Programmation fonctionnelle

L2 Info et Math-info, 2018–19

Marc Zeitoun

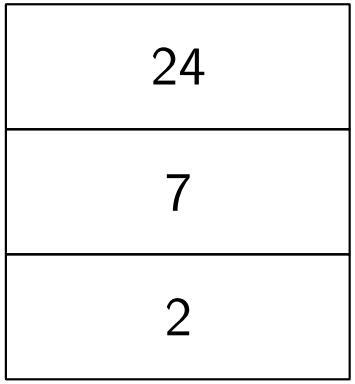
17 septembre 2018

Appels de fonction : pile d'exécution

- Pour gérer les appels de fonction **pendant l'exécution** d'un programme, et en particulier les appels récursifs, on utilise une pile.
- La pile sert à mémoriser les variables des fonctions appelées.

Appels de fonction : pile d'exécution

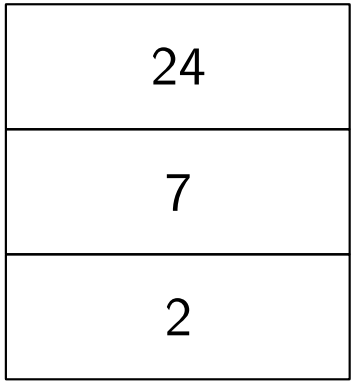
- Pour gérer les appels de fonction **pendant l'exécution** d'un programme, et en particulier les appels récursifs, on utilise une pile.
- La pile sert à mémoriser les variables des fonctions appelées.
- Pile = zone contiguë de la mémoire pouvant contenir des valeurs :



Pile contenant les valeurs 2, 7, 24

Appels de fonction : pile d'exécution

- Pour gérer les appels de fonction **pendant l'exécution** d'un programme, et en particulier les appels récursifs, on utilise une pile.
- La pile sert à mémoriser les variables des fonctions appelées.
- Pile = zone contiguë de la mémoire pouvant contenir des valeurs :

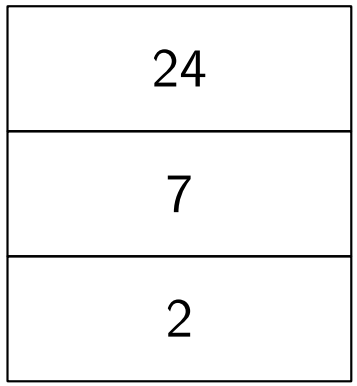


Pile contenant les valeurs 2, 7, 24

- Deux opérations sur une pile :
  - ajout d'une valeur (fait grandir la pile)
  - suppression de la valeur la plus haute (fait diminuer la pile).

Empilement

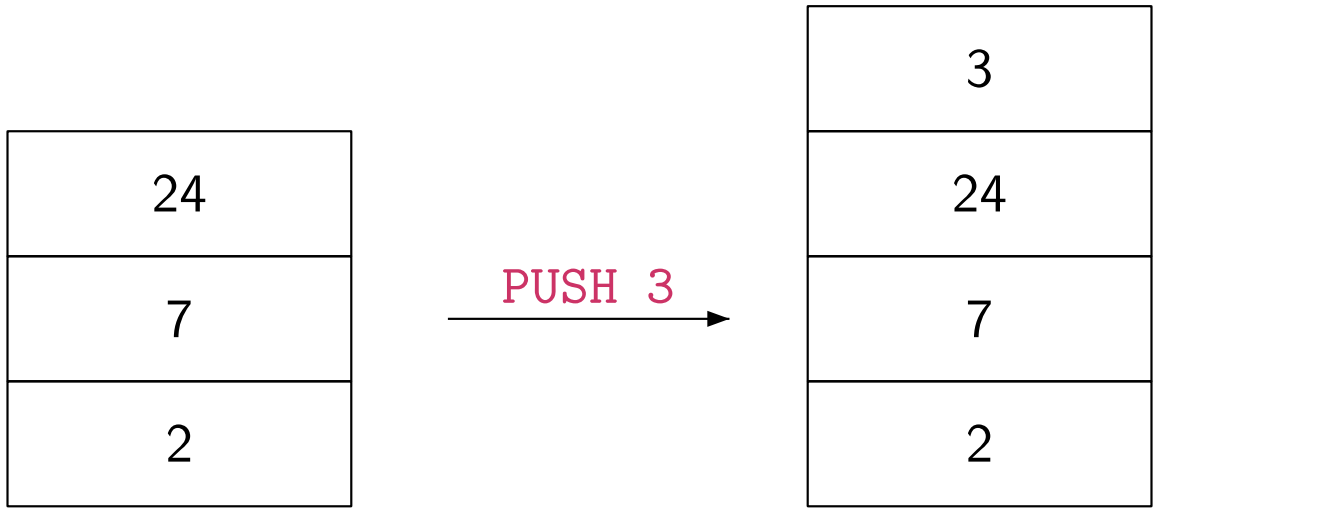
- L'ajout d'une valeur à la pile se fait toujours en haut de pile.
- L'ajout de `x` se note `PUSH x`, ou `EMPILER x`.



