

Cours d'introduction à l'informatique

Partie 4 : Les fonctions. Pourquoi les fonctions ?
Utilisation, déclaration, portée des variables et récursivité...

Motivations

Motivations

Motivations

```
1 // Un calcul...
2 var a, b, c;
3 var u, v, tmp;
4
5 u = 1;
6 v = 3;
7 while (Math.abs(u * u - 3) > 0.0001) {
8     tmp = u;
9     u = (1 / u + 1 / v) / 2;
10    v = (tmp / v) / 2;
11 }
12 a = u;
13
14 u = 1;
15 v = 5;
16 while (Math.abs(u * u - 5) > 0.0001) {
17     tmp = u;
18     u = (1 / u + 1 / v) / 2;
19     v = (tmp / v) / 2;
20 }
21 b = u;
22
23 u = 1;
24 v = 7;
25 while (Math.abs(u * u - 7) > 0.0001) {
26     tmp = u;
27     u = (1 / u + 1 / v) / 2;
28     v = (tmp / v) / 2;
29 }
30 c = u;
31
32 Ecrire(a + b + c);
33
```

Motivations

```
1 // Un calcul...
2 var a, b, c;
3 var u, v, tmp;
4
5 u = 1;
6 v = 3;
7 while (Math.abs(u * u - 3) > 0.0001) {
8     tmp = u;
9     u = (1 / u + 1 / v) / 2;
10    v = (tmp / v) / 2;
11 }
12 a = u;
13
14 u = 1;
15 v = 5;
16 while (Math.abs(u * u - 5) > 0.0001) {
17     tmp = u;
18     u = (1 / u + 1 / v) / 2;
19     v = (tmp / v) / 2;
20 }
21 b = u;
22
23 u = 1;
24 v = 7;
25 while (Math.abs(u * u - 7) > 0.0001) {
26     tmp = u;
27     u = (1 / u + 1 / v) / 2;
28     v = (tmp / v) / 2;
29 }
30 c = u;
31
32 Ecrire(a + b + c);
33
```

```
36 function Racine_Carree(x) {
37     var u, v, tmp;
38     u = 1;
39     v = x;
40     while (Math.abs(u * u - x) > 0.0001) {
41         tmp = u;
42         u = (1 / u + 1 / v) / 2;
43         v = (tmp / v) / 2;
44     }
45     return u
46 }
47
48 Ecrire(Racine_Carree(3) + Racine_Carree(5) + Racine_Carree(7));
49
```

Motivations

QUE FONT CES PROGRAMMES ???

```
6 v = 3;
7 while (Math.abs(u * u - 3) > 0.0001) {
8     tmp = u;
9     u = (1 / u + 1 / v) / 2;
10    v = (tmp / v) / 2;
11 }
12 a = u;
13
14 u = 1;
15 v = 5;
16 while (Math.abs(u * u - 5) > 0.0001) {
17     tmp = u;
18     u = (1 / u + 1 / v) / 2;
19     v = (tmp / v) / 2;
20 }
21 b = u;
22
23 u = 1;
24 v = 7;
25 while (Math.abs(u * u - 7) > 0.0001) {
26     tmp = u;
27     u = (1 / u + 1 / v) / 2;
28     v = (tmp / v) / 2;
29 }
30 c = u;
31
32 Ecrire(a + b + c);
33
```

```
36 function Racine_Carree(x) {
37     var u, v, tmp;
38     u = 1;
39     v = x;
40     while (Math.abs(u * u - x) > 0.0001) {
41         tmp = u;
42         u = (1 / u + 1 / v) / 2;
43         v = (tmp / v) / 2;
44     }
45     return u
46 }
47
48 Ecrire(Racine_Carree(3) + Racine_Carree(5) + Racine_Carree(7));
49
```

Motivations

QUE FONT CES PROGRAMMES ???

LA MÊME CHOSE !!!

```
6 v = 3;
7 while (Math.abs(u * u - 3) > 0.0001) {
14 u = 1;
15 v = 5;
16 while (Math.abs(u * u - 5) > 0.0001) {
17     tmp = u;
18     u = (1 / u + 1 / v) / 2;
19     v = (tmp / v) / 2;
20 }
21 b = u;
22
23 u = 1;
24 v = 7;
25 while (Math.abs(u * u - 7) > 0.0001) {
26     tmp = u;
27     u = (1 / u + 1 / v) / 2;
28     v = (tmp / v) / 2;
29 }
30 c = u;
31
32 Ecrire(a + b + c);
```

```
36 function Racine_Carree(x) {
37     var u, v, tmp;
38     u = 1;
39     v = x;
40     while (Math.abs(u * u - x) > 0.0001) {
41         tmp = u;
42         u = (1 / u + 1 / v) / 2;
43         v = (tmp / v) / 2;
44     }
45     return u
46 }
47
48 Ecrire(Racine_Carree(3) + Racine_Carree(5) + Racine_Carree(7));
49
```

Motivations

QUE FONT CES PROGRAMMES ???

LA MÊME CHOSE !!!

```
6 v = 3;  
7 while (Math.abs(u * u - 3) > 0.0001) {  
14 u = 1;  
15 v = 5;  
16 while (Math.abs(u * u - 5) > 0.0001) {  
17     tmp = u;  
18     u = (1 / u + 1 / v) / 2;  
19     v = (tmp / v) / 2;  
20 }  
21 b = u;  
22  
23 u = 1;  
24 v = 7;  
25 while (Math.abs(u * u - 7) > 0.0001) {  
26     tmp = u;  
27     u = (1 / u + 1 / v) / 2;  
28     v = (tmp / v) / 2;  
29 }  
30 c = u;  
31  
32 Ecrire(a + b + c);  
33
```

```
36 function Racine_Carree(x) {  
37     var u, v, tmp;  
38     u = 1;  
39     v = x;  
40     while (Math.abs(u * u - x) > 0.0001) {  
41         tmp = u;  
42         u = (1 / u + 1 / v) / 2;  
43         v = (tmp / v) / 2;  
44     }  
45     return u  
46 }  
47  
48 Ecrire(Racine_Carree(3) + Racine_Carree(5) + Racine_Carree(7));  
49
```


Motivations

DANS UN PROJET «RAISONNABLE»



Motivations

DANS UN PROJET «RAISONNABLE»



Je veux calculer
 $\sqrt{3} + \sqrt{5} + \sqrt{7}$

Motivations

DANS UN PROJET «RAISONNABLE»



Je programme

```
Ecrire(Racine(3)+Racine(5)+Racine(7));
```

Comment calculer la racine ?

Motivations

DANS UN PROJET «RAISONNABLE»



Dis donc, toi qui est fort en maths, tu pourrais m'écrire une fonction qui calcule la racine carrée ?

Motivations

DANS UN PROJET «RAISONNABLE»



Quelles sont tes spécifications (ce que prend la fonction, ce qu'elle doit donner comme résultat, de quel type) ?

Motivations

DANS UN PROJET «RAISONNABLE»



Je veux une fonction qui étant donné un réel x calcule un nombre réel égal à \sqrt{x}

Motivations

DANS UN PROJET «RAISONNABLE»



Je traduis dans mon langage:
Fonction Racine(x : réel) : réel
// retourne la racine carrée de x

Motivations

DANS UN PROJET «RAISONNABLE»

La voilà !

```
Fonction Racine(x : réel) : réel
// retourne la racine carrée de x
Variables
  u, v, tmp: réels;
Début
  u ← 1; v ← x;
  Tant que (abs(u * u - x) > 0.0001) faire
    tmp ← u;
    u ← (1 / u + 1 / v) / 2;
    v ← (tmp / v) / 2;
  fin tant que
  retourner u;
Fin
```

Tu veux que je t'explique comment ça marche ?



Motivations

DANS UN PROJET «RAISONNABLE»



Ben non si ça fait ce que je t'ai demandé !

