Programmation fonctionnelle

Cours n°2 : Fonctions récursives, mémorisation

Licence 1 Informatique, Université Paris 8

26 Septembre 2022

Récursivité

- Une **fonction récursive** est une fonction qui fait appel à elle même pour s'exécuter. En pratique, pour appliquer la fonction sur un argument donné, on va supposer qu'on connaît le résultat de l'appel de la fonction sur une donnée « **plus petite** », pour ensuite reconstruire le résultat global.

Récursivité

- Une **fonction récursive** est une fonction qui fait appel à elle même pour s'exécuter.
  En pratique, pour appliquer la fonction sur un argument donné, on va supposer qu'on connaît le résultat de l'appel de la fonction sur une donnée « **plus petite** », pour ensuite reconstruire le résultat global.

- **Exemple :** On a une liste de nombres $L = (3\ \ 6{,}5\ \ 1\ \ -2\ \ 0\ \ 7)$ dont on cherche le minimum.
  - Supposons qu'on connaisse le minimum m de la liste privée de son premier élément, ici $m = -2$.
  - Pour connaître le minimum de la liste L, il suffit alors de comparer ce minimum m avec le premier élément, et ne garder que le plus petit.

Récursivité

- Une **fonction récursive** est une fonction qui fait appel à elle même pour s'exécuter.
  En pratique, pour appliquer la fonction sur un argument donné, on va supposer qu'on connaît le résultat de l'appel de la fonction sur une donnée « **plus petite** », pour ensuite reconstruire le résultat global.

- **Exemple :** On a une liste de nombres $L = (3\ \ 6{,}5\ \ 1\ \ -2\ \ 0\ \ 7)$ dont on cherche le minimum.
  - Supposons qu'on connaisse le minimum m de la liste privée de son premier élément, ici $m = -2$.
  - Pour connaître le minimum de la liste L, il suffit alors de comparer ce minimum m avec le premier élément, et ne garder que le plus petit.

- **Question 1: Comment trouve-t-on le minimum pour la sous-liste ?**
  - En faisant le même raisonnement ! On va appeler notre fonction sur la sous-liste de celle-ci où on oublie le premier élément.

Récursivité

- Une **fonction récursive** est une fonction qui fait appel à elle même pour s'exécuter. En pratique, pour appliquer la fonction sur un argument donné, on va supposer qu'on connaît le résultat de l'appel de la fonction sur une donnée « **plus petite** », pour ensuite reconstruire le résultat global.

- **Exemple :** On a une liste de nombres $L = (3\ \ 6{,}5\ \ 1\ \ -2\ \ 0\ \ 7)$ dont on cherche le minimum.
  - Supposons qu'on connaisse le minimum m de la liste privée de son premier élément, ici $m = -2$.
  - Pour connaître le minimum de la liste L, il suffit alors de comparer ce minimum m avec le premier élément, et ne garder que le plus petit.

- **Question 1: Comment trouve-t-on le minimum pour la sous-liste ?**
  - En faisant le même raisonnement ! On va appeler notre fonction sur la sous-liste de celle-ci où on oublie le premier élément.

- **Question 2: Quand s'arrête-t-on ?**
  - Quand on ne peut plus parler de sous-liste, donc quand la liste a un seul élément. On parle de **cas d'arrêt**.
  - Dans ce cas particulier d'une liste à un seul élément, connaître le minimum est facile : c'est forcément cet élément ! On peut donc le renvoyer.

Récursivité

- Voici dont la fonction écrite en langage naturel:

