Les Fonctions en `OCaml`

Les fonctions en `OCaml` sont des objets de première classe

- les fonctions peuvent prendre en paramètre des fonctions
- les fonctions peuvent retourner comme résultat des fonctions
- les fonctions ont un type

On doit pouvoir (d)écrire des "valeurs" fonctions.

Fonction anonyme (fonction sans nom)

La fonction qui à x associe $3 * x$ est clairement définie et se note en mathématique: $x \mapsto 3 * x$

En `OCaml`, définie par une expression utilisant le mot `fun` et la syntaxe: `fun x1 x2 ... -> expression`

- les paramètres de la fonction sont `x1`, `x2`, ...
- `expression`: corps de la fonction; évaluation donne, après passage des paramètres, la valeur de retour de la fonction.

```
# fun x -> 3 * x;;
- : int -> int = <fun>
```

- Il indique que cette expression est une fonction par `<fun>`. En effet, d'habitude, pour une expression, il affiche la valeur de l'expression, mais pas dans le cas d'une fonction.
- Il donne aussi son type: `int -> int` (une fonction qui attend un `int` et retourne un `int`)

Fonction anonyme à plusieurs arguments

```
# fun x y -> 3 * x + 5 * y;;
- : int -> int -> int = <fun>
# fun x y -> float_of_int x +. y;;
- : int -> float -> float = <fun>
#  fun x y -> 2 * x + 7;;
- : int -> 'a -> int = <fun>
```

Le nombre de flèches indique le nombre d'arguments de la fonction, ici: 2 (qui sont `x` et `y`). Les types des arguments sont donnés dans l'ordre, et le dernier type, après la dernière flèche, est le type de retour de la fonction.

Inférence de type des fonctions

```
# fun x y -> 3 * x + 5 * y;;
- : int -> int -> int = <fun>
# fun x y -> float_of_int x +. y;;
- : int -> float -> float = <fun>
#  fun x y -> 2 * x + 7;;
- : int -> 'a -> int = <fun>
# fun x -> x;;
- : 'a -> 'a = <fun>
```

Concernant le typage inféré pour la seconde fonction,

- la fonction `float_of_int` est appliquée au paramètre `x`. `OCaml` peut donc en déduire que `x` est un entier, de type `int`.
- `y` est ajouté au `float` `float_of_int(x)` avec l'opérateur `+.` ce qui permet de déduire que `y` doit lui-même être un `float`.

`x` est un `int`, `y` est un `float` et la valeur de retour est un `float`.

Certains types peuvent être détectés comme étant arbitraires. Dans ce cas, ces types sont notés `'a`, `'b`, `'c`, etc. et la fonction pourra être utilisée pour n'importe quel type de l'argument correspondant.

Inférence de type des fonctions

```
# fun f x -> 2. *. (f (x+1));;
- : (int -> float) -> int -> float = <fun>
```

- l'opérateur `+` est appliqué au paramètre `x`. `OCaml` peut donc en déduire que `x` est un entier, de type `int`. Il en va de même pour l'expression `x+1`.
- `f` est appliquée à `x+1` donc `f` est une fonction à un argument et celui-ci est de type `int`. Le résultat de cette application est une opérande de l'opérateur `*.` donc la valeur de retour de `f` est de type `float`.

`x` est un `int`, `f` est un `int -> float` et la valeur de retour est un `float`.

Application (partielle) de fonctions

```
# fun x -> 3 * x ;;
- : int -> int = <fun>
# (fun x -> 3 * x) 10;;
- : int = 30
```

On aussi réaliser une application partielle avec moins d'arguments qu'attendu; le résultat est alors une fonction qui "attend" les arguments manquants

```
# fun x y -> 3 * x + 5 * y;;
- : int -> int -> int = <fun>
# (fun x y -> 3 * x + 5 * y) 2 10;;
- : int = 56
# (fun x y -> 3 * x + 5 * y) 2;;
- : int -> int = <fun>
```

Cette dernière expression "vaut" `fun y -> 3 * 2 + 5 * y`

Application (partielle) de fonctions et types

Rappel : `f e1 e2 ... en` équivaut à `(((f e1) e2) ... en)`.