Motivations

DANS UN PROJET «RAISONNABLE»



Mais tu sais, il existait aussi une fonction prédéfinie qui faisait le même travail. Elle s'appelle `Math.sqrt...`

Motivations

- C'est pratique d'avoir une fonction `valeur_absolue(x)`, `racine(x)`...
- De ne pas avoir à le recalculer à chaque fois
- On ne peut peut-être pas avoir des fonctions prédéfinies pour tout

Les fonctions sont utiles

- Pour ne pas répéter des calculs laborieux
- Pour ne pas faire des calculs laborieux
- Pour rendre lisibles les algorithmes

Définition

Définition

- C'est quoi une fonction ?
 - Un nom
 - Le type du résultat
 - Les types des arguments
 - Un effet (le lien entre les arguments et le résultat)
- En maths...
 - $\ln : \mathbb{R} \rightarrow \mathbb{R}$ est la fonction qui étant donné un réel x calcule le logarithme (népérien) de x

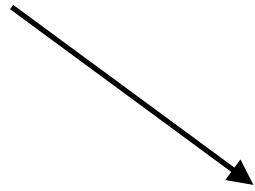
Vocabulaire

$\text{abs} : \mathbb{R} \rightarrow \mathbb{R}$

$$x \mapsto \begin{cases} -x, & \text{si } x < 0 \\ x, & \text{si } x \geq 0 \end{cases}$$

Vocabulaire

nom de la fonction



$\text{abs} : \mathbb{R} \rightarrow \mathbb{R}$

$$x \mapsto \begin{cases} -x, & \text{si } x < 0 \\ x, & \text{si } x \geq 0 \end{cases}$$

Vocabulaire

nom de la fonction

type de l'argument

abs : $\mathbb{R} \rightarrow \mathbb{R}$

$$x \mapsto \begin{cases} -x, & \text{si } x < 0 \\ x, & \text{si } x \geq 0 \end{cases}$$

Vocabulaire

nom de la fonction

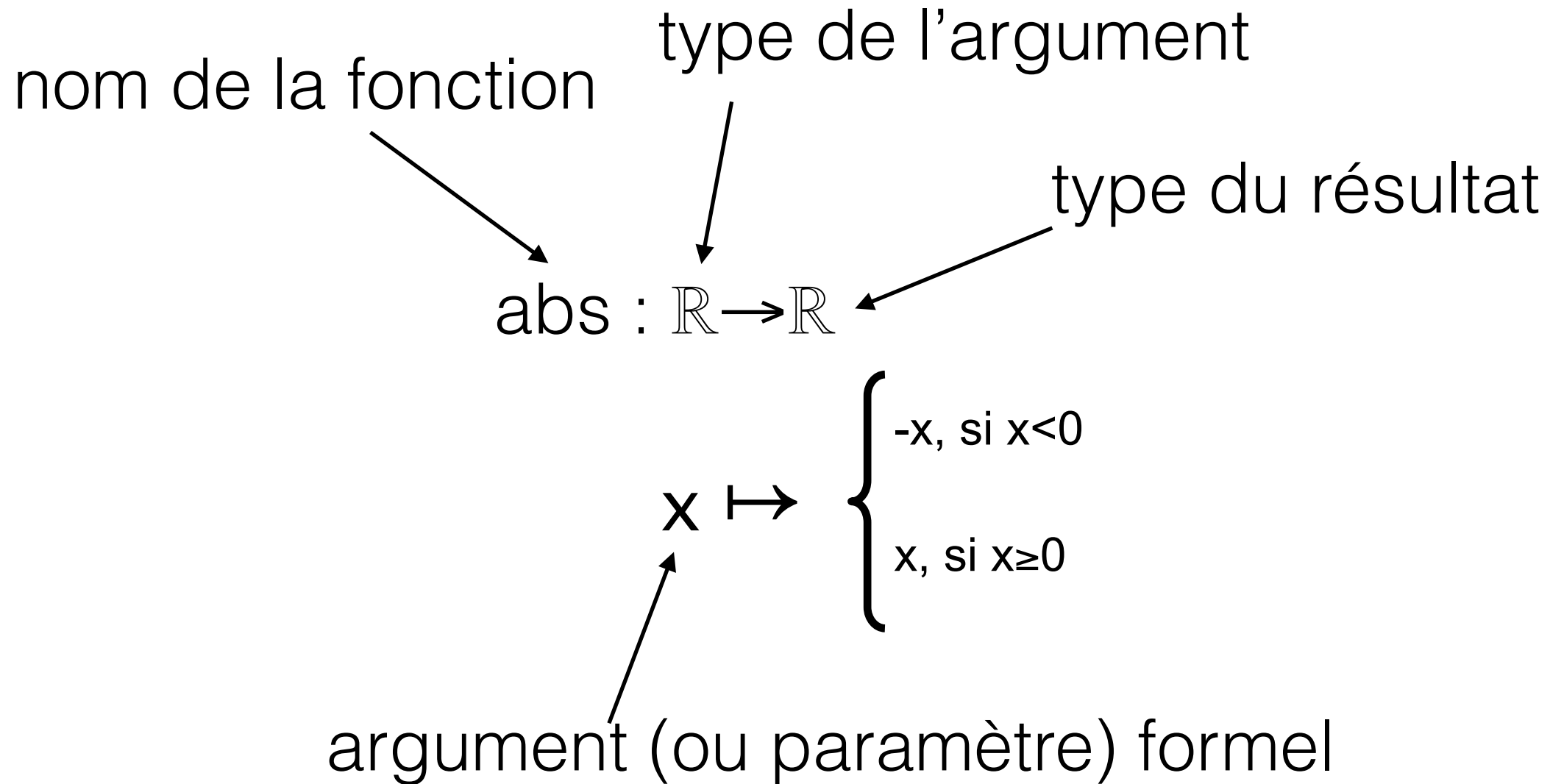
type de l'argument

type du résultat

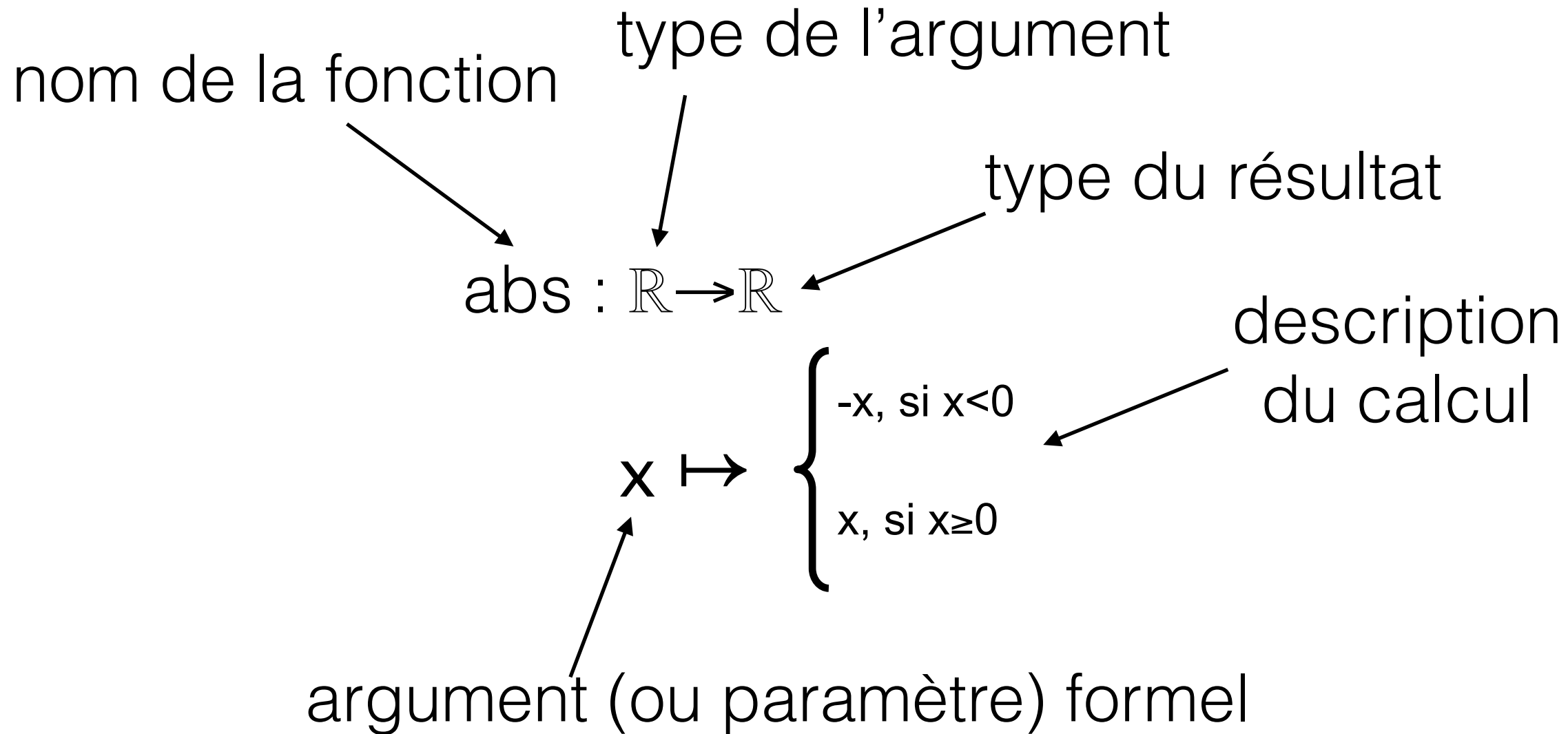
abs : $\mathbb{R} \rightarrow \mathbb{R}$

$$x \mapsto \begin{cases} -x, & \text{si } x < 0 \\ x, & \text{si } x \geq 0 \end{cases}$$

