` et de queue `r` est notée par `h::r` au lieu de `Cons (h,r)`.

Résumé

Nous allons montrer comment l'on peut définir une opération sur les listes. Toutes les démonstrations de ce paragraphe seront faites par induction structurelle en partant de la définition inductive d'une liste, suivant le schéma décrit par la proposition *informelle* suivante.

Proposition

Soit f une fonction sur les listes d'éléments de E. On suppose que

- *f fournit le résultat attendu sur la liste vide*
- *si f fournit le résultat attendu sur la liste ℓ, alors f fournit le résultat attendu sur la liste (a, ℓ) pour tout élément a de E*

Alors f fournit le résultat attendu sur toute liste d'éléments de E.

La définition d'une liste étant récursive, les procédures naturelles sur les listes sont elles mêmes récursives. Nous en donnerons néanmoins presque toujours une version itérative, nous réservant un paragraphe final pour confronter facilité d'écriture et efficacité.
Dans les premières opérations, nous privilégierons les fonctions `tete` et `queue` sur la reconnaissance de motifs, dans un souci de généralisation à d'autres langages que Caml. Mais au fur et à mesure, dans un souci de lisibilité accrue, nous privilégierons la reconnaissance de motifs ; nous conseillons au lecteur de faire l'effort de traduction dans les deux sens.

A l'aide des fonctions Tête et Queue, on peut parcourir toute une liste

$$(a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots))$$

et en afficher les divers éléments dans l'ordre (nous supposons connue une fonction `print` qui affiche les éléments de E de type `'a`)

Caml

```
#let rec print_liste l =
            match l with
                []  -> ()
             |  _   -> print (tete l); print_liste (queue l);;
print_liste : 'a Liste -> unit = <fun>
```

Proposition

La procédure `print_liste` affiche tous les éléments d'une liste dans l'ordre.

Démonstration

par induction structurelle. Si la liste est vide, la procédure ne fait rien ce qui est le résultat attendu. Si la liste est de type (a, l) et si la procédure affiche correctement l, alors la procédure commence par afficher a, puis affiche la liste l, ce qui est bien le résultat attendu.

La procédure précédente est le type même d'un sous-programme plus général qui applique une fonction f à tous les éléments d'une liste, en ignorant tous les résultats. Utilisant le caractère fonctionnel de Caml, nous pouvons en faire une procédure à deux paramètres, la fonction f et la liste l que nous nommerons `liste_iteration`. Appliquée à une fonction f et à une liste $(a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots))$ elle est équivalente à

$$\texttt{for } i = 1 \texttt{ to } n \texttt{ do } f(a_i) \texttt{ done}$$

Caml

```
#let rec liste_iteration f l =
          match l with
              [] -> ()
          |    _ -> f (tete l); liste_iteration f (queue l);;
liste_iteration : ('a -> 'b) -> 'a Liste -> unit = <fun>
```

Proposition

La procédure `liste_iteration` applique bien une fonction f à tous les éléments d'une liste, dans l'ordre, en ignorant tous les résultats obtenus.

Démonstration

par induction structurelle comme dans la démonstration précédente.

Remarque

Notre procédure d'affichage peut se définir par

```
let print_liste l = liste_iteration print l
```

ou encore plus simplement (en utilisant la programmation fonctionnelle) par

```
let print_liste = liste_iteration print
```

Dans le même ordre d'idée, étant donnée une fonction $f : E \to F$, il existe une unique application $m_f : \mathcal{L}(E) \to \mathcal{L}(F)$ qui envoie la liste vide sur la liste vide, et qui envoie la liste $(a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots))$ sur la liste $(f(a_1), (f(a_2), (\ldots, (f(a_{n-1}), (f(a_n), \emptyset))) \ldots))$ On peut la définir en Caml de manière fonctionnelle sous le nom `map_liste` par

Caml

```
let rec map_liste f l =
            match l with
                [] -> []
            |    _ -> (f (tete l)) :: (map_liste f (queue l));;
#map_liste : ('a -> 'b) -> 'a Liste -> 'b Liste = <fun>
```

ou avec une reconnaissance de motifs

Caml

```
#let rec map_liste f l =
             match l with
                [] -> []
             |  a :: lprime -> (f a) :: (map_liste f lprime);;
map_liste : ('a -> 'b) -> 'a Liste -> 'b Liste = <fun>
```

La longueur d'une liste se calcule de manière récursive

- la longueur de la liste vide est nulle
- la longueur d'une liste non vide est égale à la longueur de sa queue augmentée de 1

Caml

```
let rec longueur l =
        match l with
            []   -> 0
        |   h::r -> 1+longueur r;;
```

ou itérative : on introduit une référence sur ℓ, puis on remplace au fur et à mesure ℓ par sa queue en incrémentant un compteur i, jusqu'à obtenir la liste vide ; un invariant évident de la boucle est $i + \text{longueur}(\ell)$ après l'exécution du corps de la boucle ; on voit que la fonction est un peu plus compliqué

Caml

```
let longueur l =
       let x = ref l and i = ref 0 in
          while !x<>[] do
             i := !i+1; x := queue !x
          done;
          !i;;
```

On peut par un simple parcours rechercher le maximum d'une liste d'entiers positifs par

- le maximum de la liste vide est 0 (convention)
- le maximum d'une liste non vide est égal au plus grand des deux nombres suivants : sa tête, le maximum de sa queue.

D'où une procédure récursive

Caml

```
let rec maximum l =
        match l with
            [] -> 0
        |   h::r -> max h (maximum r);;
```

On peut également en donner une version itérative en parcourant toute la liste et en actualisant au fur et à mesure une référence sur le maximum trouvé (un invariant de la boucle est que $m = \max(a_1, \ldots, a_i)$ après la i-ième itération)

Caml

```
let maximum l =
       let x = ref l and m = ref 0 in
          while !x<>[] do
             if (tete !x)>!m then m := tete !x;
             x := queue !x
          done;
          !m;;
```

On peut par un simple parcours rechercher le n-ième élément ($n \geq 1$) d'une liste par

- le n-ième élément de la liste vide n'existe pas (on provoque une erreur `Not_found`)
- le premier élément d'une liste est sa Tête
- le n-ième élément d'une liste non vide est égal au $(n-1)$-ième élément de sa Queue

D'où une fonction récursive :

Caml

```
let rec nieme_elt l n =
            match l with
                []   -> raise Not_found
            |   h::r -> if n = 1 then h
                                 else nieme_elt r (n-1);;
```

Une procédure itérative peut être écrite suivant la même méthode que précédemment en remplaçant au fur et à mesure des itérations la liste par sa Queue, et en interrompant brutalement la boucle indexée si la liste devient vide ; un invariant de boucle est qu'après la i-ième exécution du corps de la boucle, la liste contient $a_{i+1}, \ldots, a_m$ où m est la longueur de la liste.

Caml

```
let nieme_elt l n =
        let x = ref l in
          for i = 1 to n-1 do
             if !x = [] then raise Not_found;
             x := queue !x
          done;
          tete !x;;
```

Etant donné une liste $(a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots))$, on cherche à construire la liste *miroir* (c'est à dire inversée) $(a_n, (a_{n-1}, (\ldots, (a_2, (a_1, \emptyset))) \ldots))$. Une première solution récursive se présente à nous si nous disposons d'une fonction de concaténation de deux listes comme nous allons la définir dans le paragraphe suivant. Elle se fonde sur le raisonnement inductif suivant

- l'image miroir de la liste vide est la liste vide
- l'image miroir d'une liste non vide *l* est obtenue en concaténant l'image miroir de la Queue de la liste *l* avec la liste dont le seul élément est la Tête de la liste *l*

Ce qui conduit à la fonction suivante :

Caml

```
let rec miroir_quad l = match l with
          []   -> []
       |  h::r -> concat (miroir_quad r) [h];;
```

La validité de la fonction est tout entière contenue dans le raisonnement inductif. Faisons par contre une étude de complexité. Si n désigne la longueur de la liste l, on voit que la fonction `miroir_quad` est appelée n fois, c'est à dire qu'il s'opère n concaténations d'une liste à $n-1$, puis $n-2$,..., puis 1 éléments avec une liste à 1 élément. Comme on le verra dans le paragraphe suivant, la concaténation d'une liste à p éléments a un temps de calcul proportionnel à p. On en déduit que le temps de calcul d'une image miroir par cet algorithme est proportionnel à

$(n-1)+(n-2)+\ldots+2+1=\frac{n(n-1)}{2}=O(n^2)$, d'où le nom d'algorithme quadratique. Ceci ne semble guère raisonnable, et effectivement ne l'est pas.

