Expression `let in`

Pour éviter la duplication d'expressions dans le code, il est conseillé d'utiliser l'expression `let in` qui permet de mémoriser temporairement la valeur d'une ou plusieurs expressions dans des variables temporaires. Ceci permet d'éviter

- la duplication du code
- l'évaluation multiple d'une expression

On peut donner des valeurs à plusieurs variables en parallèle à l'aide du mot-clé `and`.

```
# let x = 1 and y = 2 in x + y;;
- : int = 3
```

Pour des affectations séquentielles, on utilise le `let in` en cascade.

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

Fonction anonyme (fonction sans nom)

La fonction qui à x associe $3 * x$ est clairement définie et se note en mathématique: $x \mapsto 3 * x$
peut être définie avec une expression utilisant le mot `fun` et la syntaxe: `fun x1 x2 ... -> expression`
paramètres de la fonction: `x1`, `x2`, ...
`expression` : corps de la fonction; évaluation donne, après passage des paramètres, la valeur de retour de la fonction.
la fonction ci-dessus s'écrit

```
fun x -> 3 * x;;
```

Si on entre cette ligne dans l'interpréteur `OCaML`, l'interpréteur répond

```
- : 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.

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

Il donne aussi son type: `int -> int -> int`. 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.
De même,

```
# fun x y -> float_of_int x +. y;;
- : int -> float -> float = <fun>
# (fun x y -> float_of_int x +. y) 4;;
- : float -> float = <fun>
# (fun x y -> float_of_int x +. y) 4 2.5;;
- : float = 6.5
```

```
# fun x -> 3 * x ;;
- : int -> int = <fun>
# (fun x -> 3 * x) 10;;
- : int = 30
# fun x y -> 3 * x + 5 * y;;
- : int -> int -> int = <fun>
# (fun x y -> 3 * x + 5 * y) 2;;
- : int -> int = <fun>
# (fun x y -> 3 * x + 5 * y) 2 10;;
- : int = 56
```

Conversions

`float_of_int, int_of_float, int_of_char,`
`char_of_int,  string_of_int, string_of_bool, ...`

Requêtes

Les requêtes comme les expressions sont tapées dans la REPL.
Elles permettent de faire des opérations non purement fonctionnelles (qui modifient l'état du système).

Requête `let`

- liaison entre une variable et une valeur (donne un nom à une valeur)
- permet de définir des variables globales (indispensable pour enregistrer fonctions et données)
- effet de bord (affecte la valeur à la variable) et retourne la valeur
- ne pas confondre avec l'expression `let ... in`.

Requête `let` : exemples

```
# let x = 3 + 4;;
val x : int = 7
# x;;
- : int 7
# 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
# x;;
- : int = 7
```

Remarque: 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.

Fonction nommée

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

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

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
```

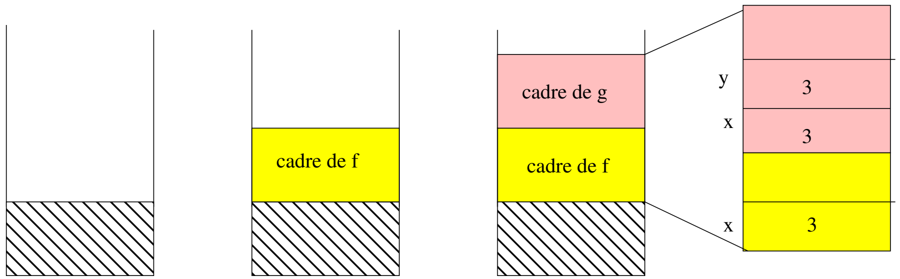
```
# let f x y = 2 * x + y;;
val f : int -> int -> int = <fun>
# f 3 4;;
- : int = 10
```

Appel de fonction

Les appels de fonction se font toujours par valeur.
Pour gérer les appels de fonction, le système utilise une *pile*. Lors d'un appel un cadre (frame) est créé et placé en sommet de pile. Il contient en particulier les variables lexicales correspondant aux paramètres de la fonction qui seront initialisées avec les valeurs résultant de l'évaluation des arguments lors du passage de paramètres. Lorsque la fonction retourne, 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
```

```
# 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
```



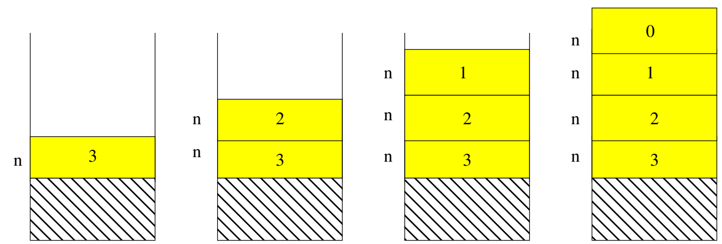
Pile

Récursivité

Ce mécanisme permet la récursivité [3]

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

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

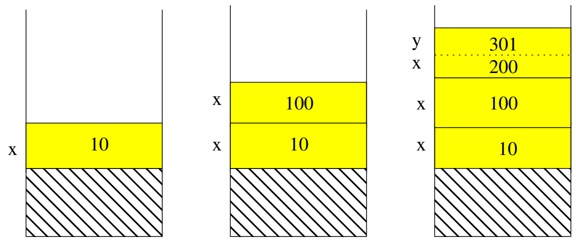


[3] Cobol, anciennes versions de Fortran: pas de récursivité: programmer la pile soi-même

Évaluation d'un `let ... in`

Le mécanisme de pile est aussi utilisé pour l'évaluation d'une expression `let x = ... in`
variable lexicale `x` créée sur la pile le temps de l'exécution du `let`

```
# let x = 10 in
      let x = x * x in
      let x = 2 * x
      and y = 3 * x + 1 in
      (x, y);;
- : int * int = (200, 301)
```



Fonctions récursives

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.
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.

Requête `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)
```

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 $< (x, y)$.

Récursivité et complexité

Pour évaluer l'efficacité d'une fonction, on évalue le nombre d'opérations élémentaires

- sur une entrée de taille n et dans le pire des cas

Dans le cas d'une fonction récursive, on est souvent amené à résoudre des équations récurrentes.

Récursivité et complexité

Exemple de calcul de 2^x pour $x > 0$.

```
let rec pow2 x =
  if x = 0 then 1
  else 2 * pow2 (x - 1)
```

Comptons le nombre $M(x)$ de multiplications effectuées:

$$M(0) = 0$$
$$M(x) = M(x-1) + 1$$

On en déduit:

$$M(x) = x$$

et donc que cette fonction est en $O(x)$.
On peut arriver à $O(log(x))$ en utilisant la méthode à la grecque.

Exponentiation à la grecque

On peut faire mieux en utilisant la méthode à la grecque.

```
let rec pow2_log x =
if x = 0 then 1
else let y = pow2_log (x / 2) in
     let p = y * y in
     if x mod 2 = 0 then p
     else p * 2
```

Comptons le nombre $G(x)$ de multiplications effectuées:

$$\begin{array}{ll} G(0) & = 0 \\ G(x) & <= G(x/2) + 2 \end{array}$$

On en déduit qu'il y a $O(log(x))$ multiplications.