Pile contenant
les valeurs 2, 7, 24

Empilement

- L'ajout d'une valeur à la pile se fait toujours en haut de pile.
- L'ajout de `x` se note `PUSH x`, ou `EMPILER x`.

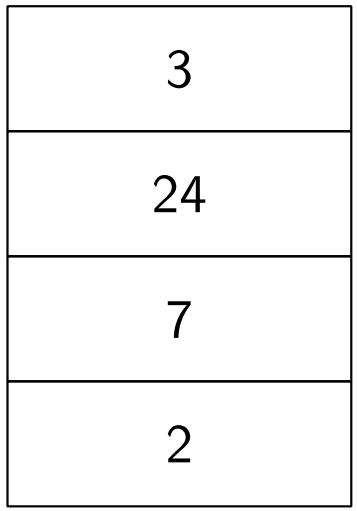


Pile contenant
les valeurs 2, 7, 24

Après `PUSH 3` : pile contenant
les valeurs 2, 7, 24, 3

Dépilement

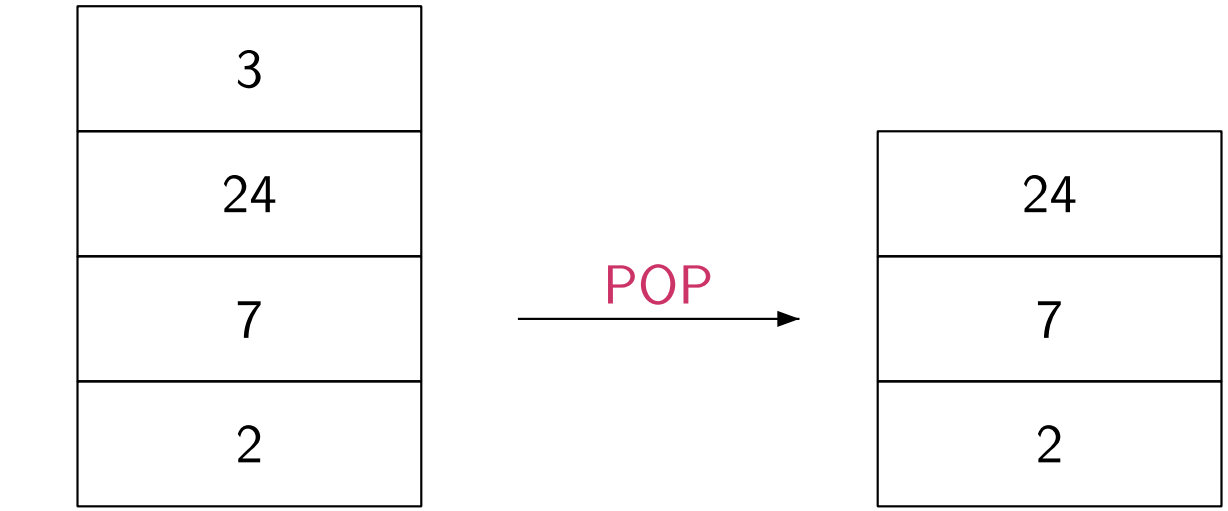
- La suppression d'une valeur à la pile se fait toujours en haut de pile.
- La suppression se note `POP`, ou `DÉPILER` et permet de récupérer la valeur qui a été supprimée.



Pile contenant
les valeurs 2, 7, 24, 3

Dépilement

- La suppression d'une valeur à la pile se fait toujours en haut de pile.
- La suppression se note `POP`, ou `DÉPILER` et permet de récupérer la valeur qui a été supprimée.



