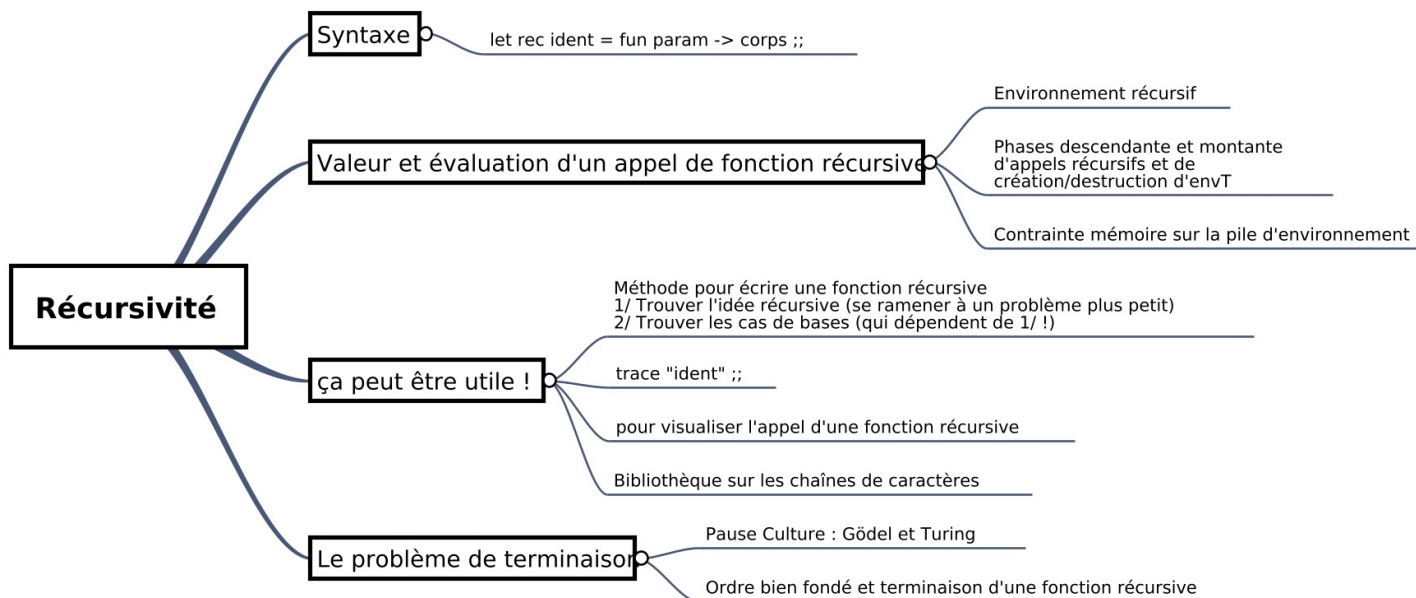


CHAPITRE 5 - RÉCURSIVITÉ SUR LES NOMBRES ET LES CHAÎNES DE CARACTÈRES

En bref



1. Fonctions récursives

1.1. Définitions

Définition 1

Une définition est dite **récursive** si elle fait au moins partiellement référence à elle-même.

Exemple 1

- Tes descendants sont tes enfants et les descendants de tes enfants.
- Plus tôt dans ce cours, nous avons défini récursivement les expressions : $exp = cste$ ou $(exp1 \text{ opérateur } exp2)$.
- En mathématiques, on peut définir récursivement la fonction factorielle par : si $n = 0$, $n! = 1$, sinon $n! = n \star (n - 1)!$.

Définition 2

Une fonction est dite **récursive** si sa définition est récursive.

La syntaxe est

```
let rec ident = fun param -> corps ; ;
```

où le corps de la fonction utilise des appels à la fonction `ident` que l'on est en train de définir.

Exemple 2

```
let rec facto = fun n ->  
    if n = 0 then 1  
    else n*facto(n-1) ; ;
```

Ou mieux, en utilisant un filtrage :

```
let rec facto = fun  
    0 -> 1  
| n -> n*facto(n-1) ; ;
```

Exercice . TP - Exercice 1

Écrire une fonction récursive `somme` : `int -> int` qui calcule la somme des n premiers entiers naturels : $f(n) = 1 + 2 + \dots + n$.

Exercice . TP - Exercice 2

1. Écrire une fonction `puissance` : `int * int -> int` qui calcule la puissance n -ième (n est un entier) d'un réel x .
Méthode : Exprimer x^n en fonction de x^{n-1} . Le cas de base et l'appel récursif seront distingués par filtrage.

