Cours 05 - Programmation récursive

MPSI - Prytanée National Militaire

Pascal Delahaye

29 mai 2015

1 Les listes

Les *listes* constituent un autre type de données très souvent utilisée en langage Caml et plus particulièrement dans la programmation récursive. Elle correspondent à un type de données plus général appelé *liste dynamique*.

Les listes, comme les vecteurs sont des suites finies d'éléments. Cependant, alors que la taille d'un vecteur est fixée à sa création, la taille d'une liste pourra varier au fur et à mesure des instructions.

La définition d'une structure de données comprend des *fonctions de manipulation*. Il s'agit des fonctions de base indispensables pour travailler avec cette structure de données. Toutes les autres fonctions agissant sur la donnée sont alors programmées à l'aide de ces fonctions de manipulation.
Pour la structure *liste*, les *fonctions de manipulation* (ou *opérations*) sont les suivantes :

1. Les constructeurs (permettent de construire la donnée) : `[] ; a::[...]`
2. Les fonctions de sélection (permettent de récupérer l'information) : `hd ; tl`
3. Les prédicats (permettent de tester la donnée) : `L = []`

`Exemple 1.` **Familiarisation avec la structure "liste"**

Instructions	Réponse	Commentaires
let liste1 = [3 ;6 ;8 ;12] ; ;		
hd liste1 ; ;		
tl liste1 ; ;		
let liste2 = 4 : :liste1 ; ;		
let liste3 = liste2@[1 ;2] ; ;		
let liste4 = [] ; ;		
let liste5 = [‘a‘ ;1 ;"toto"] ; ;		
let liste6 = [‘a‘,1,"toto"] ; ;		
let liste7 = list_of_vect [|4 ;8 ;2|] ; ;		
let liste8 = vect_of_list [3 ;2 ;1] ; ;		

A retenir ! !

1. Un liste est définie entre crochets et les éléments sont séparés par des points virgule : `[3;2;1]`
2. Contrairement aux vecteurs, nous n'avons pas directement accès à un élément quelconque de la liste. En revanche :
   (a) La commande `hd` (*head*) donne le premier élément d'une liste.
   (b) La commande `tl` (*tail*) donne la liste des éléments suivants le premier.
3. La liste vide est notée `[]`
4. Il est possible d'ajouter un élément `a` en tête de liste à l'aide de la commande `a::liste`.
5. Il est possible de concaténer deux listes par la commande `liste1@liste2`
6. Une liste ne peut contenir des éléments de types différents.

`Exemple 2.` **Image d'une liste par une fonction**
Donner le type et interpréter le sens des fonctions suivantes :

```
let f = function                          let g liste = match liste with
        | [] -> 3                                 | [] -> 3
        | x::q -> x+3 ;;                          | x::q -> x+3;;
```

A retenir ! !

1. La notation `x::q` représente une liste dont le premier élément est `x`.
   Elle n'a de sens que si la liste est non vide !
2. Dans le cas où la fonction admet une liste pour argument, il faudra impérativement utiliser le mot-clé `function` à la place du mot-clé `fun`. Vérifier que `fun` ne convient pas ...

`Exemple 3.` **Image de deux listes par une fonction**
Donner le type et interpréter le sens des fonctions suivantes :

```
let f = function                                  let f l1 l2 = match (l1,l2) with
        | ([], []) -> 3                                   | ([], []) -> 3
        | ([], _)  -> 4                                   | ([], _)  -> 4
        | (_, [])  -> 5                                   | (_, [])  -> 5
        | (x1::q1, x2::q2) -> x1+x2 ;;                    | (x1::q1, x2::q2) -> x1+x2 ;;
f ([3;6], [5;7;8]);;                              f [3;6] [5;7;8];;
```

A retenir ! !
Les fonctions usuelles sur les listes

`list_length L`	donne le nombre d'éléments contenus dans la liste L
`map f L`	permet d'appliquer la fonction f à tous les éléments de la liste L
`mem e L`	permet de savoir si un élément e est contenu dans la liste L
`L1 @ L2`	permet de concaténer les deux suites L1 et L2
`rev L`	inverse l'ordre des éléments de la liste L
`hd L`	donne le premier élément de la liste L
`tl L`	enlève le premier élément de la liste L

`Exemple 4.` Tester chacune des fonctions précédentes sur des exemples.

Exercice : 1
Les fonctions `list_length`, `map` et `mem` sont des fonctions "évoluées" portant sur les listes.
Retrouver leur programmation itérative à l'aide des fonctions de manipulation vues en introduction

2 La programmation récursive

Dans cette partie, on souhaite programmer une fonction f d'argument(s) A.
On pourra utiliser la programmation récursive dès lors que $f(A)$ peut se calculer à partir de $f(A')$ où A' est un argument "inférieur" à A.