Vocabulaire



Vocabulaire



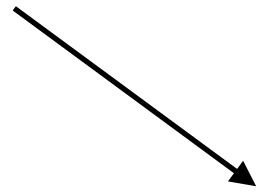
Vocabulaire

Distance : $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$

$(x, y) \mapsto \text{abs}(x-y)$

Vocabulaire

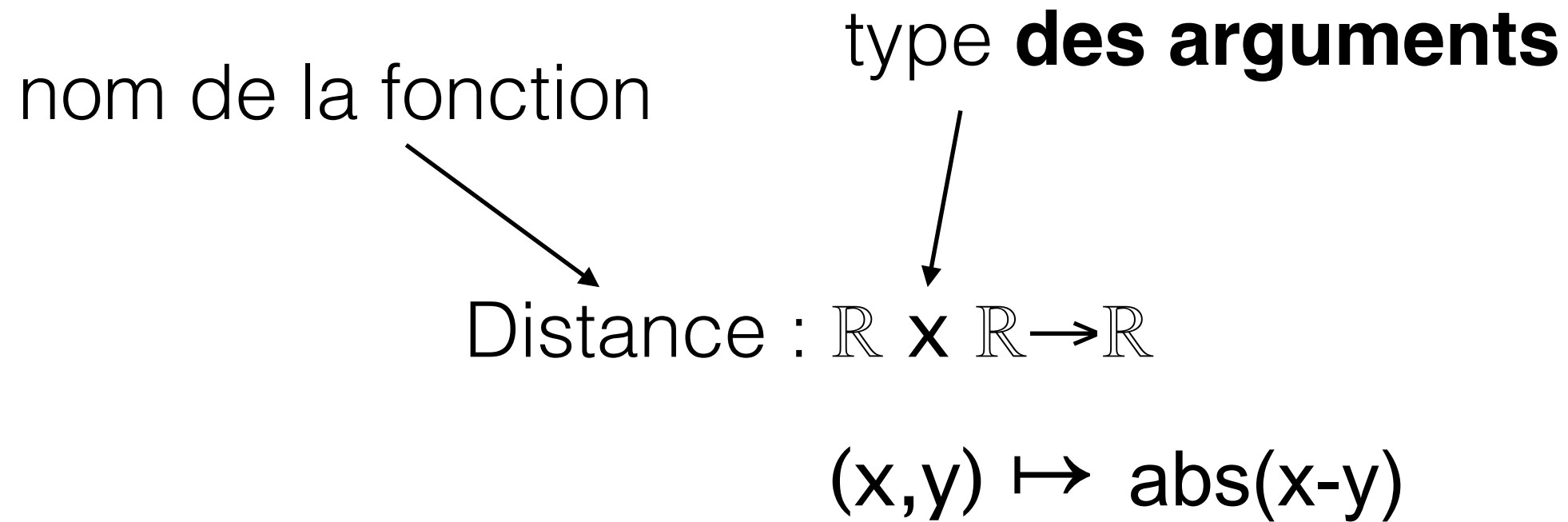
nom de la fonction



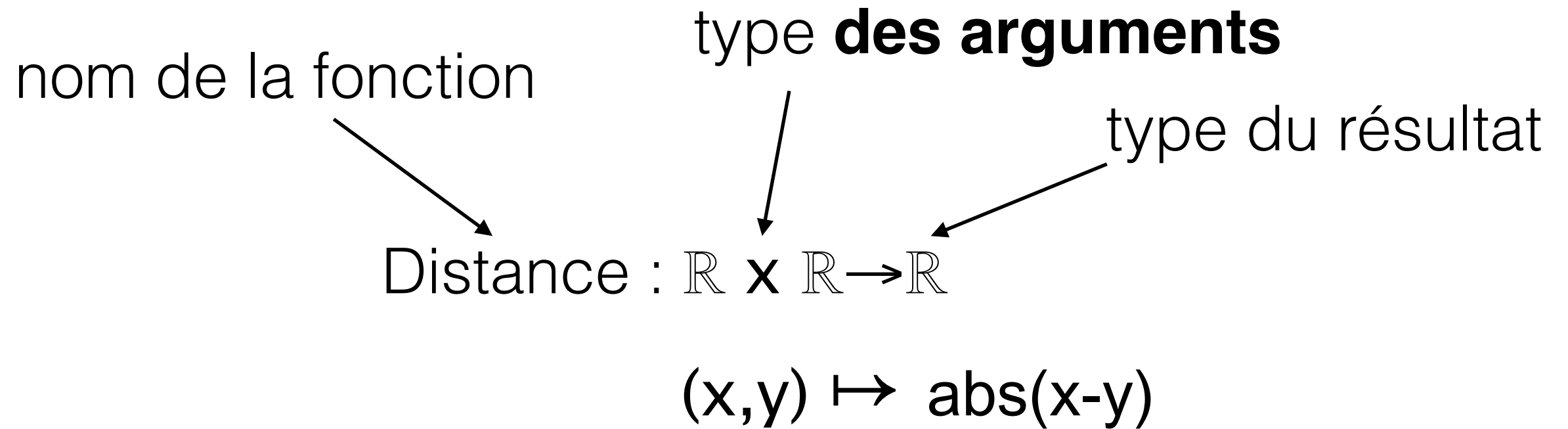
Distance : $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$

