UNIVERSITÉ Grenoble Alpes

# INF201
# Algorithmique et Programmation Fonctionnelle
## Cours 4: Fonctions et types récursifs

Année 2022


# Les précédents épisodes de INF 201

- Types de base : `bool`, `int`, `float`, `char`
- `if` ... `then` ... `else` ... expression conditionnelle
- identificateurs (locaux et globaux)
- définition, utilisation et composition de fonctions
- types avancés : synonyme, énuméré, produit, somme
- filtrage et pattern-matching

# Plan

Retour sur les fonctions

Récursivité

Terminaison

Fonctions mutuellement récursives

Types Récursifs

Conclusion

# Fonctions: SPECIFICATION et REALISATION

Il est très important de distinguer 2 concepts/étapes lors de la définition d'une fonctions (et d'un programme en général)

# Fonctions: SPECIFICATION et REALISATION

Il est très important de distinguer 2 concepts/étapes lors de la définition d'une fonctions (et d'un programme en général)

## Spécification (Le QUOI):

Une description de ce qui est attendu

- à un certain niveau d'abstraction
- doit être suffisamment précis
- proche d'une description "mathématique"
- peut utiliser des exemples *pertinents*

Un contrat


Consiste en:

- 1 description =sémantique
- 1 signature = profil = (son **type**)
- des exemples

# Fonctions: SPECIFICATION et REALISATION

Il est très important de distinguer 2 concepts/étapes lors de la définition d'une fonctions (et d'un programme en général)

## Spécification (Le QUOI):

Une description de ce qui est attendu

- à un certain niveau d'abstraction
- doit être suffisamment précis
- proche d'une description "mathématique"
- peut utiliser des exemples *pertinents*

Un contrat

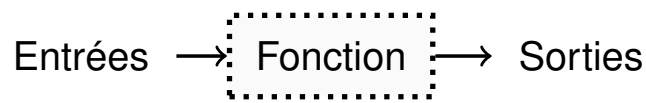

Consiste en:

- 1 description =sémantique
- 1 signature = profil = (son **type**)
- des exemples

## Réalisation (Le COMMENT):

La description de comment le réaliser

- Algorithme: une description en Français de l'algorithme utilisé.
- Implémentation: le code OCaml



# Fonctions: SPECIFICATION et REALISATION

Il est très important de distinguer 2 concepts/étapes lors de la définition d'une fonctions (et d'un programme en général)

## Spécification (Le QUOI):

Une description de ce qui est attendu

- à un certain niveau d'abstraction
- doit être suffisamment précis
- proche d'une description "mathématique"
- peut utiliser des exemples *pertinents*

Un contrat

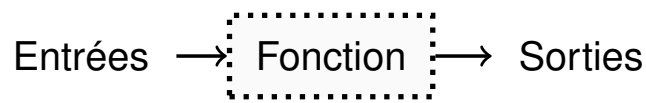

Consiste en:

- 1 description =sémantique
- 1 signature = profil = (son **type**)
- des exemples

## Réalisation (Le COMMENT):

La description de comment le réaliser

- Algorithme: une description en Français de l'algorithme utilisé.
- Implémentation: le code OCaml




Définir une fonction : Spécification PUIS Réalisation

# Plan

Retour sur les fonctions

Récursivité

Terminaison

Fonctions mutuellement récursives

Types Récursifs

Conclusion

# Pourquoi la récursivité ?

## Exemples de pbs que l'on ne sait pas encore résoudre en OCaml

- calculer la moyenne de $n$ entiers lus dans un fichier
- calculer les nombres premiers compris entre 1 et $n$
- définir une image composée de $n$ figures géomètriques
- définir un type polynôme

$\rightarrow$ **ce qu'il nous manque ?**

# Pourquoi la récursivité ?

## Exemples de pbs que l'on ne sait pas encore résoudre en OCaml

- calculer la moyenne de $n$ entiers lus dans un fichier
- calculer les nombres premiers compris entre 1 et $n$
- définir une image composée de $n$ figures géomètriques
- définir un type polynôme

$\rightarrow$ **ce qu'il nous manque ?**

1. décrire des exécutions comportant un nombre variable de calculs
2. définir des objets de taille importante/variable

# Pourquoi la récursivité ?

## Exemples de pbs que l'on ne sait pas encore résoudre en OCaml

- calculer la moyenne de $n$ entiers lus dans un fichier
- calculer les nombres premiers compris entre 1 et $n$
- définir une image composée de $n$ figures géomètriques
- définir un type polynôme

$\rightarrow$ **ce qu'il nous manque ?**

1. décrire des exécutions comportant un nombre variable de calculs
2. définir des objets de taille importante/variable

En programmation impérative (langages Python [INF101], C [INF203]) :

instruction itérative (`while`, `for`) et tableaux

En programmation fonctionnelle :

la récursivité . . .

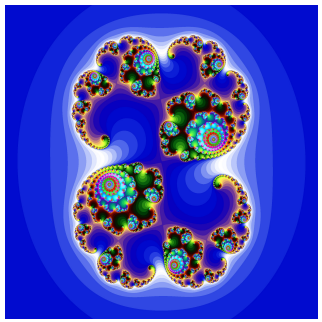
# A propos de récursivité

Qu'est-ce que la récursivité, qu'est-ce qu'une définition récursive ?

