Informatique *Cours*
Programmation fonctionnelle avec OCaml - Récursivité Ipesup

Objectifs

- Comprendre la notion de récursivité et son rôle central en Ocaml.
- Distinguer entre récursion simple, récursion mutuelle et récursion terminale.
- Comprendre l'utilisation de la pile et du tas pour les fonctions récursives.
- Définir et manipuler des types récursifs (listes d'entiers, listes polymorphes).

La récursivité est une technique omniprésente en programmation fonctionnelle. On l'utilise pour écrire des fonctions, mais également pour définir des types de données.

1 Fonctions récursives

Une fonction *récursive* est une fonction qui fait appel à elle-même dans sa propre définition.

Exemple

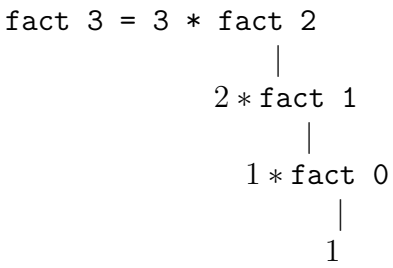
La fonction factorielle $n!$ est définie, pour tout entier naturel n, par les deux équations suivantes :

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1)! \quad \text{si } n > 0 \end{cases}$$

Pour écrire la fonction factorielle en OCaml, une solution est d'écrire une fonction `fact` avec un argument `n`, comme ceci :

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

On peut représenter l'évaluation de l'appel à `fact 3` de la manière suivante :



Cette manière de représenter l'exécution d'un programme en indiquant les différents appels effectués est appelée un *arbre d'appels*.

Pour se convaincre que la définition de la fonction factorielle est correcte, il suffit de montrer par récurrence sur $\mathbb{N}$ que l'équation `fact n = n!` est vraie.

Le cas de base correspond à `fact 0`. En substituant 0 dans la définition, on obtient

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

On a donc bien : `fact 0 = 1 = 0!`

Dans le cas récursif, on suppose que l'équation `fact (n - 1) = (n - 1)!` est vraie pour tout n strictement positif. Dans ce cas général :

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

Comme n est strictement positif la conditionnelle vaut donc `n * fact (n - 1)`. En appliquant notre hypothèse de récurrence, on obtient les égalités suivantes :

$$\texttt{fact n = n * fact (n - 1)} = n * (n-1)! = n!$$

Attention

Cette démonstration est correcte parce que non appele la fonction `fact` dans le corps de la fonction elle même. Mais la portée de l'identificateur `fact` n'inclut pas l'expression à droite de la déclaration. Mais le mot-clé `rec` permet que la variable introduite par le `let` soit visible *pendant* sa définition (et non plus uniquement après). Si le mot-clé `rec` est absent on obtient l'erreur suivante :

Exemple

```
$ ocamlopt factorielle.ml -o factorielle
File "factorielle.ml", line 2, characters 27-31:
2 | if n = 0 then 1 else n * fact (n - 1)
                           ^^^^
Error: Unbound value fact
Hint: If this is a recursive definition,
you should add the 'rec' keyword on line 1
```

1.1 Récursion mutuelle

On peut définitire une fonction « en même temps » que une autre fonction.

Exemple

On pourrait définir les entiers naturels *pairs* et *impairs* de la manière suivante :

> « Un entier $n \in \mathbb{N}$ est *pair* si $n = 0$ ou si $n - 1$ est *impair*. »
>
> « Un entier $n \in \mathbb{N}$ est *impair* si $n \neq 0$ et si $n - 1$ est *pair*. »

Les deux définitions sont *mutuellement récursives*.

```
let rec even n = (n = 0) || odd (n - 1)
let rec odd n = (n <> 0) && even (n - 1)
```

Malheureusement, la règle de portée du `let` ne permet pas de faire référence à une fonction définie au-dessous. Il faut pour cela utiliser la forme particulière des déclarations mutuellement récursives suivante :

```
let rec <id_1> = <expr_1>
and <id_2> = <expr_2>
...
and <id_k> = <expr_k>
```

Exemple

Pour les fonctions `even` et `odd` précédentes :

```
let rec even n = (n = 0) || odd (n - 1)
and odd n = (n <> 0) && even (n - 1)
```

1.2 Fonctions vs. boucles

Les boucles sont des constructions inutiles en programmation fonctionnelle. Alors, comment écrire un calcul aussi simple que celui ci-dessous (en Python) ?

```
x = 19
while x <= 42:
  x = x + 3
```

On peut considerer une fonction qui prend en argument la valeur de la mémoire *avant* la boucle, et qui renvoie la valeur de la mémoire *après* l'exécution de la boucle.
On imite alors simplement le code Python ci-dessus avec la fonction `loop` suivante :

```
let rec loop x =
  if x <= 42 then loop (x + 3) else x
```

On appellera `loop 19` pour obtenir le même résultat que la boucle ci-dessus.

