Récursivité : définition

- Définition récursive =

  Une fonction f est définie récursivement lorsque sa définition utilise f elle-même.

  Certaines fonctions ne peuvent être définies que récursivement

  Plus souvent, une définition récursive est
  - plus élégante
  - plus simple
  - plus lisible
  - plus efficace

Exemples de récursivité

– définition de factorielle (pour $n \geq 0$)

  fact(n) = fact(n-1) * n

  fact(0) = 1

– suites récurrentes (pour $i \geq 0$)

  $U_i = U_{i-1} * a$ (suite géométrique)

  $U_0 = 1$

Certains objets sont eux-mêmes récursifs

– définition d'une molécule

  Molécule == Atome

  Molécule == Molécule Liaison Molécule

Observations

Les définitions récursives viennent naturellement

Elles sont correctes !

- très différentes de l'équation x = x+1 !
- car on définit un objet structuré (une suite, une fonction) et pas une seule valeur

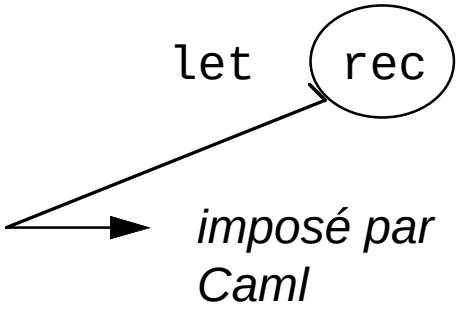
La forme est **toujours** une définition par cas

- un cas général, dans lequel l'objet défini intervient
- un cas particulier, dans lequel on a une valeur immédiate

Il peut bien sûr y avoir plusieurs cas !

Récursivité : Notation

– Il faut préciser qu'on introduit une définition récursive



– Que se passerait-il sinon ?

```
let f = function 0 -> 1
                | x -> (f (x-1))* x;;
```

Récursivité : typage

- Identique à une fonction ordinaire
- Mais, dans un environnement contenant f:σ_1 -> σ_2 en plus (le mot clé `rec` introduit cette contrainte)

```
let rec Ack =
  function
       0 -> (function y -> y + 1)
    |  x -> (function
               0 -> (Ack (x -1) 1)
             |  y -> (Ack (x-1) (Ack x (y-1))));;
```

Une fonction récursive ne peut pas être anonyme !

Pourquoi ça marche ?

- Domaines inductifs

  Certains ensembles ont une structure telle que

  (1) il existe un plus petit élément

  (2) il est possible de construire systématiquement tous les autres éléments

- Exemples

  Naturels: 0, let succ = function n -> n+1;;

  Ensembles: {}, let add_elem = function s ->
  function e -> e::s;;

  Chaînes: “”, let add_char = function ch ->
  function c -> ch^string_of_char c

Définition intuitive

Une fonction est définie récursivement lorsque la valeur de la fonction en un point x est définie par rapport à sa valeur en un point strictement « plus petit »

De ce fait, le calcul se fait de proche en proche, jusqu'à atteindre le plus petit élément pour lequel il faut une valeur immédiate (c'est-à-dire non récursive)

```
let rec f = function
        0 -> 0
      | n -> n + f (n-1);;
```

```
let rec f = function
        0. -> 0.
      | x -> x +. f (x -. 1.);;
```

C'est cassé ?

- Certains domaines ne sont pas inductifs
  - Entiers relatifs : pas de plus petit élément
  - Réels : on ne sait pas construire le réel « suivant »

  (Mais certains domaines sont « presque » inductifs, par exemple, les intervalles de longueur > ε)

- Certaines définitions ne font pas décroître l'argument

```
let rec fact = function 0 -> 1
                      | n -> (fact (n+1)) / (n+1);;
```

Régles de bonne formation

– Le corps d'une fonction récursive doit **toujours** exprimer un choix, soit par une expression conditionnelle, soit par une définition par cas

```
let rec fib = function n ->
      if n=0
          then 1
          else if n = 1
                  then 1
                  else fib (n-1) + fib (n-2);;
```

```
let rec fib = function
          0 -> 1
      |   1 -> 1
      |   n -> fib (n-1) + fib (n-2);;
```

– **Toutes** les applications de la fonction définie doivent l'être sur des valeurs **plus petites** que celle de l'argument

– Au moins un cas **terminal** rend une valeur qui **n'utilise pas** la fonction définie

Valeur d'une fonction récursive

Toujours une fermeture !

Seule nouveauté : l'expression contient le nom de la fonction

Notation

```
let rec f = function a ->
                  function
                      0 -> 1
                    | b -> a * (f a (b-1));;
```

valeur de f = « a->(0->1 | b -> a*(f a (b-1)) »

Application d'une fonction récursive

Même règle que pour l'application que nous avons déjà vue !

Attention : toute la difficulté est de ne pas s'embrouiller dans les valeurs des paramètres !

(f 2 3)	==	2 * (f 2 (3-1))
	==	2 * (f 2 (2))
	==	2* 2 * (f 2 (2-1))
	==	2* 2 * 2 * (f 2 0)
	==	2 * 2 * 2 * 1
	==	8

Exemples : séries finies

sommes de 0 à n des terme d'une suite.

terme : fonction de profil int -> float (pour simplifier)

sigma : fonction de profil n -> (profil de terme) -> float

```
let rec sigma = function
    0 -> ( function terme-> terme 0)
   | n -> (function terme ->
                    terme n +. sigma (n-1) terme);;
```

pour calculer $\pi/8 = 1/(1*3) + 1/(5*7) + 1/(9*11)$?

Exemple : puissance d'une fonction

$f^n(x) = f(f(f ....(f(x)) ...))$ (n applications de f)

```
let rec o_n =
    function f ->
        function n ->
             function x ->
                  if n = 1 then f x
                           else f ( o_n f (n-1) x);;

let inc = function n -> n + 1;;

let inc10 = o_n inc 10;;
inc10 12;;
```

Exemple : puissance d'une fonction

autre solution

```
let o = function f ->
          function g ->
               function x -> f (g x);;

let rec o_n =
  function f ->
        function n ->
           if n = 1 then f
                    else o f (o_n f (n-1));;

let inc14 = o_n inc 14;;
inc14 10;;
```

Transformations

L'écriture naïve peut être inefficace

```
let rec puiss =
  function x ->
    function
        0 -> 1.0
      | n -> if (n mod 2) = 0
               then (puiss x (n / 2))*.(puiss x (n / 2))
               else (puiss x (n / 2))*.(puiss x (n / 2))*.x;;
```

Idée : calculer les intermédiaires une seule fois, les retenir

Notation :

```
let interm = val
in  expr(interm)
```

Transformations

– Ce qui donne

```
let rec puiss =
  function x ->
    function
        0 -> 1.0
      | n -> let p = puiss x (n / 2)
             in  if (n mod 2) = 0
                    then p *.p
                    else p*.p*.x;;
```

Il est possible également de supprimer la réécriture systématique de x

Comment ?

Transformations

```
let puiss = function x ->
        let rec p_int = function
                0 -> 1.
            |  n -> let p = p_int (n / 2)
                    in if n mod 2 = 0
                           then p*.p
                            else x*.p*.p
        in function k -> p_int k;;
```