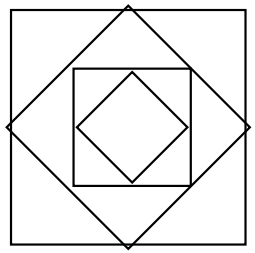
# A propos de récursivité

Qu'est-ce que la récursivité, qu'est-ce qu'une définition récursive ?

**Exemple : Quelques objets récursifs**



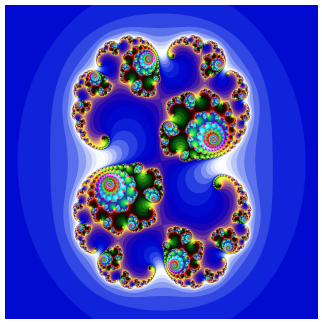
$$u_{n+1} = F(u_n)$$



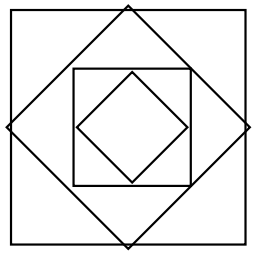
# A propos de récursivité

Qu'est-ce que la récursivité, qu'est-ce qu'une définition récursive ?

**Exemple : Quelques objets récursifs**



$$u_{n+1} = F(u_n)$$





## Fonctions récursives

- la valeur du résultat est obtenue en exécutant plusieurs fois une même fonction sur des données différentes :

$$f(f(f(...f(x_0)...)))$$

- généralisation des suites récurrentes
- un élément de base de la programmation fonctionnelle . . .
- et beaucoup (beaucoup) d'autres applications en informatique !

# Fonctions récursives en OCaml

## Un 1er exemple

**Exemple : Définition récursive de la factorielle**

$$\begin{cases} U_0 = 1 \\ U_n = n \times U_{n-1}, n \geq 1 \end{cases} \qquad \begin{cases} 0! = 1 \\ n! = n \times (n-1)!, n \geq 1 \end{cases}$$

- Cette définition fournit un résultat pour tout entier $\geq 0$ :
  $\rightarrow$ elle est bien fondée ...
  contre-exemple : $U_0 = 1$ et $U_n = n \times U_{n+1}$, $n \geq 1$
- On peut montrer sa correction par récurrence sur $\mathbb{N}$ :

$$\forall n.\ U_n = n!$$

# Fonctions récursives en OCaml

## Un 1er exemple

### Exemple : Définition récursive de la factorielle

$$\begin{cases} U_0 = 1 \\ U_n = n \times U_{n-1}, n \geq 1 \end{cases} \qquad \begin{cases} 0! = 1 \\ n! = n \times (n-1)!, n \geq 1 \end{cases}$$

- Cette définition fournit un résultat pour tout entier $\geq 0$ :
  $\rightarrow$ elle est bien fondée ...
  contre-exemple : $U_0 = 1$ et $U_n = n \times U_{n+1}$, $n \geq 1$
- On peut montrer sa correction par récurrence sur $\mathbb{N}$ :

$$\forall n.\ U_n = n!$$

### Exemple : Fonction factorielle en OCaml

```
let rec fact (n:int):int =
  if n=0 then 1
  else n * fact(n−1)
```

```
let rec fact (n:int):int =
  match n with
    0 → 1
    | n → n * fact(n−1)
```

# Définir une fonction récursive

Spécification: description, signature, exemples,

Réalisation: équations de récurrence et implémentation

## Implémentation: code de la fonction en OCaml

```
let rec fct_name (p1:t1) (p2:t2) ... (pn:tn):t = expr
```

où `expr` peut contenir zéro, un ou plusieurs appels à `fct_name`.
On distinguera différentes sous-expressions de `expr` :

- les cas de base : aucun appel à `fct_name`
- les cas récursifs : un ou plusieurs appel(s) à `fct_name`

# Définir une fonction récursive

Spécification: description, signature, exemples,

Réalisation: équations de récurrence et implémentation

## Implémentation: code de la fonction en OCaml

```
let rec fct_name (p1:t1) (p2:t2) ... (pn:tn):t = expr
```

où `expr` peut contenir zéro, un ou plusieurs appels à `fct_name`.
On distinguera différentes sous-expressions de `expr` :

- les cas de base : aucun appel à `fct_name`
- les cas récursifs : un ou plusieurs appel(s) à `fct_name`

Les règles de typage sont les mêmes que pour le fonctions non récursives.