`Exemple 5.`

1. La fonction $f(n) = \sum_{k=1}^{n} k$
2. La fonction $f(a,\ n) = a^n$
3. La fonction $f(n,\ p) = \binom{n}{p}$
4. La fonction qui donne la longueur d'une liste.

Le principe est alors le suivant :

Pour calculer $f(a)$, l'ordinateur réserve un espace mémoire dans une pile et cherche à calculer $f(a')$ pour lequel il réserve de nouveau un espace mémoire en sommet de pile puis cherche à calculer $f(a'')$ ... etc ... Comme les arguments a, a', a'' sont strictement décroissants, on arrive au bout d'un nombre fini d'étapes à un argument $a^{(n)}$ pour lequel le résultat de la fonction est évident !! Cet argument s'appelle alors un *cas de base* et pour s'assurer que le nombre d'appels récursifs de la fonction f reste fini, on programme la valeur de $f(a^{(n)})$ dans la fonction.

2.1 Un exemple de programme récursif

`Exemple 6.` **La somme des n premiers entiers :** $f(n) = \sum_{k=1}^{n} k$

1.	Nous avons la formule :	$f(n) = n + f(n-1)$
2.	Nous savons que :	$f(1) = 1$

Implémentations CAML :

1. En programmation fonctionnelle :

```
let rec somme = fun
        | 1 -> 1
        | n -> n + somme(n-1) ;;
```

```
let rec somme n = match n with
        | 1 -> 1
        | _ -> n + somme(n-1) ;;
```

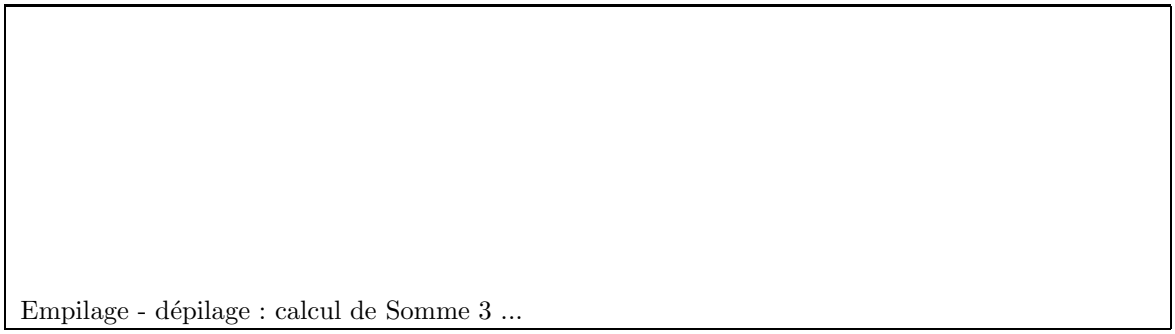
2. Avec une structure de contrôle :

```
let rec somme n =
        if n = 0 then 0
                 else n + somme(n-1) ;;
```

Remarque 1.

1. On utilise le mot-clé `rec` devant le nom de la fonction pour définir une fonction récursif.
2. Remarquer la simplicité de la programmation fonctionnelle.
3. On peut suivre les différents appels d'une fonction récursive grâce à la fonction `trace` :

```
trace "somme";;
somme 3;;
The function somme is now traced.
- : unit = ()
#somme <-- 3
somme <-- 2
somme <-- 1
somme <-- 0
somme --> 0
somme --> 1
somme --> 3
somme --> 6
- : int = 6
```



Empilage - dépilage : calcul de Somme 3 ...

Remarque 2. Comme l'ordinateur réserve un espace mémoire dans une pile à chaque appel de fonction, les programmes récursifs utilisent en général beaucoup plus de place mémoire qu'un programme itératif.

Si les cas de base de la fonction ont été mal définis, celle-ci peut en théorie, s'appeler indéfiniment. En réalité, si la totalité des places mémoire réservées dépasse la hauteur limite d'une pile, un message d'erreur de la form `Stack overflow` ou `out of memory` apparaît.

Dans les débuts de l'informatique, l'utilisation d'algorithmes récursifs était interdite pour préserver l'intégrité des machines. Aujourd'hui, les machines utilisées sont devenues beaucoup plus puissantes et la récursivité est communément utilisée essentiellement pour la simplicité de son mode de programmation.