Pile contenant
les valeurs 2, 7, 24, 3

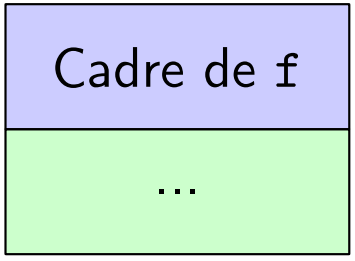
Après `POP` : pile contenant
les valeurs 2, 7, 24
`POP` « retourne » la valeur 3

Appel de fonction : fonctionnement (en pseudo-code)

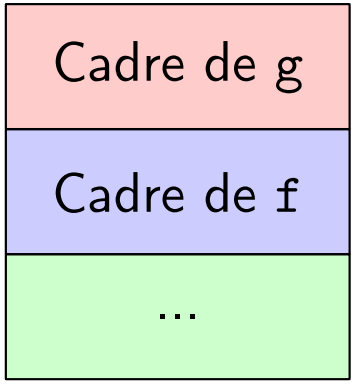
```
Algorithme f():
    var x ← 3, y ← 6
    z ← g(x,y)
    retourner z
```

```
Algorithme g(x1, x2):
    y ← 15
    retourner y - x1 * x2
```

- Schéma de la pile :



Avant appel de g



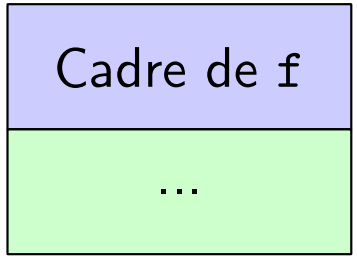
Pendant l'appel à g

Appel de fonction : fonctionnement (en pseudo-code)

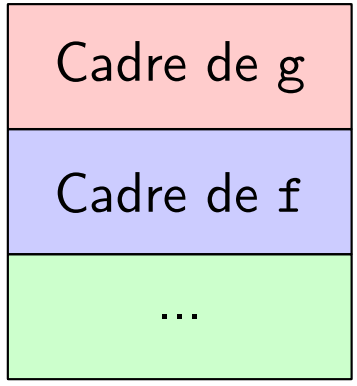
```
Algorithme f():
    var x ← 3, y ← 6
    z ← g(x,y)
    retourner z
```

```
Algorithme g(x1, x2):
    y ← 15
    retourner y - x1 * x2
```

- Schéma de la pile :



Avant appel de g



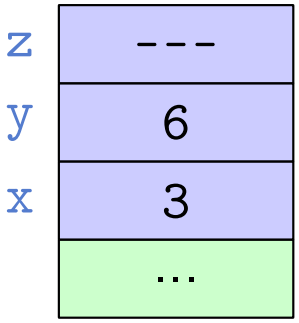
Pendant l'appel à g

- Un cadre d'appel de fonction permet de mémoriser
  - les valeurs des arguments et variables locales.
  - la valeur retour transmise à la fonction appelante.

Appel de fonction : fonctionnement

```
Algorithme f():
    x ← 3, y ← 6
    z ← g(x,y)
    retourner z
```

```
Algorithme g(x1, x2):
    y ← 15
    retourner y - x1 * x2
```

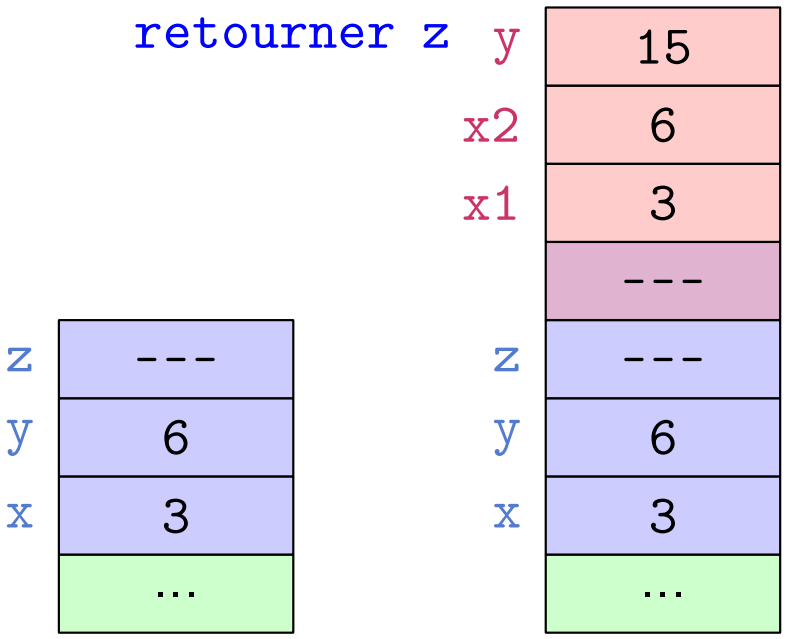


Avant appel de g

Appel de fonction : fonctionnement

```
Algorithme f():
    x ← 3, y ← 6
    z ← g(x,y)
    retourner z
```

```
Algorithme g(x1, x2):
    y ← 15
    retourner y - x1 * x2
```