**Remarque**

- `t1`, .., `tn` peuvent être des types quelconques
- Une fonction récursive ne peut pas être anonyme

# Exemples de définitions de fonctions récursives

**Exemple : Somme des entiers de 0 à $n$**

description + profil + exemples

$$\begin{cases} \sum_{i=0}^{0} i & = 0 \\ \sum_{i=0}^{n} i & = n + \sum_{i=0}^{n-1} i \quad \text{si } n > 0 \end{cases}$$

# Exemples de définitions de fonctions récursives

**Exemple : Somme des entiers de `0` à *n***

description + profil + exemples

$$\begin{cases} \sum_{i=0}^{0} i & = 0 \\ \sum_{i=0}^{n} i & = n + \sum_{i=0}^{n-1} i \quad \text{si } n > 0 \end{cases}$$

```
let rec sum (n : int) : int =
 match n with
  | 0 → 0
  | n → n + sum (n − 1)
```

# Exemples de définitions de fonctions récursives

**Exemple : Somme des entiers de 0 à $n$**

description + profil + exemples

$$\begin{cases} \sum_{i=0}^{0} i & = 0 \\ \sum_{i=0}^{n} i & = n + \sum_{i=0}^{n-1} i \quad \text{si } n > 0 \end{cases}$$

```
let rec sum (n : int) : int =
 match n with
  | 0 → 0
  | n → n + sum (n − 1)
```

**Exemple : Division entière**

description + profil + exemples

$$a/b = \begin{cases} 0 & \text{si } a < b \\ 1 + (a-b)/b & \text{si } b \leq a \end{cases}$$

# Exemples de définitions de fonctions récursives

**Exemple : Somme des entiers de 0 à $n$**

description + profil + exemples

$$\begin{cases} \sum_{i=0}^{0} i & = 0 \\ \sum_{i=0}^{n} i & = n + \sum_{i=0}^{n-1} i \quad \text{si } n > 0 \end{cases}$$

```
let rec sum (n : int) : int =
 match n with
  | 0 → 0
  | n → n + sum (n − 1)
```

**Exemple : Division entière**

description + profil + exemples

$$a/b = \begin{cases} 0 & \text{si } a < b \\ 1 + (a-b)/b & \text{si } b \leq a \end{cases}$$

```
let rec div (a : int) (b: int) : int =
 if a < b then 0
  else 1 + div (a − b) (b)
```

# Essayons . . .

## Exercice : reste de la division entière

Définir une fonction qui calcule le reste de la division entière

## Exercice : suite de Fibonacci

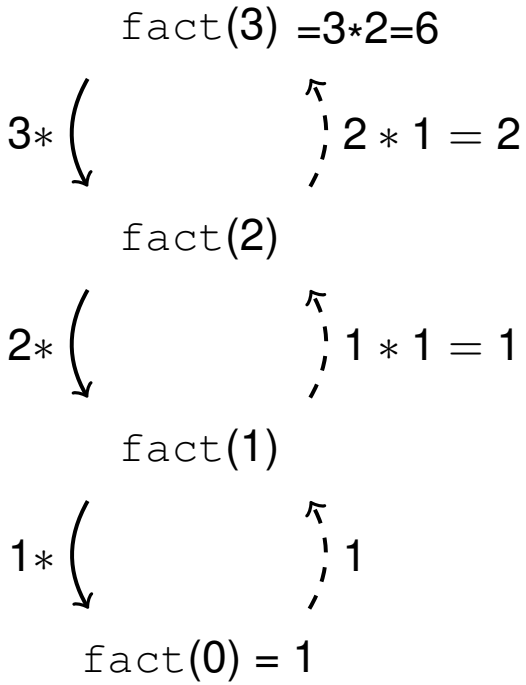
Implémenter une fonction qui renvoie le $n^{ieme}$ terme de Fibonacci où $n$ est donné comme paramètre. La suite de Fibonnaci est défini comme :

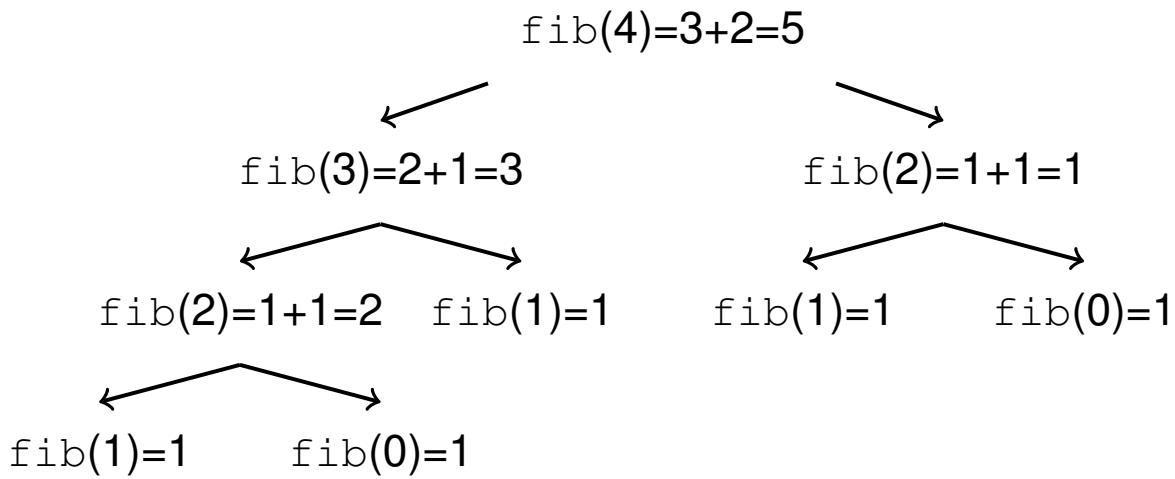
$$fib_n = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ fib_{n-1} + fib_{n-2} & \text{si } n > 1 \end{cases}$$