2.2 Comment compter le nombre de boucles récursives effectuées ?

Pour compter le nombre d'appels récursifs d'une fonction récursive `f`, il suffit de :

1. lui rajouter une variable `n`
2. d'affecter `f` de la variable `(n+1)` lors de l'appel
3. de s'assurer que la valeur de `n` est bien donnée en sortie
4. de lancer la fonction `f` en donnant à `n` la valeur 1

`Exemple 7.` Voici ce que ça donne pour l'algorithme d'euclide :

```
let rec eucl a b n = match b with
              |0 -> a,n;
              |_ -> eucl b (a mod b) (n+1);;
eucl 18 12 1 ;;
```

Programmez un lancer de dé récursif qui permet de compter le nombre de lancers nécessaires pour obtenir un 6.

2.3 Exercices

Exercice : 2

Dans les exercices suivants, vous commencerez par déterminer une formule permettant de programmer la fonction demandée sous forme récursive :

1. **Calcul de $n!$**
   Programmer en utilisant 3 syntaxes possibles, une fonction récursive donnant $n!$.
2. **Calcul de a^n**
   Programmer une fonction récursive donnant a^n où $(a,\ n) \in \mathbb{R} \times \mathbb{N}$
   (vous pourrez utiliser l'algorithme d'exponentiation rapide).
3. **Calcul du coefficient de la dérivée p^{eme} de $x \to x^n$**
   Déterminer une procédure récursive permettant de calculer le coefficient de la dérivée p^{eme} de $x \to x^n$.
4. **Calcul d'un coefficient binômial**
   Déterminer une procédure récursive permettant de calculer le coefficient coefficient binômial $\binom{n}{p}$.
5. **Multiplication Egyptienne**
   Déterminer une procédure récursive permettant de calculer pq en remarquant que si p est pair, alors $pq = (p/2).(2q)$ et si p est impair alors $pq = q + (\frac{p-1}{2}).(2q)$.
6. **Calcul du PGCD de deux entiers**
   Déterminer une procédure récursive permettant de calculer le PGCD de deux entiers a et b.
7. **Calcul de la longueur d'une liste**
   Déterminer une procédure récursive permettant de calculer la longueur d'une liste L.
   Cette fonction est préprogrammée sous Caml sous le nom de `list_length`.
8. **Recherche d'un élément dans une liste**
   Déterminer une procédure récursive permettant de dire si une liste L contient un élément e.
   Cette fonction est préprogrammée sous Caml sous le nom de `mem`.
9. **Décomposition d'un entier sous la forme $n = p.2^q$**
   Déterminer une procédure récursive `decompose` d'argument n qui renvoie le couple $(p,\ q)$ de la décomposition $n = p.2^q$ avec p impair.
10. **Tester si un nombre est premier**
    Déterminer une procédure récursive permettant de dire si un entier naturel n est divisible par un nombre inférieur ou égal à p et strictement plus grand que 1.
    En déduire une procédure permettant de tester si un nombre est premier.
11. **Suite de Fibonacci**
    Déterminer une procédure récursive permettant de terme u_n de la suite de fibonacci..
12. **Suite de carrés**
    Déterminer une procédure récursive permettant d'afficher la suite des n premiers carrés.
13. **Permutation**
    Construire une procédure vérifiant qu'un tableau de n entiers représente bien une permutation de $\mathfrak{S}_n$.
    Aide : on pourra, pour cela, construire une procédure intermédiaire permettant de savoir si un entier est présent dans une liste
14. **Nombre d'inversions dans une liste**
    Construire une procédure dénombrant le nombre d'inversions contenues dans un tableau ligne.
15. **Points les plus proches**
    Construire une procédure déterminant le couple de points les plus proches dans un nuage de points donnés.
16. **Parties d'un ensemble**
    Construire une procédure récursive déterminant les parties d'un ensemble.
17. **Image miroir d'une liste**

    (a) Construire une fonction récursive qui insère un élément à la fin d'une liste.

    (b) En déduire une fonction récursive simple qui renvoie l'image miroir d'une liste.

    Rem : la fonction `rev` de Caml permet d'effectuer cette opération.

Exercice : 3

Ecriture d'un nombre dans une base

1. Déterminer une procédure récursive permettant d'écrire un entier naturel n en base $b \geq 2$ (on pourra stocker les coefficients de la décomposition dans une liste)
2. Concevoir un programme itératif effectuant ce travail.

Exercice : 4

Que font les programmes suivants ?

Pour répondre à cette question, vous pourrez procéder de la façon suivante :

1. Vous commencerez par déterminer les types de chacune des variables intervenant.
2. Puis, vous rechercherez le résultat donné par la fonction pour des valeurs simples de la variable d'itération.