$(x, y) \mapsto \text{abs}(x-y)$

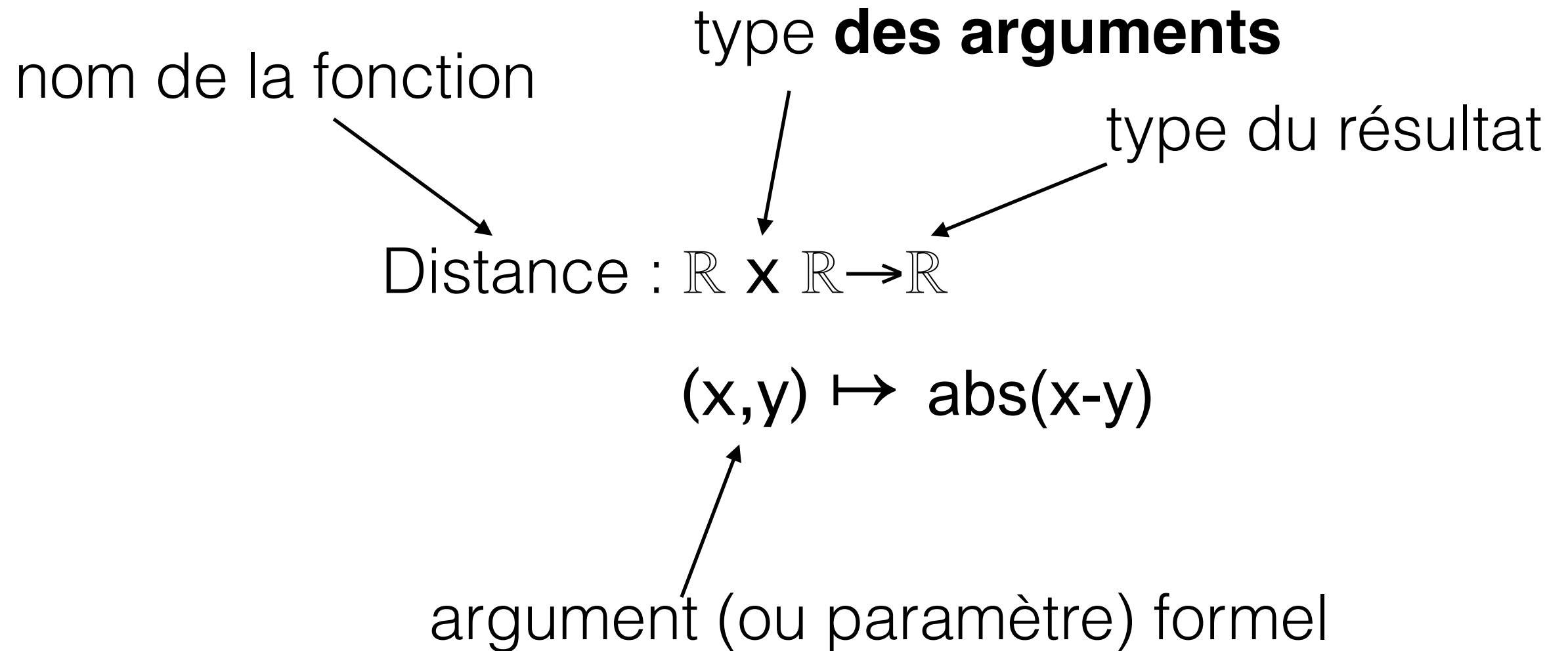
Vocabulaire



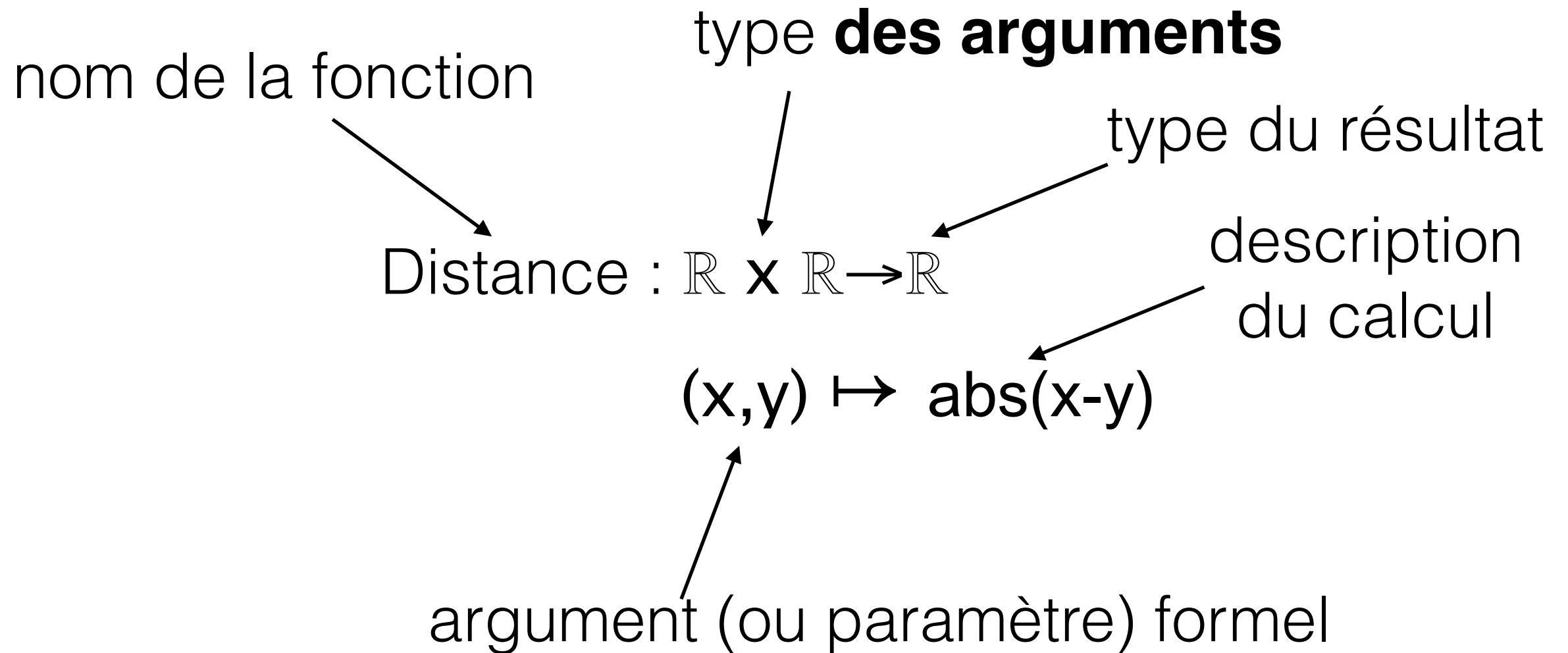
Vocabulaire



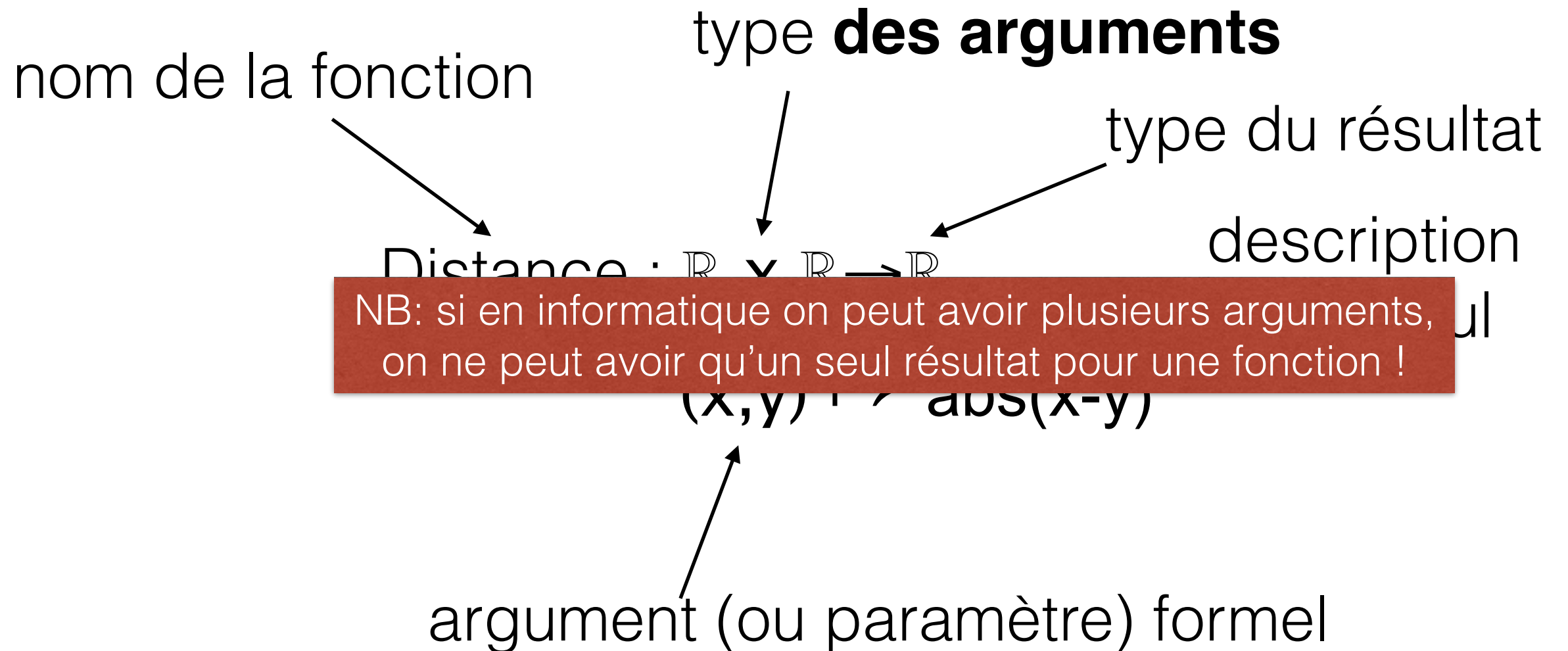
Vocabulaire



Vocabulaire



Vocabulaire



Vocabulaire

- `abs(15)`

- `abs(3*(-5))`

- `Distance(5,7)`

Arguments ou paramètres d'appel

The diagram consists of three arrows originating from a central point on the right side of the text 'Arguments ou paramètres d'appel'. One arrow points to the argument '15' in the first code example, another points to the argument '3*(-5)' in the second code example, and the third points to the first argument '5' in the third code example.

Exemple

- On sait que pour convertir les degrés Fahrenheit en degrés Celsius il convient de multiplier par 9, diviser par 5 et ajouter 32 :
- $x \mapsto C_to_F(x) = 9x/5+32$
- Autrement dit $0^{\circ}C=32^{\circ}F$, $36^{\circ}C=100.8^{\circ}F$
- (notons en passant que l'on utilise un algorithme pour convertir...)

Vocabulaire

Une fonction a un **nom**, un **type** pour le **résultat** et des **types** pour les **arguments**

Les arguments lors de la déclaration sont des **arguments formels**

Les arguments lors de l'utilisation sont des **arguments d'appel**

Utilisation de fonctions

Une fonction est employée sur des **arguments d'appel** et possède une **valeur**

(se rappeler... la valeur de la fonction **f** en...)

Fonctions prédéfinies ($\mathbb{R} \rightarrow \mathbb{R}$)

sin, cos, abs, racine, exp

sont des fonctions prédéfinies

réel \rightarrow réel

Exemple : abs est la fonction $\mathbb{R} \rightarrow \mathbb{R}$ telle que $x \mapsto \text{abs}(x)$

Remarque : x est un paramètre formel

Fonctions prédéfinies

Longueur() est une fonction prédéfinie

chaîne de caractères → entier

Exemple : Longueur('ce jour') vaut 7

Utilisation

On appelle ou utilise une fonction sur des paramètres d'appel :

`log (3)` (3 est un paramètre d'appel)

Ces paramètres d'appel sont :

des valeurs

des expressions

Il faut bien sûr que les types coïncident

`log('ceci est un beau jour ')` pose problème

Utilisation (où ?)

dans une expression

$1 + \log(3)$

$\text{abs}(7) < 8$

L'expression peut intervenir :

dans une affectation

$\text{opposé} \leftarrow \text{hypothénuse} * \sin(\text{angle})$

dans un affichage

$\text{Ecrire}(\sin(\text{angle}))$

dans une conditionnelle

si $(\sin(\text{angle}) < 0.5)$ Alors ...

Utilisation (vocabulaire)

Il faut distinguer l'expression appelante qui contient la fonction et l'expression paramètre d'appel

L'expression paramètre d'appel

opposé ← hypothénuse * sin(angle*2)

L'expression appelante

hypothénuse ← 5; angle ← 45;
opposé ← hypothénuse * sin(angle*2)

Evaluation

1. Pour évaluer l'expression, on doit d'abord évaluer « hypothénuse » et « sin(angle*2) »
2. hypothénuse vaut 5, reste à « sin(angle*2) »
3. On doit d'abord évaluer angle*2 qui vaut 90. reste à évaluer sin(90).
4. Deux questions: est-ce que sin(?) est une fonction qui existe ? Si non, ERREUR et si oui, est-ce que sin(?) peut s'appliquer à des nombres ? Si non, on essaye de transtyper et si oui, on évalue sin(90) qui vaut 1.
5. On évalue 5*1 qui vaut 5, puis on exécute l'instruction « opposé ← 5; »

Attention au type !

