Chapitre 2
Introduction pragmatique à la récursivité
Les listes Caml

I Récursivité

I.1 Introduction

La **récursivité** est un concept très général, déjà présent en mathématiques, mais également en informatique.

De manière informelle un objet (qui peut être un algorithme, une structure de données, une image,...) est récursif si la définition de cet objet fait appel à un objet de même nature.

Les cas suivants constituent des cas concrets de récursivité :

— Processus dépendant de paramètres et faisant appel à ce même processus sur d'autres paramètres plus simples (par exemple : suites récurrentes)
— Image contenant en elle-même des images similaires (voir figures 1a et 1b)
— Concept défini en invoquant le même concept
— Algorithme qui s'appelle lui-même.

(a) Vache

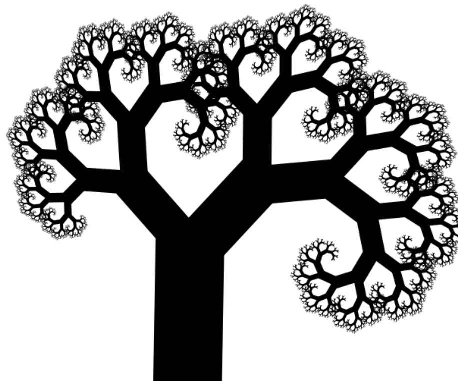

(b) Arbre récursif

Figure 1 – Exemples de dessins récursifs

I.2 Quelques exemples

On présente ici simplement quelques exemples pour s'habituer à « penser » récursif. L'étude théorique correspondante des problèmes de terminaison, des calculs de complexité, viendra plus tard. Ces exemples ne sont pas pris nécessairement dans le monde mathématique pour bien illustrer la généralité du concept.

I.2.1 Matchs d'un tournoi d'échecs

On dispose d'une liste de joueurs d'échecs et on veut créer une liste de matchs, de telle sorte que chaque joueur joue contre tous les autres joueurs exactement une seule fois.

S'il n'y a qu'un joueur, il n'y a aucun match.

S'il y a deux joueurs J_1 et J_2, il y a un seul match J_1/J_2.

S'il y a trois joueurs J_1, J_2 et J_3, il y a trois matchs J_1/J_2, J_1/J_3 et J_2/J_3.

On peut remarquer que si on rajoute un quatrième joueur J_4 il suffit pour obtenir tous les matchs du tournoi à quatre joueurs de rajouter à la liste du tournoi à trois joueurs tous les match entre J_4 et les trois autres joueurs.

De manière plus générale pour obtenir une liste de tous les matchs entre n joueurs J_i on peut obtenir une liste des matchs entre les $n-1$ premiers joueurs, puis rajouter tous les matchs J_i/J_n $(i \leqslant n-1)$ entre J_n et tous les autres joueurs.

Cette procédure est donc récursive puisque pour le problème à n joueurs, on utilise la même procédure (ici appliquée à $n-1$ joueurs).

I.2.2 Calcul récursif de la factorielle

La définition de la factorielle d'un entier n'est pas intrinsèquement récursive, mais elle se prête bien à un calcul récursif, dès lors que l'on remarque que $n! = n \times (n-1)!$.

Notons que si on se limite à cette relation, le calcul de 4! amènera ceux successivement de 3!, 2!, 1!, 0!, $-1!$, etc... et que le calcul ne s'arrêtera pas... Il faut donc disposer de cas élémentaires pour lesquels on connaît la réponse sans calcul récursif. Ici c'est par exemple $0! = 1$. Il est fondamental d'identifier ces cas de base pour assurer le bon fonctionnement du calcul.

I.2.3 Calcul du nombre de façons de construire une barrière

On veut construire une barrière de n mètres de long (n entier), à l'aide d'éléments de barrière de longueur 2 m ou 3 m. On cherche à dénombrer les façons de faire, que l'on note $f(n)$.

Clairement si $n = 0$ ou $n = 1$ il n'y a aucune façon de faire et donc $f(0) = f(1) = 0$.

Si $n = 2$ ou $n = 3$ il n'y a qu'une seule façon et donc $f(2) = f(3) = 1$.