Pour obtenir un algorithme récursif qui travaille en un temps $O(n)$ nous allons généraliser notre problème d'image miroir en envisageant une fonction de concaténation à l'envers. On veut ici une fonction qui reçoit en paramètres deux listes ℓ_1 d'éléments $a_1, \ldots, a_m$ et ℓ_2 d'éléments $b_1, \ldots, b_n$ et qui retourne la liste concaténée $\ell_1 * \ell_2$ d'éléments $a_m, \ldots, a_1, b_1, \ldots, b_n$. La fonction récursive travaille naturellement sur la liste ℓ_1

- si ℓ_1 est vide, $\ell_1 * \ell_2$ est la liste ℓ_2
- sinon, on commence par ajouter en première position de ℓ_2 la Tête de ℓ_1 puis on concatène l'inverse de la Queue de ℓ_1

Ceci se traduit en Caml par

Caml

```
#let rec concatene_envers l1 l2 =
                match l1 with
                        []    -> l2
                    |    h::r -> concatene_envers r (h::l2);;
concatene_envers : 'a list -> 'a list -> 'a list = <fun>
```

Proposition

La fonction `concatene_envers` *renvoie le résultat de la concaténation de l'image miroir de* ℓ_1 *avec* ℓ_2*.*

Démonstration

par induction structurelle. C'est clair si ℓ_1 est la liste vide puisqu'elle renvoie ℓ_2. Supposons que la tête de ℓ_1 soit h et la queue r, et que la proposition soit vraie pour r. Soit ℓ_1 d'éléments $a_1, \ldots, a_m$ et ℓ_2 d'éléments $b_1, \ldots, b_n$. Alors $h :: \ell_2$ est la liste d'éléments $a_1, b_1, \ldots, b_n$, h est la liste d'éléments $a_2, \ldots, a_m$ et la concaténation de l'image miroir de h avec $h :: \ell_2$ est la liste d'éléments $a_m, \ldots, a_1, b_1, \ldots, b_n$, ce que l'on voulait.

Une fois cette fonction de concaténation à l'envers construite, il suffit simplement pour construire l'image miroir d'une liste ℓ de la concaténer à l'envers avec la liste vide. La fonction `concatene_envers` travaille manifestement dans un temps proportionnel à la longueur de la liste ℓ_1, et donc cette fonction `miroir` calcule l'image miroir d'une liste de longueur n en un temps $O(n)$ (linéaire).

Caml

```
let miroir l = concatene_envers l [];;
```

La même démarche conduit à une procédure itérative de concaténation à l'envers en utilisant deux références, l'une sur la liste ℓ_1 qui décroît, l'autre sur la liste ℓ_2 qui s'adjoint au fur et à mesure les éléments de têtes de ℓ_1, avec l'invariant de boucle : après la i-ième itération, x_1 est la liste d'éléments $a_{i+1}, \ldots, a_n$ et x_2 est la liste d'éléments $a_i, \ldots, a_1, b_1, \ldots, b_n$; la terminaison est garantie par la décroissance stricte de la longueur de la liste x_1.

Caml

```
let concatene_envers l1 l2 =
     let x2 = ref l2 and x1 = ref l1 in
         while !x1<>[] do
            x2 := (tete !x1)::!x2;
            x1 := queue !x1
         done;
       !x2;;
```

L'image miroir peut alors se construire comme précédemment par concaténation inverse avec la liste vide, ce qui conduit à :

Caml

```
let miroir l =
     let x = ref l and  accu = ref [] in
         while !x<>[] do
            accu := (tete !x)::!accu;
            x := queue !x
         done;
       !accu;;
```

On veut ici une fonction qui reçoit en paramètres deux listes l_1 d'éléments $a_1, \ldots, a_m$ et l_2 d'éléments $b_1, \ldots, b_n$ et qui retourne la liste concaténée $l_1 * l_2$ d'éléments $a_1, \ldots, a_m, b_1, \ldots, b_n$.
La fonction récursive travaille naturellement sur la liste l_1

- si l_1 est vide, $l_1 * l_2$ est la liste l_2
- sinon, on commence par concaténer la Queue de l_1 avec l_2 et ensuite on ajoute en première position la Tête de l_1

ce qui s'écrit :

Caml

```
let rec concat l1 l2 =
       match l1 with
            [] -> l2
         |   h::r -> h::(concat r l2);;
```

La validité de cette fonction se démontre trivialement par induction structurelle sur la liste l_1.
Une version itérative est *a priori* plus difficile à construire car de toute évidence l'appel récursif n'est pas terminal (il faut ensuite ajouter h en tête). On constate que l'on est un peu bloqué pour construire itérativement la liste d'éléments $a_1, \ldots, a_m, b_1, \ldots, b_n$, sachant qu'à la première étape on n'accède aisément qu'à a_1 et b_1 :

- on ne peut pas construire facilement $a_m, b_1, \ldots, b_n$ car on accède difficilement à a_m
- on ne peut pas construire facilement $a_1, \ldots, a_m, b_1$ car on ne sait pas ajouter facilement un élément en bout de liste

La solution consiste simplement à utiliser les fonctions itératives d'image miroir du paragraphe précédent et à concaténer à l'envers l'image miroir de l_1 avec l_2, ce qui nous donne

Caml

```
let concat l1 l2 = concat_rev (miroir l1) l2;;
```

Nous recherchons une fonction qui étant donnée une liste d'éléments $a_1, \ldots, a_m$, un entier $n \in [1, m+1]$ et un élément x de E, nous construise la liste $a_1, \ldots, a_{n-1}, x, a_n, \ldots, a_m$, où x a été inséré à la n-ième place L'induction structurelle se formule ainsi

- si $n = 1$, il suffit d'ajouter x en tête de la liste l
- si $n > 1$ et si la liste est vide, il y a erreur
- si $n > 1$ et la liste est non vide, le résultat est la liste dont la tête est la tête de l et donc la queue est obtenue en insérant l'élément x à la $n-1$-ième place de la queue de l.

Nous obtenons la fonction suivante (démonstration évidente par récurrence sur n)

Caml

```
#let rec insere x l n =
        if n = 1
          then x::l
          else
            match l with
              [] ->  raise Invalid_argument "insere"
            |  h::r -> h::(insere x r (n-1));;
insere : 'a -> 'a list -> int -> 'a list = <fun>
```

De nouveau ici la récursivité n'est pas terminale, et il n'est pas facile de donner une version itérative de cette fonction d'insertion. En effet le parcours obligatoire de la liste à partir du premier élément nous oblige a priori à ajouter les éléments successifs en queue de l'accumulateur qui va recevoir le résultat, puis à ajouter x en queue de cet accumulateur et enfin à ajouter les éléments successifs de la liste en queue de l'accumulateur, ce que nous ne pouvons pas faire facilement puisqu'on ne sait ajouter efficacement des éléments qu'en tête d'une liste.

Nous contournerons la difficulté en ajoutant tous ces éléments en tête de l'accumulateur. Le résultat sera bien entendu non pas la liste avec x inséré, mais son image miroir. Il suffit ensuite de prendre l'image miroir de ce résultat stocké dans l'accumulateur. Un invariant de la première boucle est : après la i-ième itération, `accu` contient $a_i, \ldots, a_1$ et $n \leq m$; un invariant de la deuxième boucle est : après la i-ième itération, `accu` contient $a_{n+i}, a_{n+i-1}, \ldots, a_{n+1}, x, a_n, \ldots, a_1$.

Caml

```
let insere x l n =
      let l1 = ref l and accu = ref [] in
        for i = 1 to n-1 do
            if !l1 = [] then raise Not_found;
            accu := (tete !l1)::!accu;
            l1 := queue !l1
        done;
        accu := x::!accu;
        while !l1<>[] do
            accu := (tete !l1)::!accu;
            l1 := queue !l1
        done;
        miroir !accu;;
```

Nous cherchons une fonction qui à une liste ℓ et un élément x associe la liste obtenue en supprimant *tous* les éléments de la liste égaux à x. L'induction structurelle est

- si la liste est vide on ne fait rien
- si la liste est non vide, on supprime x de la queue de ℓ et on met en tête la tête de ℓ si celle ci n'est pas égale à x

ce qui conduit à

Caml

```
let rec suppr l x =
          match l with
             []    -> []
          |  h::r -> if h = x then (suppr r x)
                                else h::(suppr r x);;
```

Une version itérative est aisément obtenue en cherchant l'image miroir de la liste ℓ et en *oubliant* d'ajouter les éléments égaux à x. Il suffit ensuite de renvoyer l'image miroir de l'accumulateur. Un invariant de la boucle est : après la i-ième exécution du corps de la boucle, `accu` contient ceux des élément $a_i, \ldots, a_1$ non égaux à x, dans cet ordre.