Dans une affectation

$$A \leftarrow (3,5 < B) \text{ ou } C$$

on ne peut remplacer B que par un appel de fonction dont le résultat est numérique et C par un appel de fonction dont le résultat est booléen

Exemple

Pour B `abs(17), log(0,7)...`

Pour C `premier(47), pair(22)...`

Fonctions prédéfinies

Dans les différents langages de programmation il y a des fonctions prédéfinies

Dans chaque cas, la fonction est prévue pour fonctionner avec des arguments de types particuliers

Il peut y avoir des bibliothèques de fonctions contenant de très nombreuses fonctions (pas uniquement numériques)

Écriture d'une fonction

Déclaration de fonctions

Déclarer une fonction c'est :

1. la **spécifier** (types)
2. expliciter le **calcul** permettant de passer des paramètres à la valeur résultat

Déclaration (exemple)

Fonction C_to_F(temp_en_C : réel) : réel

Variables:

res: réel;

Début

res ← 1,8*temp_en_C + 32

retourner res;

Fin

Déclaration (exemple)

Fonction C_to_F(temp_en_C : réel) : réel

Variables:

res: réel;

Début

res ← 1,8*temp_en_C + 32

retourner res;

Fin

Nom



Déclaration (exemple)

Fonction C_to_F(temp_en_C : réel) : réel

Variables:

res: réel;

Début

res ← 1,8*temp_en_C + 32

retourner res;

Fin

Déclaration (exemple)

Fonction C_to_F(temp_en_C : réel) : réel

Variables:

res: réel;

Début

res ← 1,8*temp_en_C + 32

retourner res;

Fin

Paramètre(s)



Déclaration (exemple)

Fonction C_to_F(temp_en_C : réel) : réel

Variables:

res: réel;

Début

res ← 1,8*temp_en_C + 32

retourner res;

Fin

Déclaration (exemple)

Fonction C_to_F(temp_en_C : réel) : réel

Variables:

res: réel;

Début

res ← 1,8*temp_en_C + 32

retourner res;

Fin

Type du paramètre



Déclaration (exemple)

Fonction C_to_F(temp_en_C : réel) : réel

Variables:

res: réel;

Début

res ← 1,8*temp_en_C + 32

retourner res;

Fin

Déclaration (exemple)

Fonction C_to_F(temp_en_C : réel) : réel

Variables:

res: réel;

Début

res ← 1,8*temp_en_C + 32

retourner res;

Fin



Type de la fonction

Déclaration (exemple)

Fonction C_to_F(temp_en_C : réel) : réel

Variables:

res: réel;

Début

res ← 1,8*temp_en_C + 32

retourner res;

Fin

Déclaration (exemple)

Fonction C_to_F(temp_en_C : réel) : réel

Variables:

res: réel;

Début

res ← 1,8*temp_en_C + 32

retourner res;

Fin



Calcul

Déclaration (exemple)

Fonction C_to_F(temp_en_C : réel) : réel

Variables:

res: réel;

Début

res ← 1,8*temp_en_C + 32

retourner res;

Fin

Déclaration (exemple)

Fonction C_to_F(temp_en_C : réel) : réel

Variables:

res: réel;

Début

res ← 1,8*temp_en_C + 32

retourner res;

Fin

Un mot clé important !

Déclaration (écriture)

Quasiment comme un algorithme ...

Fonction `nom_f` (paramètres formels: types) : type

Variables:

variables utilisées par la fonction

Début

instructions;

retourner expression;

Fin

`nom_f` est un identificateur

Il peut y avoir 0, 1 ou plusieurs paramètres formels

La fonction peut ne pas utiliser de variables

Le type de l'expression est nécessairement celui de la fonction

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Ecrire un algorithme qui demande à l'utilisateur de saisir une année de naissance et calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Ecrire un algorithme qui demande à l'utilisateur de saisir une année de naissance et calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Algorithme Calcule âge

Variables

année : entier
age : entier

Début

année ← Saisie();

age ← 2020 - année;

Ecrire(age);

Fin

Ecrire un algorithme qui demande à l'utilisateur de saisir une année de naissance et calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Algorithme Calcule âge

Variables

année : entier
age: entier

Début

année ← Saisie();

age ← 2020 - année;

Ecrire(age);

Fin

Ecrire un algorithme qui demande à l'utilisateur de saisir une année de naissance et calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Fonction *Calcule_âge*():

Variables

année : entier
age: entier

Début

année ← *Saisie*();
age ← 2020 - *année*;
Ecrire(*age*);

Fin

Ecrire un algorithme qui demande à l'utilisateur de saisir une année de naissance et calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Fonction *Calcule_âge*():

Variables

année : entier
age : entier

Début

année ← *Saisie*();

age ← 2020 - *année*;

Ecrire(*age*);

Fin

Ecrire un algorithme qui demande à l'utilisateur de saisir une année de naissance et calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Fonction *Calcule_âge*():

Variables

année : entier
age : entier

Début

année ← *Saisie*();

age ← 2020 - *année*;

Ecrire(*age*);

Fin

Ecrire un algorithme qui demande à l'utilisateur de saisir une année de naissance et calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Fonction *Calcule_âge(année : entier)*:

Variables

age : entier

Début

année ← Saisie();

age ← 2020 - année;

Ecrire(age);

Fin

Ecrire un algorithme qui demande à l'utilisateur de saisir une année de naissance et calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Fonction *Calcule_âge(année : entier):*

Variables

age: entier

Début

age ← 2020 - année;

Ecrire(age);

Fin

Ecrire un algorithme qui demande à l'utilisateur de saisir une année de naissance et calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Fonction *Calcule_âge(année : entier):*

Variables

age: entier

Début

age ← 2020 - année;

Ecrire(age);

Fin

Ecrire un algorithme qui demande à l'utilisateur de saisir une année de naissance et calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Restitution du résultat

Fonction *Calcule_âge(année : entier)*:

Variables

age: entier

Début

age ← 2020 - année;

Ecrire(age);

Fin

Ecrire un algorithme qui demande à l'utilisateur de saisir une année de naissance et calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Restitution du résultat

Fonction *Calcule_âge(année : entier) : entier*

Variables

age : entier

Début

age ← 2020 - année;

Retourner(age);

Fin

Ecrire un algorithme qui demande à l'utilisateur de saisir une année de naissance et calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Restitution du résultat

Fonction *Calcule_âge*(*année : entier*): *entier*

Variables

age : entier

Début

age ← 2020 - *année*;

Retourner(*age*);

Fin

en Javascript:

```
function Calcule_age(annee) {  
  var age;  
  age = 2020 - annee;  
  return age;  
}
```

Ecrire un algorithme qui demande à l'utilisateur de saisir une année de naissance et calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Restitution du résultat

Fonction *Calcule_âge(année : entier) : entier*
// fonction qui calcule l'âge au 31/12/2020 étant donné l'année de naissance.

Variables

age : entier

Début

age ← 2020 - année;

Retourner(age);

Fin

en Javascript:

```
function Calcule_age(annee) {  
  var age;  
  age = 2020 - annee;  
  return age;  
}
```


Ecrire un algorithme qui demande à l'utilisateur de saisir une année de naissance et calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Un exercice typique ?

Ecrire une fonction qui, étant donné une année de naissance, calcule l'âge qu'aura la personne au 31 décembre 2020 à minuit.

Restitution du résultat

Fonction *Calcule_âge(année : entier) : entier*

// fonction qui calcule l'âge au 31/12/2020 étant donné l'année de naissance.

Vas

Moralité 1 : Si on sait écrire un algorithme, il n'y a pas de raison qu'on ne sache pas écrire une fonction !

Dé

Moralité 1 (suite) : Les paramètres de la fonction généralisent les saisies. La valeur retournée est une généralisation d'un affichage du résultat du calcul.

Algorithme: *Saisies* \rightsquigarrow *Calculs* \rightsquigarrow *Affichage(s)*

Fonction: *Paramètres* \rightsquigarrow *Calculs* \rightsquigarrow *Retour du résultat*

Fin

Les bonnes pratiques

1. Pas de saisie ni d'affichage dans une fonction (sauf si on définit des fonctions dédiées à des saisies complexes ou des affichages complexes).
2. Bien commenter une fonction
3. Bien déclarer toutes les variables utilisées par la fonction (source de bugs en TP).
4. Il faut proposer une manière de tester le bon fonctionnement de la fonction (tests unitaires, utilisation de la fonction dans un algorithme,...).

Quand doit-on écrire une fonction ?

- Dès que possible.....
- Les bonnes pratiques de programmation consistent à découper le problème en sous tâches qui paraissent élémentaire.
- Par exemple, dès qu'on a deux boucles imbriquées, il faut songer à écrire une fonction...

Quand doit-on écrire une fonction ?

- Dès que possible.....
- Les bonnes pratiques de programmation consistent à découper le problème en sous tâches qui paraissent élémentaire.

