# Algorithmique et programmation avancée

Licence SPI 2e année

Loïc Barrault (Loic.Barrault@univ-lemans.fr)
Bruno Jacob (Bruno.Jacob@univ-lemans.fr)
et Aina Lekira, Grégor Dupuy (TP)

# Organisation

- De janvier à avril
- 12 cours, 12 TD, 12 TP
- Documents
  - Polycopié « Le langage C » de L1
- Contrôle des connaissances
  - Contrôle continu
  - TP notés
- Travail personnel
  - Relire le cours
  - Refaire les exercices des TD
  - Préparer les TP

# Plan du cours

**Partie I**

- Retour (rapide) sur les variables et leur portée
- Récursivité
- Types abstraits
  - Arbres
  - Tables

# Chapitre 1

Retour rapide sur les variables et leur portée

# 1) Définition

- Variable = zone mémoire
  - accessible par un nom
  - lecture et écriture
  - Taille définie par le **type**
- Syntaxe déclaration :
  - <type> NOM = init_val**;**
  - <type> NOM1 {=init_val}, NOM2 [=init_val2]**;**
  - <type> TAB[10], NOM**;**
  - <type> * ptr;
    - Un pointeur est une variable pouvant contenir une adresse !

# 2) Notion de bloc

- Bloc : espace défini par une accolade ouvrante et une fermante
  - Structure de contrôle
    - if()**{BLOC},** while()**{BLOC}**, etc.
  - Fonctions
    - int fct(<params) **{}**
- Dans un bloc : deux variables ne peuvent avoir le même nom

# 3) Variable locale

- Déclarée à l'intérieur d'un bloc
- Existe à partir de la déclaration
- **Est détruite à la fin du bloc !**

# 4) Variables globales

- Déclarée en dehors de tout bloc
- Existe à partir de la déclaration
- Est détruite à la fin du programme

- Accessible depuis tout endroit du fichier (module)
  - Mot clé **extern** pour exploiter une variable globale d'un autre module

# Variable locale vs globale

- Une variable locale peut avoir le même nom qu'une variable globale
  - Masquage de la variable globale

```
int i=0;
int main(void)
{
        int i=2;
        printf("i=%i », i);
}
```

# Chapitre 2

Récursivité

# 1) Définition

Une fonction **récursive** est une fonction qui fait appel à elle-même.

Équivalent en mathématiques: la définition par récurrence

# Exemple : calcul de factorielle

La factorielle de *n* est définie par

- 1! = 1
- n! = n * (n-1)!  pour n > 1

1. Cas où le résultat est immédiat
2. Cas où le résultat se calcule par récursivité

# Exemple : fonction factorielle

```
int fact(int n)
/* fonction qui renvoie n! */
{
        if (n == 1)
                return 1;
        return n * fact(n-1);
}
```

Appel : printf( ‘‘%i’’, fact(3))

# Exemple

fact(3) = 3*fact(2) = 6

fact(2) = 2*fact(1) = 2



fact(1) = 1

# Exemple

Dans cette fonction, les calculs s'effectuent *au retour* des appels récursifs

fact(3) → fact(2) → fact(1)

↓

6 ← 2 ← 1

# Variante de la fonction factorielle

```
int fact2(int n, int result)
/* fonction qui renvoie n! */
{
        if (n == 1)
                return result;
        return fact2(n-1, n*result);
}
```

Appel : printf(‘‘%i’’, factorielle(3,1))

# Variante

fact(3,1) = fact(2,3) = fact(1,6) = 6

# Variante

Dans cette fonction, les calculs s'effectuent *au moment* des appels récursifs

fact(3,1) → fact(2,3) → fact(1,6)