1.
```
let rec comp f x = fun
                   |1 -> x
                   |n ->comp f (f x) (n-1);;
comp (fun x ->x**2.) 3. 4;;
```

2.
```
let rec iter f = fun
                |1 -> print_string ""
                |n -> print_newline();
                      f (n);
                      iter f (n-1);;
iter (fun x -> print_int (x*x)) 10;;
```

3.
```
let rec mystere1 n = match n with
         | 0 -> print_newline()
         | _ -> print_int n;
                print_char ' ';
                mystere1 (n-1);
                print_int n;
                print_char ' ';;
mystere 5;;
```

4.
```
(* renvoie le caractère en minuscule *)
let minuscule c = char_of_int (32 + int_of_char c);;

let rec mystere2 s i =
           if i = string_length s then print_newline()
           else begin print_char s.[i];
                      s.[i] <- minuscule s.[i];
                      mystere2 s (i + 1);
                      print_char s.[i]
           end;;
```

2.4 Récursivité terminale et non terminale

On distingue deux types de fonctions récursives :

1. **Les fonctions récursives terminales :**

   Il s'agit des fonctions récursives qui donnent le résultat attendu lors du dernier appel de la fonction.
   Par exemple : le calcul du PGCD par l'algorithme d'Euclide.
   Pour ces fonctions, on constate qu'il n'est pas nécessaire de réserver de la mémoire destinée à stocker les résultats des appels récursifs intermédiaires. Certain langage de programmation (dont CAML) sont capables de reconnaître une récursivité terminale lors de la compilation du programme et se dispense donc de mobiliser de la mémoire inutilement.

2. **Les fonctions récursives non terminales :**

   Ces fonctions récursives, en revanche, utilisent le résultat du dernier appel de la fonction pour évaluer le résultat de l'appel précédent et on "remonte" ainsi de suite jusqu'à obtenir la valeur du premier appel.
   Inutile de préciser que les fonctions récursives non terminales sont les plus gourmandes en temps de calcul. Par exemple : le calcul du nième terme d'une suite définie par une relation de récurrence.

Fonction récursive terminale :	Fonction récursive non terminale :

On peut parfois rendre "terminale" une fonction récursive "non terminale" en utilisant un accumulateur.

`Exemple 8.` Dans le cas de la fonction $n!$, on peut :

1. utiliser la formule de récursivité `n! = n.(n-1)!` qui donne une fonction récursive non terminale
2. utiliser une fonction auxilliaire `aux(n,acc)` qui "accumule" n dans l'accumulateur (en faisant $n \times acc$) en décroissant n de 1.
   — la formule de récursivité est `aux(n,acc) = aux(n-1,n.acc)` qui donne une fonction récursive terminale
   — il suffit alors d'appeler `aux(n,1)` pour obtenir $n!$.

```
let fact n = let rec aux n acc = match n with
                                  |0 -> acc
                                  |p -> aux (p-1) p*acc
             in aux n 1 ;;
```

Exercice : 5

(**) Proposer pour chacune des fonctions suivantes, deux procédures récursives, l'une terminale et l'autre non terminale.

1. $f(n) = \sum_{k=1}^{n} k$
2. $g(a,n) = a^n$
3. $h(f,n,x) = f^n(x)$

Observer le dépassement de capacité mémoire en calculant g(1000,3000) par la fonction récursive non terminale.

Intérêt de la récursivité terminale :

```
let rec puiss1 a = fun
        |0 -> 1.
        |n -> a*.(puiss1 a (n-1));;

puiss1 2. 3;;

let puiss2 a n = let rec aux a acc = fun
                    |0 -> acc
                    |n -> aux a (a*.acc) (n-1)
             in aux a 1. n;;

puiss2 2. 3;;
```

2.5 Récursivité croisée

Deux fonctions sont dîtes *récursives croisées* si elles s'appellent l'une l'autre.

`Exemple 9.`

Il s'agit ici de programmer les fameuses *suites imbriquées* définies par $\begin{cases} a_0 = 4 \\ a_n = \dfrac{a_{n-1}+b_{n-1}}{2} \end{cases}$ et $\begin{cases} b_0 = 5 \\ b_n = \sqrt{a_{n-1}.b_{n-1}} \end{cases}$.

Pour cela, on doit utiliser la syntaxe suivante :

```
let rec a = fun
       | 0 -> 4.
       | n -> (a(n-1) +. b(n-1)) /. 2.
    and b = fun
       | 0 -> 5.
       | n -> sqrt(a(n-1) *. b(n-1));;
```

Essayer de définir les fonctions a et b de façon indépendante ... Que constatez-vous ?

Remarque 3.