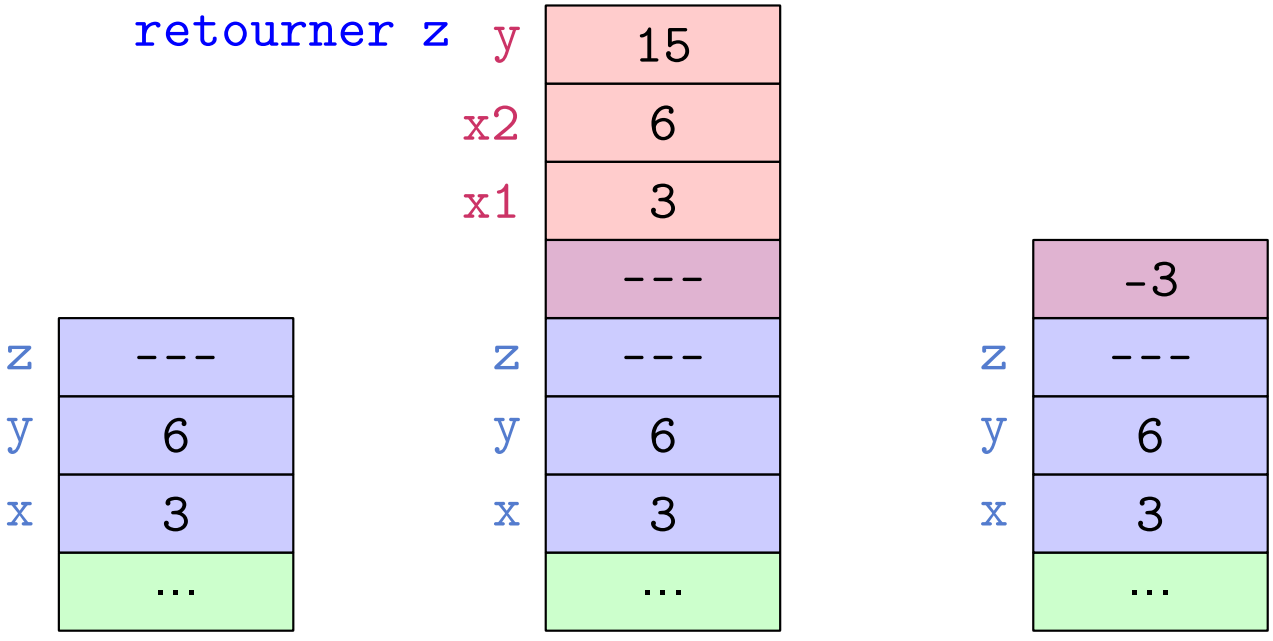


Avant appel de g Pendant appel à g

Appel de fonction : fonctionnement

```
Algorithme f():
    x ← 3, y ← 6
    z ← g(x,y)
    retourner z
```

```
Algorithme g(x1, x2):
    y ← 15
    retourner y - x1 * x2
```

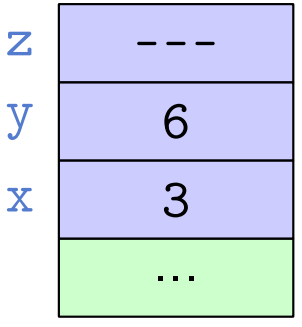


Avant appel de g | Pendant appel à g | Après retour de g

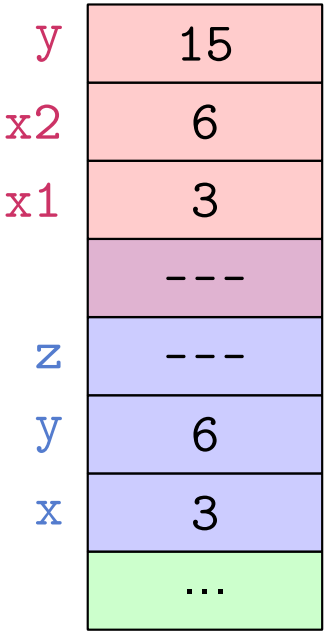
Appel de fonction : fonctionnement

```
Algorithme f():
    x ← 3, y ← 6
    z ← g(x,y)
    retourner z
```