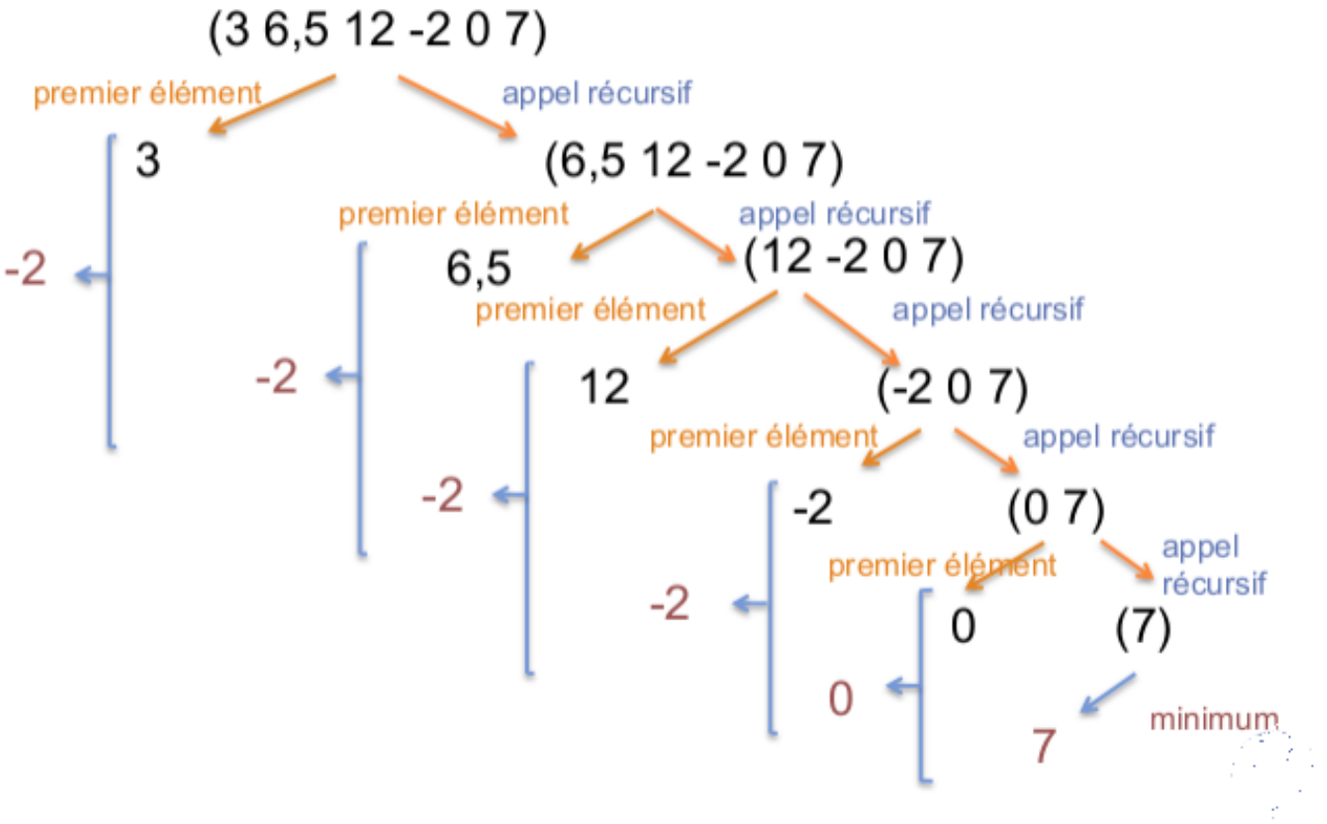
```
Définition de la fonction minimum (L)

Si vide?(reste(L)) Alors
        retourne premier(L)
Sinon
        Si premier(L) < minimum(reste(L)) Alors
                retourne premier(L)
        Sinon
                retourne minimum(reste(L))
        FinSi
FinSi
```

- La première ligne représente le **cas d'arrêt**: si la liste ne contient qu'un élément, autrement dit que son reste est vide (ce qui est plus simple à tester), on retourne le premier élément.

- L'instruction **minimum(reste(L))** constitue **l'appel récursif** de la fonction minimum sur la liste reste(L).

Récursivité

- **Illustration :**



- Pour écrire une fonction récursive, il faut donc (dans cet ordre).
  - 1) Choisir sur quelle donnée plus petite faire l'appel récursif. En pratique, pour un entier n ce sera $n-1$; pour une liste L, ce sera L moins son premier élément etc.
  - 2) Déterminer comment passer du résultat de l'appel récursif au résultat cherché.
  - 3) Identifier le cas d'arrêt.
- C'est ici ce qu'on appelle de la récursivité **en remontant**. Nous verrons plus tard un autre type de récursivité.

Quelques fonctions récursives

- **Factorielle :** On calcule ici, pour en entier n, la quantité $n! := 1 \times 2 \times \cdots \times n$. Par convention, $0! = 1$.