↓

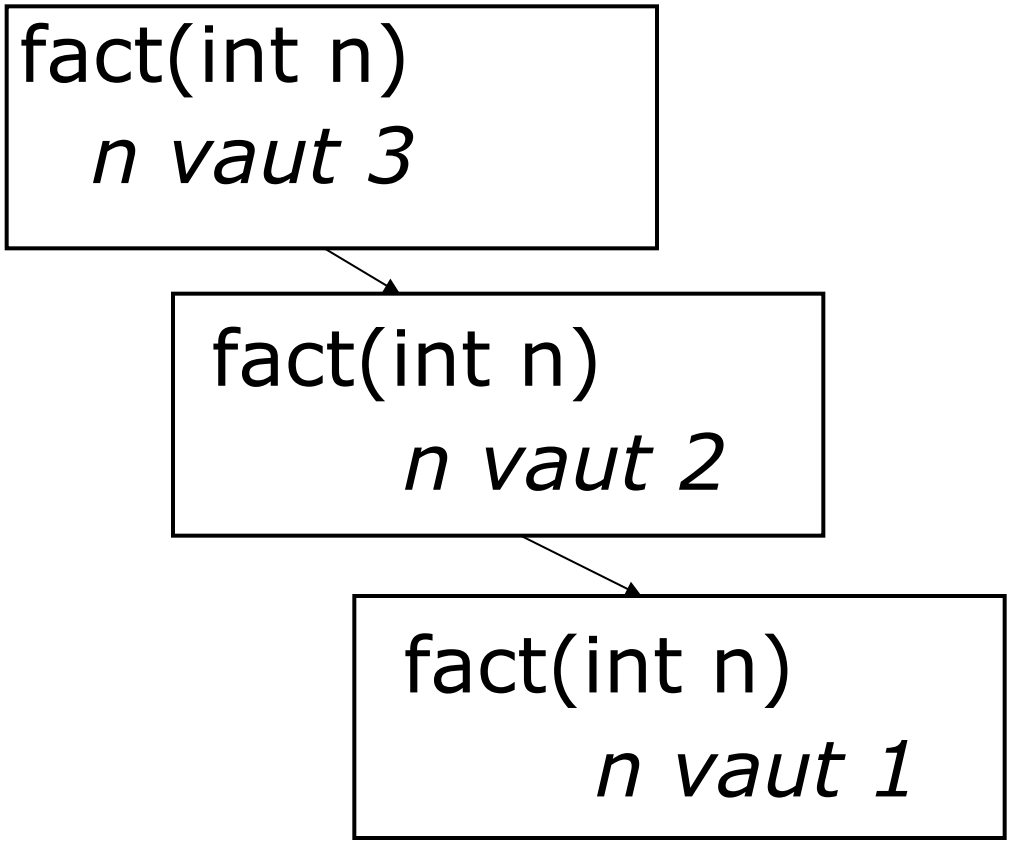
6 ← 6 ← 6

# 2) Objets locaux et globaux

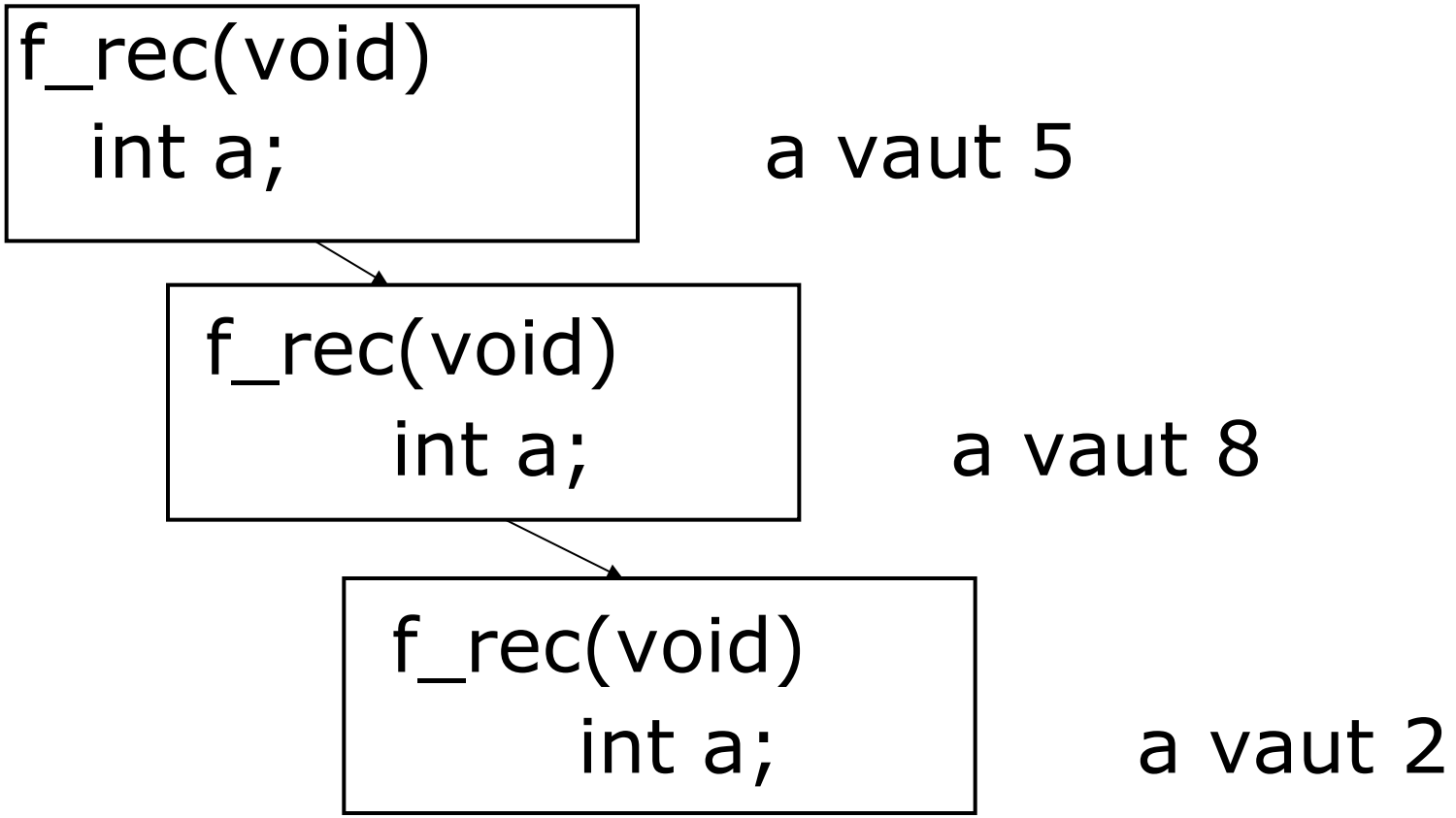
A l'exécution, il est créé en mémoire autant d'exemplaires de la fonction qu'il y a d'appels récursifs.

Chaque exemplaire de la fonction contient un exemplaire des variables locales et des paramètres fixes.

# Exemple : duplication du paramètre fixe



# Exemple : duplication des variables locales



# Objets globaux

En revanche, les variables globales n'existent qu'en un seul exemplaire et sont partagées par tous les exemplaires de la fonction récursive.

Il en va de même pour les paramètres modifiables (pointeurs).

# Exemple

Problème : lire une suite d'entiers positifs terminée par un marqueur et calculer la somme ou le produit selon que le marqueur vaut -1 ou -2

5 8 2 -1 résultat 5+8+2=15

1 2 1 4 -2 résultat 1*2*1*4=8

# Exemple

Problème : lire une suite d'entiers positifs terminée par un marqueur et calculer la somme ou le produit selon que le marqueur vaut -1 ou -2

Écrire un programme récursif pour traiter ce problème (sans utiliser de pile ni de tableau)

# Remarques

On ne peut pas faire le calcul au fur et à mesure car l'opérateur n'est connu qu'à la fin → il faut mémoriser chaque valeur lue dans une variable locale

L'opérateur et le résultat sont uniques → on peut utiliser une variable globale ou un paramètre pointeur

# Programme (1)

```
int marqueur; /* -1 ou -2 */
int resultat     /* valeur du résultat */

void calcul (int * result)
{
        int x;
        scanf( ‘ ‘%i’ ’ , &x);
```

# Programme (2)

```
if (x == -1){
        marqueur = -1;
        *result = 0;
}
else if (x == -2){
        marqueur = -2;
        *result = 1;
}
```