Caml

```
let suppr l x =
     let l1 = ref l and  accu = ref [] in
         while !l1<>[] do
           let tete = tete !l1 in
            if tete <> x then accu := tete::!accu;
            l1 := queue !l1
         done;
       !accu;;
```

Résumé

Nous avons construit dans le paragraphe précédent les opérations de base sur les listes avec chaque fois une version récursive et une version itérative. Nous cherchons donc à comparer ces deux modes de programmation sur les listes selon trois points de vue

- clarté et démonstration de la méthode
- efficacité en temps de calcul
- efficacité en occupation mémoire

En ce qui concerne la clarté et la concision, la balance penche de façon évidente en faveur des versions récursives ; ceci favorise également leur démonstration puisque la formulation même des fonctions suit l'induction structurelle qui en constitue par là-même la démonstration.

En ce qui concerne l'efficacité en temps de calcul, il est clair que tous les algorithmes récursifs que nous avons construits ont des temps de calcul en $O(n)$, si n désigne la longueur de la liste traitée. Il peut en sembler de même pour les algorithmes itératifs, qui tous sont constitués d'une boucle qui est effectuée au plus n fois. En fait une partie du travail est masquée par les affectations des références et en particulier les affectations `x:=queue !x` qui font décroître la liste sur laquelle on travaille : il est clair que le processeur doit commencer par procéder à une recopie de la liste avant de l'affecter et que le temps de cette recopie ne peut être que proportionnel à la longueur de la liste. Nos versions itératives ne sont peut-être pas si linéaires que cela, et un temps de calcul en $O(n^2)$ est fort probable.

En ce qui concerne l'efficacité en occupation mémoire, les versions récursives obligent la machine à une sauvegarde du contexte à chaque appel récursif. Par contre, dans les versions itératives, l'encombrement mémoire est uniquement celui des références et peut sembler moindre ; ce n'est vrai que dans la mesure où la machine parvient à récupérer efficacement les emplacements mémoires occupés par les listes qui ne sont plus pointées par les références après leurs affectations répétées, ce qui n'est peut être pas garanti (cela l'est en Caml). De toute façon, cette libération de mémoire prend du temps.
En conclusion, en ce qui concerne les listes, comme en général pour toute structure définie inductivement, les sous-programmes récursifs sont souvent les plus concis, les plus clairs, les moins susceptibles de contenir des erreurs, et les plus efficaces. Il faut donc les privilégier.

Résumé

Nous allons ici étudier les méthodes de tri de listes, en traitant le cas de listes d'entiers ; le lecteur généralisera facilement au cas du tri d'éléments d'un ensemble ordonné quelconque. Nous n'allons étudier ici que deux tris, particulièrement bien adaptés aux listes : le tri par insertion en $O(n^2)$ qui est simple et facile à programmer, le tri par fusion en $O(n \log_2 n)$ qui est très efficace en temps de calcul. Nous n'étudierons que des versions récursives de ces tris, les versions itératives ne présentant, comme nous l'avons vu dans le paragraphe précédent, aucun intérêt.

Recherchons tout d'abord comment faire l'insertion (à la bonne place) d'un élément x dans une liste déjà triée ℓ. L'induction structurelle est la suivante

- si la liste est vide, le résultat est la liste dont le seul élément est x
- si la liste est non vide et si x est inférieur ou égal à la tête de ℓ, le résultat est la liste dont la tête est x et la queue est ℓ
- si la liste est non vide et si x est supérieur à la tête de ℓ, le résultat est la liste dont la tête est la tête de ℓ et la queue est le résultat de l'insertion de x dans la queue de ℓ.

On obtient une fonction Caml (dont le temps de calcul est en $O(n)$ si n est la longueur de ℓ)

Caml

```
let rec insere_tri x l =
        match l with
              [] -> [x]
          |  h::r -> if x <= h then x::l
                                else h::(insere_tri x r);;
```

Le tri par insertion d'une liste est alors très simple

- si la liste est vide, on ne fait rien
- si la liste est non vide, on trie la queue de ℓ et on y insère la tête de ℓ

Ceci conduit à une fonction de tri dont le temps de calcul vérifie $T(n) = T(n-1) + O(n)$ (le $O(n)$ étant dû à l'insertion), soit $T(n) = O(n^2)$.

Caml

```
let rec tri_insertion l =
        match l with
            [] -> []
          | h::r -> insere_tri h (tri_insertion r);;
```

Note de programmation Le lecteur pourra généraliser à un ensemble ordonné quelconque muni d'une relation d'ordre total $\prec$ en passant en paramètre une fonction booléenne *f* qui renvoie `true` si $x \prec y$. On aura alors

Caml

```
#let rec insere_tri x l f =
          match l with
             [] -> [x]
           |  h::r -> if (f x h) then x::l
                                 else h::(insere_tri x r f);;
insere_tri : 'a -> 'a list -> ('a -> 'a -> bool) -> 'a list = <fun>
#let rec tri_insertion l f=
         match l with
             [] -> []
          |  h::r -> insere_tri h (tri_insertion r) f;;
```

Résumé

Nous renvoyons le lecteur au paragraphe sur les tris pour ce qui concerne le principe du tri par fusion. Rappelons simplement que nous avons besoin de deux fonctions : une fonction de partition qui partage une liste en deux listes de tailles similaires, une fonction de fusion qui fusionne deux listes déjà triées en une nouvelle liste triée. Les démonstrations de ce paragraphe par induction structurelle sont laissées au soin du lecteur : elles ne présentent aucune difficulté.

La fonction de partition opère de manière inductive

- si la liste est vide, le résultat est le couple formé de deux listes vides
- si la liste a un seul élément, le résultat est le couple formé de la liste ℓ et de la liste vide
- si la liste a au moins deux éléments, on partitionne la queue de la queue de ℓ (c'est à dire ℓ privé de ses deux premiers éléments) et on ajoute le premier élément de ℓ à la première liste et le deuxième élément de ℓ à la deuxième liste

On obtient la fonction Caml `partage` :

Caml

```
let rec partage l =
          match l with
               []  -> ([],[])
            |  [x] -> ([x],[])
            |   h1::h2::r -> match partage r with
                        (r1,r2) -> (h1::r1,h2::r2);;
```

La fusion de deux listes triées l_1 et l_2 opère également de manière inductive

- si l'une des listes est vide, le résultat est l'autre liste
- si la tête de ℓ_1 est inférieure ou égale à la tête de ℓ_2, on fusionne la queue de ℓ_1 avec ℓ_2 et on ajoute la tête de ℓ_1 en tête du résultat obtenu
- si la tête de ℓ_1 est supérieure à la tête de ℓ_2, on fusionne la queue de ℓ_2 avec ℓ_1 et on ajoute la tête de ℓ_2 en tête du résultat obtenu

On obtient la fonction Caml suivante qui procède par une double reconnaissance de motifs.

Caml

```
#let rec fusion l1 l2 =
         match (l1,l2) with
               ([],_) ->  l2
           |   (_,[]) ->  l1
           |   (h1::r1,h2::r2) ->
                        if h1 <= h2
                              then h1::(fusion r1 l2)
                              else h2::(fusion l1 r2);;
fusion : int list -> int list -> int list = <fun>
```

Le tri par fusion procède alors simplement par partage de la liste, tri récursif des deux listes obtenues et fusion de ces deux listes. Le partage et la fusion ayant visiblement des temps de calcul en $O(n)$, le temps de calcul $T(n)$ du tri par fusion vérifiera $T(n) = 2T(n/2) + O(n)$, ce qui conduit à $T(n) = O(n \log_2 n)$.

Caml

```
let rec tri_fusion = function
              []  -> []
          |  [x] ->  [x]
          |   l  ->  match partage l with
                       (l1,l2) ->  fusion (tri_fusion l1) (tri_fusion l2);;
```

Résumé

Les listes sont particulièrement adaptées à des structures mathématiques creuses, c'est à dire où beaucoup d'éléments ont des valeurs par défauts, la plupart du temps 0. C'est ainsi que si l'on travaille avec le polynôme $1 + 3X^2 + X^{100}$, c'est un épouvantable gâchis de le stocker dans un tableau de taille 101 dont 98 éléments vaudront 0. Nous allons étudier ces objets *creux* sur deux exemples : les polynômes et les matrices.

L'idée est de stocker un polynôme comme une liste ordonnée de monômes non nuls et de stocker un monôme $a_i X^i$ sous la forme du couple (i, a_i). Un polynôme creux à coefficients réels sera donc stocké comme une liste de couples formés d'un entier positif et d'un nombre réel.

- le polynôme nul est stocké comme la liste vide
- le polynôme $1 + 3X^2 + X^{100}$ est stocké comme $[(0, 1.), (2, 3.), (100, 1.)]$.