Le type `t1 -> t2 -> ... -> tn -> t` de `f` correspond à :

```
t1 -> (t2 -> (... -> (tn -> t)) ...)
```

Le parenthésage se fait de gauche à droite pour l'application de fonctions et droite à gauche pour le type des fonctions.

Avec `e1` de type `t1`, `e2` de type `t2`, ..., `en` de type `tn`,

- `(f e1)` est de type `(t2 -> (... -> (tn -> t)) ...)`
- `(f e1) e2)` est de type `(t3 -> (... -> (tn -> t)) ..)`
- ....
- `(((f e1) e2) ... en)` est de type `t`

Fonction nommée

Puisqu'une fonction anonyme est une expression, on peut la nommer avec la requête `let`.

```
  let cube = fun x -> x * x * x

# let cube = fun x -> x * x * x;;
val cube : int -> int = <fun>
# cube 3;;
- : int = 27

# let f = fun x y -> 2 * x + 7;;
val f : int -> 'a -> int = <fun>
# f;;
- : int -> 'a -> int = <fun>
# f 3;;
- : 'a -> int = <fun>
# f 3 4;;
- : int = 13
```

Fonction nommée (sucre syntaxique)

La requête `let f = fun x y ... -> corps` s'écrit de fait en utilisant la syntaxe **plus spécifique**:

```
let f x y ... = corps
```

```
# let cube x = x * x * x;;
val cube : int -> int = <fun>

# let f x y = 2 * x + y;;
val f : int -> int -> int = <fun>
# f 3 4;;
- : int = 10
```

Appel de fonction par valeur

En `OCaml`, les appels de fonction `f e1 e2 ... en` se font toujours par valeur:

- les valeurs des arguments `e1`, `e2` , ..., `en` sont calculées
- on applique `f` à ces valeurs

```
# let f x y = 2*x;;
val f : int -> 'a -> int = <fun>
# f 3 (12/0);;
Exception: Division_by_zero.
```

Problème aussi si l'évaluation du second paramètre ne termine pas.

S'oppose au passage par nom où la valeur du paramètre est calculé chaque fois qu'il est nécessaire et uniquement là.

