Fonctions récursives

Définition

Une définition est dite récursive (ou auto-référente) si elle fait au moins partiellement référence à elle-même.

Exemple :

- Tes descendants sont tes enfants et les descendants de tes enfants.
- On a utilisé dans ce cours des définitions récursives pour définir une expression :
  exp=cste ou (exp op exp).
- En maths : on définit parfaitement la fonction factorielle par $n! = 1$ si $n = 0$ et $n * (n-1)!$ si $n > 0$.

Fonctions récursives en CAML

Définition

Une fonction est dite récursive si sa définition est récursive.

Syntaxe :

$$\textbf{let rec ident = fun param} -> \textbf{corps}$$

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

Exemple

```
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);;
```

Premiers exercices

1. Ecrire une fonction récursive qui calcule la somme des n premiers entiers naturels : $f(n) = 1 + 2 + ... + n$.
2. Ecrire une fonction permettant de calculer les puissances entières n^p. (puissance naïve).
3. Ecrire une fonction permettant la conversion d'un nombre entier de la base 10 vers la base 2 par divisions euclidiennes successives :

```
#convert(45);;
- : int = 101101
```

Premiers exercices

1. Ecrire une fonction permettant la conversion d'un nombre entier de la base 2 vers la base 10 par le schéma de Horner :

```
#horner(101101);;
- : int = 45
```

2. Ecrire une fonction donnant la somme des chiffres d'un nombre donné.

Valeur d'une fonction récursive

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

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

où la fermeture de la fonction est

`<f>=<param , corps, Env_Rec>`

et non pas Env-def comme pour une fonction classique. Env-Rec s'utilise donc lui-même dans sa propre définition.

Bien noter la différence entre les deux définitions suivantes :
Si on est dans l'environnement Env0 :

```
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]
```

Evaluation d'un appel de fonction récursive

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

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

#f(3);;
1a)                 Env2=[(n,3)><Env1]
1b) évaluation de 3*f(2) dans Env2                    Res=6
1c) Destruction de Env2
Appel de fonction  f(2)
        2a)         Env3=[(n,2)><Env1]
        2b) évaluation de 2*f(1) dans Env3            Res=2
        2c) Destruction de Env3
```

```
Appel de fonction f(1)
        3a)                 Env4=[(n,1)><Env1]
        3b) évaluation de 1*f(0)          Res=1
        3c) Destruction de Env4
        Appel de fonction f(0)
                4a)     Env5=[(n,0)><Env1]
                4b) évaluation de 1       Res=1
                4c) destruction de Env5
```

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
```

On distingue donc :

- une phase descendante ou les appels de fonction s'enchaînent récursivement et où des nouveaux environnements sont créés
- la phase montante où les résultats se propagent et où les environnements sont détruits.

Trace

L'interface CAML permet de visualiser ces deux phases lors d'un appel de fonction récursive :
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

```
#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";;
```

Exercice

1. Analyse d'une fonction récursive Caml : Soit la fonction Caml suivante :

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

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

Mémoire

L'utilisation des fonctions récursives nécéssite une importante place en mémoire puisqu'il faut gérer une "pile" d'environnements :
Dans notre exemple 5 environnement 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.

Le problème de terminaison

On a 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'en nombre fini d'appels récursifs à f.

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

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

Définition

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

Remarque : En pratique on sera limité par la "terminaison sur nos machines" dépendant de la place allouée en mémoire au système CAML .

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

Le problème de terminaison

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 !
Exercice : Prouver par récurrence la terminaison de la fonction récursive factorielle pour tout n positif.

Méthode d'écriture des fonctions récursives

Il existe des conditions suffisantes "simples" de terminaison.
Les fonctions écrites suivant cette méthode sont dites "primitives récursives".
Nous n'écrirons cette année (presque) que des fonctions primitives récursives. Il est bon de savoir qu'il en existe bien d'autres qui terminent.

Un peu de maths

Définition

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

Exemple $(\mathbb{N}, \leq)$ est bien ordonné d'élément minimal 0, mais pas $(\mathbb{Z}, \leq)$.
$\mathbb{N}^2$ 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 (des mots du dictionnaire) se généralise facilement à $\mathbb{N}^p$ en faisant un ensemble bien ordonné.

Regardons quelques exemples avec des ordres partiels. Dans ce cas on n'a plus nécessairement unicité des éléments minimaux : $\mathbb{N}\backslash\{0,1\}$ muni de l'ordre de divisibilité

$$n \leq p \iff n|p$$

est bien ordonné. Les éléments minimaux sont les nombres premiers. Soit E un ensemble fini non vide. L'ensemble P des parties non vides de E muni de l'inclusion $A \leq B \iff A \subseteq B$ est bien ordonné d'éléments minimaux les singletons.

Premier théorème de terminaison

Théorème

Soit f une fonction récursive, A l'ensemble de ses arguments, on suppose ici que A est 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 qu'on appellera les cas de base).*
- $\forall x \in A$, *l'appel à* $f(x)$ *n'engendre que des appels récursifs* $f(y)$ *à des éléments* $y \in A$ *vérifiant* $y < x$,

ALORS $f(x)$ *termine* $\forall x \in A$.