L'addition se fait par une variante très simple de la procédure utilisée pour la fusion de deux listes triées : au lieu de simplement fusionner les deux listes, on procède par addition des coefficients lorsque les degrés des monômes de tête sont les mêmes (et la somme des coefficients non nulle), par simple fusion dans le cas contraire, et ceci récursivement.

Caml

```
type  monome == int*float;;
type  polynome == monome  list;;

#let rec add_pol p1 p2=
         match (p1,p2) with
               ([],_) ->  p2
             | (_,[]) ->  p1
             | ((d1,a1)::r1,(d2,a2)::r2) ->
                    if d1<d2 then (d1,a1)::(add_pol r1 p2)
                    else if d2<d1 then (d2,a2)::(add_pol p1 r2)
                    else if a1+.a2 <>0. then (d1,a1+. a2)::(add_pol r1 r2)
                    else add_pol r1 r2;;
add_pol : (int * float) list -> (int * float) list -> (int * float) list =
```

Pour définir le produit de deux polynômes, on commence par définir le produit d'un polynôme par un monôme de la manière évidente :

$$aX^d \sum_{i=0}^{n} b_i X^i = \sum_{i=0}^{n} (ab_i) X^{d+i}$$

Caml

```
#let rec prod_mon_pol (d,a) p =
        if a = 0. then [] else
            match p with
               [] -> []
            |  (d1,a1)::r ->  (d+d1,a *. a1)::(prod_mon_pol (d,a) r);;
prod_mon_pol : int * float -> (int * float) list -> (int * float) list = <f
```

Puis on définit le produit de deux polynômes par

$$(\sum_{i=0}^{m} a_i X^i) P_2(X) = \sum_{i=0}^{m} (a_i X^i P_2(X))$$

soit en Caml :

Caml

```
#let rec prod_pol p1 p2 =
      match (p1,p2) with
            ([],_) -> [] (* par souci d'efficacité *)
        |   (_,[]) -> []
        |   (m1::r1 , _ ) -> add_pol (prod_mon_pol m1 p2) (prod_pol r1 p2);;
prod_pol : (int * float) list -> (int * float) list
                              -> (int * float) list = <fun>
```

Nous laissons le soin au lecteur d'écrire quelques procédures utiles sur les polynômes creux : multiplication par un scalaire, soustraction, puissance, dérivée, dérivée n-ième.

Le principe est le même que pour les polynômes creux, sauf que l'on remplacera le degré du monôme par le couple (i, j) qui indexe l'élément $a_{i,j}$, ces éléments étant ordonnés par exemple selon l'ordre lexicographique.

Caml

```
#let lex (i1,j1) (i2,j2) =  (* ordre lexico strict *)
       (i1<i2) or (i1 = i2 & j1<j2);;
lex : int * int -> int * int -> bool = <fun>
```

L'addition des matrices creuses est tout à fait similaire à l'addition des polynômes creux, à la seule différence près que l'on utilise l'ordre lexicographique :

Caml

```
#let rec add_mat m1 m2 =
          match (m1,m2) with
                ([],_) -> m2
              | (_,[]) -> m1
              | ((d1,a1)::r1,(d2,a2)::r2) ->
                      if (lex d1 d2) then (d1,a1)::(add_mat r1 m2)
                      else if (lex d2 d1) then (d2,a2)::(add_mat m1 r2)
                      else if a1+.a2 <>0. then (d1,a1+. a2)::(add_mat r1 r2)
                      else add_mat r1 r2;;
add_mat : ((int * int) * float) list -> ((int * int) * float) list ->
                     ((int * int) * float) list = <fun>
```

En ce qui concerne le produit des matrices creuses, on commence par définir le produit d'une matrice B par une matrice élémentaire du type $aE_{i,j}$ où $E_{i,j}$ est la matrice qui a des zéros partout, sauf à l'intersection de la i-ième ligne et de la j-ième colonne où figure un 1. On rappelle que $E_{i,j}E_{k,l} = \delta_j^k E_{i,l}$ avec $\delta_j^k = 1$ si $j = k$ et $\delta_j^k = 0$ sinon. On en déduit que si $B = bE_{k,l} + B_1$ alors

$$aE_{i,j}B = ab\delta_j^k E_{i,l} + aE_{i,j}B_1$$

Le lemme suivant nous garantit que l'ordre lexicographique sur les indices correspondant à des termes non nuls est conservé.

Lemme

L'ordre lexicographique sur les matrices $E_{i,j}$ est compatible avec la multiplication à gauche et à droite.

Démonstration

Bien entendu notre affirmation est un peu abusive, à moins de prendre par convention que les deux affirmations $A \prec 0$ et $0 \prec A$ sont vraies. Ceci signifie simplement que si $(i,j) \prec (i',j')$ et si les produits sont tous deux non nuls, $E_{i,j}E_{k,l} \prec E_{i',j'}E_{k,l}$ et de même $E_{k,l}E_{i,j} \prec E_{k,l}E_{i',j'}$.
Examinons le premier cas. Le fait que les deux produits soient non nuls nécessite $j = j' = k$. Comme $(i,j) \prec (i',j')$, c'est donc que $i < i'$ et donc $(i,l) \prec (i',l)$ soit $E_{i,j}E_{k,l} = E_{i,l} \prec E_{i',l} = E_{i',j'}E_{k,l}$.
Dans le second cas, le fait que les deux produits soient non nuls nécessite $i = i' = l$. Comme $(i,j) \prec (i',j')$, c'est donc que $j < j'$ et donc $(k,j) \prec (k,j')$ soit $E_{k,l}E_{i,j} = E_{k,j} \prec E_{k,j'} = E_{k,l}E_{i',j'}$.

Ceci conduit à la fonction Caml

Caml

```
#let rec prod_mat_elem ((i,j),a) B =
       match B with
             [] -> []
        |    ((k,l),b)::B1 ->
                 if j = k then
                    let c = a *. b in
                       if c<> 0.0 then
                           ((i,l),c)::(prod_mat_elem ((i,j),a) B1)
                       else prod_mat_elem ((i,j),a) B1
                  else prod_mat_elem ((i,j),a) B1;;

prod_mat_elem : ('a * 'b) * float -> (('b * 'c) * float) list ->
       (('a * 'c) * float) list = <fun>
```

On peut alors écrire une fonction de produit de matrices creuses comme pour les polynômes creux. Le lemme précédent nous garantit encore une fois que l'ordre lexicographique sur les indices des termes non nuls est conservé.

Caml

```
#let rec prod_mat m1 m2 =
      if m2 = [] then []  (* par souci d'efficacité *)
      else
         match m1 with
            [] ->  []
          |  a1::r1  ->  add_mat (prod_mat_elem a1 m2) (prod_mat r1 m2);;
prod_mat : (int * float) list -> (int * float) list -> (int * float) list =
```

Résumé

Il arrive souvent qu'en algorithmique on ait besoin de sauvegarder des valeurs. On utilise alors généralement une pile. Qu'est ce qu'une pile ? C'est tout simplement un objet informatique modifiable qui dispose de deux méthodes (sous-programmes), une procédure permettant de ranger une valeur dans cette pile (en général notée `push`), une fonction retournant une valeur rangée dans cette pile et la supprimant au passage de la pile (en général notée `pop`).
On voit donc qu'une pile est particulièrement adaptée à ranger provisoirement des valeurs dont on n'aura besoin qu'une seule fois par la suite.
Bien entendu, les opérations de rangement dans la pile (on dit plutôt *empilage*) et les opérations de récupération/suppression dans la pile (on dit plutôt *dépilage*) ont tendance à être très imbriquées. On peut avoir deux empilages, suivis d'un dépilage, suivi de trois empilages, suivis de quatre dépilages, etc.
On distingue deux grandes méthodes de dépilages.

Le premier type de pile est similaire à une pile d'assiettes rangées dans un placard. Lorsqu'on y empile une assiette, elle se trouve sur le dessus. Lorsque l'on dépile une assiette, on la prend sur le dessus. Autrement dit, l'assiette que l'on dépile est toujours la dernière qui a été empilée. On parle alors de pile LIFO pour *Last In First Out*, soit *dernier entré premier sorti*. Ces piles portent encore en anglais le nom de *stack* ; ce sont celles que l'on rencontre le plus fréquemment en informatique, en particulier dans tous les problèmes liés à la récursivité : en effet lorsqu'on revient d'un sous programme appelé, c'est toujours dans le dernier sous programme appelant. C'est ce type de piles que nous allons considérer.