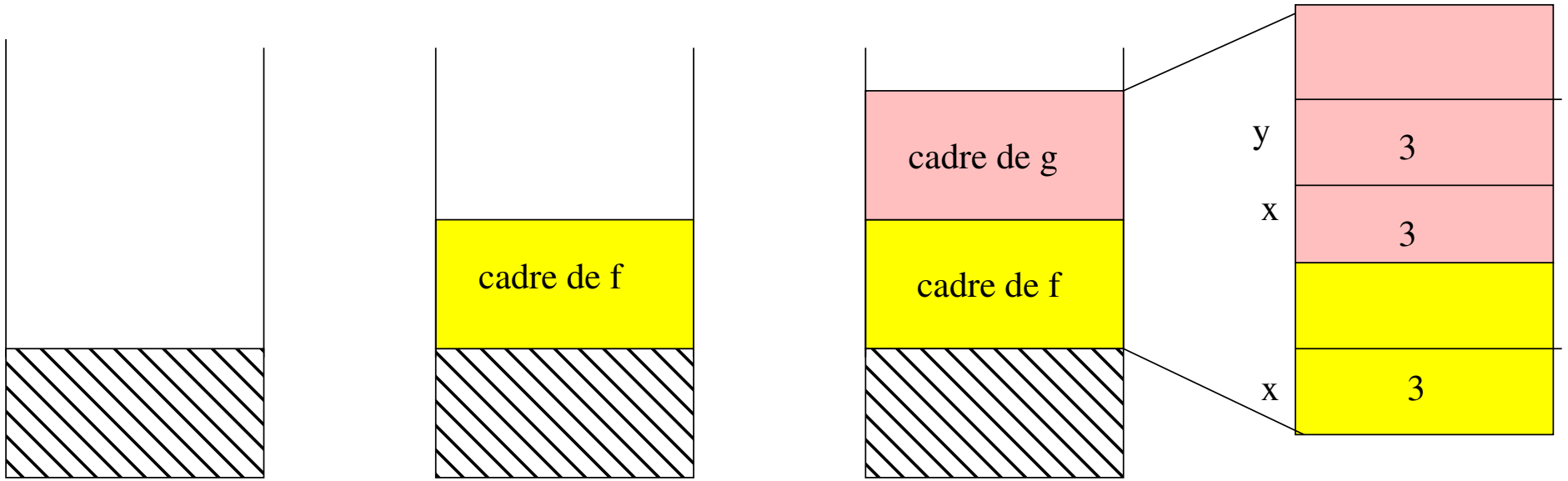
Mécanismes d'appel de fonction

Pour gérer les appels de fonction, le système utilise une *pile*. Lors d'un appel un cadre (frame) est créé et empilé. Il contient en particulier les variables correspondant aux paramètres de la fonction initialisées avec les valeurs résultant de l'évaluation des arguments lors du passage de paramètres. Lorsque la fonction retourne (fin de son évaluation), le cadre est dépilé.

```
# let g x y = 2 * x + y;;
val g : int -> int -> int = <fun>
# let f x = g x x;;
val f : int -> int = <fun>
# f 3;;
- : int = 9
```

Mécanismes d'appel de fonction

```
# let g x y = 2 * x + y;;
val g : int -> int -> int = <fun>
# let f x = g x x;;
val f : int -> int = <fun>
# f 3;;
- : int = 9
```



Pile

Fermetures en OCaml

Une fermeture est une structure qui contient à la fois

- la valeur (le code) d'une fonction
- un environnement qui associe à chaque variable libre de la fonction (les variables utilisées dans la fonction qui ne sont ni des variables locales, ni des paramètres) la valeur qu'elle possédait au moment de la définition de la fonction

```
# let y = 10;;
val y : int = 10
# let f x = x+y;;
val f : int -> int = <fun>
# f 4;;
- : int = 14
# let y = 20;;
val y : int = 20
# f 4;;
- : int = 14
```

Fermetures en OCaml

S'oppose à la notion de portée dynamique

L'application partielle crée une fermeture

```
# let f x y z= (x+y)*z;;
val f : int -> int -> int -> int = <fun>
# let g = f 3 4;;
val g : int -> int = <fun>
```

$$g = (\texttt{fun x y z -> (x+y)*z}, \{(x,3),(y,4)\})$$

function

Lorsqu'une fonction ne possède qu'un argument et pour laquelle la définition se fait par cas implicitement sur cet argument

```
let  f = fun x -> match x with
       0 -> "vaut 0"
     | 1 -> "vaut 1"
     | y  ->  (string_of_int y)^" est différent de 0,1"
```

mais plutôt

```
let  f = function
       0 -> "vaut 0"
     | 1 -> "vaut 1"
     | y ->  (string_of_int y)^" est différent de 0,1"
```

Fonctions récursives

En l'absence de boucle, il faut un mécanisme supportant l'itération de calcul : la récursivité

Une fonction *récursive* est appelée dans sa propre définition.

- pour écrire une fonction récursive, il est donc nécessaire de *nommer* la fonction pour pouvoir l'appeler par son nom dans sa définition.
- En `OCaml`, la construction `let f ... = ...` ne permet de définir que des fonctions **non** récursives.

```
# let fact n =
  if n = 0 then 1 else n * fact (n-1);;
Error: Unbound value fact
```

À noter que un langage disposant d'une conditionnelle et de fonctions récursives a la puissance des machines de Turing, c'est à dire permet de programmer tout ce qui est programmable.