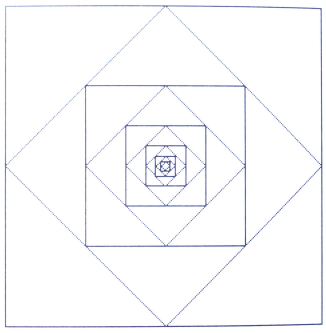
1. La programmation Caml précédente des fonctions récursives croisées correspond exactement à la définition mathématiques de celles-ci.
2. Vous remarquerez qu'il ne faut pas répéter le mot clé `rec` lors de la définition de la deuxième fonction récursive.

Exercice : 6

Construire un algorithme faisant intervenir deux fonctions récursives croisées `pair` et `impair` permettant de tester la parité d'un entier naturel n.

Exercice : 7

Construire un programme faisant intervenir deux fonctions récursives croisées `carre1` et `carre2` permettant de tracer le dessin suivant représentant 20 carrés imbriqués.
On supposera connue la fonction `trace : float*float -> float*float -> unit` permettant de tracer un carré connaissant deux sommets opposés.



3 Preuve d'un algorithme

La preuve d'un algorithme consiste à vérifier que cet algorithme se termine et renvoie le bon résultat.
On distingue donc dans la preuve deux étapes :

1. La preuve de la terminaison
2. La preuve de la correction

3.1 Preuve de la terminaison

Comme dans le cas itératif avec les boucles `WHILE`, un algorithme récursif mal construit peut ne jamais se terminer.
Nous allons ici exposer une "méthode" permettant de s'assurer de la terminaison d'un algorithme récursif.
En gros, cette preuve consiste à vérifier :

1. la décroissance stricte des arguments de la fonction au sens d'une fonction à déterminer.

2. que les appels récursifs aboutissent nécessairement à des cas traités par l'algorithme.

`Exemple 10.` **Un algorithme récursif croisé qui ne se termine pas**

```
let rec tic () = print_string "tic"; tac()
    and tac () = print_string "tac"; tic();;
```

Remarque 4. Dans l'algorithme précédent, il n'y a aucune décroissance stricte des arguments de la fonction. Evidemment puisqu'il n'y a pas d'argument du tout ! !

`Exemple 11.` Expliquer pourquoi les algorithmes programmés précédemment terminent effectivement.

3.1.1 Un peu de théorie

DÉFINITION 1 : **Terminaison**
On dira qu'une fonction f (ou qu'un algorithme) termine pour l'argument x si le calcul de $f(x)$ nécessite un nombre fini d'opérations élémentaires (calcul, échange de valeurs ...).

Remarque 5. On dira qu'une fonction f termine sur l'ensemble de ses arguments si elle termine pour tous ses arguments.

DÉFINITION 2 : **Elément minimal**
Soit $\preceq$ une relation d'ordre partielle ou totale sur un ensemble non vide E. Soit $A \subset E$.
$a \in A$ de $(E, \preceq)$ est dit *minimal* dans A si n'existe aucun élément de A strictement plus petit que a.

Remarque 6. Si l'ordre est total, la notion d'élément minimal coincide avec celle de minimum.

DÉFINITION 3 : **Ensemble bien fondé**
Soit $\preceq$ une relation d'ordre partielle ou totale sur un ensemble non vide E
$(E, \preceq)$ est *bien fondé* si et seulement si toute partie non vide de E admet au moins un élément minimal.
Ou encore :
$(E, \preceq)$ est *bien fondé* si et seulement si toute suite strictement décroissante de E est nécessairement finie

Preuve 0 : On admettra que ces deux définitions sont équivalentes.

`Exemple 12.`

1. $(\mathbb{N}, \geq)$ est bien fondé alors que $(\mathbb{Z}, \geq)$ ne l'est pas.
2. $(\mathbb{N}^2, \preceq)$ où $\preceq$ est l'ordre lexicographique ou l'ordre produit est un ensemble bien fondé.

3.1.2 Méthode

Preuve de terminaison d'une fonction récursive
Soit f une fonction récursive prenant ses arguments dans un ensemble $\mathcal{A}$.

I] On choisit judicieusement une application φ de $\mathcal{A}$ dans un ensemble bien fondé $(E, \preceq)$.

II] On détermine :

1. $\mathcal{B}$ la partie de $\mathcal{A}$ telle que $\forall b \in \mathcal{B}$, $\varphi(b)$ est un élément minimal de $\varphi(\mathcal{A})$ (les cas de base).
2. $\mathcal{L}$ la partie de $\mathcal{A}$ telle que $\forall l \in \mathcal{L}$, la formule de récursivité ne s'applique pas pour l (les cas limites).

