Cours de programmation fonctionnelle et logique

Cours 4 : récursivité

Table des matières

De l'art d'écrire des programmes qui résolvent des problèmes que l'on ne sait pas résoudre soi-même !

1 Fonctions récursives simples

1.1 Notion de récursivité

Définition : une définition récursive est une définition dans laquelle intervient ce que l'on veut définir. C'est donc *une définition qui se mord la queue.*

```
#let rec z = 1::z;;
```

Risque que la définition soit dénuée de sens :

```
#let rec f x = f x;;
 val f : 'a -> 'b = <fun>
```

Difficultés :

- détection des cas où la récursivité entraîne le non-sens ;
- vérification de la validité de la définition.

1.2 Portée statique et définitions récursives

En mathématiques, comme en `caml`, on ne peut parler que de ce qui a été préalablement défini. Donc, pour pouvoir utiliser un identificateur, il faut qu'il ait été défini.

Portée statique : on peut trouver la définition indépendamment de l'exécution. (*portée dynamique* : la valeur dépend de la façon dont le calcul se déroule.)

Pour connaître la valeur associée à un identificateur `x`, il suffit d'examiner le texte du programme qui précède `x` pour trouver la liaison qui définit `x`.

Exemples

L'expression :

```
#let x = 1 in
   let x = x+1 in
     x+3;;
     - : int = 5
```

est équivalente à :

```
#let y = 1 in
   let x = y+1 in
     x+3;;
     - : int = 5
```

L'expression :

```
#let x = 1 in
   let x = x+1 and y = x + 2 in
     x+y;;
     - : int = 5
```

est équivalente à :

```
#let z = 1 in
   let x = z+1 and y = z + 2 in
     x+y;;
     - : int = 5
```

Nécessité d'une définition préalable

Un nom fait toujours référence à une définition préalable.

`let f = ... f ...`

Le nom `f` en membre droit ne fait pas référence au nom `f` en membre gauche puisque celui-ci n'est pas encore défini (nous sommes à l'intérieur de la définition).

```
#let fact = function
   0 -> 1
 | n -> n*(fact (n-1));;
     Characters 39-43:
Unbound value fact
```

D'où la nécessité du mot-clef `rec` :

```
#let rec fact = function
   0 -> 1
 | n -> n*(fact (n-1));;
     val fact : int -> int = <fun>
```

1.3 Ordre des appels récursifs

Fonction qui énumère les entiers par ordre décroissant, de `n` jusqu'à 1 :
- si `n` vaut 0, ne rien faire ;
- sinon, on imprime `n` puis on énumère les nombres par ordre décroissant de `n` jusqu'à 1.

```
#let rec compte_à_rebours = function
   0 -> ()
 | n -> begin print_int n; print_string " "; compte_à_rebours (n-1) end;;
     val compte_à_rebours : int -> unit = <fun>
```

```
#compte_à_rebours 10;;
 10 9 8 7 6 5 4 3 2 1 - : unit = ()
```

Interversion de l'affichage et de l'appel récursif :

```
#let rec compte = function
   0 -> ()
 | n -> begin compte (n-1); print_int n; print_string " " end;;
     val compte : int -> unit = <fun>
```

```
#compte 10;;
 1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

L'impression a maintenant lieu au retour des appels récursifs.

1.4 Traçage des appels

```
#let successeur x = x+1;;
 val successeur : int -> int = <fun>

##trace successeur;;
 successeur is now traced.
   #successeur 2;;
    successeur <-- 2       Appel de fonction et argument.
   successeur --> 3        Retour de fonction et valeur retournée.
   - : int = 3
```

Retour sur les fonctions d'énumérations

```
##trace compte;;
 compte is now traced.

##trace compte_à_rebours;;
 compte_à_rebours is now traced.