Fonctions locales

On utilise habituellement des définitions locales de fonctions pour des "boucles".
Par exemple, le programme suivant en Python :

```
x = 0
for i in range(1,4):
  for j in range(1,4):
    x = x * i + j
```

peut s'écrire en utilisant deux fonctions récursives locales `for1` et `for2` :

```
let x =
  let rec for2 (x, i, j) =
        if j >= 4 then x else for2 (x * i + j, i, j + 1)
  in let rec for1 (x, i) =
           if i >= 4 then x else for1 (for2 (x, i, 1), i + 1)
  in for1 (0, 1)
```

2 Récursion terminale

Considérons le programme suivant qui calcule la somme des nombres de 0 à 1 000 000.

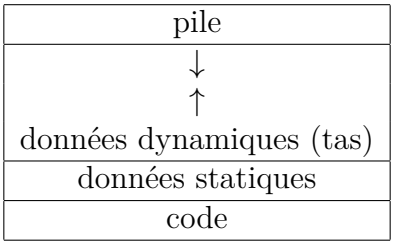
```
let rec somme n =
  if n = 0 then 0 else n + somme (n - 1)

let v = somme 1_000_000
```

Après avoir compilé ce programme, on obtient le message suivant en l'exécutant :

```
$ ocamlopt -o test test.ml
$ ./test
Fatal error: exception Stack_overflow
```

Cette erreur indique un débordement de pile (*Stack_overflow*). Pour bien comprendre ce qui s'est passé, il faut expliquer comment OCaml utilise la *pile* pour exécuter des fonctions récursives.

On rappelle dans le schéma suivante l'organisation de la mémoire :



Lors d'un programme OCaml, la mémoire est organisée en quatre zones principales :

- Code : contient la séquence d'instructions machine à exécuter.
- Données statiques : stocke les constantes du programme.
- Tas (données dynamiques) : sert à créer les structures de données (*n-uplets*, enregistrements, etc.) pendant l'exécution.
- Pile : gère la mémoire utilisée par les appels de fonctions et les variables locales.

Entre les deux flèches, la zone mémoire est *non occupée*. Au démarrage, la pile et le tas sont presque vides :

- la pile commence à une adresse haute et croît vers les adresses basses ;
- le tas commence à une adresse basse et croît vers les adresses hautes.

La gestion de ces zones est assurée par le système d'exploitation : il alloue de la mémoire au tas ou à la pile selon les besoins du programme et il garantit que ces zones ne se chevauchent jamais — aucune donnée du tas ne peut écraser la pile, et inversement.

2.1 Gestion de la pile

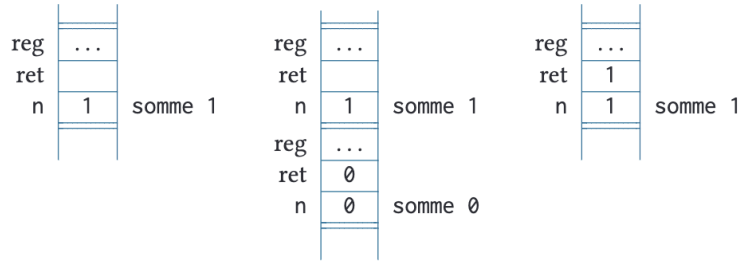
La pile gère la mémoire utilisée par les fonctions et libère *automatiquement* cet espace dès qu'elles se terminent.

Chaque appel de fonction crée un *contexte d'exécution* contenant :

- les arguments de l'appel ;

— les variables locales ;
— la valeur de retour ;
— et certains registres du microprocesseur.

La figure suivante illustre cette gestion dans la fonction `somme` :



Gestion du segment de pile pour l'appel somme 1.