III] On vérifie que :

1. $f(x)$ termine pour tout $x \in \mathcal{B} \cup \mathcal{L}$
2. La formule de récursivité calculant $f(x)$ fait apparaître des appels récursifs $f(y)$:
   (a) en nombre fini
   (b) tels que $\varphi(y) \prec \varphi(x)$

On peut alors conclure que : $f(x)$ termine pour tout x dans $\mathcal{A}$.

Preuve d'un algorithme récursif :

Remarque 7.
Très souvent, l'ensemble bien fondé choisi sera soit $(\mathbb{N},\ \leq)$ soit $\mathbb{N}^p$ muni de l'ordre lexicographique ou éventuellement de l'ordre produit. Voici quelques exemples de choix possibles :

$\mathcal{A}=\mathbb{N}$	$\varphi(n)=n$	un seul cas de base : $n=0$
$\mathcal{A}=\mathbb{N}^2$	$\varphi(n,\ p)=n+p$	un seul cas de base : $n=0$
	$\varphi(n,\ p)=\min(n,\ p)$	une infinité de cas de base : $(0,\ p)$ et $(n,\ 0)$
	$\varphi(n,\ p)=(n,\ p)$	un seul cas de base : $(0,\ 0)$ si ordre lexico
	$\varphi(n,\ p)=p$	une infinité de cas de base : $(n,\ 0)$

Exercice : 8

Déterminer les cas de base et les cas limites dans le cas où la formule de récursivité utilisée est de la forme :

1. $f(n)=g(f(n-1),\ f(n-2))$
2. $f(n,\ p)=g(f(n,\ p-1))$
3. $f(n,\ p)=g(f(n-p,\ p-1))$
4. $f(n,\ p)=g(f(n/p,\ p),\ f(n,\ (p+1))$ $(\varphi(n,\ p)=n-p^2)$
5. $f(n,\ p)=g(f(n+1,\ p-1))$

`Exemple 13.` **La fonction de ackermann**
Prouvons que l'algorithme récursif suivant valable pour des arguments dans $\mathbb{N}^2$ termine :

```
let rec acker = fun
        | (0,p) -> p + 1
        | (n,0) -> acker ((n - 1),1)
        | (n,p) -> acker (n-1,acker (n,p - 1)) ;;
```

Il s'agit ici d'étudier uniquement la terminaison de la fonction :

- On a $\mathcal{A}=\mathbb{N}^2$.
  On introduit par exemple l'application $\varphi=\mathrm{id}_{\mathbb{N}^2}$ en munissant $\mathbb{N}^2$ de l'ordre lexicographique.
- $\mathbb{N}^2$ admet pour élément minimal $(0,\ 0)$, dont le seul antécédent par φ est $(0,\ 0)$ donc $\mathcal{B}=\{(0,\ 0)\}$.

- La formule de récursivité n'est valable que pour tout argument $(n,\ p)$ tel que $\left\{ \begin{array}{l} n-1 \geq 0 \\ p-1 \geq 0 \end{array} \right.$ c'est à dire $\left\{ \begin{array}{l} n \geq 1 \\ p \geq 1 \end{array} \right.$.
  L'ensemble des cas limites est donc $\mathcal{L} = \{(n,\ 0),\ (0,\ p) \mid (n,\ p) \in \mathbb{N}^2\}$ (qui inclut l'unique cas de base).

1. Les cas de base et cas limites sont bien traités par l'algorithme.
2. La formule de récursivité fait apparaître 3 appels récursifs (nombre fini !) d'arguments $\left\{ \begin{array}{l} (n-1,\ 1) \\ (n,\ p-1) \\ (n-1,\ X) \end{array} \right.$.
   On a bien la décroissance stricte des arguments : $\left\{ \begin{array}{l} \varphi(n-1,\ 1) < \varphi(n,\ p) \\ \varphi(n,\ p-1) < \varphi(n,\ p) \\ \varphi(n-1,\ X) < \varphi(n,\ p) \end{array} \right.$.

Par conséquent, cet algorithme termine bien pour tout argument $(n,\ p) \in \mathbb{N}^2$!

Remarque 8.
1. Vérifier si la preuve fonctionne correctement avec les autres fonctions φ possibles.
2. La fonction de ackermann a pour particularité d'être la fonction connue admettant la plus forte croissance. Outre son intérêt pédagogique, elle est aussi parfois utilisée pour tester les performences d'un ordinateur !