#compte_à_rebours 3;;
 compte_à_rebours <-- 3
3 compte_à_rebours <-- 2
2 compte_à_rebours <-- 1
1 compte_à_rebours <-- 0
compte_à_rebours --> ()
compte_à_rebours --> ()
compte_à_rebours --> ()
compte_à_rebours --> ()
- : unit = ()

#compte 3;;
 compte <-- 3
compte <-- 2
compte <-- 1
compte <-- 0
compte --> ()
1 compte --> ()
2 compte --> ()
3 compte --> ()
- : unit = ()
```

Remarquez quand les impressions ont lieu : avant l'appel récursif pour `compte_à_rebours`, et après pour `compte`.

2 Exemple : les palindromes

Un palindrome est un mot qui se lit bien à l'endroit qu'à l'envers (cf. figure 1).

- La chaîne vide est un palindrome.
- Un mot réduit à un seul caractère est un palindrome.
- Un mot d'au moins deux caractères est un palindrome si :
  - ses premier et dernier caractères sont identiques ;
  - le mot situé entre ces caractères est un palindrome.

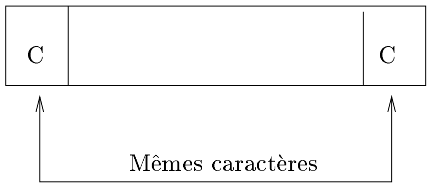


FIG. 1: Structure d'un palindrome

```
#let rec palindrome s =
   match String.length s with
     0 -> true
   | 1 -> true
   | n -> if s.[0]=s.[n-1]
          then palindrome (String.sub s 1 (n-2))
          else false;;
             val palindrome : string -> bool = <fun>
```

Opérateurs booléens : et = &, ou = or.

```
#let rec palindrome s =
   match String.length s with
     0 -> true
   | 1 -> true
   | n -> (s.[0]=s.[n-1]) & palindrome(String.sub s 1 (n-2));;
         val palindrome : string -> bool = <fun>
```

Version sans construction de sous-chaînes

On utilise deux indices, i et j, désignant respectivement le début et la fin de la sous-chaîne traitée. Initialement, i vaut 0 et j la longueur de la chaîne moins 1. À chaque étape on compare les caractères d'indice i et j :

- s'ils sont égaux on continue ;
- s'ils sont différents on arrête ;
- si i≥j, on a atteint ou dépassé le milieu et on arrête.

```
#let rec palin s i j =
   (i >= j) or ((s.[i]=s.[j])&(palin s (i+1) (j-1)));;
   val palin : string -> int -> int -> bool = <fun>
```