Le deuxième type est similaire à certains distributeurs de gobelets, où l'on introduit les gobelets par le haut et où on les récupère par le bas. Dans ce cas, le gobelet que l'on récupère est toujours le premier qui a été empilé. On parle alors de pile FIFO pour *First In First Out*, soit *premier entré premier sorti*. Ces piles portent encore le nom de *queue* ou *files d'attente* (analogue à une queue pour aller au cinéma ou pour franchir un péage d'autoroute). Elles sont moins fondamentales pour nous et interviennent principalement dans les gestions de files d'attente et tous les phénomènes où la chronologie a de l'importance (gestion du clavier, des clics de la souris, etc.).

Résumé

Nous connaissons déjà des structures analogues aux piles LIFO, ce sont les listes. L'élément d'une liste auquel on peut accéder directement est le dernier élément qui y a été entré. Tout ce qu'il nous faut c'est une pile qui est modifiable tout au long du travail et qui dispose de deux sous-programmes `push` et `pop`. Le premier sous-programme est une procédure qui ajoute un élément x au sommet de la pile. Le second est une fonction qui renvoie l'élément du sommet de la pile tout en le supprimant de la pile.
Une référence sur une liste est tout à fait adaptée à cet effet et on peut simuler une pile en Caml avec les deux sous programmes. Bien entendu, il est prudent de prévoir un message d'erreur pour le cas où l'on essaye de dépiler dans une pile vide ce qui arrive plus fréquemment qu'on ne le voudrait.

Caml

```
#type 'a pile == ('a list) ref;;
Type pile defined.
#let push s x = s := x::!s;;
push : 'a list ref -> 'a -> unit = <fun>
#let pop s =
       match !s with
           []-> failwith "pile vide"
         |  h::r -> s := r; h;;
pop : 'a list ref -> 'a = <fun>
```

Note de programmation En fait Caml dispose déjà dans sa bibliothèque du type `stack` qui possède deux méthodes `push` et `pop`, ainsi qu'une troisième méthode `new`, fonction qui retourne une pile vide. Si l'on regarde l'implémentation de ce type, il est un tout petit peu plus compliqué qu'on ne pourrait le croire puisqu'on y trouve

Caml

```
type 'a t = { mutable c : 'a list };;

let new () = { c = [] };;

let push x s = s.c <- x :: s.c;;

let pop s =
    match s.c with
         hd::tl -> s.c <- tl; hd
     |   []     -> raise Empty;;
```

Note de programmation Pourquoi un type enregistrement modifiable comportant uniquement une liste plutôt qu'une référence sur une liste ? Tout simplement pour permettre à Caml de reconnaître le type d'une pile. En utilisant une référence, nous avons eu beau définir le type `'a pile` comme synonyme de `('a list) ref`, nous voyons que lorsque Caml nous affiche le type de la fonction `push` il nous indique `'a list ref -> 'a -> unit` et non pas `'a pile -> 'a -> unit`. Le synonyme est pour vous, mais vous ne pouvez pas forcer Caml à l'utiliser. Si par contre vous posez

```
type 'a pile = { mutable pile_contenu : 'a list }
```

le type `'a pile` sera bien défini à l'intérieur de Caml, Caml pourra le reconnaître à coup sûr grâce à l'étiquette `pile_contenu` et on obtiendra :

Caml

```
type 'a pile = { mutable pile_contenu : 'a list };;
Type pile defined.

let push x s = s.pile_contenu <- x :: s.pile_contenu;;
#push : 'a pile -> 'a -> unit = <fun>
```

C'est une méthode élégante et à retenir pour définir de nouveaux types construits.

Remarque

Une autre méthode pour construire des piles est d'utiliser un tableau et un compteur indiquant le sommet de la pile. La procédure `push` est alors simplement `i:=!i+1; tab.(!i)<-x` (augmenter le compteur et stocker l'élément à la nouvelle place) et la fonction `pop` est alors `i:=!i+1; tab.(!i+1)` (diminuer le compteur et renvoyer l'élément suivant). L'inconvénient de cette technique est bien entendu d'obliger dès le départ à estimer la taille nécessaire sous peine de débordement de la pile et à réserver toute cette place, même si on n'en a pas vraiment besoin tout au long du déroulement du programme. On peut également réallouer le vecteur lors d'un débordement.

Résumé

Les piles FIFO à construire car on ne dispose pas directement de structure dynamique de type FIFO. Nous allons décrire un moyen utilisé par exemple dans les *buffers* ou *tampons* de claviers (emplacements mémoires destinés à recevoir les codes des touches sur lesquelles vous avez appuyé, en attendant que le logiciel puisse les traiter).
Si l'on sait que l'on n'aura pas plus de n objets à stocker on utilise un tableau $a_0, \ldots, a_{n-1}$ et deux indices indiquant le début et la fin de la pile. On stocke les éléments de manière circulaire dans la pile, c'est à dire que lorsque l'on déborde vers la droite, on revient à gauche de la pile ; autrement dit, on considère ces indices modulo n.

On obtient alors les déclarations suivantes :

Caml

```
type 'a pile_FIFO = { pf_contenu:'a vect; pf_taille:int;
                      mutable pf_debut:int; mutable pf_fin:int};;

let new_FIFO n =  {pf_contenu = make_vect (n+1) 0;
                               (* remplacer 0 par un élément de type 'a *)
                   pf_taille = (n+1); pf_debut = 0; pf_fin = 0};;
```

Dans cette déclaration, `pf_contenu` désigne le tableau de stockage de la pile, `pf_taille` la taille de ce tableau stockée une fois pour toute pour éviter de la recalculer à chaque empilage ou dépilage, `pf_debut` pointe sur le premier élément entré (donc celui qui sera le premier à sortir) et `pf_fin` l'indice de la case qui suit le dernier entré. Nous avons utilisé une petite astuce qui consiste à ajouter 1 à la taille souhaitée et à travailler modulo $n+1$. Ceci permet de distinguer aisément une pile vide (quand `pf_debut=pf_fin`) d'une pile pleine (quand `pf_debut` désigne la case d'après la case d'indice `pf_fin`), mais en sacrifiant une place dans la pile.

Caml

```
#let push_FIFO s x =
        let nouvelle_fin = (s.pf_fin +1) mod s.pf_taille in
            if nouvelle_fin = s.pf_debut then
                                failwith "debordement de pile";
            s.pf_contenu.(s.pf_fin) <- x;
            s.pf_fin <- nouvelle_fin;;
push_FIFO : 'a pile_FIFO -> 'a -> unit = <fun>

#let pop_FIFO s =
        let nouveau_deb = s.pf_debut +1 mod s.pf_taille
        and x = s.pf_contenu.(s.pf_debut) in
            if s.pf_fin = s.pf_debut then failwith "pile vide";
            s.pf_debut <- nouveau_deb ;
            x;;
pop_FIFO : 'a pile_FIFO -> 'a = <fun>
```

Résumé

Un moyen classique de gérer des files d'attente en mode FIFO (First In First Out, premier entré, premier sorti) ou LIFO (Last In First Out, dernier entré premier sorti) ou en mode mixte, est de chainer des éléments. Ceci se fait tout naturellement en Caml. C'est ainsi que le type des listes en Caml pourrait encore s'écrire

Caml

```
type 'a cell = {suivante : 'a liste; contenu: 'a}
and 'a liste = Vide | Car of 'a cell;;

let car = function
        |  Vide -> failwith "liste vide"
        |  Car c -> c.contenu;;

let cdr = function
        |  Vide -> failwith "liste vide"
        |  Car c -> c.suivante;;

let cons x liste =Car {suivante=liste; contenu=x};;
```

Un code similaire permet alors de gérer des piles LIFO avec une structure de donnée chainée

Caml

```
type 'a pile = {lapile: 'a liste};;

let nouvellePile () = {lapile=Vide};;
let pop = function
        |  {lapile=Vide} -> failwith "pile vide"
        |  {lapile=Car c} as  p -> p.lapile <- c.suivante; c.contenu;;
let push p x = let c = {suivante=p.lapile; contenu=x} in
                       p.lapile <- Car c;;
```

Un code similaire va nous permettre de construire des files d'attente LIFO. Il nous suffit de garder un pointeur sur le dernier élément de la liste chainée ; on empilera les éléments en fin de liste, on les dépilera en début de liste ; ceci nécessite de modifier un peu le typage des cellules pour rendre le champ `suivante` modifiable

Caml

```
type 'a cell = {mutable suivante : 'a liste; contenu: 'a}
and 'a liste = Vide | Car of 'a cell;;
type 'a pile_fifo = {mutable premier: 'a liste;
                             mutable dernier: 'a liste};;

let nouvelle_pile_fifo () = {premier=Vide; dernier = Vide};;
let pop_fifo = function
       |  {premier=Vide} -> failwith "pile vide"
       |  {premier=Car c} as p ->
                          p.premier <- c.suivante;
                          if p.premier = Vide then p.dernier <- Vide;
                           c.contenu;;
```