`Exemple 14.` **Attention aux pièges : la fonction de Morris !**
Tenter de prouver la terminaison du programme récursif suivant.

```
let rec morris = fun
        | 0 _ -> 1
        | m n -> morris (m-1) (morris m n) ;;
```

Remarque 9. Rappelons qu'il n'est théoriquement pas toujours possible de vérifier la terminaison d'un algorithme. Ainsi, bien que l'algorithme de syracus semble toujours se terminer, rien ne nous permet d'affirmer qu'il sera un jour possible de le prouver ou éventuellement de prouver qu'il peut ne pas se terminer.

3.2 Preuve de la correction

Une fois la terminaison assurée, on peut prouver la correction de l'algorithme étudié par une méthode s'apparentant à une récurrence forte.

Preuve de correction d'une fonction récursive :

Soit f une fonction récursive prenant ses arguments dans un ensemble $\mathcal{A}$.
Il s'agit de prouver que $f(a)$ donne le bon résultat pour tout $a \in \mathcal{A}$.

On introduit $\left\{ \begin{array}{l} \bullet\ \mathcal{B} \text{ l'ensemble des cas de base} \\ \bullet\ \mathcal{L} \text{ l'ensemble des cas limites} \end{array} \right.$.

---

1. On montre que $f(a)$ donne le bon résultat pour tout $a \in \mathcal{B} \cup \mathcal{L}$
2. On fixe $a \in \mathcal{A}$ et on suppose que $f(a')$ donne le bon résultat pour tout $a' \in \mathcal{A}$ tel que $\varphi(a') \prec \varphi(a)$.
   On montre alors que $f(a)$ donne le bon résultat.

`Exemple 15.` **Cas d'une fonction à un argument entier naturel !**
Prouvons que l'algorithme récursif suivant termine et est correct :

```
let rec somme = fun
        | 0 -> 0
        | n -> n + somme(n-1) ;;
```

On a ici $\mathcal{A} = \mathbb{N}$. On introduit alors l'application $\varphi = \mathrm{id}_{\mathbb{N}} : \begin{array}{ccc} \mathbb{N} & \longrightarrow & \mathbb{N} \\ n & \mapsto & n \end{array}$.
$\mathbb{N}$ admet pour élément minimal 0, dont le seul antécédent par φ est 0 donc $\mathcal{B} = \{0\}$.
La formule de récursivité n'est plus valable si $n - 1 < 0$, c'est à dire $n = 0$ donc $\mathcal{L} = \{0\}$.

1. Terminaison :
   - Les cas de base et cas limites sont bien traités par l'algorithme
   - La formule de récursivité ne fait apparaître qu'un appel récursif donc l'argument est bien $\prec n$.

2. Correction :
   - Les cas de base et cas limite renvoient bien le bon résultat
   - on suppose que `somme (n-1)` renvoie le bon résultat. `somme n` renvoie alors le bon résultat.

`Exemple 16.` **Cas d'une fonction à deux arguments entiers naturels !**
Pour tout $(n,\ p) \in \mathbb{N}^2$, construisons une procédure récursive permettant de calculer le coefficient binômial $\binom{n}{p}$ à l'aide de la formule d'addition :

$$\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1} \quad \forall n,\ p \in \mathbb{N}^*$$

1. Nom de la fonction : f
2. Arguments : $(n,\ p) \in \mathbb{N}^2$ (on aurait pu restreindre encore plus cet ensemble ...)
3. Formule de récursivité : f(n, p) = f(n-1, p)+ f(n-1, p-1).
4. Recherche des cas limites :
   La formule de récursivité précédente n'est valable que pour les couples $(n,\ p)$ tels que $\begin{cases} n \geq 1 \\ p \geq 1 \end{cases}$.
   L'algorithme doit donc nécessairement préciser les valeurs de la fonction pour ces *arguments limites*.
   Il faut donc indiquer que : $\begin{cases} f(n,\ 0) = 1 \text{ pour tout } n \in \mathbb{N} \\ f(0,\ p) = 0 \text{ pour tout } p \in \mathbb{N}^* \end{cases}$.
5. Formalisation :
   On a ici $\mathcal{A} = \mathbb{N}^2$. Prenons alors par exemple $E = \mathbb{N}^2$ muni de l'ordre lexicographique et l'application $\varphi = \mathrm{id}_{\mathbb{N}^2}$.
   (a) $\mathbb{N}^2$ admet pour élément minimal $(0,\ 0)$, donc l'ensemble des cas de base est $\mathcal{B} = \{(0,\ 0)\}$.
   L'algorithme doit donc préciser la valeur de la fonction en $(0,\ 0)$, c'est à dire : f(0, 0)=1.
   (b) D'autre part, la fonction f ne fait appel qu'à un nombre fini d'appels récursifs tels que : $\begin{cases} \varphi(n-1,\ p) < \varphi(n,\ p) \\ \varphi(n-1,\ p-1) < \varphi(n,\ p) \end{cases}$
   (c) Enfin, on fait confiance aux mathématiciens pour nous avoir donné une formule juste !