- Par exemple deux boucles imbriquées à écrire une fonction...

```
// Algorithme Triangle
var ch, i, j;
for (i = 1; i <= 4; i = i + 1) {
  ch = "";
  for (j = 1; j <= i; j = j + 1) {
    ch = ch + '*';
  }
  Ecrire(ch);
}
```


Liens entre déclaration et utilisation

Dans un programme il existera des règles très précises nous disant où il faut placer la déclaration de la fonction.

En algorithmique ce n'est pas un enjeu mais on veillera toujours à ce qu'une fonction qui a été déclarée soit utilisée (soit par une autre fonction, soit par un algorithme).

Une bonne habitude :

Fonction truc(x : réel, n : entier) : réel;

Variables

i : entier;
res : réel;

Début

res ← 1;
pour i allant de 1 à n faire
 res ← res * x;
fin pour
retourner res;

Fin

Algorithme utiliser truc

Variables

nombre : réel;

Début

nombre ← Saisie();
Ecrire(truc(nombre,2));

Fin

aration et n

es règles très
cer la déclaration

enjeu mais on
ion qui a été
autre fonction, soit

Une bonne habitude :

paration et

Fonction truc(x : réel, n : entier) : réel;

Variables

i : entier;
res : réel;

Début

res ← 1;
pour i allant de 1 à n faire
 res ← res * x;
fin pour
retourner res;

Fin

Algorithme utiliser truc

Variables

nombre : réel;

Début

nombre ← Saisie();
Ecrire(truc(nombre,2));

Fin

```
function truc(x,n) {  
    var i,res;  
    res=1;  
    for(i=1; i<=n; i=i+1) {  
        res=res*x;  
    }  
    return res;  
}
```

```
// Programme utiliser truc  
var nombre  
nombre=Saisie();  
Ecrire(truc(nombre,2));
```

Une bonne habitude :

Effet de l'appel

Les **paramètres d'appel** sont évalués

La valeur obtenue est transférée aux **paramètres formels** (il y a un **contrôle** ou une **conversion** de types)

Le corps de la fonction est exécuté avec les valeurs retenues jusqu'à ce qu'une instruction **retourner(exp)** soit exécutée

La valeur de la fonction est alors remplacée dans l'expression appelante

Rq: une fonction peut ne rien retourner. Dans ce cas, on donne « rien » comme type de retour de la fonction. On parle parfois de « procédure » (ex: Ecrire(chaine); est de ce type).

Exemple très simple

Algorithme utiliser

Variables:

v : entier;

Début

1) v ← double(3);

Fin

Fonction double(i : entier) : entier

Début

a) i ← 2*i ;

b) retourner (i);

Fin

Exemple très simple

Algorithme utiliser

Variables:

v : entier;

Début

1) v ← double(3);

Fin

Fonction double(i : entier) : entier

Début

a) i ← 2*i ;

b) retourner (i);

Fin

Instruction	v	i
-------------	---	---

Exemple très simple

Algorithme utiliser

Variables:

v : entier;

Début

1) v ← double(3);

Fin

Fonction double(i : entier) : entier

Début

a) i ← 2*i ;

b) retourner (i);

Fin

Instruction	v	i
Avant 1)	?	inconnue

Exemple très simple

Algorithme utiliser

Variables:

v : entier;

Début

1) v ← double(3);

Fin

Fonction double(i : entier) : entier

Début

a) i ← 2*i ;

b) retourner (i);

Fin

Instruction	v	i
Avant 1)	?	inconnue
Avant a)	connue ?	3

passage de la valeur 3 à i

Exemple très simple

Algorithme utiliser

Variables:

v : entier;

Début

1) v ← double(3);

Fin

Fonction double(i : entier) : entier

Début

a) i ← 2*i ;

b) retourner (i);

Fin

Instruction	v	i
Avant 1)	?	inconnue
Avant a)	connue ?	3
Après a)	connue ?	6

passage de la valeur 3 à i

Exemple très simple

Algorithme utiliser

Variables:

v : entier;

Début

1) v ← double(3);

Fin

Fonction double(i : entier) : entier

Début

a) i ← 2*i ;

b) retourner (i);

Fin

Instruction	v	i	
Avant 1)	?	inconnue	
Avant a)	connue ?	3	passage de la valeur 3 à i
Après a)	connue ?	6	
Après b)	connue ?	6	envoi de la valeur 6 à l'expression appelante

Exemple très simple

Algorithme utiliser

Variables:

v : entier;

Début

1) v ← double(3);

Fin

Fonction double(i : entier) : entier

Début

a) i ← 2*i ;

b) retourner (i);

Fin

Instruction	v	i
Avant 1)	?	inconnue
Avant a)	connue ?	3 <i>passage de la valeur 3 à i</i>
Après a)	connue ?	6
Après b)	connue ?	6 <i>envoi de la valeur 6 à l'expression appelante</i>
Après 1)	6	inconnue

Pourquoi doit-on écrire des fonctions ?

Donner un nom à une portion d'algorithme

---> lisibilité, modularité

Pourquoi doit-on écrire des fonctions ?

« factoriser » une portion d'algorithme

---> facilité d'écriture/débogage

- lisibilité
- longueur du code source
- **maintenabilité**
- " écrire une fois, utiliser plusieurs fois "

Pourquoi doit-on écrire des fonctions ?

« partager » une portion d'algorithme avec d'autres algorithmes

Bibliothèques de fonctions

Ré-utilisabilité: un exemple

- **But: tracer cette flèche**
- La première ligne contient
5 espaces et 1 *
- La deuxième ligne contient
4 espaces et 3 *
- Etc, etc.....

```
-----*
```

```
-----***
```

```
-----*****
```

```
-----*****
```

```
-----*****
```

```
-----*****
```

```
-----***
```

```
-----***
```

```
-----***
```

```
-----***
```

```
-----***
```

Ré-utilisabilité: un exemple

Fonction Repeter(nb : entier, symbole: caractère):chaîne de caractères

Variables:

i: entier;

res : chaîne de caractères

Début

res ← “”;

pour i allant de 1 à nb faire

 res ← res + symbole;

fin pour

retourner(res);

Fin

Ré-utilisabilité: un exemple

Algorithme Flèche

Variables:

i : entier;

Début

pour i allant de 1 à 5 faire

Ecrire(**Repeter**(6-i, '␣')+**Repeter**(2*i-1, '*'));

fin pour

pour i allant de 1 à 5 faire

Ecrire(**Repeter**(4, '␣')+**Repeter**(3, '*'));

fin pour

Fin

Ré-utilisabilité: un exemple

Algorithme Flèche

Variables:

i : entier;

Début

pour i allant de 1 à 5 faire

Ecrire(**Repeter**(6-i, '␣')+**Repeter**(2*i-1, '*'));

fin pour

pour i allant de 1 à 5 faire

Ecrire(**Repeter**(4, '␣')+**Repeter**(3, '*'));

fin pour

Fin

**DÉFINIE 1 FOIS
UTILISÉE 4 FOIS !!!**

Pourquoi doit-on écrire des fonctions ?

"partager le travail"

une portion d'algorithme peut ainsi être écrite par un tiers

---> rapidité d'écriture

Bilan...

Les avantages des fonctions sont :

Réutilisabilité

Généricité

Rend la lecture plus simple

Partage du travail

Modularité

Portée des variables

Portée des variables

Les variables définies dans une fonction ne sont pas accessibles à l'extérieur de cette fonction.

Elles sont appelées variables **locales** (à la fonction) par opposition aux variables **globales** (au programme).

La valeur d'une variable dépend donc de son **contexte**.

Portée des variables: exemple

Fonction Repeter(nb : entier, symbole: caractère): chaîne de caractères

Variables: res: chaîne, i:entier;

Début

```
1   res ← “”;  
   pour i allant de 0 à nb-1 faire  
2       res ← res + symbole;  
   fin pour  
3   retourner(res);
```

Fin

Algorithme Flèche

Variables: i : entier;

Début

```
   pour i allant de 1 à 5 faire  
A   chaine1 ← Repeter(6-i, '_');  
B   chaine2 ← Repeter(2*i-1, '*');  
C   Ecrire(chaine1+chaine2);  
   fin pour
```

Fin

Portée des variables: exemple

Fonction Repeter(nb : entier, symbole: caractère): chaîne de caractères

Variables: res: chaîne, i:entier;

Début

```

1   res ← "";
    pour i allant de 0 à nb-1 faire
2       res ← res + symbole;
    fin pour
3   retourner(res);

```

Fin

Algorithme Flèche

Variables: i : entier;

Début

```

    pour i allant de 1 à 5 faire
A   chaine1 ← Repeter(6-i, ' ');
B   chaine2 ← Repeter(2*i-1, '*');
C   Ecrire(chaine1+chaine2);
    fin pour

```

Fin

	i	chaine1	chaine2	res
Avant A	1	?	?	?
Avant 1	?	?	?	?
Après 1	?	?	?	"
Après 2	0	?	?	' '
Après 3	?	?	?	' '
Après B	1	"	?	?
Etc, etc				

Récurtivité

Appels récursifs

Que se passe-t-il lorsqu'on appelle une fonction dans le corps de celle-ci ?