Caml

```
let push_fifo p x =
       match p with
               |  {dernier=Vide} ->  (* aucun élément *)
                          p.premier <- Car {suivante=Vide; contenu=x};
                          p.dernier <- p.premier
               |  {dernier = Car c1} when p.premier=p.dernier->
                         (* un seul element *)
                          let c = {suivante=Vide; contenu=x} in
                          c1.suivante <- Car c; p.dernier <- Car c
               |  {dernier=Car c1}  ->  (* plusieurs elements *)
                          let c = {suivante=Vide; contenu=x} in
                          c1.suivante <- Car c; p.dernier <- Car c ;;
```

Résumé

On peut également utiliser des listes doublement chaînées, qui nous permettront d'empiler ou de dépiler en tête ou en queue de la liste

Caml

```
type 'a cell2 = {mutable next : 'a liste2;
                 mutable prev: 'a liste2; contenu: 'a}
and 'a liste2 = Vide | Car of 'a cell2;;

type 'a pile2 = {mutable first: 'a liste2;
                 mutable last: 'a liste2};;
```

Caml

```
let new_pile2 () = {first=Vide; last=Vide};;
let new_cell x = {next = Vide; prev = Vide; contenu = x};;
let push_first p x = let c = new_cell x in
        match p.first with
             | Vide -> p.first <- Car c; p.last<-p.first
             | Car t -> c.next <- Car t; t.prev <- Car c;
                        p.first <- Car c;;
let pop_first p = match p.first with
             | Vide -> failwith "pile vide"
             | Car t -> begin match t.next with
                        | Vide -> p.last <- Vide;
                                  p.first <- Vide
                        | Car c -> c. prev <- Vide;
                                  p.first <- Car c
                   end;
                   t.contenu;;
```

Caml

```
let push_last p x =  let c = new_cell x in
        match p.last with
              | Vide -> p.last <- Car c; p.first <- Car c
              | Car t -> c.prev <- Car t; t.next <- Car c;
                           p.last <- Car c;;
let pop_last p = match p.last with
              | Vide -> failwith "pile vide"
              | Car t -> begin match t.prev with
                           | Vide -> p.first <- Vide; p.last <- Vide
                           | Car c -> c.next <- Vide; p.last <- Car c
                     end;
                t.contenu;;
```

La notation habituelle des expressions algébriques, sous forme dite *infixe*, où les opérateurs (addition, multiplication, soustraction, division) figurent entre leurs deux opérandes, souffre *a priori* d'une grande ambiguïté si l'on n'introduit pas de priorités entres les opérateurs. C'est ainsi que la notation $2 + 3 * 4$ peut aussi bien désigner $2 + (3 * 4) = 14$ que $(2 + 3) * 4 = 20$. Des parenthèses ou des règles de priorité sont donc nécessaires pour lever cette ambiguïté. Nous allons étudier ici une autre notation, appelée notation algébrique postfixée ou encore notation polonaise inversée qui ne souffre pas de ces inconvénients. Cette notation est utilisée par certains langages de programmation comme Forth ou certaines calculatrices.

Résumé

Soit E un ensemble (les *nombres*), $\mathcal{O}$ un ensemble d'applications de $E \times E$ dans E (les *opérateurs*) et $\mathcal{F}$ un ensemble d'applications de E dans E (les *fonctions*). On considère l'ensemble $\mathcal{E}$ des suites $a_1 \ \ldots \ a_n$ où les a_i sont dans $E \cup \mathcal{O} \cup \mathcal{F}$, qu'on appellera l'ensemble des expressions algébriques.

Exemple

On pourra prendre $E = \mathbb{R}$, $\mathcal{O} = \{+, -, *, /\}$ et $\mathcal{F} = \{\sin, \cos, \exp, \log\}$. Les suites ci-dessous sont des expressions algébriques

- 2 + 3 * 4
- 2 3 + sin 4 * 5 6 + -
- + 3 5 * sin cos

Définition

On appelle ensemble des expressions algébriques postfixées le sous ensemble $\mathcal{EP}$ de $\mathcal{E}$ construit inductivement par les règles suivantes

- *pour tout élément $a \in E$, la suite à un élément a est une expression algébrique postfixée*
- *si $A = a_1 \ \ldots \ a_m$ et $B = b_1 \ \ldots \ b_n$ sont deux expressions algébriques postfixées et c un opérateur, alors $A\ B\ c = a_1 \ \ldots \ a_m\ b_1 \ \ldots \ b_n\ c$ est encore une expression algébrique postfixée*
- *si $A = a_1 \ \ldots \ a_m$ est une expression algébrique postfixée et f une fonction, alors $A\ f = a_1 \ \ldots \ a_m\ f$ est encore une expression algébrique postfixée.*

Proposition

Toute expression algébrique postfixée commence par un nombre et toute expression algébrique postfixée de longueur strictement supérieure à 1 se termine par un opérateur ou par une fonction.

Démonstration

par induction structurelle en appliquant les règles de construction précédentes. Les expressions algébriques de longueur 1 sont les suites n'ayant qu'un seul élément, donc leur premier élément est un nombre ; de plus si A commence par un nombre, aussi bien $A\ B\ c$ que $A\ f$ commencent par un nombre. Ceci montre que toute expression algébrique postfixée commence par un nombre. D'autre part, le seul moyen d'obtenir une expression algébrique postfixée de longueur strictement supérieure à 1 est d'appliquer une des deux dernières règles de construction, et donc l'expression algébrique postfixée se termine forcément par un opérateur ou une fonction.

Exemple

On reprend les expressions de l'exemple précédent

- 2 + 3 * 4 n'est pas une expression algébrique postfixée, car elle est de longueur plus grande que 1 et se termine par un nombre
- 2 3 + sin 4 * 5 6 + - est une expression algébrique postfixée comme le montre le parenthésage suivant qui donne les différentes étapes de la construction ((((2 3 +) sin) 4 *) (5 6 +) -)
- + 3 5 * sin cos n'est pas une expression algébrique postfixée car elle commence par un opérateur
- 2 + n'est pas une expression algébrique postfixée bien qu'elle commence par un nombre et se termine par un opérateur

Ce dernier exemple montre qu'il n'est pas facile de caractériser immédiatement les expressions algébriques postfixées. Nous allons travailler en deux temps et introduire les deux définitions suivantes

Définition

Soit $p : E \cup \mathcal{O} \cup \mathcal{F} \to \mathbb{N}$ *définie par* $p(x) = 1$ *si* $x \in E$, $p(c) = -1$ *si* $c \in \mathcal{C}$ *et* $p(f) = 0$ *si* $f \in \mathcal{F}$. *Si* $A = a_1 \ \ldots \ a_m$ *est une expression algébrique, on appelle poids de A le nombre entier noté* $p(A)$ *défini par* $p(A) = \sum_{i=1}^{m} p(a_i)$.

Définition

Soit $A = a_1 \ \ldots \ a_m$ *une expression algébrique. On appelle préfixes stricts de A les expressions algébriques* $A_i = a_1 \ \ldots \ a_i$ *pour* $1 \leq i \leq m - 1$.

Proposition

Soit A une expression algébrique postfixée. Alors A est de poids égal à 1 et tout préfixe de A a un poids supérieur ou égal à 1.

Démonstration

par récurrence sur la longueur de A. C'est clair si A est de longueur 1, puisqu'alors $A = a$ avec $a \in E$ et que $p(A) = p(a) = 1$; de plus A n'a dans ce cas aucun préfixe strict.
Supposons que A est construit par application de la règle 2 ; on a alors $A = B\ C\ c$ où B et C sont des expressions algébriques postfixées (de longueur strictement inférieure) et c un opérateur ; l'hypothèse de récurrence donne alors $p(A) = p(B) + p(C) + p(c) = 1 + 1 - 1 = 1$. Soit A' un préfixe strict de A. Alors soit A' est un préfixe de B ou même B, auquel cas par l'hypothèse de récurrence $p(A') \geq 1$, soit $A' = B\ C'$ où C' est ou bien un préfixe strict de C ou bien C ; alors par l'hypothèse de récurrence $p(C') \geq 1$, soit $p(A') = p(B) + p(C') = 1 + p(C') \geq 2$.

Démonstration (suite)

Supposons A construit par application de la règle 3 ; on a alors $A = B\ f$ où B est une expression algébrique postfixée (de longueur strictement inférieure) et f une fonction. Alors $p(A) = p(B) + p(f) = 1 + 0 = 1$ par l'hypothèse de récurrence. De plus si A' est un préfixe strict de A, c'est soit B de poids 1, soit un préfixe de B qui par hypothèse de récurrence est de poids supérieur ou égal à 1.

Théorème

Soit A une expression algébrique postfixée de longueur strictement supérieure à 1.