  Remarque clé : $n! = n \times (n-1)!$

Quelques fonctions récursives

- **Factorielle :** On calcule ici, pour en entier n, la quantité $n! := 1 \times 2 \times \cdots \times n$. Par convention, $0! = 1$.

Remarque clé : $n! = n \times (n-1)!$

```
#lang racket
(define facto
  (lambda (n); n un entier -> renvoie un nombre entier
    (if (= 0 n)
        1
        (* n (facto (- n 1))))))
```

Quelques fonctions récursives

- **Factorielle :** On calcule ici, pour en entier n, la quantité $n! := 1 \times 2 \times \cdots \times n$. Par convention, $0! = 1$.

  Remarque clé : $n! = n \times (n-1)!$

```
#lang racket
(define facto
  (lambda (n); n un entier -> renvoie un nombre entier
    (if (= 0 n)
        1
        (* n (facto (- n 1))))))
```

- **Minimum d'une liste :**

```
#lang racket
(define minimum
  (lambda (L); L une liste -> renvoie un entier
    (if (null? (cdr L))
        (car L)
        (min (car L) (minimum (cdr L))))))
```

  - Il reste ici des fonctions mystérieuses, notamment **car** et **cdl**, que nous verrons plus tard...

Correction de l'Exercice 4 du TD1

- Fonction récursive pour calculer la somme des carrés de 1 à N:

Correction de l'Exercice 4 du TD1

- Fonction récursive pour calculer la somme des carrés de 1 à N:

```
1 #lang racket
2 (define (sommeCarrés N)
3   (if (= N 1)
4       1
5       (+ (* N N) (sommeCarrés (- N 1)))))
```

Correction de l'Exercice 4 du TD1

- Fonction récursive pour calculer la somme des carrés de 1 à N:

```
1 #lang racket
2 (define (sommeCarrés N)
3   (if (= N 1)
4       1
5       (+ (* N N) (sommeCarrés (- N 1)))))
```

- Fonction récursive pour calculer le $n$ème nombre $F(n)$ de la suite de Fibonacci, qui est définie par

$F(0) = 1,\ F(1) = 1$,et pour $n \geq 2$, $F(n) = F(n-1) + F(n-2)$ **(relation de récurrence)**

Correction de l'Exercice 4 du TD1

- Fonction récursive pour calculer la somme des carrés de 1 à N:

```
1 #lang racket
2 (define (sommeCarrés N)
3   (if (= N 1)
4       1
5       (+ (* N N) (sommeCarrés (- N 1)))))
```

- Fonction récursive pour calculer le $n$ème nombre $F(n)$ de la suite de Fibonacci, qui est définie par

$F(0) = 1,\ F(1) = 1$,et pour $n \geq 2$, $F(n) = F(n-1) + F(n-2)$ **(relation de récurrence)**

```
1 #lang racket
2 (define (fibo n)
3   (if (or (= n 0) (= n 1))
4       1
5       (+ (fibo (- n 1)) (fibo (- n 2)))))
```

Correction de l'Exercice 4 du TD1

- Fonction récursive pour calculer la somme des carrés de 1 à N:

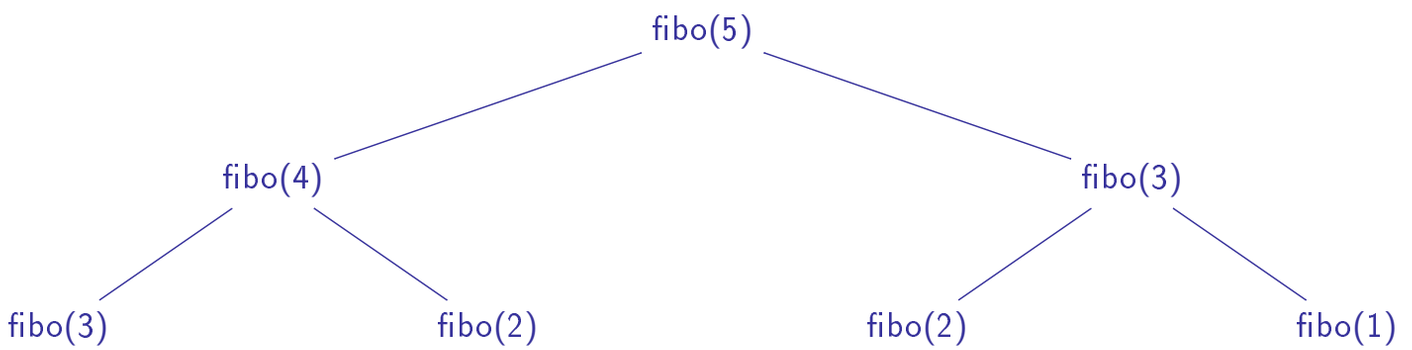
```
1 #lang racket
2 (define (sommeCarrés N)
3   (if (= N 1)
4       1
5       (+ (* N N) (sommeCarrés (- N 1)))))
```