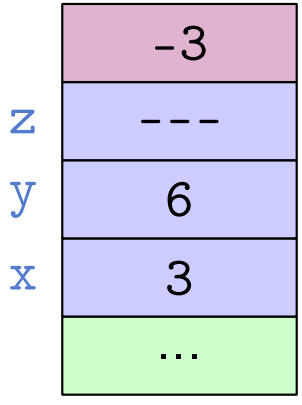
```
Algorithme g(x1, x2):
    y ← 15
    retourner y - x1 * x2
```



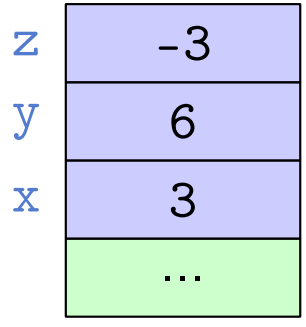
Avant appel de g



Pendant appel à g



Après retour de g



Après z=g(x,y)

Récursivité

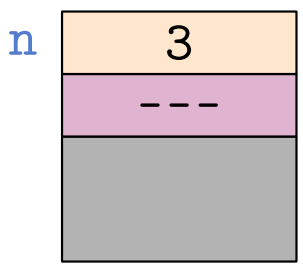
- On peut, dans une définition de fonction `f`, appeler la fonction `f`.
- Les règles d'appel de fonction s'appliquent normalement.
- Bien adapté aux définitions de fonctions récursives.
- Exemple :
  - $0! = 1! = 1$,
  - $n! = n \times (n-1)!$ pour $n \geqslant 1$.

```
let rec fact = fun n ->
    if n <= 1
    then 1
    else n * fact (n-1)

fact 3
```

Récursivité

- On peut, dans une définition de fonction `f`, appeler la fonction `f`.
- Les règles d'appel de fonction s'appliquent normalement.
- Bien adapté aux définitions de fonctions récursives.
- Exemple :
  - $0! = 1! = 1$,
  - $n! = n \times (n-1)!$ pour $n \geqslant 1$.

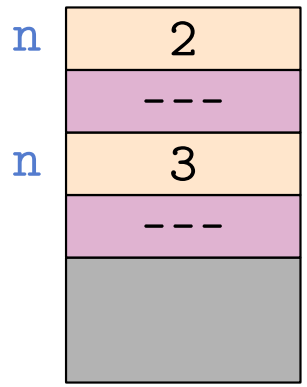
```
let rec fact = fun n ->
    if n <= 1
    then 1
    else n * fact (n-1)

fact 3
```



Récursivité

- On peut, dans une définition de fonction `f`, appeler la fonction `f`.
- Les règles d'appel de fonction s'appliquent normalement.
- Bien adapté aux définitions de fonctions récursives.
- Exemple :
  - $0! = 1! = 1$,
  - $n! = n \times (n-1)!$ pour $n \geqslant 1$.

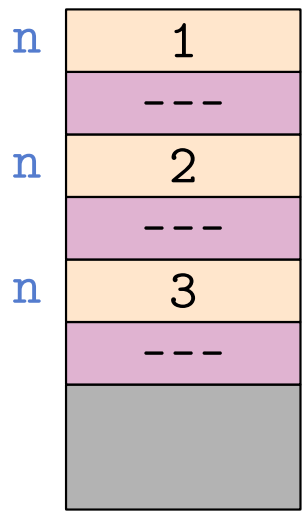
```
let rec fact = fun n ->
    if n <= 1
    then 1
    else n * fact (n-1)

fact 3
```



Récursivité

- On peut, dans une définition de fonction `f`, appeler la fonction `f`.
- Les règles d'appel de fonction s'appliquent normalement.
- Bien adapté aux définitions de fonctions récursives.
- Exemple :
  - $0! = 1! = 1$,
  - $n! = n \times (n-1)!$ pour $n \geqslant 1$.

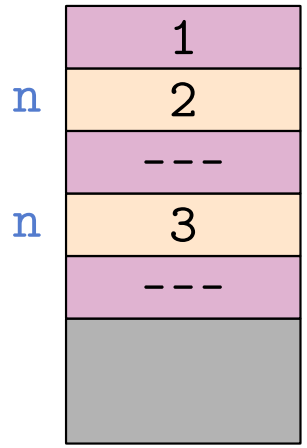
```
let rec fact = fun n ->
    if n <= 1
    then 1
    else n * fact (n-1)

fact 3
```



Récursivité

- On peut, dans une définition de fonction `f`, appeler la fonction `f`.
- Les règles d'appel de fonction s'appliquent normalement.
- Bien adapté aux définitions de fonctions récursives.
- Exemple :
  - $0! = 1! = 1$,
  - $n! = n \times (n-1)!$ pour $n \geqslant 1$.