On obtient alors le programme suivant :

```
let rec binom = fun
        | (n,0) -> 1
        | (0,p) -> 0
        | (n,p) -> binom (n-1,p-1) + binom(n-1,p) ;;
```

Exercice : 9
Prouver ou reprenez l'élaboration des algorithmes récursifs précédents en suivant la démarche vue précédemment !

3.3 Cas particulier d'une fonction récursive ayant une liste pour argument

Dans le cas d'une telle fonction :

1. L'ensemble $\mathcal{A}$ des arguments est l'ensemble des listes.
2. L'ensemble bien fondé que l'on choisit est $(\mathbb{N},\ \leq)$.

3. L'application $\varphi : \mathcal{A} \mapsto \mathbb{N}$ choisie sera alors la fonction qui à toute liste associe sa longueur.
4. L'unique élément minimal de $\mathbb{N}$ est 0.
5. Il existe un unique cas de base qui est la liste vide : $\varphi^{-1}(\{0\}) = \{[]\}$

Dans ce cas, la méthode précédente donne :

Preuve de correction et de terminaison
Pour prouver qu'une fonction se termine et renvoie le bon résultat pour toute liste donnée en argument :

1. On vérifie que la fonction renvoie le bon résultat pour la liste vide
2. On suppose que la liste renvoie le bon résultat pour une liste `q` et on montre qu'elle renvoie alors le bon résultat pour une liste `x::q`

`Exemple 17.` Programmation des fonctions `list_length` et @

1. Ecrire une fonction récursive qui compte le nombre d'éléments d'une liste
2. Ecrire une fonction récursive qui concatène deux listes.

4 Exercices

Exercice : 10
Programmer de façon récursive la recherche dichotomique d'un élément dans un vecteur trié.
Prouver votre programme.

Exercice : 11
Lorsqu'on introduit l'ensemble $\mathbb{N}$ à l'aide des axiomes de Peano, la première opération que l'on présente est l'addition. La multiplication dans $\mathbb{N}$ est alors définie récursivement de la façon suivante :

- $\forall n \in \mathbb{N}$, $n * 0 = 0$
- $\forall (n,\ p) \in \mathbb{N} \times \mathbb{N}^*$, $n * p = n * (p-1) + n$

En suivant cette définition, programmer et prouver la fonction `Mult n p`.

Exercice : 12
Il s'agit ici d'écrire une fonction récursive `change` qui détermine une façon de payer une somme d'argent s à l'aide d'une liste de valeurs de billets disponibles. Nous supposerons que cette liste est donnée de façon décroissante.
Par exemple : `change 1790 [500;200;100;50;10]`.
Le résultat pourra être donné sous la forme d'une liste des billets nécessaires. Ex : `[500;500;500;200;50;10;10;10;10]`.

Exercice : 13
On considère un jeu traditionnel de 32 cartes. A chaque carte numéroté, on associe la valeur 0, puis la valeur 1 aux valets, la valeur 2 aux dames et la valeur 3 aux rois.

1. Définir un type "carte" de la façon suivante : `type carte = sept | ... | as ;;`.
2. Programmer un algorithme récursif permettant de donner la valeur d'une main contenant des cartes.

Exercice : 14
Le jeu de *fizzbuzz* consiste à, partant de $n = 1$ demander à chaque personne autour d'une table de citer le nombre entier suivant en respectant les règles suivantes données par ordre de priorité :

— Si n est un multiple de 35, la personne doit dire *fizzbuzz*
— Si n est un multiple de 5, la personne doit dire *buzz*
— Si n est un multiple de 7 ou contient un 7 dans son écriture décimale, la personne doit dire *fizz*
— Sinon, la personne doit simplement dire le nombre n.

L'objectif de cet exercice est de construire un programme donnant la liste des mots à prononcer par les joueurs.
Pour cela, nous allons décomposer le problème en sous-problème :

1. Définir les fonctions suivantes :
   (a) la fonction `charlist-of-string` qui convertit une chaîne de caractères en la liste de ses caractères.
   (b) la fonction `contient c s` qui teste si le caractère c apparaît dans la chaîne de caractère s.
2. Définir la fonction `fizzbuzz` qui écrit à l'écran la valeur qui doit être prononcée en fonction de n.
3. En déduire la fonction `List-fizzbuzz n` qui affiche la séquence des `fizzbuzz k` pour k allant de 1 à n.