Pour une longueur $n \geqslant 3$, pour le premier élément de barrière on a le choix entre une longueur de 2 mètres (et il restera alors $n-2$ mètres à réaliser), et une longueur de 3 mètres (et il restera à réaliser $n-3$ mètres à réaliser).

On peut donc écrire dans ce cas $f(n) = f(n-2) + f(n-3)$.

Ce dénombrement peut donc se réaliser récursivement.

Le but de cet exemple est de montrer trois choses :

- on peut faire plusieurs appels récursifs (ici 2)
- les appels n'ont pas nécessairement lieu sur un problème de taille juste inférieure à celle du traitement courant. Ici pour calculer $f(n)$ on appelle $f(n-2)$ (problème de taille $n-2$) et $f(n-3)$ (problème de taille $n-3$), alors que dans les deux exemples précédents, on faisait un appel récursif avec un problème de taille $n-1$.
- il n'y a pas forcément qu'un seul cas de base.

I.2.4 Les tours de Hanoï

Il s'agit d'un exemple très intéressant (et très prisé des informaticiens) de récursivité. Il sera présenté en cours. Sa particularité est qu'il s'agit d'une procédure récursive qui décrit des mouvements à réaliser sur un système concrêt pour réaliser une tâche donnée. D'une part on ne calcule pas une valeur numérique et d'autre part la programmation récursive est étonnament simple alors que la résolution purement itérative du problème semble hors de portée du cerveau humain dès que la taille du problème est supérieure à quelques unités...

I.3 Algorithme récursif

Il découle des exemples précédents les caractères généraux suivants, qui permettent de donner une définition un peu informelle d'un algorithme récursif :

Définition Un algorithme récursif se caractérise par :

1. une **base** de récursivité correspondant à un ou plusieurs cas particuliers pour lesquels le résultat est connu ou se calcule sans faire appel à l'algorithme lui-même. On parle des **cas de base**
2. le traitement des autres cas faisant appel à **un ou plusieurs appels récursifs, i.e. de l'algorithme lui-même**, généralement sur des données de « taille »(notion à définir au cas par cas...) inférieure au cas que l'on est en train de traîter.

II Description des listes en Caml

II.1 Introduction

Le type 'a list [1] existe en Caml mais ces listes sont **très** différentes des listes python :

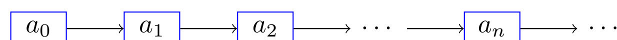
— les listes peuvent grossir dynamiquement, mais les éléments sont toujours rajoutés en **tête de liste** ;

— on ne peut accéder directement **qu'à la tête de la liste.**

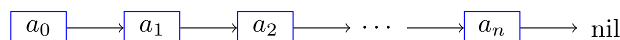
— tous les éléments d'une liste sont du même type (listes homogènes)

Présentons de manière un peu plus abstraite cette notion de liste : en informatique, une *structure de données* est la description d'une structure **logique** destinée à organiser et à agir sur des données **indépendamment** de la mise en œuvre effective de ces structures (i.e. de la façon dont sont codées en interne les opérations sur ces données). Nous allons en étudier plusieurs cette année, et nous commençons par les *listes*.

Une liste est une collection séquentielle et de taille arbitraire de données de même type : chaque élément possède, en plus de la donnée, d'un pointeur vers l'élément suivant de la liste. On peut donc représenter une liste par la figure suivante [2] :



Néanmoins, un tel schéma est incomplet car taille arbitraire ne signifie pas taille infinie : il est nécessaire que notre liste se termine. Nous allons donc adjoindre à cette description un élément particulier caractérisant la terminaison de la liste, et qu'on appelle le *nil* (abréviation du latin *nihil*, autrement dit, rien).



Ainsi, le type de données abstrait définissant une liste est le suivant :

$$\boxed{\text{Liste} = \text{nil} + \text{Élément} \times \text{Liste},}$$

ce qui signifie qu'une liste est soit la liste vide, soit un couple constitué d'un élément et d'une liste. On notera que cette définitation de la liste est **récursive** ! Ce sera donc notre premier terrain de jeux pour la programmation récursive.