1.2. Valeur d'une fonction récursive

L'environnement créé par la définition globale d'une fonction récursive est lui-même récursif. Après définition d'une fonction récursive le nouvel environnement est

```
Env_Rec = [(f, <f>)><Env_def]
```

où la fermeture de la fonction est $\langle f \rangle = \langle \text{param}, \text{cors}, \text{Env_Rec} \rangle$ et non pas Env_def comme pour une fonction classique. Env_Rec s'utilise donc lui-même dans sa propre définition.

Exemple 3

Bien noter la différence entre les deux définitions suivantes.

Si on est dans l'environnement Env_0 :

```
let f = fun
  0 -> 1
|n -> n*(n-1) ; ;
```

crée l'environnement :

```
Env1 = [(f, <filtres, Env0>) >< Env0 ]
```

alors que

```
let rec f = fun
  0 -> 1
| n -> n * f(n-1) ; ;
```

crée l'environnement :

```
Env1 = [(f, <filtres, Env1>) >< Env0 ]
```

1.3. Évaluation d'un appel de fonction récursive

Considérons sur l'exemple précédent l'appel $f(3)$; ; et regardons comment évoluent les environnements :

Exemple 4

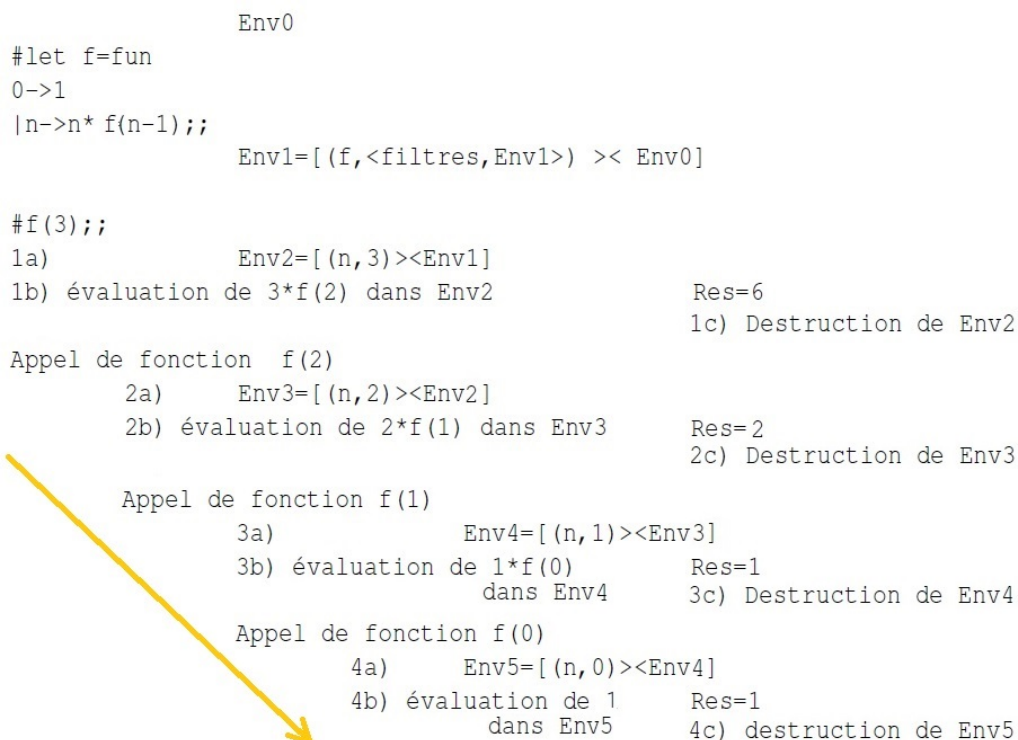
```
Env0
#let f=fun
0->1
|n->n*f(n-1);;
Env1=[(f,<filtres,Env1>) >< Env0]

#f(3);;
1a) Env2=[(n,3)><Env1]
1b) évaluation de 3*f(2) dans Env2
1c) Destruction de Env2
Res=6

Appel de fonction f(2)
2a) Env3=[(n,2)><Env2]
2b) évaluation de 2*f(1) dans Env3
2c) Destruction de Env3
Res=2

Appel de fonction f(1)
3a) Env4=[(n,1)><Env3]
3b) évaluation de 1*f(0) dans Env4
3c) Destruction de Env4
Res=1

Appel de fonction f(0)
4a) Env5=[(n,0)><Env4]
4b) évaluation de 1 dans Env5
4c) destruction de Env5
Res=1
```



Ce que l'on résume en ne notant que les appels et les résultats par :

```
f(3)                6
  |-> 3*f(2)         6
    |-> 2*f(1)        2
      |-> 1*f(0)       1
        |-> 1
```

Nous distinguons donc :

- Une *phase descendante* où les appels de fonction s'enchaînent récursivement et où de nouveaux environnements sont créés
- Une *phase montante* où les résultats se propagent et où les environnements sont détruits.