- *Si $A = B\ C\ c$ où B et C sont des expressions algébriques postfixées et c un opérateur, alors B est le plus long préfixe strict de A de poids 1*
- *Si $A = B\ f$ où B est une expression algébrique postfixée et f une fonction, alors B est le plus long préfixe strict de A de poids 1*

Démonstration

la deuxième assertion est triviale puisque B est le plus long préfixe strict de A et qu'il est de poids 1 d'après la proposition précédente. Montrons maintenant la première. On sait que B est un préfixe strict de A et qu'il est de poids 1 (d'après la proposition précédente). Soit A' un préfixe strict de A, strictement plus long que B. Alors $A' = B\ C'$ où C' est un préfixe strict de C ou bien C lui même ; on sait alors que $p(C') \geq 1$ et donc
$p(A') = p(B) + p(C') = 1 + p(C') \geq 2$. Donc B est bien le plus long préfixe strict de poids 1.

La proposition précédente montre que, contrairement à la notation infixe, la notation postfixée ne présente aucune ambiguïté, ce que nous allons traduire sous forme de corollaire.

Corollaire

L'écriture d'une expression algébrique postfixée de longueur strictement supérieure à 1 sous l'une des deux formes

- *$A = B\ C\ c$ où B et C sont des expressions algébriques postfixées et c un opérateur*
- *$A = B\ f$ où B est une expression algébrique postfixée et f une fonction*

est unique.

Démonstration

en effet dans les deux cas, B est parfaitement caractérisée par le fait d'être le plus long préfixe strict de poids 1 et dans le premier cas, C n'est autre que A privé de B et du dernier élément de A.

Nous allons revenir à la caractérisation des expressions algébriques postfixées.

Théorème

Une expression algébrique A est une expression algébrique postfixée si et seulement si elle est de poids 1 et tout préfixe strict a un poids supérieur ou égal à 1.

Démonstration

nous avons déjà vu que la condition est nécessaire. Montrons qu'elle est suffisante. Soit $A = a_1 \ \ldots \ a_m$ une expression algébrique de poids 1 telle que tout préfixe strict soit de poids supérieur ou égal à 1. Nous allons montrer par récurrence sur m que A est une expression algébrique postfixée. Si $m = 1$, on a $1 = p(A) = p(a_1)$ donc a_1 est un nombre et A est bien une expression algébrique postfixée. Si $m > 1$, remarquons que si $A' = a_1 \ \ldots \ a_{m-1}$, alors A' est un préfixe strict de A donc $p(A') \geq 1$. Comme $p(A) = p(A') + p(a_m) = 1$, on a nécessairement $p(a_m) \leq 0$, donc a_m est soit une fonction, soit un opérateur. Si a_m est une fonction f, on a $p(A') = 1$; comme tout préfixe strict de A' est un préfixe strict de A, il a un poids supérieur ou égal à 1, donc par l'hypothèse de récurrence, A' est une expression algébrique postfixée, soit $A = A' \ f$ est aussi un expression algébrique postfixée.

Démonstration (suite)

Si a_m est un opérateur c, on a $p(A') = 2$. A admet au moins un préfixe strict de poids 1, à savoir a_1. Soit donc B un préfixe strict de A de poids 1 de longueur maximale ; comme tout préfixe strict de B est un préfixe strict de A, par l'hypothèse de récurrence B est une expression algébrique postfixée. Comme de plus $p(A') = 2$, on a $B \neq A'$ soit $A' = B\ C$. On a $2 = p(A') = p(B) + p(C) = 1 + p(C)$, donc $p(C) = 1$. Soit C' un préfixe strict de C. Alors $B\ C'$ est un préfixe strict de A strictement plus long que B, donc $p(B\ C') \geq 1$ et $p(B\ C') \neq 1$, soit $p(B\ C') \geq 2$. Mais $p(C') = p(B\ C') - p(B) = p(B\ C') - 1 \geq 1$; par l'hypothèse de récurrence, C est une expression algébrique postfixée. Mais alors $A = B\ C\ c$ est encore une expression algébrique postfixée.

Résumé

Pour le moment, nos expressions algébriques postfixées sont uniquement des objets formels, des suites de symboles, sans signification. Nous avons seulement appris à les reconnaître et à les manipuler, c'est-à-dire leur syntaxe. Nous allons maintenant donner un sens à ces objets formels, c'est à dire leur définir une sémantique, à travers le théorème suivant.

Théorème

Il existe une unique application Ev *de l'ensemble des expressions algébriques postfixées dans l'ensemble* E *définie inductivement par*

- $\mathrm{Ev}(A) = a$ *si* $A = a$ *avec* $a \in E$
- $\mathrm{Ev}(A) = c\big(\mathrm{Ev}(B), \mathrm{Ev}(C)\big)$ *si* $A = B\ C\ c$ *où* B *et* C *sont des expressions algébriques postfixées et* c *un opérateur*
- $\mathrm{Ev}(A) = f\big(\mathrm{Ev}(B)\big)$ *si* $A = B\ f$ *où* B *est une expression algébrique postfixée et* f *une fonction*

Démonstration

ceci découle immédiatement de l'unicité de l'écriture d'une expression algébrique postfixée sous l'une des trois formes, et du chapitre sur la récursion et les définitions inductives de fonctions.

Définition

L'application Ev *est appelée l'évaluation. L'élément de* E, $\mathrm{Ev}(A)$ *est appelé l'évaluation de l'expression algébrique postfixée* A.

Résumé

La méthode précédente pour l'évaluation d'une expression algébrique postfixée se heurte à la difficulté de déterminer B et C pour le cas où $A = B\ C\ c$ avec c un opérateur. La définition de B comme le plus long préfixe de poids 1 se prête mal à un algorithme. Nous allons donc donner une méthode plus pratique utilisant une pile.
Nous allons définir une suite d'applications de l'ensemble des expressions algébriques postfixées dans l'ensemble des suites d'éléments de E par récurrence de la manière suivante. Soit $A = a_1\ \ldots\ a_m$ une expression algébrique postfixée.

- $f_1(A) = a_1$
- Posons $f_i(A) = b_1\ \ldots\ b_k$; alors $f_{i+1}(A)$ est défini par
  - $f_{i+1}(A) = f_i(A) = b_1\ \ldots\ b_k$ si $i \geq m$
  - $f_{i+1}(A) = f_i(A)\ a_{i+1} = b_1\ \ldots\ b_k\ a_{i+1}$ si $a_{i+1} \in E$
  - $f_{i+1}(A) = b_1\ \ldots\ b_{k-2}\ c(b_{k-1}, b_{k-2})$ si $a_{i+1} = c$ est un opérateur (défini si $k \geq 2$)
  - $f_{i+1}(A) = b_1\ \ldots\ b_{k-1}\ f(b_k)$ si $a_{i+1} = f$ est une fonction

Lemme

Pour tout $i \leq m$, $f_i(A)$ est défini et la longueur de $f_i(A)$ est égale au poids de $a_1 \ \ldots \ a_i$.

Démonstration

nous allons le montrer par récurrence sur i. C'est clair pour $i = 1$. Supposons que ce soit vrai pour $i \leq m - 1$ et montrons le pour $i + 1$.
Si $a_{i+1} \in E$, alors il est clair que $f_{i+1}(A)$ est définie et sa longueur est égale à la longueur de $f_i(A)$ plus 1, soit, par l'hypothèse de récurrence, au poids de $a_1 \ \ldots \ a_i$ augmenté de $1 = p(a_{i+1})$, donc au poids de $a_1 \ \ldots \ a_{i+1}$.
Si a_{i+1} est un opérateur, comme on sait que $p(a_1 \ \ldots \ a_{i+1}) \geq 1$ et que $p(a_{i+1}) = -1$, on a $p(a_1 \ \ldots \ a_i) \geq 2$; par l'hypothèse de récurrence ce nombre est aussi égal à la longueur de $f_i(A)$ ce qui montre que $k \geq 2$, donc que $f_{i+1}(A)$ est bien définie. Mais alors la longueur de $f_{i+1}(A)$ est, d'après sa définition, égale à la longueur de $f_i(A)$ diminuée de 1, donc au poids de $a_1 \ \ldots \ a_i$ diminué de 1, donc au poids de $a_1 \ \ldots \ a_{i+1}$.
Enfin, si a_{i+1} est une fonction, alors il est clair que $f_{i+1}(A)$ est définie et sa longueur est égale à la longueur de $f_i(A)$, soit, par l'hypothèse de récurrence, au poids de $a_1 \ \ldots \ a_i$, donc au poids de $a_1 \ \ldots \ a_{i+1}$ (puisque $p(a_{i+1} = 0)$. Ceci achève la démonstration du lemme.