II.2 Construction d'une liste

Les listes sont délimitées par des crochets `[` et `]` , les éléments (qui doivent être de même type) étant séparés par un point-virgule. Par exemple :

Code Caml 1

```
1  # [4; 3; 2; 1] ;;
2  - : int list = [4; 3; 2; 1]
3  # ['a'; 'b'; 'c'] ;;
4  - : char list = [ 'a'; 'b'; 'c']
5  # [4; 'a'; 3; 2; 1] ;;
6  Entrée interactive :
7  >[4; 'a '; 3; 2; 1] ;;
8  >^^^^^^^^^^^^^^
9  Error: This expression has type char but an expression was expected of type
10 int
```

Bien entendu, l'élément nil sera représenté par la liste vide `[]` et aura pour type 'a list .

L'ajout d'un élément en tête de liste est représenté par le constructeur infixe [3] : : qu'on appelle *conse* (pour constructeur de liste). Ce constructeur est associatif à droite, ce qui permet d'éviter l'écriture de nombeuses parenthèses inutiles.

1. liste d'éléments de type 'a , notez le polymorphisme
2. Pour la culture, on parle de liste simplement chaînée
3. infixe pour signifier qu'il se place entre ses arguments : `arg1 op arg2`

Code Caml 2

```
# 5 :: [4; 3; 2; 1] ;;
-: int list = [5; 4; 3; 2; 1]
# 'a' :: ('b' :: ('c' :: [])) ;; (* version avec parenthèses (
   inutiles) *)
-: char list = ['a'; 'b'; 'c']
# 'a' :: 'b' :: 'c' :: [] ;; (* version sans parenthèses, é
   criture conseillée *)
-: char list = ['a'; 'b'; 'c']
```

À l'inverse, les fonctions `hd` (head) et `tl` (tail) permettent d'obtenir la tête (le premier élément de la liste) et la queue (la liste privée de son premier élément) d'une liste :

Code Caml 3

```
# List.hd [4; 3; 2; 1] ;;
- : int = 4
# List.tl [4; 3; 2; 1] ;;
- : int list = [3; 2; 1]
```

Notons qu'il faut faire précéder les fonctions du nom du module correspondant (ici `List`) et d'un point ".". Notez également que ces deux fonctions déclenchent une exception lorsqu'on essaye de les appliquer à la liste vide :

Code Caml 4

```
# List.hd [] ;;
Exception: Failure "hd".
```

II.3 Filtrage et récursivité

Compte tenu de la nature récursive de la définition des listes, nous serons amenés à programmer essentiellement avec des fonctions récursives qui traiteront les divers cas (cas de base, cas de récurrence) par filtrage (`match`). Dans ce cadre le motif `tete :: queue` est reconnu par toute liste non vide, et dans la suite de l'évaluation `tete` prendra la valeur de la tête et `queue` celle de la queue. Le motif pour la liste vide est bien sûr `[]`. Il nous est donc facile de redéfinir à titre d'exemple les fonctions `hd` et `tl` :

Code Caml 5

```
# let rec head liste =
    match liste with
    | [] -> failwith "liste vide"
    | tete :: queue -> tete ;;
val head : 'a list -> 'a = <fun>
# let rec tail liste =
    match liste with
    | [] -> failwith "liste vide"
    | tete :: queue -> queue ;;
val tail : 'a list -> 'a list = <fun>
```

Associé à la récursivité, nous avons là le mode principal de définition d'une fonction agissant sur les listes.

II.4 Premières fonctions élémentaires sur les listes.

Nous allons nous intéresser à des fonctionnalités basiques mais importantes sur les listes (ce que j'appelle les gammes...). Il vous est demandé pour chaque fonction de réfléchir à la description en français

de la programmation récursive, puis éventuellement de proposer le code Caml correspondant. Tout ceci sera présenté/corrigé en cours. Certaines fonctions sont déjà programmées en OCaml ; elles seront signalées.

Le calcul de la longueur d'une liste : Fonction qui renvoie le nombre d'éléments d'une liste :

Code Caml 6

```
# let rec longueur liste =
À compléter.
```

Caml dispose déjà de l'instruction correspondante `List.length`.

L'obtention du dernier élément d'une liste Fonction qui retourne le dernier élément d'une liste, et lève une exception si la liste est vide.