- Fonction récursive pour calculer le $n$ème nombre $F(n)$ de la suite de Fibonacci, qui est définie par

$F(0) = 1,\ F(1) = 1$,et pour $n \geq 2$, $F(n) = F(n-1) + F(n-2)$ **(relation de récurrence)**

```
1 #lang racket
2 (define (fibo n)
3   (if (or (= n 0) (= n 1))
4       1
5       (+ (fibo (- n 1)) (fibo (- n 2)))))
```

- Notons qu'une telle fonction est très coûteuse en temps de calcul : par exemple, pour calculer $F(5)$, nous ferons 11 appels récursifs.



Correction de l'Exercice 4 du TD1

- Fonction récursive pour déterminer si un nombre est une puissance de 2 ou non.
  - Être une puissance de 2 (c'est à dire être un nombre de la forme 2^n) signifie que si on divise le nombre par 2, il doit rester un entier, et qu'on peut répéter le processus jusqu'à arriver à 1.
  - On pourrait imaginer une fonction **itérative** avec une boucle *while*, mais on veut une fonction récursive.
  - La remarque ci-dessus suggère que l'on va tester si $n/2$ est entier, et ensuite faire un appel récursif de notre fonction **non pas sur** $n-1$**, mais sur** $n/2$!
  - Reste à déterminer le cas d'arrêt: on peut s'arrêter lorsqu'on arrive à la première puissance de 2, à savoir $1 = 2^0$.

Correction de l'Exercice 4 du TD1

- Fonction récursive pour déterminer si un nombre est une puissance de 2 ou non.
  - Être une puissance de 2 (c'est à dire être un nombre de la forme 2^n) signifie que si on divise le nombre par 2, il doit rester un entier, et qu'on peut répéter le processus jusqu'à arriver à 1.
  - On pourrait imaginer une fonction **itérative** avec une boucle *while*, mais on veut une fonction récursive.
  - La remarque ci-dessus suggère que l'on va tester si $n/2$ est entier, et ensuite faire un appel récursif de notre fonction **non pas sur** $n-1$**, mais sur** $n/2$!
  - Reste à déterminer le cas d'arrêt: on peut s'arrêter lorsqu'on arrive à la première puissance de 2, à savoir $1 = 2^0$.

```
1 #lang racket
2 (define (puissance2 x); x un entier, renvoie un booléen
3   (if (= x 1)
4       #t
5       (if (integer? (/ x 2))
6           (puissance2 (/ x 2))
7           #f)))
```

Correction de l'Exercice 4 du TD1

- Fonction récursive pour déterminer si un nombre est une puissance de 2 ou non.
  - Être une puissance de 2 (c'est à dire être un nombre de la forme 2^n) signifie que si on divise le nombre par 2, il doit rester un entier, et qu'on peut répéter le processus jusqu'à arriver à 1.
  - On pourrait imaginer une fonction **itérative** avec une boucle *while*, mais on veut une fonction récursive.
  - La remarque ci-dessus suggère que l'on va tester si $n/2$ est entier, et ensuite faire un appel récursif de notre fonction **non pas sur** $n-1$**, mais sur** $n/2$!
  - Reste à déterminer le cas d'arrêt: on peut s'arrêter lorsqu'on arrive à la première puissance de 2, à savoir $1 = 2^0$.

```
1 #lang racket
2 (define (puissance2 x); x un entier, renvoie un booléen
3   (if (= x 1)
4       #t
5       (if (integer? (/ x 2))
6           (puissance2 (/ x 2))
7           #f)))
```

```
1 #lang racket
2 (define (puissance2 x); x un entier, renvoie un booléen
3   (if (= x 1)
4       #t
5      (and (integer? (/ x 2)) (puissance2 (/ x 2)))))
```

- La condition **(integer? (/ x 2))** peut aussi être remplacée par **(= 0 (modulo x 2))**.