# Appel et exécution d'une fonction récursive

"Déplier le corps de la fonction" $\rightarrow$ ré-écriture

Exemple : arbre des appels des fonctions `factorielle` et `fibonacci`



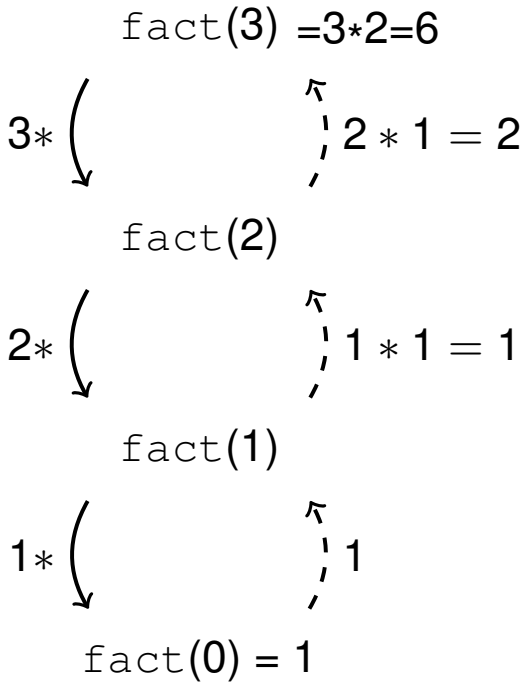



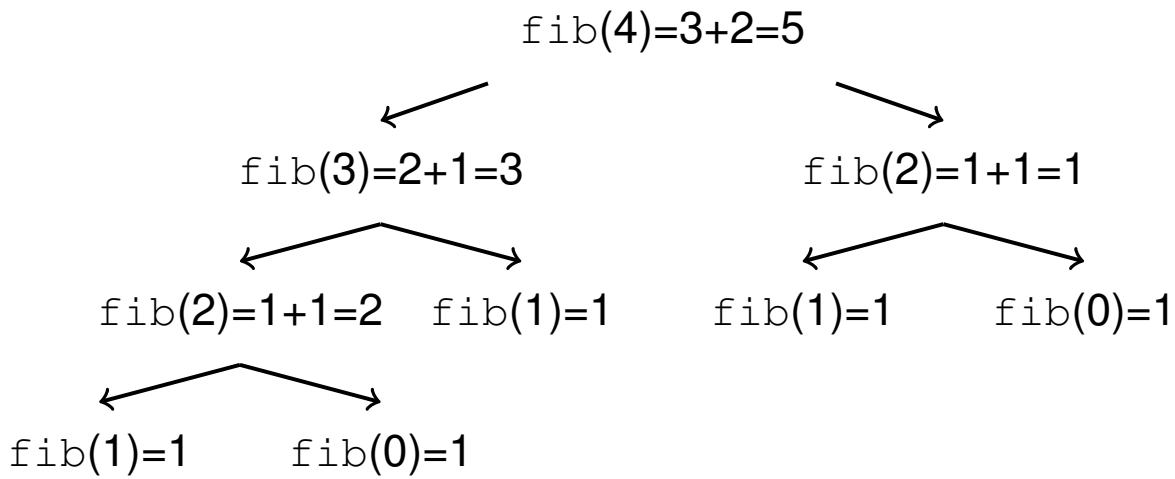

- $\longrightarrow$: ré-écriture des appels générés et suspension des opérations en cours
- $\dashrightarrow$: évaluation (en ordre inverse) des opérations suspendues

# Appel et exécution d'une fonction récursive

"Déplier le corps de la fonction" $\rightarrow$ ré-écriture

Exemple : arbre des appels des fonctions `factorielle` et `fibonacci`




- $\longrightarrow$: ré-écriture des appels générés et suspension des opérations en cours
- $\dashrightarrow$: évaluation (en ordre inverse) des opérations suspendues

En OCaml: directive \#`trace`

DEMO: Tracer une fonction

## Exercice: la fonction puissance (2 versions)

$$\begin{cases} x^0 & = 1 \\ x^n & = x * x^{n-1} \quad \text{si } 0 < n \end{cases} \qquad \begin{cases} x^0 & = 1 \\ x^n & = (x * x)^{n/2} \quad \text{si } n \text{ est pair} \\ x^n & = x * (x * x)^{\frac{n-1}{2}} \quad \text{si } n \text{ est impair} \end{cases}$$

- Donnez 2 implémentations de la fonction `power`: `int` $\rightarrow$ `int` $\rightarrow$ `int` en vous basant sur ces 2 définitions équivalentes.
- Quelle est la différence entre ces deux versions ?

## Exercice: la fonction puissance (2 versions)

$$\begin{cases} x^0 & = 1 \\ x^n & = x * x^{n-1} \quad \text{si } 0 < n \end{cases} \qquad \begin{cases} x^0 & = 1 \\ x^n & = (x * x)^{n/2} \quad \text{si } n \text{ est pair} \\ x^n & = x * (x * x)^{\frac{n-1}{2}} \quad \text{si } n \text{ est impair} \end{cases}$$

- Donnez 2 implémentations de la fonction `power`: `int` $\rightarrow$ `int` $\rightarrow$ `int` en vous basant sur ces 2 définitions équivalentes.
- Quelle est la différence entre ces deux versions ?