On parle d'appel récursif (ou récursivité)

Analogie en mathématiques: suites définies récursivement

ex: factorielle: $u_0=1$ et $u_n=n*u_{n-1}$ si $n>0$

Un autre moyen de faire des boucles...

Appels récursifs : schéma

Fonction factorielle(n : entier) : entier

Variables: res : entier

Début

si (n=0)

Alors res ← 1;

Sinon res ← n*factorielle(n-1);

fin si

retourner(res);

Fin

Appels récursifs : schéma

Fonction factorielle(n : entier) : entier

Variables: res : entier

Début

si (n=0)

Alors res ← 1;

Sinon res ← n*factorielle(n-1);

fin si

retourner(res);

Fin

Condition d'arrêt



Appels récursifs : schéma

Fonction factorielle(n : entier) : entier

Variables: res : entier

Début

si (n=0)

Alors res ← 1;

Sinon res ← n*factorielle(n-1);

fin si

retourner(res);

Fin

Appels récursifs : schéma

Fonction factorielle(n : entier) : entier

Variables: res : entier

Début

si (n=0)

Alors res ← 1;

Sinon res ← n*factorielle(n-1);

fin si

retourner(res);

Fin

Valeur d'arrêt



Appels récursifs : schéma

Fonction factorielle(n : entier) : entier

Variables: res : entier

Début

si (n=0)

Alors res ← 1;

Sinon res ← n*factorielle(n-1);

fin si

retourner(res);

Fin

Appels récursifs : schéma

Fonction factorielle(n : entier) : entier

Variables: res : entier

Début

si (n=0)


Alors res ← 1;

Sinon res ← n*factorielle(n-1);

fin si

retourner(res);

Fin


Appel récursif

Appels récursifs : schéma

Fonction factorielle(n : entier) : entier

Variables: res : entier

Début

si (n=0)

Alors res ← 1;

Sinon res ← n*factorielle(n-1);

fin si

retourner(res);

Fin

Appels récursifs : Dangers

Avoir une condition d'arrêt qui ne soit jamais satisfaite (boucle infinie comme dans le tant que)

Il faut que l'appel récursif modifie les paramètres !!!

Problèmes de saturation de la mémoire (à chaque appel récursif, l'état courant de la mémoire est sauvegardé pour pouvoir le récupérer tel quel après l'appel.

Appels récurrents : Dangers

Factorielle(5)
Retourner 5*?

MEMOIRE

Appels récurrents : Dangers

Factorielle(4)
Retourner 4*?

Factorielle(5)
Retourner 5*?

MEMOIRE

Appels récurrents : Dangers

Factorielle(3)
Retourner 3*?

Factorielle(4)
Retourner 4*?

Factorielle(5)
Retourner 5*?

MEMOIRE

Appels récursifs : Dangers

Factorielle(3)
Retourner 3^* ?

Factorielle(4)
Retourner 4^* ?

Factorielle(5)
Retourner 5^* ?

MEMOIRE

Factorielle(2)
Retourner 2^* ?

Appels récursifs : Dangers

Factorielle(2)
Retourner 2*?

Factorielle(3)
Retourner 3*?

Factorielle(4)
Retourner 4*?

Factorielle(5)
Retourner 5*?

MEMOIRE

Factorielle(1)
Retourner 1*?

Appels récurrents : Dangers

Factorielle(1)
Retourner 1*?

Factorielle(2)
Retourner 2*?

Factorielle(3)
Retourner 3*?

Factorielle(4)
Retourner 4*?

Factorielle(5)
Retourner 5*?

MEMOIRE

Factorielle(0)
Retourner 1

Appels récursifs : Dangers

Factorielle(2)
Retourner 2^* ?

Factorielle(3)
Retourner 3^* ?

Factorielle(4)
Retourner 4^* ?

Factorielle(5)
Retourner 5^* ?

MEMOIRE

Factorielle(1)
Retourner 1^*1

Appels récursifs : Dangers

Factorielle(3)
Retourner 3*?

Factorielle(4)
Retourner 4*?

Factorielle(5)
Retourner 5*?

MEMOIRE

Factorielle(2)
Retourner 2*1

Appels récurifs : Dangers

Factorielle(3)
Retourner $3*2$

Factorielle(4)
Retourner $4*?$

Factorielle(5)
Retourner $5*?$

MEMOIRE

Appels récursifs : Dangers

Factorielle(4)
Retourner 4*6

Factorielle(5)
Retourner 5*?

MEMOIRE

Appels récursifs : Dangers

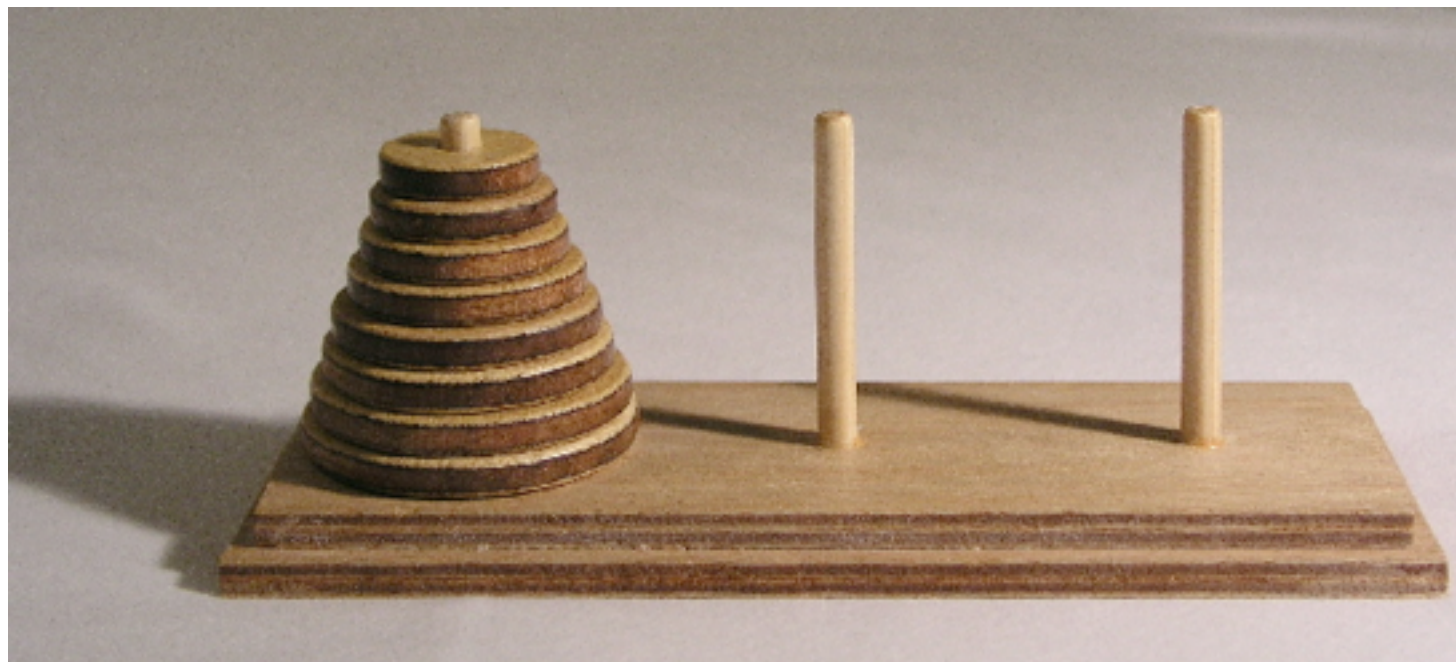
Factorielle(5)
Retourner 5×24

MEMOIRE

Appels récursifs : Puissance

Les tours de Hanoi: Comment déplacer le tas du premier au dernier piquet.

- 1 disque à la fois
- On n'a pas le droit de placer un disque sur un autre de taille inférieure.



Appels récursifs : Puissance

- Exemple: Les tours de Hanoi animé



Appels récurrents : Puissance

En séquentiel:

Je déplace le premier disque au milieu

Je déplace le second à la fin

Puis le premier du milieu à la fin

... et pour plus de disques ?????

En récursif:

Pour déplacer 3 disques, il suffit de mettre les disques 1&2 au milieu en attendant, déplacer le disque 3 à la fin et remettre les disques 1&2 dessus.