Mémorisation

- Il est parfois nécessaire d'utiliser un même calcul ou une même fonction avec les mêmes arguments dans une expression : pour éviter l'exécution redondante d'appels de fonctions, il est possible d'utiliser des variables locales, liant les expressions entre elles.

- **Exemple :** Si on veut calculer $\sqrt{x^2+1}+\sqrt{x^2+2}$, on peut écrire

```
#lang racket
(define f
  (lambda (x)
    (+ (sqrt (+ 1 (* x x))) (sqrt (+ 2 (* x x))))))
```

- Toutefois, on calcule la quantité **(* x x)** deux fois, alors qu'on aurait pu le faire qu'une fois. La fonction **let** permet de définir une variable équivalente à ces appels et précise à l'utilisateur qu'il suffit de l'exécuter une seule fois et réutiliser la ou les valeurs retournées.

Mémorisation

- La syntaxe de **let** est la suivante:

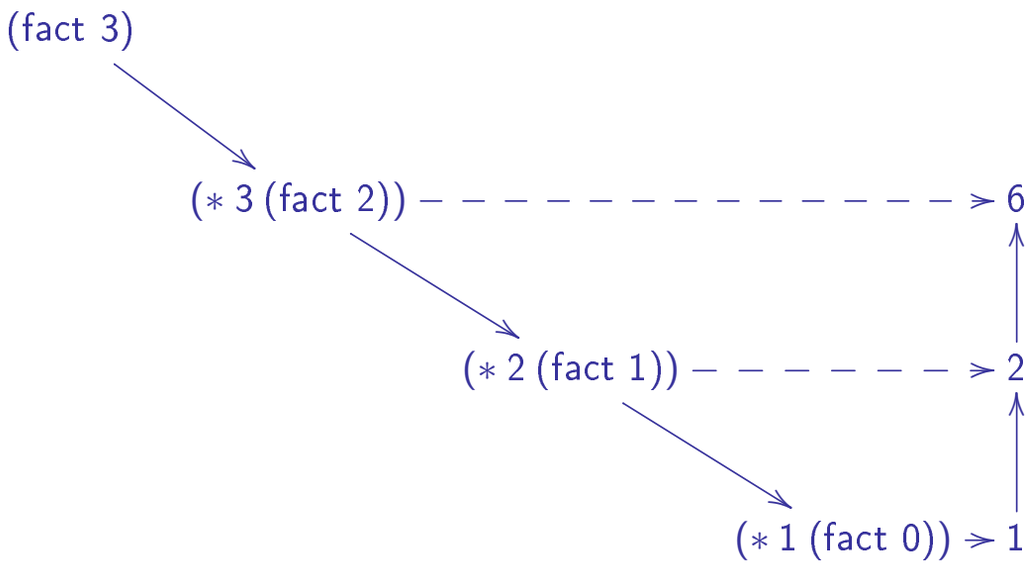
```
(let (
       (ident1 val1)
       (ident2 val2)
       ...
       (identN valN)
     )
  corps
)
```

```
#lang racket
(define f
  (lambda (x)
    (let ((x2 (* x x)))
      (+ (sqrt (+ 1 x2)) (sqrt (+ 2 x2))))))
```

- **Remarques :**
  - Les parenthèses rouges à l'intérieur du **let** (séparant les différentes mémorisations) peuvent être remplacées par des crochets.
  - Il existe d'autres fonctions telles que **let*** et **letrec** héritées de Scheme, ou encore la fonction **local** de Racket; que nous utiliserons moins. Vous pouvez regarder la documentation.
  - En pratique, on utilisera souvent une mémorisation pour éviter des appels récursifs redondants, et ainsi économiser en coût de calcul.