— Après l'appel `somme 1`, la pile contient les registres (`reg`), la case de retour (`ret`) et l'argument `n = 1`.
— Lors de l'appel récursif `somme 0`, un *nouvel environnement* est ajouté avec `n = 0` et une case `ret`. La fonction renvoie 0.
— À son retour, l'environnement de `somme 0` est retiré, la valeur 0 est récupérée et la somme `1 + somme 0 = 1` est stockée dans la case de retour de `somme 1`.

Ainsi, la pile évolue dynamiquement pour stocker et libérer les contextes d'appels récursifs au fur et à mesure de l'exécution.

Le calcul récursif de `somme n` provoque une suite d'appels « en cascade » à la fonction `somme`, chacun empilant un nouvel environnement d'exécution.

$$\texttt{somme n} = \texttt{n + somme (n - 1)}$$
$$\Big|$$
$$(n-1) + \texttt{somme (n - 2)}$$
$$\Big|$$
$$\cdots$$
$$\Big|$$
$$1 + \texttt{somme 0}$$
$$\Big|$$
$$0$$

Ainsi, pour un appel initial `somme n`, la pile contient $n+1$ environnements juste après l'appel final à `somme 0`. Chaque environnement stocke la valeur de l'argument `n`, la case de retour (`ret`) et les registres nécessaires. Cette organisation consomme rapidement beaucoup de mémoire.

La taille de la pile est souvent limitée par le système d'exploitation : sous Linux, la commande `ulimit -s` affiche la taille maximale de la pile (ex. 8 Ko par défaut). On peut augmenter cette limite avec :

```
$ ulimit -s unlimited
```

Même avec une pile agrandie, allouer un environnement pour chaque appel reste lent et coûteux en mémoire. La solution est d'utiliser une forme optimisée appelée récursion terminale, où l'appel récursif constitue la dernière opération effectuée par la fonction.

2.2 Function récursives terminales

Une fonction est dite *récursive terminale* si elle renvoie directement le résultat de l'appel récursif, sans calcul supplémentaire. Par exemple :

```
let f1 x = g x
let f2 x = if ... then g x else ...
```

Dans ces cas, l'appel à `g x` est en position terminale, permettant au compilateur d'optimiser la récursion sans empilement supplémentaire.

Exemple

Exemples de fonctions terminales

```
let f3 x = if ... then ... else g x
let f4 x =
  let y = ... in
  g y
let f5 x =
  match x with
  | ... -> ...
  | ... -> g x
  | ... -> ...
```

Comme on peut le constater, chaque appel `g x` est bien en position d'être la dernière opération effectuée par ces fonctions.
À l'inverse, les appels à `g` dans les deux fonctions suivantes ne sont pas terminaux car il reste à faire l'opération `x + ...` dans `f6` et `y + 1` dans `f7`.

```
let f6 x = x + g x
let f7 x = let y = g x in
           y + 1
```

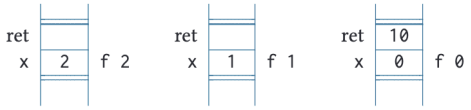
Une fonction récursive est dite récursive terminale lorsque tous ses appels récursifs apparaissent en *position finale*, c'est-à-dire qu'ils constituent la dernière opération de la fonction.

L'intérêt principal est qu'il n'est pas nécessaire d'allouer un nouvel environnement d'exécution à chaque appel : la fonction *réutilise le même espace mémoire*, car le résultat de l'appel récursif terminal est directement la valeur renvoyée par la fonction.

Exemple

```
let rec f x =
  if x = 0 then 10 else f (x - 1)
```

Pour l'appel `f 2`, la pile contient une case pour l'argument `x = 2` et une pour la valeur de retour (`ret`). Lors de l'appel récursif `f 1`, aucun nouvel espace n'est ajouté : le même contexte est *réutilisé*, car la valeur précédente de `x` n'est plus utile. Le même mécanisme s'applique jusqu'à `f 0`, qui place la valeur finale `10` dans la case `ret`.



Ainsi, une fonction récursive terminale peut s'évaluer sans empiler de contextes d'évaluation, ce qui rend son exécution aussi rapide qu'un programme implémenté avec une boucle [1].