# Programme (3)

```
        else   /* x est un entier positif */
        {
                calcul(result);
                if (marqueur == -1)
                        *result = *result + x;
                else
                        *result = *result * x;
        }
}
```

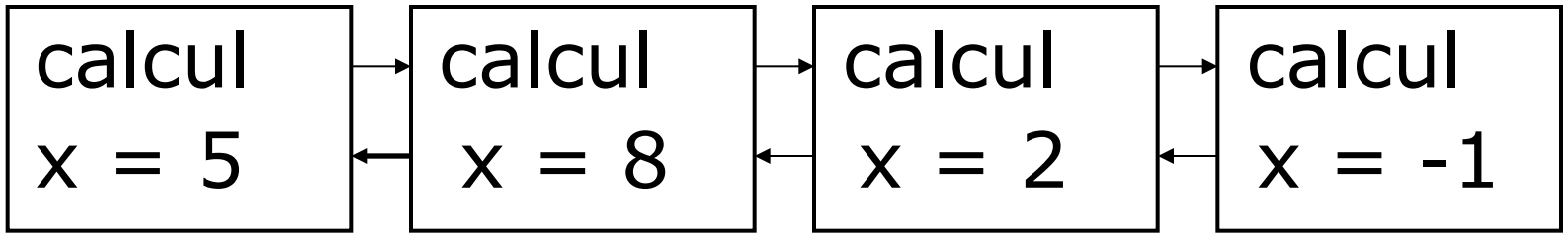
# Programme (4)

```
int main()
{
   calcul (&resultat);
   printf( ‘‘La valeur est :
   %i’’,resultat);

}
```

# Exemple

résultat : 15 marqueur : -1




5 8 2 -1

# Remarque

Dans cet exemple, chacune des variables globales *marqueur* et *resultat* pourrait être passée en paramètre ou non

```
void calcul (int * result, int * marq)
void calcul (int * result)
void calcul (int * marq)
void calcul (void)
```

# 3) Récursivité et itération

Tout algorithme récursif peut être transformé en algorithme itératif, et réciproquement.

# Factorielle récursif ↔ itératif

```
int fact(int n)
{
   if (n == 1)
        return 1;
   return n*fact(n-1);
}
```

```
int fact(int n)
{
        int res;
        res = n;
        while(n!=1){
            n = n - 1;
            res = res*n;
        }
        return res;
}
```

# Choisir entre itératif et récursif

Version récursive

- Elle est plus naturelle quand on part d'une définition récursive
- Elle est plus simple en cas de récursivité multiple (ex: parcours d'arbres)
- Elle s'impose en programmation fonctionnelle (Lisp, Caml) ou déclarative (Prolog)

# Choisir entre itératif et récursif

Version itérative

- Elle est beaucoup plus économique à l'exécution (pas de duplication de la fonction)

# Choisir entre itératif et récursif

Bilan

- En programmation impérative, il faut établir un compromis entre simplicité des algorithmes et coût d'exécution
- On préfère en général la version itérative, pour son efficacité

# 4) Transformation récursif → itératif

Tout algorithme récursif peut être transformé en un algorithme itératif équivalent : c'est la *dérécursivation*.

La méthode à suivre dépend du type de récursivité de l'algorithme.

# Récursivité terminale et non-terminale

Un algorithme est dit *récursif terminal* s'il ne contient aucun traitement après un appel récursif.

# Exemple

La fonction fact2 est récursive terminale : elle n'effectue aucun traitement après l'appel récursif

```
int fact2(int n, int result)
/* fonction qui renvoie n! */
{
        if (n == 1)
                return result;
        return fact2(n-1, n*result);
}
```

# Fonction récursive terminale

```
type f_rec(params)
        if (condition)
                traitement1
        else
        {
                traitement2
                f_rec(nouv_params)
        }
```

# Sur l'exemple

| | |
|---|---|
| *type* f_rec(*params*) | |
| if (*condition*) | n == 1 |
| *traitement1* | return result |
| else | |
| { | |
| *traitement2* | (vide) |
| f_rec(*nouv_params*) | n-1, n*result |
| } | |

# Fonction itérative équivalente

```
type f_iter(params)
        while (non condition)
        {
                traitement2
                params = nouv_params
        }
        traitement1
```

# Exemple

```
int fact2_iter(int n, int result)
{
        while (n!=1)
        {
                n = n - 1;
                result = result*n;
        }
        return result;
}
```

# Récursivité terminale et non-terminale

Réciproquement, une fonction est *récursive non terminale* si elle effectue un traitement après un appel récursif.