Définition `let rec`

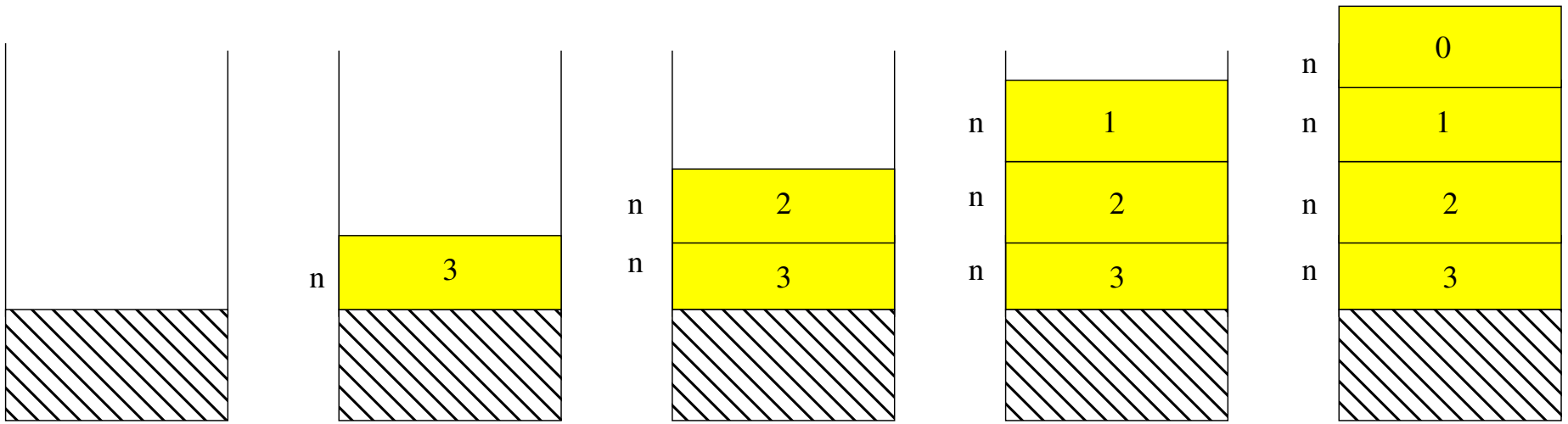
Pour définir une fonction récursive, il faut explicitement le préciser par la requête `let rec`. Par exemple, la fonction factorielle définie sur les entiers naturels s'écrit de façon naïve:

```
let rec fact n =
  if n = 0 then 1
     else n * fact (n-1)
```

Evaluation de la récursivité

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n - 1)

# fact 3;;
- : int = 6
```



Principe de la récursivité

fonction récursive basée sur ordre bien fondé
ordre bien fondé $<$:

- pas de suite infinie strictement décroissante
- nombre fini d'éléments minimaux

Sous ces hypothèses, on peut

- décrire le calcul sur les éléments minimaux (cas de base ou d'arrêt)
- décrire le calcul des éléments non minimaux en fonction d'éléments plus petits (cas récursifs)

Récursivité (suite)

- En particulier, pour une fonction `f` d'un entier naturel (comme factorielle), $<$ est un ordre bien fondé pour les entiers naturels et il existe un unique élément minimal: 0.
- Il suffit d'indiquer comment calculer `f 0` puis d'indiquer comment calculer `f n` pour `n` $>$ `0` en fonction de `f i` pour `i` $<$ `n`.
- il faut que les appels récursifs se fassent sur des arguments plus petits que ceux passés en paramètres et il faut prévoir les cas d'arrêt (pour tous les éléments minimaux qui se calculent sans appel récursif).

Exemples d'ordres bien fondés

- Entiers naturels munis de $<$. On définit la fonction pour 0 puis pour $n > 0$ en utilisant des appels récursifs sur des valeurs $< n$.
- Couples d'entiers naturels munis de l'ordre lexicographique.

$$(x, y) < (x', y') \text{ ssi } \text{ soit } x < x', \text{ soit } x = x' \text{ et } y < y'$$

On définit la fonction pour $(0, 0)$, puis pour $(x, y) > (0, 0)$ avec des appels sur des valeurs $< (x, y)$.

Correction et complexité de fonctions récursives

La définition récursive d'une fonction simplifie :

- les preuves de terminaison : trouver une valeur qui décroit le long des appels
  - dans le cas impératif : trouver un variant de boucle
- les preuves de correction (partielle) : souvent preuve par récurrence/induction sur les objets manipulés
  - dans le cas impératif: trouver un invariant de boucle
- le calcul de la complexité : souvent résolution d'équations récurrentes.
  - dans le cas impératif: estimation du nombre de tours de boucle

Fonctions mutuellement récursives

Une fonction `f1` qui appelle `f2` tandis que `f2` appelle `f1`.

De manière plus générale, il existe un cycle dans le graphe d'appels des fonctions

Utilisation de la définition `let rec ... and ...`

```
let rec
  est_pair n =
          if n = 0 then true
          else est_impair (n - 1)
and
  est_impair n =
          if n = 0 then false
          else est_pair (n - 1)
```