Deuxième théorème de terminaison

Théorème

Soit f une fonction récursive, A l'ensemble de ses arguments, $(E, \leq)$ un ensemble bien ordonné.
$\phi : A \to E$, 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 qu'on appellera les cas de base).*
- $\forall x \in A$, *l'appel à* $f(x)$ *n'engendre (au premier ordre) que des appels récursifs* $f(x')$ *à des éléments* $x' \in A$ *vérifiant* $\phi(x') < \phi(x)$.

ALORS $f(x)$ *termine* $\forall x \in A$.

L'intérêt du paramétrage d'une fonction récursive est donc de ramener dans tous les cas l'ensemble des paramètres à un ensemble bien ordonné (en général à un des N^p.)
Réfléchir posément au paramétrage permet également de bien déterminer les cas de base à traiter élémentairement.
Exemple : Ecrire une fonction qui détermine le plus grand chiffre d'un nombre entier.
On peut paramétrer par $\phi(n) =$ le nombre de chiffres de n. On a alors $A = \mathbb{Z}$ et $B = \phi(A) = \mathbb{N}^*$ bien ordonné d'élément minimal 1. Le cas de base sera donc le cas des nombres à un chiffre.

```
# let max= fun (a,b)-> if a>b then a else b;;
max : 'a * 'a -> 'a = <fun>
# let rec pgc = fun n-> let q= n/10 and d=n mod 10 in
 if q=0 (* nb à un chiffre *) then n
 else max(pgc(q),d);;
pgc : int -> int = <fun>
```

Ce paramétrage par la taille de l'argument sera très souvent utilisé par la suite (longueur des chaînes de caractères, taille des listes...)

Méthode d'écriture d'une fonction récursive

La fonction f et l'ensemble A de ses arguments sont fixés par l'énoncé.

1. Réfléchir à une "idée récursive" qui permette de ramener le problème à un problème "plus petit" au sens d'un "certain paramétrage."
   Cette étape détermine ϕ et E. E doit être bien ordonné sinon on part à la recherche d'une autre idée !
2. Déterminer les ensembles M et B (les cas de base) et trouver un moyen de les traiter élémentairement. Là encore si on n'y parvient pas, il faudra chercher une autre idée de départ.

1. Utiliser l'idée du 1) pour ramener le calcul de $f(x)$ à des $f(y)$ avec $\phi(y) < \phi(x)$.
2. On peut alors réfléchir à l'efficacité de la fonction et choisir de traiter élémentairement certains cas afin de réduire le temps de traitement de la fonction.
   On est alors en mesure d'écrire la fonction récursive.

Récursivité et chaînes de caractères

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

- On utilise presque toujours dans ce cas le paramétrage par 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.

Bibliothèque sur les chaînes

On pourra utiliser les fonctions suivantes de la bibliothèque standard du système Caml Light :

```
string_length s    (longueur d'une chaîne s),
nth_char s n       (nième caractère d'une chaîne s),
sub_string s n m   (extrait d'une chaîne s entre
                     les caractères n et m) et
string_of_char     (conversion caractère -> chaîne)
```

Les caractères d'une chaîne sont comptés à partir de 0.

Tete et reste

On utilisera aussi les functions *tetec*, *tetes* et *reste*, qui pour une chaine non vide fournissent respectivement le premier caractère de la chaîne sous forme de caractère ou de chaîne et la chaîne privée de son premier caractère.

```
let tetec = fun s->
 if s= "" then failwith "Erreur : chaine vide!"
 else nth_char s 0;;

let tetes = fun s->string_of_char (tetec(s));;

let reste = fun s->
if s="" then "Erreur : chaine vide!"
else sub_string s 1 (string_length s -1);;
```

Exercices

A l'aide de ces conseils et des fonctions tete et reste, écrire les fonctions (récursives) :

1. longueur qui calcule la longueur d'une chaîne.
2. miroir qui calcule le symétrique d'une chaîne.
3. palindrome qui détermine si une chaîne constitue un palindrome.
4. appartient qui détermine si un caractère donné appartient à une chaîne.

Solutions

A l'aide de ces conseils et des fonctions tete et reste, écrire les fonctions (récursives) :

1. longueur qui calcule la longueur d'une chaîne.

```
#let rec longueur = fun s->
if s="" then 0 else 1+ longueur(reste(s));;
longueur : string -> int = <fun>
#longueur "bonjour";;
- : int = 7
```

2. miroir qui calcule le symétrique d'une chaîne.

```
#let rec miroir = fun s->
if s="" then "" else miroir(reste s)^tetes(s);;
miroir : string -> string = <fun>
#miroir "bonjour";;
- : string = "ruojnob"
```

1. palindrome qui détermine si une chaîne constitue un palindrome.

```
#let palindrome = fun s-> s=miroir(s);;
palindrome : string -> bool = <fun>
#palindrome("laval");;
- : bool = true
```

2. appartient qui détermine si un caractère donné appartient à une chaîne.

```
#let rec appartient=fun (c,s)->if s="" then false
else if c=tetec(s) then true else appartient(c,reste(s
appartient : char * string -> bool = <fun>
#appartient (`a`,"albi");;
- : bool = true
#appartient (`c`,"albi");;
- : bool = false
```