# Exemple

La fonction fact est récursive non terminale : elle effectue une multiplication après l'appel récursif

```
int fact(int n)
/* fonction qui renvoie n! */
{
        if (n == 1)
                return 1;
        return n * fact(n-1);
}
```

# Fonction récursive **non** terminale

```
type f_rec(params)
        if (condition)
                traitement1
        else
        {
                traitement2
                f_rec(nouv_params)
                traitement3(nouv_params)
        }
```

# Dé-récursivation d'une fonction récursive non terminale

Première méthode

Transformer la fonction pour obtenir une fonction récursive terminale, puis se ramener au premier cas.

En général, cela nécessite d'ajouter un paramètre.

**Exemple**

fact(int n) → fact(int n, int result)

# Dé-récursivation d'une fonction récursive non terminale

Deuxième méthode

Quand la transformation n'est pas possible (récursivité multiple), on introduit une pile dans laquelle on stocke les paramètres de l'appel récursif.

Cette méthode est plus complexe à réaliser.

# Dé-récursivation d'une fonction récursive non terminale

```
void quicksort(i,j)

int milieu;


if(i<j){
   m = partition(i,j);
   quicksort(i,m-1);
   quicksort(m+1,j);
}
```

```
void quicksort (i,j)

empiler(0,N-1);
while(!pilevide()){
   depiler(i,j);
   if(i<j){
        m = partition(i,j);
        empiler(i,m-1);
        empiler(m+1,j);
   }
}
```

# 5) Différents types de récursivité

- Récursivité simple
- Récursivité multiple
- Récursivité croisée
- Récursivité imbriquée

# a) Récursivité multiple

La fonction effectue plusieurs appels récursifs consécutifs.

**Exemple**

Calcul des combinaisons $C_n^p$

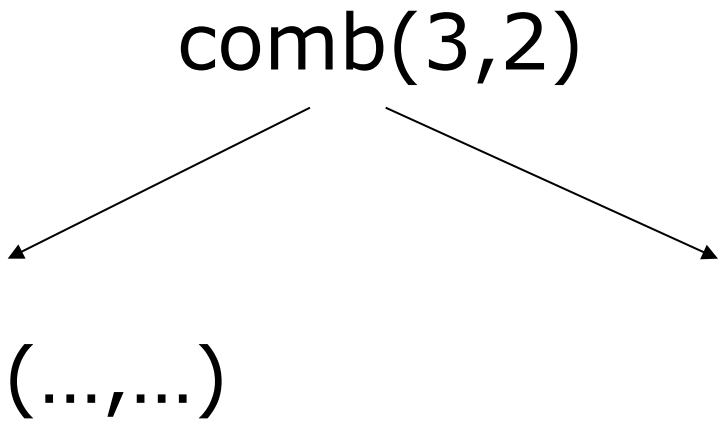
- $C_n^p = 1$ si $p = 0$ ou $p = n$
- $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$ sinon