```
#let palindrome s = palin s 0 (String.length s - 1);;
 val palindrome : string -> bool = <fun>
```

Et avec une fonction locale

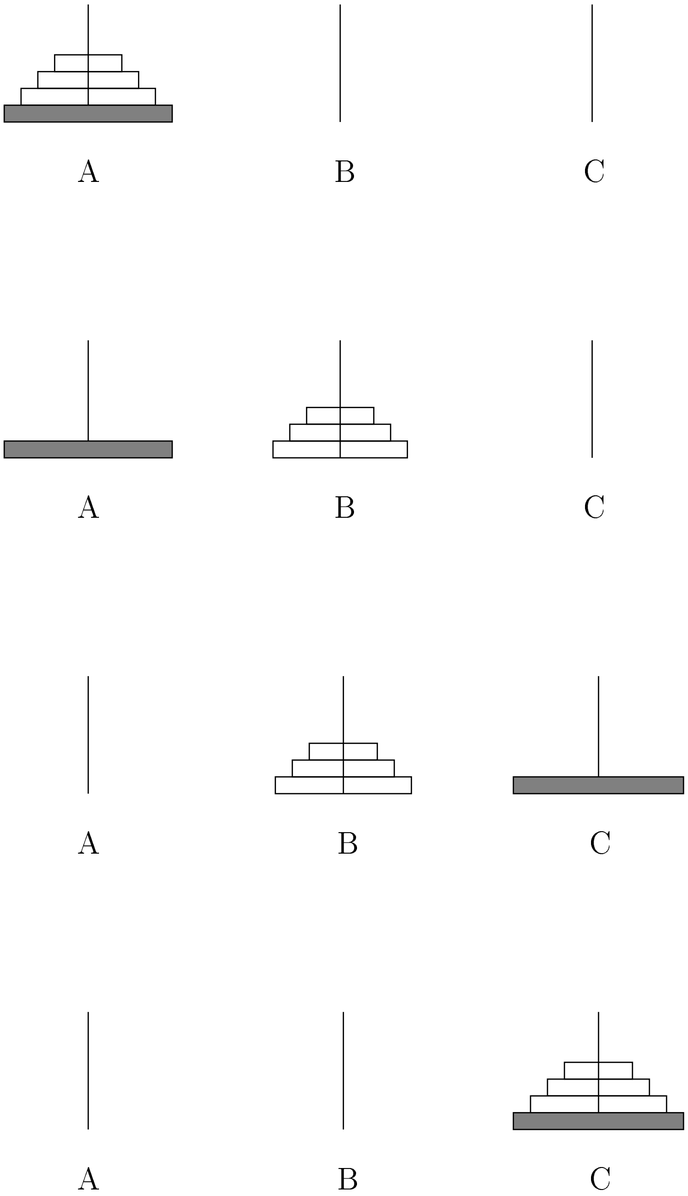
```
#let palindrome s =
   let rec palin i j =
     (i >= j) or ((s.[i]=s.[j])&(palin (i+1) (j-1)))
   in palin 0 (String.length s - 1);;
       val palindrome : string -> bool = <fun>
```

3 Les tours de Hanoï

3.1 Le problème

Le jeu est constitué d'une plaquette de bois où sont plantées trois tiges. Sur ces tiges sont enfilés des disques de diamètres tous différents. Les seules règles du jeu, sont que l'on ne peut déplacer qu'un seul disque à la fois, et qu'il est interdit de poser un disque sur un disque plus petit.

Au début tous les disques sont sur la tige de gauche, et à la fin sur celle de droite.



Principe : nous supposons que nous savons résoudre le problème pour (n-1) disques.
Algorithme : on déplace les (n-1) plus petits disques de `A` vers `B`, puis on déplace le plus gros disque de `A` vers `C`, puis on déplace les (n-1) plus petits disques de `B` vers `C`.
Validité : il n'y a pas de viols possibles puisque le plus gros disque est toujours en « bas » d'une tige et que l'hypothèse nous assure que nous savons déplacer le « bloc » de (n-1) disques en respectant les règles.

3.2 Programme

```
#let mouvement de vers =
   print_string("Déplace un disque de la tige "^de^" à la tige "^vers);
   print_newline();;
     val mouvement : string -> string -> unit = <fun>
```

```
#let rec hanoi départ milieu arrivée = function
    0 -> ()
  | n -> hanoi départ arrivée milieu (n-1);
         mouvement départ arrivée;
         hanoi milieu départ arrivée (n-1);;
          val hanoi : string -> string -> string -> int -> unit = <fun>
#print_newline(); hanoi "A" "B" "C" 3;;

Déplace un disque de la tige A à la tige C
Déplace un disque de la tige A à la tige B
Déplace un disque de la tige C à la tige B
Déplace un disque de la tige A à la tige C
Déplace un disque de la tige B à la tige A
Déplace un disque de la tige B à la tige C
Déplace un disque de la tige A à la tige C
- : unit = ()
```

Il ne faut pas chercher à comprendre *comment* ça marche, mais *pourquoi* ça marche...

3.3 Complexité