# Plan

Retour sur les fonctions

Récursivité

Terminaison

Fonctions mutuellement récursives

Types Récursifs

Conclusion

# Terminaison

Pensez vous que l'exécution de cette fonction termine ? (Fonction 91 de McCarthy (cf. wikipedia))

$$mac(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ mac(mac(n + 11)) & \text{si } n \leq 100 \end{cases}$$

# Terminaison

Pensez vous que l'exécution de cette fonction termine ? (Fonction 91 de McCarthy (cf. wikipedia))

$$mac(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ mac(mac(n + 11)) & \text{si } n \leq 100 \end{cases}$$

Et celles-ci ?

La fonction puissance

$$\begin{cases} x^0 & = 1 \\ x^n & = x * x^{n-1} \quad \text{si } 0 < n \end{cases}$$

La fonction factorielle

$$\begin{cases} fact(0) & 1 \\ fact(1) & = 1 \\ fact(n) & = \frac{fact(n+1)}{n+1} \end{cases}$$

# Terminaison

Pensez vous que l'exécution de cette fonction termine ? (Fonction 91 de McCarthy (cf. wikipedia))

$$mac(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ mac(mac(n + 11)) & \text{si } n \leq 100 \end{cases}$$

Et celles-ci ?

La fonction puissance

$$\begin{cases} x^0 & = 1 \\ x^n & = x * x^{n-1} \quad \text{si } 0 < n \end{cases}$$

La fonction factorielle

$$\begin{cases} fact(0) & 1 \\ fact(1) & = 1 \\ fact(n) & = \frac{fact(n+1)}{n+1} \end{cases}$$

Il est **fondamental** de savoir décider si une fonction termine ou non

# Terminaison

Pensez vous que l'exécution de cette fonction termine ? (Fonction 91 de McCarthy (cf. wikipedia))

$$mac(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ mac(mac(n + 11)) & \text{si } n \leq 100 \end{cases}$$

Et celles-ci ?

La fonction puissance

$$\begin{cases} x^0 & = 1 \\ x^n & = x * x^{n-1} \quad \text{si } 0 < n \end{cases}$$

La fonction factorielle

$$\begin{cases} fact(0) & 1 \\ fact(1) & = 1 \\ fact(n) & = \frac{fact(n+1)}{n+1} \end{cases}$$

Il est **fondamental** de savoir décider si une fonction termine ou non

Il n'existe pas (et il ne peut pas exister) de programme qui dit quels fonctions terminent.

# Terminaison

Pensez vous que l'exécution de cette fonction termine ? (Fonction 91 de McCarthy (cf. wikipedia))

$$mac(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ mac(mac(n + 11)) & \text{si } n \leq 100 \end{cases}$$

Et celles-ci ?

La fonction puissance

$$\begin{cases} x^0 & = 1 \\ x^n & = x * x^{n-1} \quad \text{si } 0 < n \end{cases}$$

La fonction factorielle

$$\begin{cases} fact(0) & 1 \\ fact(1) & = 1 \\ fact(n) & = \frac{fact(n+1)}{n+1} \end{cases}$$

Il est **fondamental** de savoir décider si une fonction termine ou non

Il n'existe pas (et il ne peut pas exister) de programme qui dit quels fonctions terminent.

Il faut donc trouver soit même une preuve de terminaison.

# Terminaison

Pensez vous que l'exécution de cette fonction termine ? (Fonction 91 de McCarthy (cf. wikipedia))

$$mac(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ mac(mac(n + 11)) & \text{si } n \leq 100 \end{cases}$$

Et celles-ci ?

La fonction puissance

$$\begin{cases} x^0 & = 1 \\ x^n & = x * x^{n-1} \quad \text{si } 0 < n \end{cases}$$

La fonction factorielle

$$\begin{cases} fact(0) & 1 \\ fact(1) & = 1 \\ fact(n) & = \frac{fact(n+1)}{n+1} \end{cases}$$

Il est **fondamental** de savoir décider si une fonction termine ou non

Il n'existe pas (et il ne peut pas exister) de programme qui dit quels fonctions terminent.

Il faut donc trouver soit même une preuve de terminaison.

Ceci est valable pour tous les langages de programmation.

# Comment prouver qu'une fonction récursive termine ?

## En utilisant une mesure . . .

### Theorem

*Toute suite positive strictement décroissante converge*

# Comment prouver qu'une fonction récursive termine ?

## En utilisant une mesure ...

### Theorem

*Toute suite positive strictement décroissante converge*

### Méthode générale pour prouver qu'une fonction *f* termine ?

Trouver une mesure $\mathcal{M}$ t.q. :

- $\mathcal{M}$ a les mêmes paramètres que la fonction *f*.
- $\mathcal{M}$ décroit strictement entre deux appels récursifs
- $\mathcal{M}$ est positive

# Comment prouver qu'une fonction récursive termine ?

En utilisant une mesure ...

## Theorem

*Toute suite positive strictement décroissante converge*

## Méthode générale pour prouver qu'une fonction *f* termine ?

Trouver une mesure $\mathcal{M}$ t.q. :

- $\mathcal{M}$ a les mêmes paramètres que la fonction *f*.
- $\mathcal{M}$ décroit strictement entre deux appels récursifs
- $\mathcal{M}$ est positive

## Exemple : Terminaison de la fonction `sum`

```
let rec sum (x : int) : int =
  match x with
   | 0 → 0
   | x → x + sum (x − 1)
```

# Comment prouver qu'une fonction récursive termine ?

En utilisant une mesure ...