1.4. Trace

L'interface CAML permet de visualiser ces deux phases lors d'un appel de fonction récursive ; en effet, la commande

```
trace "ident" ; ;
```

permet de "tracer" la fonction récursive de nom `ident`, c'est-à-dire qu'à la suite de cette commande, tous les appels à la fonction `ident` seront détaillés en explicitant tous les appels récursifs de la phase descendante et tous les résultats intermédiaires de la phase ascendante.

Pour y mettre fin, on utilise la commande

```
untrace "ident" ; ;
```

Exemple 5

```
#trace "facto" ; ;
La fonction facto est dorénavant tracée.
- : unit = ()
#facto(4) ; ;
facto <-- 4
facto <-- 3
facto <-- 2
facto <-- 1
facto <-- 0
facto --> 1
facto --> 1
facto --> 2
facto --> 6
facto --> 24
- : int = 24
#untrace "facto" ; ;
La fonction facto n'est plus tracée.
```

Exercice . TD - Exercice A

Soit la fonction Caml suivante :

```
let rec f = fun n ->
  if n = 0 then 0
  else 2*n - 1 + f(n-1) ; ;
```

1. Détailler l'évolution des environnements au cours du calcul de $f(4)$.
2. Que calcule la fonction f ? Démontrer-le par récurrence.

1.5. Mémoire

L'utilisation des fonctions récursives nécessite une importante place en mémoire puisqu'il faut gérer une "pile" d'environnements. Dans notre exemple ci-dessus, 5 environnements doivent être empilés.

La mémoire allouée au système Caml limitera le nombre d'environnements qui peuvent être empilés, et donc le nombre maximum d'appels récursifs qui pourront être effectués avant de bloquer le système par manque de place.

Ainsi se pose le très important problème de la terminaison des fonctions récursives.

2. Le problème de terminaison

Nous avons vu qu'un appel à $f(3)$ engendre des appels récursifs à $f(2), f(1), f(0)$ pour lesquels, à chaque fois, il est nécessaire de créer un environnement temporaire qui s'ajoute à la pile.

La place disponible en mémoire étant toujours finie (même si elle peut être grande), il est nécessaire pour que l'appel d'une fonction récursive termine, que l'appel $f(n)$ n'engendre qu'un nombre fini d'appels récursifs à f .

Exemple 6

Avec l'exemple précédent,

```
f(-1)
  (-1)*f(-2)
    (-2)*f(-3)
      etc .
```

On finit par avoir le message d'erreur : OUT of Memory.

2.1. Appel récursif qui termine

Définition 3

Étant donné une fonction récursive f et un argument x , nous disons que l'appel $f(x)$ termine s'il n'engendre récursivement qu'un nombre fini d'appels à f .

Remarque : En pratique, nous serons limité par la "terminaison sur nos machines" dépendant de la place allouée en mémoire au système Caml.

2.2. Pause culture

Le problème de la terminaison des fonctions récursives est un problème passionnant mais difficile. Gödel et Turing ont montré qu'il n'existe pas d'algorithme général permettant de déterminer si une fonction termine ou pas.

En effet, si cet algorithme existait, on pourrait le programmer en Caml :

```
let rec gödel = fun
  n -> if termine(gödel) then gödel(n)
        else 0 ; ;
```

Si cette fonction termine, elle ne termine pas et réciproquement. Elle ne peut donc pas exister !

2.3. Preuve de terminaison

Un peu de maths

Définition 4

Un ensemble E muni d'une relation d'ordre totale ou partielle est dit **bien ordonné** si et seulement si il n'existe pas de suite d'éléments de E strictement décroissante ou de manière équivalente, si et seulement si toute partie non vide de E admet un élément minimal.

Exemple 7

- (\mathbb{N}, \leq) est bien ordonné d'élément minimal 0
- (\mathbb{Z}, \leq) n'est pas bien ordonné
- $\mathbb{N} \times \mathbb{N}$, muni de l'ordre lexicographique :

$$(x, y) \leq (a, b) \iff (x < a) \text{ ou } (x = a \text{ et } y \leq b)$$

est bien ordonné d'élément minimal $(0, 0)$.

- Cet ordre lexicographique se généralise facilement à \mathbb{N}^p , en faisant un ensemble bien ordonné.