```
#let rec compte_hanoi départ milieu arrivée = function
    0 -> 0
  | n -> compte_hanoi départ arrivée milieu (n-1) +
         1 +
         compte_hanoi milieu départ arrivée (n-1);;
          val compte_hanoi : 'a -> 'a -> 'a -> int -> int = <fun>
#let rec compte_hanoi = function
    0 -> 0
  | n -> compte_hanoi (n-1) + 1 + compte_hanoi (n-1);;
      val compte_hanoi : int -> int = <fun>
#let rec compte =
   function   0 -> 0
            | n -> 2*(compte (n-1))+1;;
      val compte : int -> int = <fun>
#let rec compte_rapide n = 2.0 ** n -. 1.0;;
 val compte_rapide : float -> float = <fun>
#compte_rapide 64.0;;
 - : float = 1.84467440737e+19
```

Pour arriver au même résultat, `compte_hanoi` aurait dû faire $2^{64} - 1$ additions, ce qui lui aurait pris plus de 5845 ans à raison de 100 10^6 additions par secondes.

Moralité : il ne suffit pas que les programmes soient corrects...

4 Récursivité et boucles

Fonction qui épelle un mot :

```
#let rec épelle_aux s i =
    if i < String.length s then
      begin
      print_char(s.[i]);
      print_char ' ';
```

```
      épelle_aux s (i+1)
      end;;
              val épelle_aux : string -> int -> unit = <fun>
```

```
#let épelle s = épelle_aux s 0;;
 val épelle : string -> unit = <fun>
```

```
#let épelle s =
   let rec épelle_aux i =
    if i < String.length s then
      begin
      print_char(s.[i]);
      print_char ' ';
      épelle_aux (i+1)
      end
  in épelle_aux 0;;
                 val épelle : string -> unit = <fun>
```

Il s'agit en fait d'une boucle déguisée :

```
#let épelle s =
    for i=0 to String.length s - 1 do
      print_char(s.[i]); print_char ' ';
    done;;
       val épelle : string -> unit = <fun>
```

Mais tous les programmes ne peuvent pas être *facilement* dérécursivés.

5 Validité d'une définition récursive

Il faut s'assurer que la fonction est complètement définie, c'est-à-dire, qu'elle est définie sur tout son domaine d'applications.

Définition par décomposition : il faut vérifier que toutes les formes d'objets du type décomposé sont décrits par la fonction (à vérifier même si la définition n'est pas récursive).

```
#let f = function true -> 1;;
 Characters 8-26:
Warning: this pattern-matching is not exhaustive.
val f : bool -> int = <fun>
```

Le filtrage est exhaustif mais la fonction peut ne pas être définie pour toute valeur.

```
#let rec f p =
    if p >= 0
      then match p with
        0 -> 1
      | n -> (f (n-2)) + 1
      else failwith "probleme";;
            val f : int -> int = <fun>
```

Il faut s'assurer qu'il n'y a pas d'applications indéfinies (fonction définie uniquement si `a=1` ou si `b=1`).

```
#let rec produit (a,b) =
    if a = 1 then b else produit (b,a);;
   val produit : int * int -> int = <fun>
```

Critères d'exhaustivité

- Existence d'un ordre strict tel que les valeurs successives des arguments invoqués par la définition soit strictement décroissantes à partir d'un certain rang.

```
#let rec nombre n =
   if n < 10 then 1 else 1 + nombre (n/10);;
   val nombre : int -> int = <fun>
```

(nombre de chiffres de la représentation décimale d'un entier.)

– La suite des valeurs successives des arguments atteint toujours une valeur pour laquelle la fonction est explicitement définie.

```
#let rec diviseur (a,b) =
  if a < 0
   then failwith "diviseur"
   else
     if a >=b
       then a = b
       else diviseur (a,b-a);;
             val diviseur : int * int -> bool = <fun>
```

La suite des valeurs `b`, `b-a`, `b-2*a`, etc. est strictement décroissante, car `a` est strictement négatif, et on fini toujours pas aboutir à un couple d'arguments `(a,b)` tel que `b-a` est négatif, cas défini explicitement.