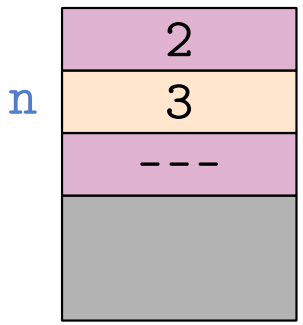
```
let rec fact = fun n ->
    if n <= 1
    then 1
    else n * fact (n-1)

fact 3
```



Récursivité

- On peut, dans une définition de fonction `f`, appeler la fonction `f`.
- Les règles d'appel de fonction s'appliquent normalement.
- Bien adapté aux définitions de fonctions récursives.
- Exemple :
  - $0! = 1! = 1$,
  - $n! = n \times (n-1)!$ pour $n \geqslant 1$.

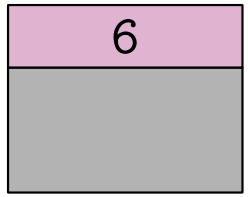
```
let rec fact = fun n ->
    if n <= 1
    then 1
    else n * fact (n-1)

fact 3
```



Récursivité

- On peut, dans une définition de fonction `f`, appeler la fonction `f`.
- Les règles d'appel de fonction s'appliquent normalement.
- Bien adapté aux définitions de fonctions récursives.
- Exemple :
  - $0! = 1! = 1$,
  - $n! = n \times (n-1)!$ pour $n \geqslant 1$.

```
let rec fact = fun n ->
    if n <= 1
    then 1
    else n * fact (n-1)

fact 3
```



Récursivité

Pour définir une fonction récursive,

- on a un ordre $\leqslant$ sur les entrées d'un problème.
  - Bien fondé : il n'y a pas de suite infinie strictement décroissante.
  - Il y a un nombre fini d'éléments minimaux pour $\leqslant$.

Récursivité

Pour définir une fonction récursive,

- on a un ordre $\leqslant$ sur les entrées d'un problème.
  - Bien fondé : il n'y a pas de suite infinie strictement décroissante.
  - Il y a un nombre fini d'éléments minimaux pour $\leqslant$.

Sous ces hypothèses, pour écrire un algorithme, on peut

- décrire son comportement sur les éléments $\leqslant$-minimaux,
- spécifier son comportement sur les autres entrées en fonction de son comportement sur des entrées strictement plus petites.

Exemple : marches d'escalier

Supposément posé par les recruteurs chez Amazon.

- `https://youtu.be/5o-kdjv7FD0`
- Une personne monte un escalier à n marches.
- Elle monte à chaque pas soit une, soit deux marches.
- De combien de façons peut-elle monter les escaliers ?

Appel de fonction : récapitulatif

1. Un algorithme récursif s'appelle lui-même sur des entrées « plus petites ».
   $\rightsquigarrow$ bien adapté pour des définitions mathématiques récursives.

Appel de fonction : récapitulatif

1. Un algorithme récursif s'appelle lui-même sur des entrées « plus petites ».
   $\rightsquigarrow$ bien adapté pour des définitions mathématiques récursives.

- **Ne pas oublier** de traiter les « cas de base ».

Appel de fonction : récapitulatif

1. Un algorithme récursif s'appelle lui-même sur des entrées « plus petites ».
   $\rightsquigarrow$ bien adapté pour des définitions mathématiques récursives.

- **Ne pas oublier** de traiter les « cas de base ».
- Une fonction récursive doit s'appeler elle-même !

Appel de fonction : récapitulatif

1. Un algorithme récursif s'appelle lui-même sur des entrées « plus petites ».
   $\rightsquigarrow$ bien adapté pour des définitions mathématiques récursives.

   - **Ne pas oublier** de traiter les « cas de base ».
   - Une fonction récursive doit s'appeler elle-même !
   - En Ocaml : ne pas oublier le mot-clé **rec**.

2. **Note.** Pour tout algorithme récursif, il existe un algorithme non récursif effectuant le même calcul.

   Il suffit pour cela de savoir gérer « manuellement » la pile d'exécution.