## Theorem

*Toute suite positive strictement décroissante converge*

## Méthode générale pour prouver qu'une fonction *f* termine ?

Trouver une mesure $\mathcal{M}$ t.q. :

- $\mathcal{M}$ a les mêmes paramètres que la fonction *f*.
- $\mathcal{M}$ décroit strictement entre deux appels récursifs
- $\mathcal{M}$ est positive

## Exemple : Terminaison de la fonction `sum`

```
let rec sum (x : int) : int =
  match x with
   | 0 → 0
   | x → x + sum (x − 1)
```

Mesure :

- $\mathcal{M}(n) = n \geq 0$, $\mathcal{M}(n) \in \mathbb{N}$
- $\mathcal{M}(n) > \mathcal{M}(n-1)$ puisque $n > n-1$

# Terminaison de quelques fonctions

## Exercice: trouver les *mesures*

Prouvez que les fonctions `factorielle`, `puissance`, `quotient`, `reste` terminent . . .

# Terminaison de quelques fonctions

## factorielle et puissance

Terminaison de `fact`:

```
let rec fact (n:int):int =
  match n with
    0 → 1
    | n → n * fact(n−1)
```

# Terminaison de quelques fonctions

## factorielle et puissance

Terminaison de `fact`:

```
let rec fact (n:int):int =
  match n with
    0 → 1
    | n → n * fact(n−1)
```

- Définissons $\mathcal{M}(n) = n \in \mathbb{N}$
- $\mathcal{M}(n) > \mathcal{M}(n-1)$
  puisque $n > n-1$

# Terminaison de quelques fonctions

## factorielle et puissance

### Terminaison de `fact`:

```
let rec fact (n:int):int =
  match n with
    0 → 1
    | n → n * fact(n−1)
```

- Définissons $\mathcal{M}(n) = n \in \mathbb{N}$
- $\mathcal{M}(n) > \mathcal{M}(n-1)$ puisque $n > n-1$

### Terminaison de `power`:

```
let rec power (a:float) (n:int):float =
   if (n=0) then 1.
  else (if n>0 then a *. power a (n−1)
  else 1. /. (power a (−n)))
```

# Terminaison de quelques fonctions

## factorielle et puissance

Terminaison de `fact`:

```
let rec fact (n:int):int =
  match n with
    0 → 1
    | n → n * fact(n−1)
```

- Définissons $\mathcal{M}(n) = n \in \mathbb{N}$
- $\mathcal{M}(n) > \mathcal{M}(n-1)$ puisque $n > n-1$

Terminaison de `power`:

```
let rec power (a:float) (n:int):float =
  if (n=0) then 1.
 else (if n>0 then a *. power a (n−1)
 else 1. /. (power a (−n)))
```

- Définissons $(\mathcal{M}\ a\ n) = n$ si $n \geq 0$ et $(\mathcal{M}\ a\ n) = -n+1$ sinon
- $(\mathcal{M}\ a\ n) \in \mathbb{N}$
- Pour tout appel récursif avec $n > 0$, $(\mathcal{M}\ a\ n) = n > (\mathcal{M}\ a\ (n-1)) = n-1$
- Pour tout appel récursif avec $n < 0$, $(\mathcal{M}\ a\ n) = -n+1 > (\mathcal{M}\ a\ (-n)) = -n$

# Terminaison de quelques fonctions

```
let rec quotient (a:int) (b:int):int =
 if (a<b) then 0
 else 1 + quotient (a−b) b
```

```
let rec reste (a:int) (b:int):int =
  if (a<b) then a
  else reste (a−b) b
```

# Terminaison de quelques fonctions

```
let rec quotient (a:int) (b:int):int =
 if (a<b) then 0
 else 1 + quotient (a−b) b
```

```
let rec reste (a:int) (b:int):int =
  if (a<b) then a
  else reste (a−b) b
```

Terminaison de `quotient` et `reste`:

- Définissons $\mathcal{M}\ a\ b = a$
- $\mathcal{M}\ a\ b \in \mathbb{N}$ (d'après la spec)
- $\mathcal{M}\ a\ b > \mathcal{M}\ (a - b)\ b$ puisque $b > 0$

# Plan

Retour sur les fonctions

Récursivité

Terminaison

Fonctions mutuellement récursives

Types Récursifs

Conclusion

# Fonctions *mutuellement* récursives

## Sur un exemple

Récursivité “directe” : appels récursifs à une seule fonction
Qu’en est-il d’une fonction `f` qui appelle `g` qui appelle `f` qui appelle `g` . . .
$\hookrightarrow$ fonctions mutuellement récursives (récursivité croisée)

# Fonctions *mutuellement* récursives

## Sur un exemple

Récursivité “directe” : appels récursifs à une seule fonction
Qu’en est-il d’une fonction `f` qui appelle `g` qui appelle `f` qui appelle `g` . . .
$\hookrightarrow$ fonctions mutuellement récursives (récursivité croisée)

**Exemple :**
Comment déterminer si un entier est pair ou impair sans utiliser /, `*`, `mod`
(donc en utilisant uniquement $-$ et =) ?