Récursivité terminale

- Jusqu'à présent, nous n'avons vu que des fonctions récursives utilisant le principe de la **récursivité en remontant**.



- Il existe une autre forme de récursivité, appelée **récursivité terminale** (ou **récursivité en descendant**) dont le principe est d'ajouter un paramètre à la fonction, qui va s'incrémenter au fur et à mesure des appels récursifs jusqu'au résultat souhaité lorsqu'on arrive au cas d'arrêt.

Récursivité terminale

- **Exemple :** On va écrire une fonction **fact_term** qui va prendre 2 paramètres: l'entier n dont on veut calculer la factorielle, et une variable **res** (entière puisqu'on veut que la fonction renvoie un nombre) qui va s'incrémenter à chaque appel récursif via:

  (fact_term (- n 1) (* n res))

- Lorsqu'on arrive au cas d'arrêt, on renvoie alors **res** qui doit contenir le résultat souhaité.

Récursivité terminale

- **Exemple :** On va écrire une fonction **fact_term** qui va prendre 2 paramètres: l'entier n dont on veut calculer la factorielle, et une variable **res** (entière puisqu'on veut que la fonction renvoie un nombre) qui va s'incrémenter à chaque appel récursif via:

  (fact_term (- n 1) (* n res))

- Lorsqu'on arrive au cas d'arrêt, on renvoie alors **res** qui doit contenir le résultat souhaité.

- **Implémentation Racket :**

```
1 | #lang racket
2 | (define (fact_term n res)
3 |   (if (= n 0)
4 |       res
5 |       (fact_term (- n 1) (* n res))))
```

Récursivité terminale

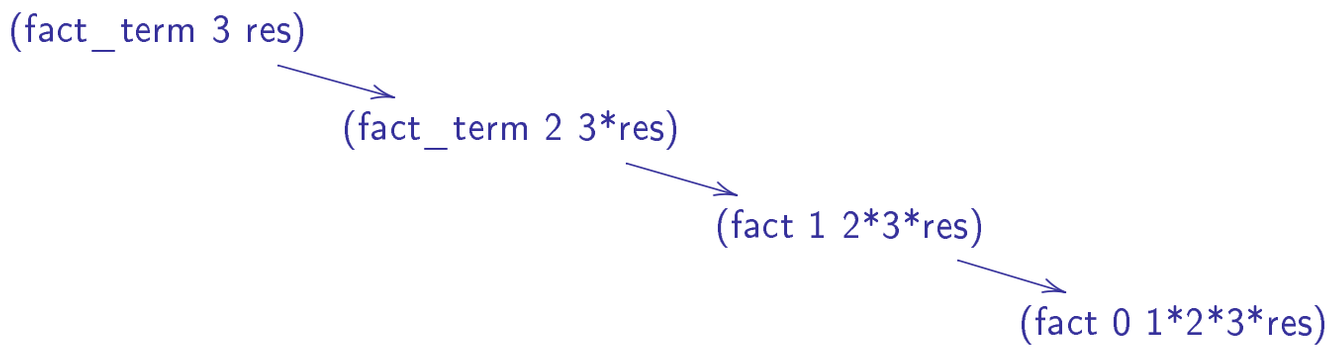
- **Exemple :** On va écrire une fonction **fact_term** qui va prendre 2 paramètres: l'entier n dont on veut calculer la factorielle, et une variable **res** (entière puisqu'on veut que la fonction renvoie un nombre) qui va s'incrémenter à chaque appel récursif via:

  (fact_term (- n 1) (* n res))

- Lorsqu'on arrive au cas d'arrêt, on renvoie alors **res** qui doit contenir le résultat souhaité.