Théorème 1

Soit f une fonction récursive, A l'ensemble de ses arguments que l'on suppose bien ordonné. Soit B l'ensemble des éléments minimaux de A appelés **cas de base**.

Avec ces notations, si

- $\forall b \in B, f(b)$ termine (ce que l'on appellera les cas de base)
- $\forall x \in A$, l'appel à $f(x)$ n'engendre que des appels récursifs $f(y)$ sur des éléments $y \in A$ tels que $y < x$.

Alors $f(x)$ termine $\forall x \in A$.

Exercice . TD - Exercice B

Montrer la terminaison de la fonction factorielle et de la fonction somme (exercice TP 1 plus haut).

2.4. Méthode d'écriture d'une fonction récursive

Important

Pour écrire une fonction récursive, qui termine, il faut donc au préalable :

- Réfléchir à l'**idée récursive** : comment ramener le problème à un problème *plus petit* ?
- Déterminer les cas de base et trouver un moyen de les traiter simplement.

Attention ! La tentation est souvent grande de commencer par déterminer les cas de base, c'est une erreur ! En effet, les cas de base dépendent du choix de l'idée récursive, c'est donc bien dans cet ordre qu'il faut procéder.

Exercice . TP - Exercice 3

Écrire les fonctions récursives suivantes :

1. `repet(c,n) : int * int -> int` qui construit le nombre obtenu en répétant n fois le chiffre c .

```
repet(5,4) ; ;  
- : int = 5555
```

2. `unChiffre(n,c) : int * int -> bool`, qui renvoie `true` si et seulement si le nombre n n'est constitué que du chiffre c .

```
unChiffre(555755,5) ; ;  
- : bool = false  
unChiffre(55555,5) ; ;  
- : bool = true
```

3. `pgd(n,d)` qui retourne le plus grand nombre entier inférieur ou égal à d et divisant n .
On rappelle que 1 divise tous les entiers...

```
pgd(18,12) ; ;  
- : int = 9  
pgd <-- 7, 6  
- : int = 1
```

4. `nbPairChif(n)` qui renvoie `true` si et seulement si n contient un nombre pair de chiffres.

```
#nbPairChif(456789) ; ;  
- : bool = true  
#nbPairChif(4567899) ; ;  
- : bool = false
```

Exercice . TD - Exercice C

Montrer la terminaison des fonctions `repet` et `unChiffre` de l'exercice précédent.

Exercice . TP - Exercice 4

Les égyptiens du deuxième millénaire avant J.C. utilisaient une numération additionnelle de base 10 (ce que vous connaissez de plus proche est la numération romaine...). Avec ce type de numération, il est très simple d'ajouter deux nombres, donc de faire des produits par 2 et de faire des divisions entières par deux. Pour les autres produits, c'est moins simple...

Les scribes égyptiens ont mis au point une technique de multiplication récursive n'utilisant que des sommes et des partages en 2. Elle peut se décrire avec le vocabulaire d'aujourd'hui de la manière suivante :

- Si n est pair, alors $n \times p = (n/2) \times (p + p)$
- Si n est impair, alors $n \times p = (n - 1) \times p + p$.

1. Écrire une fonction `mult_egypt(n,p) : int * int -> int` qui calcule $n \times p$ en exploitant cette idée récursive.
2. Décrivez à la main la suite des appels (récursifs) engendrés par `mult_egypt(59,17)`.
3. Vérifiez votre prédiction en traçant l'exécution de la fonction `mult_egypt`.

Exercice . TP - Exercice 5

On souhaite écrire en Caml des fonctions calculant la somme des chiffres d'un nombre entier représenté en base 10.

1. Définir une fonction récursive `s_chif : int -> int` qui calcule la somme des chiffres d'un entier naturel.

```
s_chif (302091) ; ;  
- : int = 15
```

2. Écrire une fonction `som_chif : int -> int` qui poursuit le calcul de la somme des chiffres d'un nombre jusqu'à se ramener à un seul chiffre.

```
som_chif(18925) ; ;  
- : int = 7
```


3. Récursivité et chaînes de caractères

Les chaînes de caractères constituent un bon champ d'applications des fonctions récursives.

3.1. Terminaison

Pour montrer la terminaison d'une fonction récursive, nous utiliserons cette variante du théorème 1 :

Théorème 2

Soit f une fonction récursive, A l'ensemble de ses arguments, (E, \leq) un ensemble bien ordonné et une fonction $\phi : A \rightarrow E$, qu'on appelle un paramétrage de la fonction.