# Fonctions *mutuellement* récursives

## Sur un exemple

Récursivité “directe” : appels récursifs à une seule fonction
Qu’en est-il d’une fonction `f` qui appelle `g` qui appelle `f` qui appelle `g` . . .
$\hookrightarrow$ fonctions mutuellement récursives (récursivité croisée)

**Exemple :**
Comment déterminer si un entier est pair ou impair sans utiliser /, `*`, `mod` (donc en utilisant uniquement $-$ et =) ?

- $n \in \mathbb{N}$ est impair si $n - 1$ est pair
- $n \in \mathbb{N}$ est pair si $n - 1$ est impair
- 0 est pair
- 0 n’est pas impair

# Fonctions *mutuellement* récursives

## Sur un exemple

Récursivité “directe” : appels récursifs à une seule fonction
Qu’en est-il d’une fonction `f` qui appelle `g` qui appelle `f` qui appelle `g` . . .
$\hookrightarrow$ fonctions mutuellement récursives (récursivité croisée)

**Exemple :**
Comment déterminer si un entier est pair ou impair sans utiliser /, `*`, `mod` (donc en utilisant uniquement $-$ et =) ?

- $n \in \mathbb{N}$ est impair si $n - 1$ est pair
- $n \in \mathbb{N}$ est pair si $n - 1$ est impair
- 0 est pair
- 0 n’est pas impair

```
let rec
  pair (n:int):bool = if n=0 then true else impair (n−1)
  and
  impair (m:int):bool = if m=0 then false else pair (m−1)
```

DEMO: pair et impair, récursivité croisée

# Fonctions mutuellement récursives

## Généralisation

```
let rec fct1 [parametres+type resultat] = expr_1
  and fct2 [parametres+ type resultat] = expr_2
     ....
  and fctn [parametres+type resultat] = expr_n
```

où

`expr_1`, `expr_2`, ..., `expr_n` peuvent appeler `fct1`, `fct2`, ..., `fctn`

# Plan

Retour sur les fonctions

Récursivité

Terminaison

Fonctions mutuellement récursives

Types Récursifs

Conclusion

# Types récursifs : pour faire quoi ?

## fonction récursive

- définie *en "fonction d'elle-même"* (cas de base, cas récursifs)
- permet de décrire des **suites de calcul** de longueur arbitraire
  **ex :** (`fact` 5), (`fact` 10), etc.
- problème de **terminaison**

## Type récursif

- défini *en "fonction de lui-même" . . .* (cas de base, cas récursifs)
- permet de décrire des **données** de taille arbitraire
- problème de **terminaison** : type "bien fondés"

## Exemples d'application :

définir des ensembles, des séquences, des arborescences . . .

# Types récursifs : définition et exemple

## Exemple :

```
type t =
     C of char (* constructeur non recursif *)
    |S of int * t   (* constructeur recursif *)
```

Exemple de valeurs dy type `t` ?

# Types récursifs : définition et exemple

## Exemple :

```
type t =
     C of char (* constructeur non recursif *)
    |S of int * t   (* constructeur recursif *)
```

Exemple de valeurs dy type `t` ?

`C('x')`

# Types récursifs : définition et exemple

## Exemple :

```
type t =
     C of char (* constructeur non recursif *)
    |S of int * t   (* constructeur recursif *)
```

Exemple de valeurs dy type `t` ?

`C('x')`

# Types récursifs : définition et exemple

## Exemple :

```
type t =
     C of char (* constructeur non recursif *)
    |S of int * t   (* constructeur recursif *)
```

Exemple de valeurs dy type `t` ?

`C('x')`  `S(5, C('x'))`

# Types récursifs : définition et exemple

## Exemple :

```
type t =
     C of char (* constructeur non recursif *)
    |S of int * t   (* constructeur recursif *)
```

Exemple de valeurs dy type `t` ?

`C`('x') `S`(5, `C`('x')) `S`(12, `S`(5, `C`('x'))) etc.

# Types récursifs : définition et exemple

## Exemple :

```
type t =
      C of char (* constructeur non recursif *)
     |S of int * t   (* constructeur recursif *)
```

Exemple de valeurs dy type `t` ?

`C`('x') `S`(5, `C`('x')) `S`(12, `S`(5, `C`('x'))) etc.

$\rightarrow$ séquence d'entiers terminée par un caractère . . .

## Définition générale

`type nouveau_type` = ... `nouveau_type` ...

Pour être "bien fondé", `nouveau_type` doit être :

- un type **somme**
- avec au moins un constructeur non récursif

DEMO: exemples de définition de types réursifs (bien fondés ou non)

# Un type récursif : les entiers de Peano

## le point de vue mathématique et le point de vue OCaml

Les entiers de Peano (NatPeano) : une manière de définir $\mathbb{N}$

Définition récursive de NatPeano:

- une base : le constructeur “non récursif” Zero
- un *constructeur* “récursif”:
  Suc: le successeur d’un élément de NatPeano
- Zero est le successeur d’aucun élément de NatPeano
- deux élément de NatPeano qui ont même successeur sont égaux

$\hookrightarrow \mathbb{N}$ est le plus petit ensemble contenant Zero et le successeur de tout élément de $\mathbb{N}$