Version terminale de la somme

Pour éviter le *débordement de pile*, la fonction `somme` doit être écrite en forme récursive terminale.
On part d'une version Python utilisant une boucle `for` et un accumulateur `acc` :

```
def somme(n):
  acc = 0
  for i in range(n, 0, -1):
    acc = i + acc
  return acc
```

La variable `acc`, initialisée à 0, accumule la somme des entiers de n à 0 (ordre décroissant [a]). Pour reproduire ce calcul en OCaml sans boucle, on définit une fonction récursive prenant en argument les variables modifiées par la boucle, ici `i` et `acc`.
La condition d'arrêt correspond à la fin de la boucle : si `i = 0`, on renvoie `acc`, sinon on poursuit le calcul.

```
let rec somme (i, acc) =
  if i = 0 then acc else somme (i - 1, i + acc)
```

L'appel récursif est terminal car il ne fait aucun calcul après le retour : il transmet simplement les nouvelles valeurs de `i` et `acc`.
Pour calculer la somme de 0 à 1 000 000 :

```
let v = somme (1_000_000, 0)
let () = print_int v
```

Le compilateur OCaml détecte automatiquement les fonctions récursives terminales et optimise leur exécution en supprimant l'empilement des contextes — ce qui les rend aussi efficaces qu'une boucle.

a. Le parcours décroissant correspond mieux à la logique récursive.

3 Gestion du tas en OCaml

La pile ne suffit pas quand une valeur doit survivre au retour d'une fonction.

Exemple

```
let paire x =
  let p1 = (x, x + 1) in
  let p2 = (x - 1, x) in
  (p1, p2)
```

Les paires `p1`, `p2` et la paire `(p1, p2)` doivent être allouées dans le tas, sinon elles disparaîtraient à la fin de l'appel.

1. En effet, l'instruction en assembleur générée pour un appel récursif terminal est un simple saut à l'adresse de début de la fonction. C'est la même technique qui est utilisée pour revenir au début du code d'une boucle.

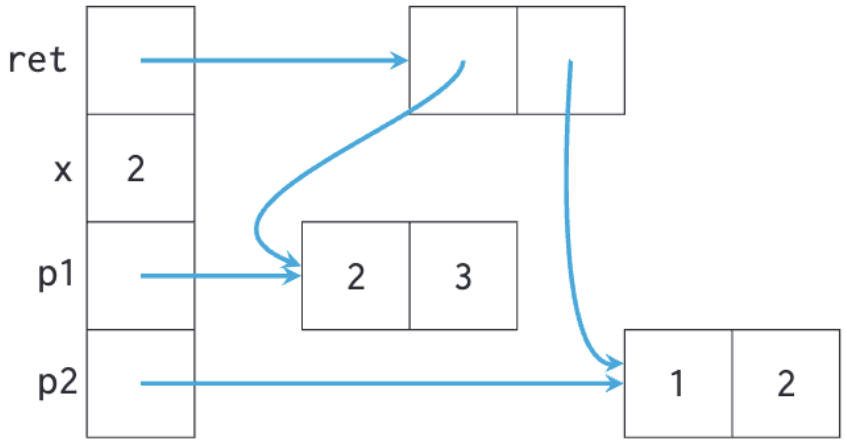
Le tas est un tableau contigu de cases mémoires, géré plus librement que la pile. Dans certains langages (C) l'allocation est *explicite* (`malloc`). En OCaml, elle est *automatique* et transparente.

Presque toutes les valeurs sont des pointeurs vers des données du tas, avec exception (représentées en entiers immédiats) : `int`, `char`, `unit`, `bool` et constructeurs nuls des types sommes.

Après `paire 2` :

- `x` (`int`) est dans la pile ;
- `p1` et `p2` sont allouées dans le tas ;
- la pile ne stocke que des *pointeurs* vers `p1` et `p2` ;
- la case `ret` contient un pointeur vers la paire `(p1, p2)` située dans le tas ;
- chaque composant de `(p1, p2)` pointe vers `p1` et `p2`.

Ainsi, ces valeurs survivent au retour de la fonction.



Segments de pile et de tas pour l'appel `paire 2`.

Décider quand libérer une zone du tas est difficile (risque de manque d'espace ou de libération trop tôt ⇒ erreur à l'exécution). OCaml utilise un ramasse-miettes (*glaneur de cellules*, *garbage collector*) qui, *pendant l'exécution*, détecte automatiquement les zones du tas devenues inutiles et les libère. Le programmeur OCaml n'a donc pas à gérer manuellement l'allocation/libération du tas.