# Récursivité multiple

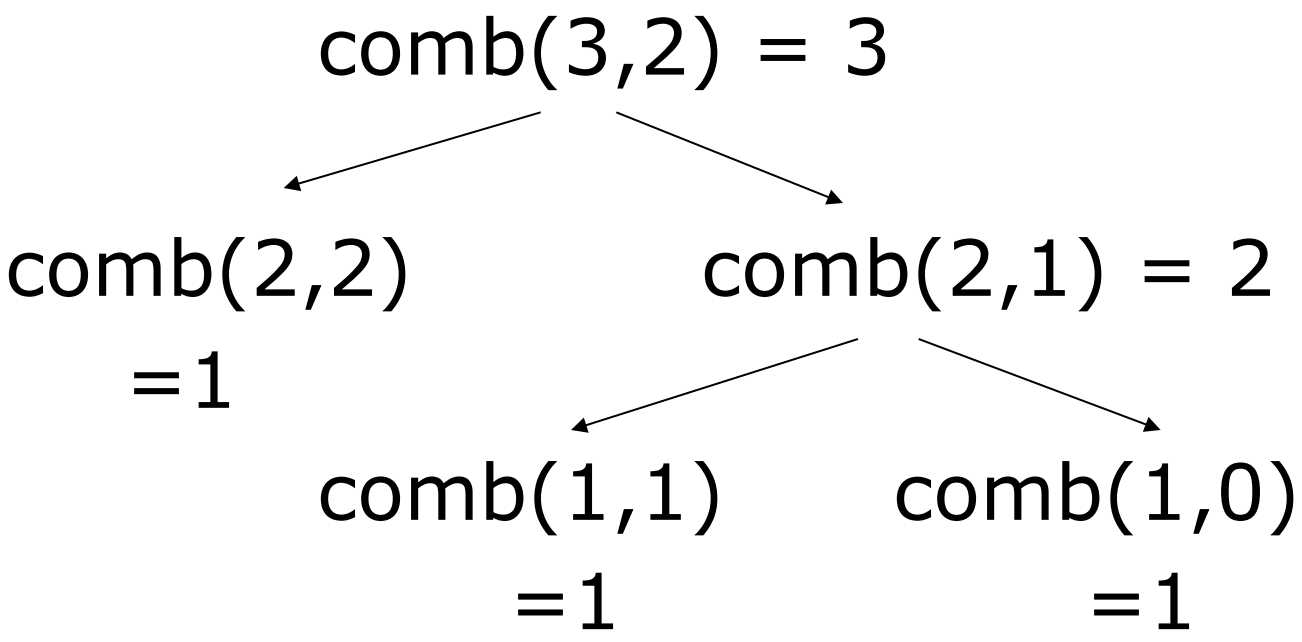
```
int comb(int n, int p)
/* calcule récursivement C_n^p */
{
   if (p == 0 || p == n)
        return 1;
   else
        return comb(n-1,p) + comb(n-1,p-1);
}
```

# Application

Dérouler l'exécution de comb(3,2) en faisant apparaître chaque appel récursif et donner le résultat



# Application



# b) Récursivité mutuelle (ou croisée)

Des définitions sont mutuellement récursives si elles dépendent l'une de l'autre.

**Exemple**

Définition de la parité

- n est pair si n=0 ou si (n-1) est impair
- n est impair si n=1 ou si (n-1) est pair

# Récursivité mutuelle

```
int pair(int n){
        if(n==0)
                return vrai;
        else
                return impair(n-1);
}

int impair(int n){
        if(n==0)
                return faux;
        else
                return pair(n-1);
}
```