Code Caml 7

```
# let rec dernier liste =
À compléter.
```

Le test d'appartenance à une liste Fonction qui retourne un booléen traduisant l'appartenance d'un élément à une liste.

Code Caml 8

```
# let rec membre x liste =
À compléter.
```

Caml dispose déjà de l'instruction correspondante `List.mem` .

L'obtention du n^e élément d'une liste Fonction qui retourne le n-ième élément d'une liste (en comptant les élément à partir de 0, comme le font les informaticiens). Attention l'analyse et la programmation sont plus subtiles... et fait appel a un filtrage sur les deux paramètres d'entrée...

On n'utilisera pas la fonction `longueur` définie plus haut pour des raisons de compexité.

Code Caml 9

```
# let rec nieme n liste =
À compléter.
```

La concaténation de deux listes Il s'agit de fabriquer une nouvelle liste à partir des deux listes reçues en paramètres, avec tous les éléments de la première liste en tête (et dans l'ordre initial) puis tous ceux de la deuxième liste (aussi dans l'ordre initial).

Code Caml 10

```
# let rec concat lst1 lst2 =
À compléter.
```

Caml dispose de l'instruction `@` qui est la forme **infixe** (i.e. que l'opérateur se met **entre** les deux opérandes) de cette fonction :

Code Caml 11

```
# [1; 2; 3] @ [4; 5; 6] ;;
- : int list = [1 ;2; 3; 4; 5; 6]
```

II.5 Complexité des fonctions sur les listes

Mis à part les fonctions `List.hd` et `List.tl` , **aucune** des fonctions ci-dessus n'a un coût temporel constant : si n désigne la longueur de la liste, la fonction `List.length` et la fonction `List.mem` ont une complexité en $O(n)$.

Quand à la concaténation `lst1 @ lst2` , son coût est proportionnel à la longueur de la liste `lst1` .

III Fonctionnelles agissant sur les listes

On s'intéresse maintenant à des fonctionnalités plus avancées. Nous ne les utiliserons a priori que très peu en première année, mais vous risquez d'en avoir un usage plus important en deuxième année.

III.1 Les fonctions List.map et List.iter

Étant données une fonction f de type 'a -> 'b et une liste $[a_0; \cdots; a_{n-1}]$ de type 'a list , la fonctionnelle `map` a pour objet de créer la liste $[f(a_0); \cdots; f(a_{n-1})]$. Proposer une définition OCaml

Code Caml 12

```
1 # let rec map f liste =
2 À compléter.
```

Son type est ('a -> 'b) -> 'a list -> 'b list ; notons que cette fonction est prédéfinie en OCaml . Dans l'exemple suivant on applique la fonction qui calcule la longueur d'une chaîne de caractères à chacune des chaînes de la liste (ce qui permet de connaître les premières décimales de π...) :

Code Caml 13

```
1 # List.map String.length ["Que"; "j"; "aime"; "a"; "faire"; "
    apprendre"; "ce"; "nombre"; "utile"; "aux"; "sages"] ;;
2 -: int list = [3; 1; 4; 1; 5; 9; 2; 6; 5; 3; 5]
```

Étant données une fonction f de type 'a -> unit et une liste $[a_0; \cdots; a_{n-1}]$ de type 'a list , la fonctionnelle `List.iter` a pour objet d'effectuer la **séquence** (notion que nous reverrons plus tard, plutôt dans le contexte de la programme impérative) $f(a_0); f(a_1); \cdots; f(a_{n-1})$ (ce qui n'a d'intérêt que si f a un effet sur l'environnement). Sa définition est la suivante. Là je donne le code car nous ne connaissons par encore les séquences.

Code Caml 14

```
1 # let rec iter f liste = function
2   | [] -> ()
3   | tete :: queue -> f tete ; iter f queue ;;
```

Son type est ('a -> 'b) -> 'a list -> unit ; cette fonction est, elle aussi, prédéfinie en Caml .

Code Caml 15

```
1 # List.iter print_string ["Que"; "j"; "aime"; "a"; "faire"; "
    apprendre"; "ce"; "nombre"; "utile"; "aux"; "sages"];;
2 Quejaimeafaireapprendrecenombreutileauxsages- : unit = ()
```