Attention

Représentation des valeurs : pointeurs vs. entiers
En OCaml, les valeurs sont soit des pointeurs, soit des entiers. Comme une adresse mémoire est aussi un entier, le langage doit les distinguer pour que le *ramasse-miettes* puisse identifier correctement les pointeurs à suivre dans le tas.
Principe de distinction : OCaml réserve le *bit de poids faible* :

- pour les pointeurs, ce bit vaut `0` (adresses alignées, multiples de 2 ou 4) ;
- pour les entiers, il vaut `1`.

4 Types récursifs

Le principe de récursivité s'applique aussi aux types de données. Comme pour une fonction, un type récursif `t` se définit en deux étapes :

— définir les valeurs de base, construites sans récursion ;
— définir comment construire récursivement les autres valeurs.

Exemple

Le type `ilist` des listes d'entiers. Une valeur `l` de type `ilist` est soit vide, soit formée d'un entier et d'une autre liste :

```
type ilist = Empty | Cell of int * ilist
```

— `Empty` : représente la liste vide ;
— `Cell(i, s)` : représente une liste non vide, composée d'une **tête** `i` (de type `int`) et d'une **suite** `s` (de type `ilist`).

Ainsi, le type `ilist` permet de décrire toute liste d'entiers de manière purement récursive.

Exemples de valeurs :

```
let l1 = Empty
let l2 = Cell(42, Empty)
let l3 = Cell(10, Cell(2, Cell(7, Empty)))
```

`l1` est la liste vide ; `l2` contient un seul élément 42 ; `l3` représente la liste `[10; 2; 7]`.

Fonctions récursives sur des types récursifs

Puisque `ilist` est défini récursivement, toute fonction qui le manipule l'est aussi.
Une fonction `f` sur une liste `l` suit en général une structure de *filtrage par cas* :

```
let rec f l =
  match l with
  | Empty -> ... (* cas de base *)
  | Cell (v, s) -> ... (* cas rcursif *)
```

— Le cas de base correspond à la liste vide : la fonction renvoie une valeur directe.
— Le cas récursif (`Cell(i, s)`) traite une liste non vide : la fonction applique un appel récursif sur la suite `s`.

Exemple

La fonction `mem` testant la présence d'un entier dans une liste :

```
let rec mem (x, l) =
  match l with
  | Empty -> false
  | Cell (i, s) -> x = i || mem (x, s)
```

Si `l` est vide, le résultat est `false` : aucun élément n'est trouvé.
Sinon, `l = Cell(i, s)`. Si `x = i`, l'expression `x = i || ...` retourne immédiatement `true`. Sinon, l'évaluation passe à l'appel récursif `mem (x, s)`.

5 Les listes en OCaml

La liste est une structure fondamentale en programmation fonctionnelle. OCaml fournit un type prédéfini `list`, une syntaxe dédiée et un ensemble de fonctions pour les manipuler.

Le type `list` est **polymorphe** : il peut contenir des éléments de n'importe quel type, mais tous de même type. Ainsi :

— `int list` représente une liste d'entiers ;
— `(float * int) list` une liste de paires flottant-entier.

Syntaxe des listes

La liste vide se note `[]`, quel que soit le type.
Une liste non vide formée d'une tête `i` et d'une suite `s` se note `i :: s`.

Exemple

La valeur `1::2::3::[]` correspond à la liste des entiers 1, 2 et 3.

OCaml offre aussi une notation équivalente et plus compacte :

`[e1; e2; ...; en] = e1::e2::...::en::[]`

Ces notations peuvent être combinées :

```
let l = [4] :: [[5; 8; 1]; []; [2]]
```

La fonction `mem` définie précédemment peut s'exprimer directement avec la syntaxe des listes :

```
let rec mem (x, l) =
  match l with
  | [] -> false
  | i::s -> x = i || mem (x, s)
```

5.1 Fonctions sur les listes

OCaml regroupe les opérations sur les listes dans le module `List`. On appelle ses fonctions par la notation `List.<id>`.

Exemple

— `List.hd l` retourne la tête d'une liste `l` ;
— `List.tl l` retourne la suite (le reste) ;
— `List.length l` calcule sa longueur.

Ce module définit aussi l'opérateur `@` pour concaténer deux listes de même type : `l1 @ l2`

Attention

Le coût de `@` est linéaire en la longueur de `l1`. Son implémentation n'étant pas récursive terminale, un *débordement de pile* peut survenir pour de très longues listes.