- **Implémentation Racket :**

```
1 | #lang racket
2 | (define (fact_term n res)
3 |   (if (= n 0)
4 |       res
5 |       (fact_term (- n 1) (* n res))))
```



Récursivité terminale

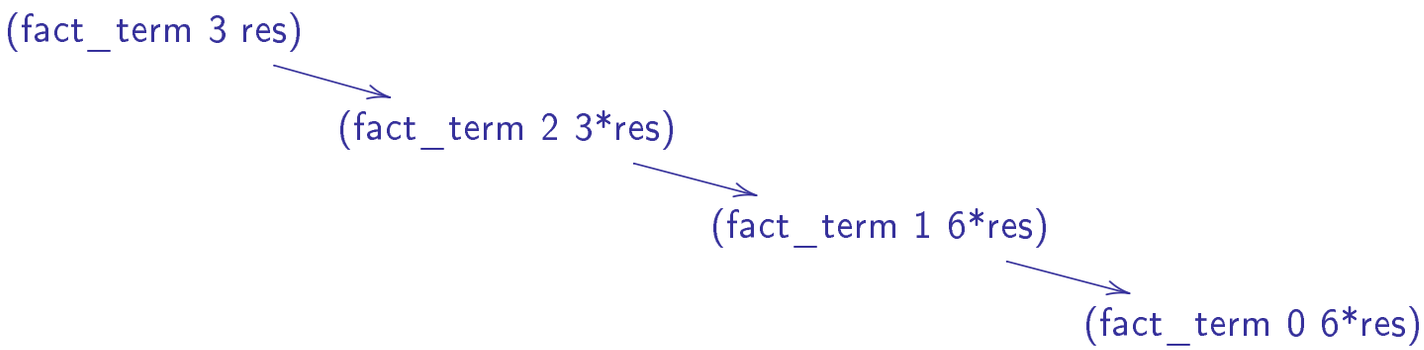
- **Exemple :** On va écrire une fonction **fact_term** qui va prendre 2 paramètres: l'entier n dont on veut calculer la factorielle, et une variable **res** (entière puisqu'on veut que la fonction renvoie un nombre) qui va s'incrémenter à chaque appel récursif via:

  (fact_term (- n 1) (* n res))

- Lorsqu'on arrive au cas d'arrêt, on renvoie alors **res** qui doit contenir le résultat souhaité.

- **Implémentation Racket :**

```
1 | #lang racket
2 | (define (fact_term n res)
3 |   (if (= n 0)
4 |       res
5 |       (fact_term (- n 1) (* n res))))
```



Récursivité terminale

- Ainsi, si au départ on prend la variable **res** égale à 1, on a à la fin le résultat souhaité. Pour obtenir la factorielle voulue, il suffit alors juste d'appeler

(fact_term n 1)

- On peut donc écrire simplement une nouvelle fonction qui calcule $n!$:

```
7 | (define (facto n)
8 |   (fact_term n 1))
```

- La fonction précédente utilisant la variable **res** est alors une fonction **auxiliaire** nous permettant d'écrire notre fonction factorielle définitive.

Exemple : Fibonacci avec récursivité terminale

- On rappelle que la suite de Fibonacci est définie par

$$F(0) = 1,\ F(1) = 1, \text{ et pour } n \geq 2,\ F(n) = F(n-1) + F(n-2).$$

- Ici, on va ajouter deux paramètres entiers a et b (qui vaudront au départ $F(0)$ et $F(1)$, donc 1 et 1. Ensuite, à chaque appel récursif, le paramètre a va prendre la somme $a + b$ des deux précédents (c'est le calcul de Fibonacci), et le paramètre b va reprendre la valeur du paramètre précédent, à savoir a.

- Lors du dernier appel récursif, pour le cas d'arrêt on renvoie ainsi le paramètre a.

```
1 #lang racket
2 (define (fibo_term n a b)
3   (if (= n 1)
4       a
5       (fibo_term (- n 1) (+ a b) a)))
6
7 (define (fibo n)
8   (fibo_term n 1 1))
```

- **Exercice :** Testez la fonction récursive classique et la récursive terminale pour Fibonacci, et essayez de les appeler pour $n = 10$, $n = 20$, et $n = 50$.

Récursivité terminale avec encapsulation

- On peut en fait utiliser de la récursivité terminale avec une seule fonction principale. Pour cela, on peut **encapsuler** la fonction auxiliaire dans la fonction principale en utilisant une définition locale de l'auxiliaire dans la principale, par exemple :

```
1 #lang racket
2 (define (factorielle n)
3   (define (fact_aux n res)
4     (if (= n 0)
5         res
6         (fact_aux (- n 1) (* n res))))
7   (fact_aux n 1))
```