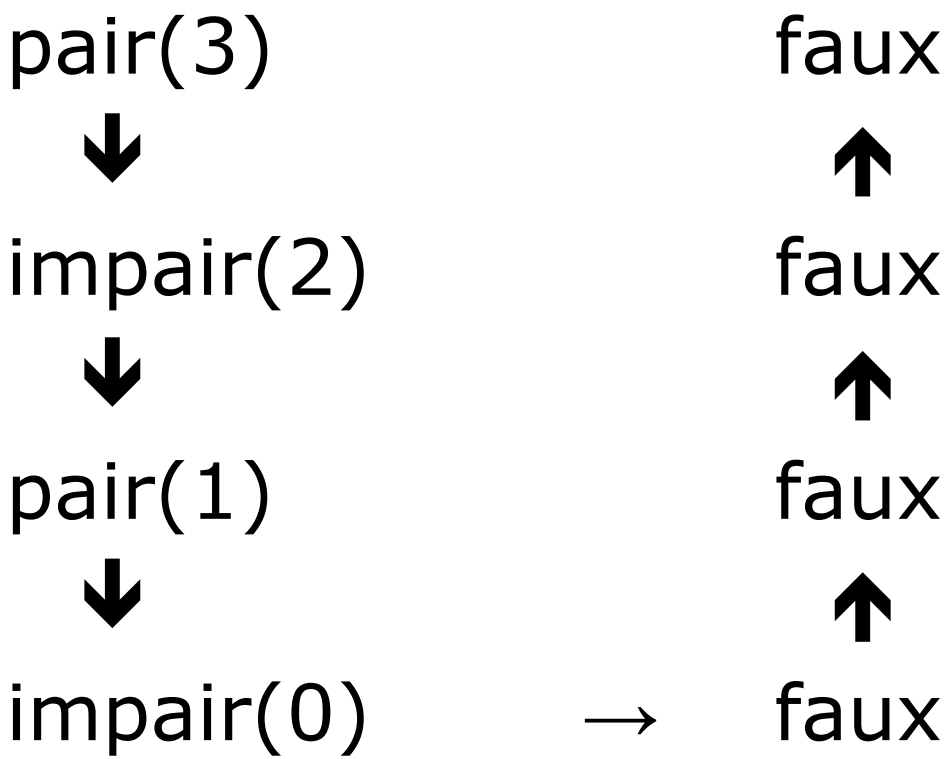
# Application

Dérouler l'exécution de

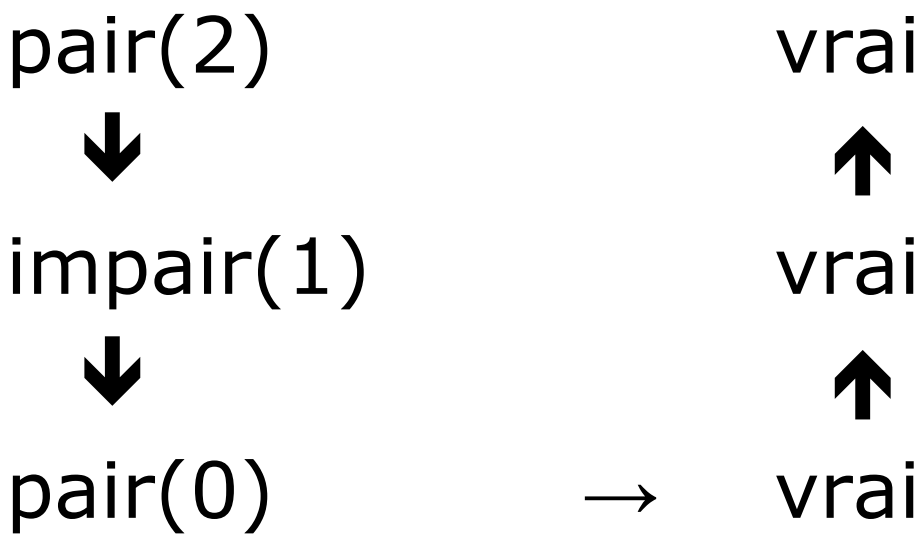
- pair(3)
- pair(2)

en faisant apparaître chaque appel récursif et donner le résultat

# Application : 3 est-il pair ?



# Application : 2 est-il pair ?



# c) Récursivité imbriquée

Un paramètre de l'appel récursif est lui-même issu d'un appel récursif

**Exemple**

Calcul de la fonction d'Ackermann

- A(m,n)=n+1 si m=0
- A(m,n)=A(m-1,1) si m>0 et n=0
- A(m,n)=A(m-1,A(m,n-1)) sinon

# Récursivité imbriquée

```
int ack(int m, int n)
{
        if (m==0)
                return n+1;
        else if (m>0 && n==0)
                return ack(m-1,1);
        else
                return ack(m-1, ack(m,n-1));
}
```

# Application

Dérouler l'exécution de ack(1,1) en faisant apparaître chaque appel récursif et donner le résultat

# Application

ack(1,1) **= 3**

↓ **c**

ack(0, ack(1, 0)) → ack(0,2)

↓ **b** =3 **a**

ack(0,1)

=2 **a**