Soit M l'ensemble des éléments minimaux de E et B la préimage par ϕ de M . Avec ces notations, si

- $\forall b \in B, f(b)$ termine (ce que l'on appellera les cas de base)
- $\forall x \in A$, l'appel à $f(x)$ n'engendre que des appels récursifs $f(y)$ sur des éléments $y \in A$ tels que $\phi(y) < \phi(x)$.

Alors $f(x)$ termine $\forall x \in A$.

Le paramétrage dans le cas de chaînes de caractères sera généralement le nombre de caractères de la chaîne. Le cas de base est alors constitué de la chaîne vide et on cherche à ramener le problème au cas d'une chaîne plus courte.

3.2. Bibliothèque sur les chaînes

Le fichier *chaines.ml* vous fournit quelques primitives de traitement des chaînes de caractères. Nous utiliserons principalement les fonctions suivantes :

1. *tetec s* : donne l'initiale d'un mot sous forme de caractère
2. *tetes s* : donne l'initiale d'un mot sous forme de chaîne (pour pouvoir concaténer)
3. *reste s* : donne le mot privé de son initiale

Exercice . TP - Exercice 6

Écrire les fonctions suivantes :

1. *majuscule* : $\text{char} \rightarrow \text{bool}$, *minuscule* : $\text{char} \rightarrow \text{bool}$ et *lettre* : $\text{char} \rightarrow \text{bool}$, des fonctions booléennes qui indiquent la propriété éponyme à un caractère donné en paramètre. Par exemple, *majuscule c* doit rendre vrai si et seulement si *c* est une lettre majuscule.

Rappel : on a une relation d'ordre sur les caractères telle que $'A' < 'Z' < 'a' < 'z'$.

2. *appartient* : $\text{char} * \text{string} \rightarrow \text{bool}$: fonction récursive booléenne déterminant si un caractère appartient à une chaîne
3. Écrire une fonction booléenne *debut* : $\text{string} * \text{string} \rightarrow \text{bool}$ qui indique si une chaîne est le début d'une autre.
4. Écrire une fonction booléenne *incluse* : $\text{string} * \text{string} \rightarrow \text{bool}$ déterminant si une chaîne est incluse dans une autre.

Utilisez la fonction *debut*.

5. Écrire une fonction `frequence` : `char * string -> int` donnant le nombre d'occurrences d'un caractère donné dans une chaîne.
6. Écrire une fonction `elimine` : `char * string -> string` qui élimine toutes les occurrences d'un caractère donné dans une chaîne donnée.
7. Écrire une fonction `renverse` : `string -> string` qui renverse une chaîne.
8. En déduire une fonction `palindrome` : `string -> bool` qui détermine si une chaîne constitue un palindrome.

Exercice . TD - Exercice D

Montrer la terminaison de deux fonctions récursives sur les chaînes de caractères, à choisir dans l'exercice précédent.

Exercice . TP - Exercice 7

On souhaite écrire des fonctions permettant l'évaluation de nombres écrits en chiffres romains.

- On se restreindra aux nombres s'écrivant à l'aide des seuls chiffres romains I, V et X.
- On suppose dans tout l'exercice que les nombres fournis par l'utilisateur ont des syntaxes correctes, les fonctions à écrire n'auront donc pas à le vérifier.
- On suppose enfin que l'on dispose des fonctions `tetec`, `tetes` et `reste` définies sur les chaînes des caractères.

1. On suppose dans un premier temps que l'on n'utilise pas les écritures IV pour 4 ni IX pour 9. (Le système est donc purement additif).

- a) Écrire une fonction `chiffre` : `char -> int`, définie par filtrage, qui à un caractère parmi I, V et X, représentant un chiffre romain associe sa valeur entière.

```
chiffre('V') ; ;          (*- : int = 5 *)
```

- b) Écrire une fonction récursive `rom1` : `string -> int`, qui à une chaîne de caractères représentant un nombre romain associe sa valeur.

```
rom1 "XXVIII" ; ;        (* - : int = 28 *)
```

2. On suppose maintenant que l'on autorise les écritures IV pour 4 et IX pour 9.

- a) Écrire une fonction `valeurI` : `char -> int`, qui à un caractère `c` parmi I, V, X, associe la valeur entière du nombre romain "Ic".

```
valeurI 'X' ; ;  
- : int = 9
```

- b) Écrire une fonction récursive `romain` : `string -> int`, qui à une chaîne de caractères représentant un nombre romain associe sa valeur.

```
romain "XXIV" ; ;        (*- : int = 24*)  
romain "XXXIX" ; ;       (*- : int = 39*)
```