Théorème

Soit A une expression algébrique postfixée de longueur m. Alors $f_m(A) = \mathrm{Ev}(A)$.

Démonstration

par récurrence sur m. Posons $A = a_1 \ \ldots \ a_m$. C'est clair si $m = 1$ puisqu'alors $f_1(A) = a_1 = \mathrm{Ev}(A)$. Supposons le résultat vrai pour toute expression algébrique postfixée de longueur strictement inférieure à m. Si $a_m = f$ est une fonction, alors $A = B\ f$. Par hypothèse de récurrence $f_{m-1}(A) = f_{m-1}(B) = \mathrm{Ev}(B)$ et donc $f_m(A) = f\big(\mathrm{Ev}(B)\big) = \mathrm{Ev}(A)$.
Si $a_m = c$ est un opérateur, alors $A = B\ C\ c$ où B et C sont des expressions algébriques postfixées. Appelons p la longueur de B, q la longueur de C si bien que $m = p + q + 1$. Pour tout $i \leq p$, on a $f_i(A) = f_i(B)$ et en particulier pour $i = p$, par l'hypothèse de récurrence, $f_p(A) = f_p(B) = \mathrm{Ev}(B)$.

Démonstration (suite)

On montre maintenant par récurrence sur $i \in [1, q]$ que $f_{p+i}(A) = \mathrm{Ev}(B)\ f_i(C)$. C'est clair si $i = 1$; supposons le donc pour i et distinguons suivant le type de a_{p+i+1}. Si $a_{p+i+1} \in E$, alors

$$f_{p+i+1}(A) = f_{p+i}(A)\ a_{p+i+1} = \mathrm{Ev}(B)\ f_i(C)\ a_{p+i+1} = \mathrm{Ev}(B)\ f_{i+1}(C)$$

Si a_{p+i+1} est une fonction f, posons $f_i(C) = b_1\ \ldots\ b_k$ avec $k \geq 1$; on a $f_{p+i}(A) = \mathrm{Ev}(B)\ = b_1\ \ldots\ b_k$ et donc

$$f_{p+i+1}(A) = \mathrm{Ev}(B)\ b_1\ \ldots\ f(b_k) = \mathrm{Ev}(B)\ f_{i+1}(C)$$

Si maintenant a_{p+i+1} est un opérateur d, posons $f_i(C) = b_1\ \ldots\ b_k$ avec $k \geq 1$; comme B est le plus grand préfixe de poids 1, le poids de $a_1\ \ldots\ a_p\ a_{p+1}\ \ldots a_{p+i+1}$ est au mois égal à 2 et comme $p(a_{p+i+1}) = -1$, le poids de $a_1\ \ldots\ a_p\ a_{p+1}\ \ldots a_{p+i}$ est au mois égal à 3 ; mais ce nombre est aussi la longueur de $f_{p+i}(A) = \mathrm{Ev}(B)\ f_i(C)$ (par l'hypothèse de récurrence).

Démonstration (suite)

On a donc $k \geq 2$; on en déduit que

$$f_{p+i+1}(A) = \mathrm{Ev}(B)\ b_1\ \ldots\ b_{k-2}\ d(b_{k-1}, b_k) = \mathrm{Ev}(B)\ f_{i+1}(C)$$

Pour $i = q$, on a donc $f_{m-1}(A) = \mathrm{Ev}(B)\ f_q(C) = \mathrm{Ev}(B)\ \mathrm{Ev}(C)$ par l'hypothèse de récurrence. On a alors $f_m(A) = c\big(\mathrm{Ev}(B), \mathrm{Ev}(C)\big) = \mathrm{Ev}(A)$.
Dans tous les cas, on a bien $f_m(A) = \mathrm{Ev}(A)$.

Remarque

Cette méthode fournit en même temps un vérificateur de la syntaxe de l'expression algébrique A. D'après la caractérisation des expressions algébriques postfixées, A est une expression algébrique postfixée si et seulement si $f_m(A)$ est définie (ce qui correspond à tout préfixe strict est de poids supérieur ou égal à 1) et $f_m(A)$ est de longueur 1 (ce qui correspond à $p(A) = 1$).

Pour écrire une procédure Caml, nous allons utiliser une structure de liste pour l'expression algébrique : l'expression algébrique $a_1 \ \ldots \ a_m$ sera entrée dans une liste Caml $[a_1, \ldots, a_m]$ ce qui permettra d'en extraire les éléments a_i dans l'ordre de leur apparition. Pour permettre d'entrer dans cette liste aussi bien des nombres (ici réels) que des opérateurs (ici $+, -, *, /$) et des fonctions (ici $\sin, \cos, \exp, \log$), nous définirons un type construit `eap_elt` (pour *él*émen*t* d'une *e*xpression *a*lgébrique *p*ostfixée) qui pourra être de l'un des ces trois types, les opérateurs étant symbolisés par des caractères et les fonctions par leur nom :

Caml

```
#type eap_elt=Nb of float | Op of char | Fct of string;;
Type eap_elt defined.
```

Les manipulations à faire pour calculer les $f_i(A)$ sont clairement d'adjoindre un élément a_{i+1}, de retirer deux éléments b_{k-1} et b_k puis d'adjoindre $c(b_{k-1}, b_k)$, de retirer b_k et d'adjoindre $f(b_k)$. Une structure de pile est donc idéale pour cela, puisque ce sont toujours les derniers éléments entrés que l'on doit retirer. Ceci conduit directement à la fonction suivante où nous avons défini une pile `accu` qui est une référence sur une liste, initialement vide, et deux fonctions adaptées `push` (qui adjoint un élément à la pile) et `pop` (qui retire un élément de la pile et renvoie la valeur de cet élément).

La procédure `eval` effectue tout le travail de manière récursive sur l'expression algébrique. L'invariant évident de ces appels récursifs est qu'après le i-ième appel de la fonction `eval`, B contient $[a_{i+1}; \ldots; a_m]$ et `accu` contient $[f_i(A)] = [b_k; \ldots; b_1]$ (dans l'ordre inverse puisque les listes Caml ont pour premier élément le dernier entré) ; la terminaison est garantie par la stricte décroissance de la liste B ; tout ceci démontre la validité de la fonction.

Caml

```
#let evalue_eap A =
   let accu = ref [] in
      let push x =  accu := x::!accu
      and pop () =
         match !accu with
               [] -> failwith "erreur de syntaxe: trop d'opérateurs"
          |   h::r -> accu := r; h
      in
         let rec eval B =
             match B with
                [] ->  ()
             |  (Nb x)::r -> push x; eval r
```

Caml

```
|   (Fct f)::r -> let x = pop () in
        begin
          match f with
            "sin" ->  push (sin x)
          | "cos" ->  push (cos x)
          | "exp" ->  push (exp x)
          | "log" ->  push (cos x)
          |  _  ->  failwith "erreur de syntaxe: fonction inco
        end;
        eval r
|    (Op c)::r -> let y = pop () and x = pop () in
        begin
          match c with
            `+` ->  push (x +. y)
          | `-` ->  push (x -. y)
          | `*` ->  push (x *. y)
          | `/` ->  push (x /. y)
          |  _  ->  failwith "erreur de syntaxe: opérateur inc
        end;
        eval r
```

Caml

```
in
  eval A;
  match !accu with
      []   -> failwith "erreur de syntaxe: trop d'opérateurs"
    | h::r -> if r<>[] then failwith
                              "erreur de syntaxe: trop de nombres"
                            else h;;
```

Avec quelques essais :

Caml

```
evalue_eap : eap_elt list -> float = <fun>
#evalue_eap [Nb 2. ; Nb 3. ; Nb 4. ; Op `+`; Op `*`];;
[]- : float = 14
#evalue_eap [Nb 2. ; Nb 3. ; Fct "sin" ; Op `+`; Op `*`];;
Uncaught exception: Failure "erreur de syntaxe: trop d'opérateurs"
#evalue_eap [Nb 2. ; Nb 3. ; Fct "sin" ; Op `+`];;
[]- : float = 2.14112000806
```

Cette fonction d'évaluation peut être facilement enrichie avec toutes les fonctions et tous les opérateurs usuels.

Remarque

les puristes objecteront que la fonction précédente a un double rôle : l'un de vérification de la syntaxe de l'expression algébrique, l'autre d'évaluation de cette même expression ; de ce point de vue, il y a confusion entre la syntaxe et la sémantique. Mais ceci est inhérent aux expressions algébriques postfixées : l'évaluation de l'expression en vérifie en même temps la correction. Une deuxième approche plus générale sera vue à propos des arbres et des expressions algébriques *infixes* : dans ce cas il y a séparation claire entre l'analyse syntaxique (qui consiste à construire l'arbre de l'expression) et l'évaluation sémantique.