# Un type récursif : les entiers de Peano

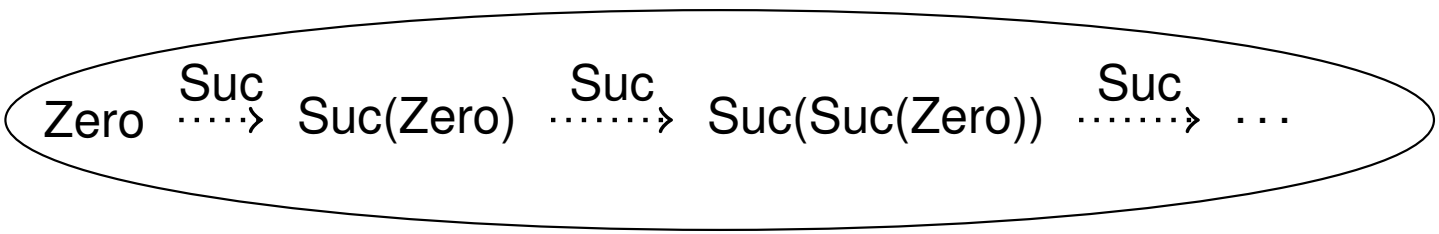
## le point de vue mathématique et le point de vue OCaml

Les entiers de Peano (NatPeano) : une manière de définir $\mathbb{N}$

Définition récursive de NatPeano:

- une base : le constructeur “non récursif” Zero
- un *constructeur* “récursif”:
  Suc: le successeur d’un élément de NatPeano
- Zero est le successeur d’aucun élément de NatPeano
- deux élément de NatPeano qui ont même successeur sont égaux

$\hookrightarrow \mathbb{N}$ est le plus petit ensemble contenant Zero et le successeur de tout élément de $\mathbb{N}$




Définition de NatPeano en OCaml:

```
type natPeano = Zero | Suc of natPeano
```

$\hookrightarrow$ `natPeano` est **un type récursif**

# Entiers de Peano

## Conversion vers/depuis les entiers

### Convertir un entier de Peano en entier

- Description: `natPeano2int` traduit un entier de Peano en son équivalent entier
- Profil/Signature: `natPeano2int`: `natPeano` $\rightarrow$ `int`
- Ex.: `natPeano2int Zero` = 0, `natPeano2int Suc(Suc(Suc Zero))`=3

# Entiers de Peano

## Conversion vers/depuis les entiers

### Convertir un entier de Peano en entier

- Description: `natPeano2int` traduit un entier de Peano en son équivalent entier
- Profil/Signature: `natPeano2int`: `natPeano` → `int`
- Ex.: `natPeano2int Zero` = 0, `natPeano2int Suc(Suc(Suc Zero))`=3

```
let rec natPeano2int (n:natPeano):int =
 match n with
    Zero → 0
   | Suc (nprime) → 1+ natPeano2int nprime
```

# Entiers de Peano

## Conversion vers/depuis les entiers

### Convertir un entier de Peano en entier

- Description: `natPeano2int` traduit un entier de Peano en son équivalent entier
- Profil/Signature: `natPeano2int`: `natPeano` → `int`
- Ex.: `natPeano2int Zero` = 0, `natPeano2int Suc(Suc(Suc Zero))`=3

```
let rec natPeano2int (n:natPeano):int =
 match n with
    Zero → 0
   | Suc (nprime) → 1+ natPeano2int nprime
```

### Convertir un entier en entier de Peano

comme ci-dessus, mais dans le sens inverse . . . !

# Entiers de Peano

## Conversion vers/depuis les entiers

### Convertir un entier de Peano en entier

- Description: `natPeano2int` traduit un entier de Peano en son équivalent entier
- Profil/Signature: `natPeano2int`: `natPeano` → `int`
- Ex.: `natPeano2int Zero` = 0, `natPeano2int Suc(Suc(Suc Zero))`=3

```
let rec natPeano2int (n:natPeano):int =
 match n with
    Zero → 0
  | Suc (nprime) → 1+ natPeano2int nprime
```

### Convertir un entier en entier de Peano

comme ci-dessus, mais dans le sens inverse . . . !

```
let rec int2natPeano (n:int):natPeano=
 match n with
   0 → Zero
   | nprime → Suc (int2natPeano (n−1))
```

DEMO: Fonctions natPeano2int et int2natPeano

## Exercice: somme de deux entiers de Peano

- Définir une fonction qui effectue la *somme de deux entiers de Peano* sans utiliser les fonction de conversion depuis/vers les entiers
- Prouver que votre fonction termine

## Exercice: produit de deux entiers de Peano

- Définir une fonction qui *multiplie deux entiers de Peano*
- Prouver que votre fonction termine

## Exercice: factorielle d'un entier de Peano

- Définir une fonction qui *calcule la factorielle d'un entier de Peano*
- Prouver que votre fonction termine

# Conclusion

## La récursivité : une notion fondamentale . . .

On a vu deux formes de récursivité :

- les fonctions récursives
  - équations récursives
  - terminaison
  - définition = spécification (description, profil, équations récursives, exemples)
    + implémentation
    + arguments de terminaison

- les types/valeurs/objets récursifs
  - définition (“bien fondée”)
  - fonctions récursives portant sur des types récursifs :
    $\rightarrow$ construites selon la définition du type récursif