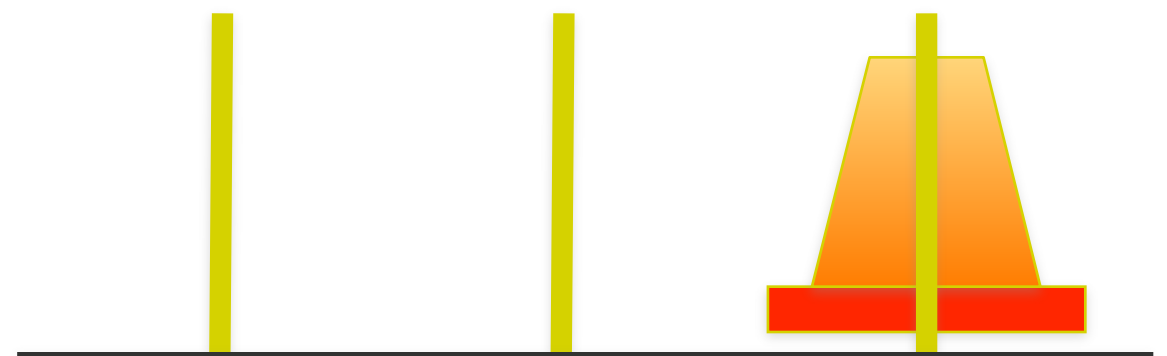
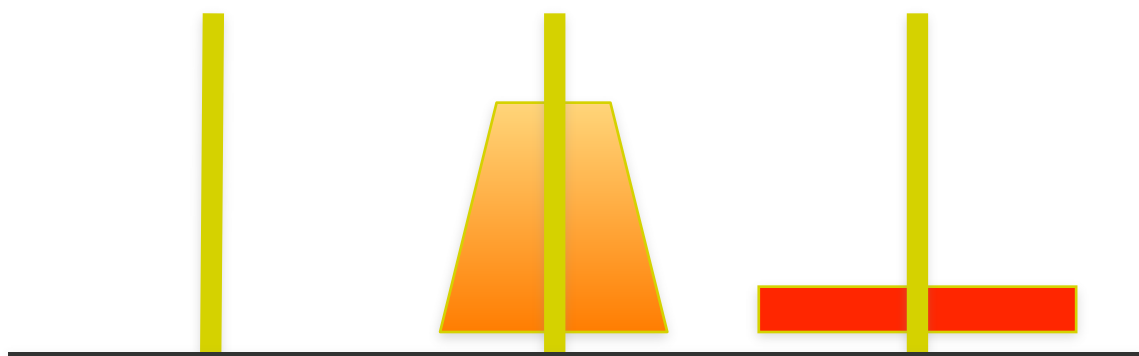
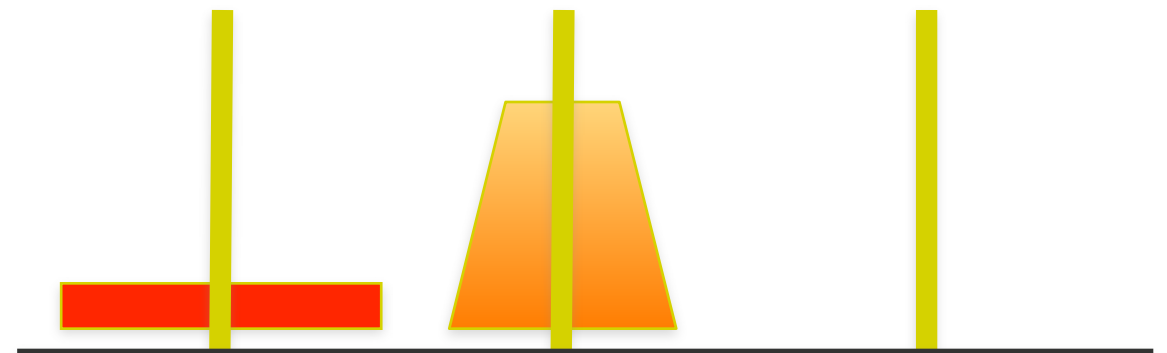
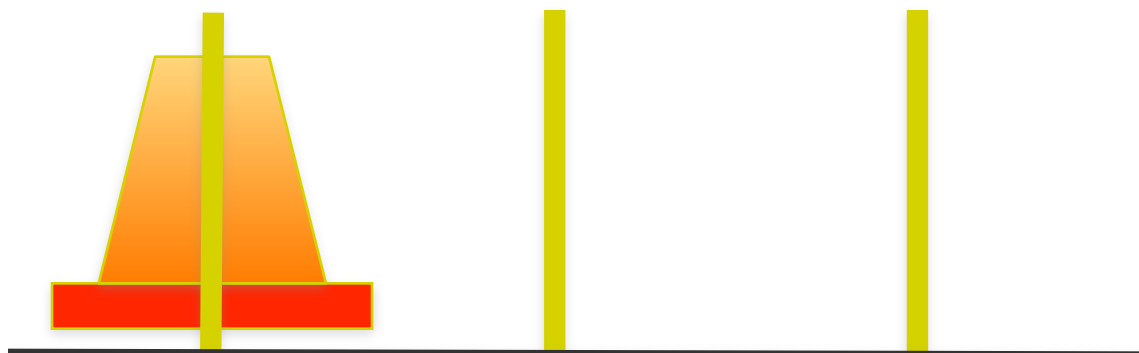
Appels récursifs : Puissance

- En récursif: Pour déplacer 8 disques, il suffit d'en mettre 7 au milieu en attendant, déplacer le huitième à la fin et remettre les 7 disques dessus.
- il faut au minimum $2^N - 1$ coups pour déplacer N disques (255 pour 8 disques)



Appels récurrents : Puissance

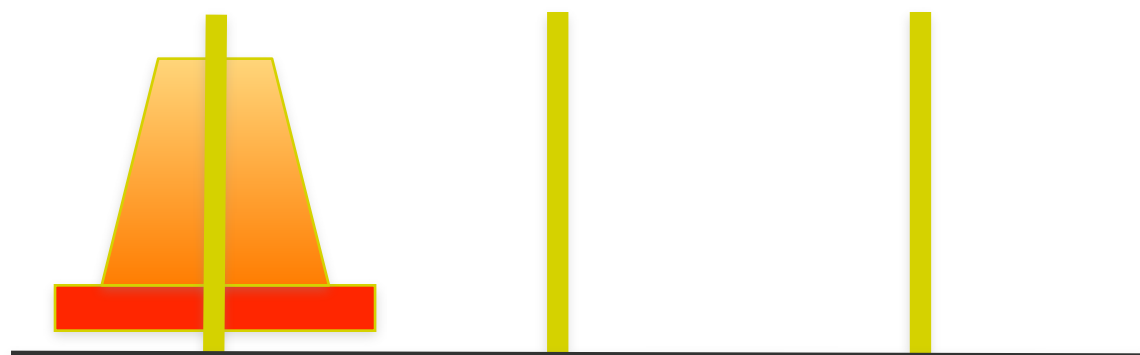
En récursif: Pour déplacer 8 disques, il suffit d'en mettre 7 au milieu en attendant, déplacer le huitième à la fin et remettre les 7 disques dessus.



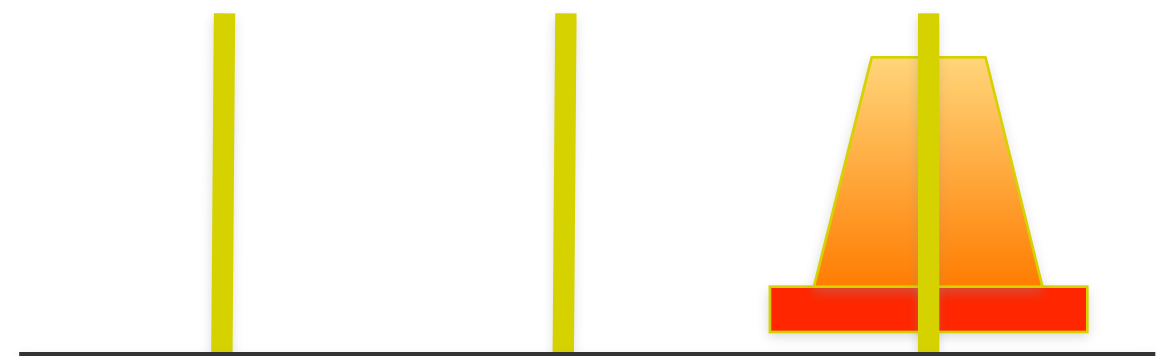
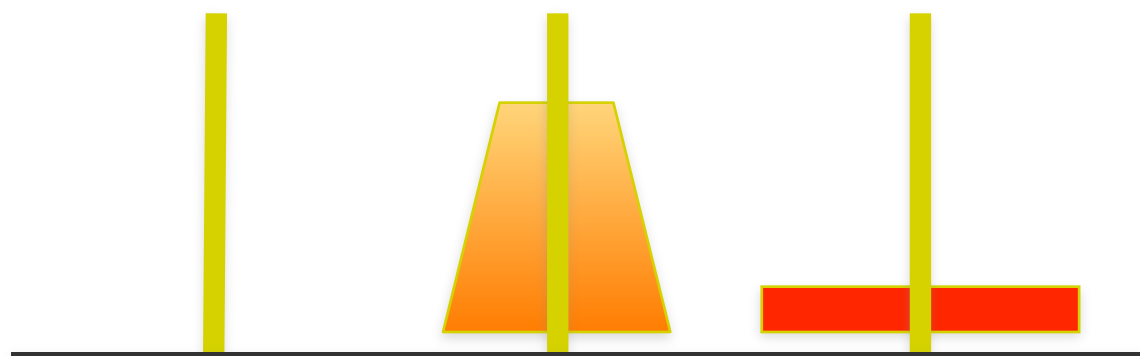
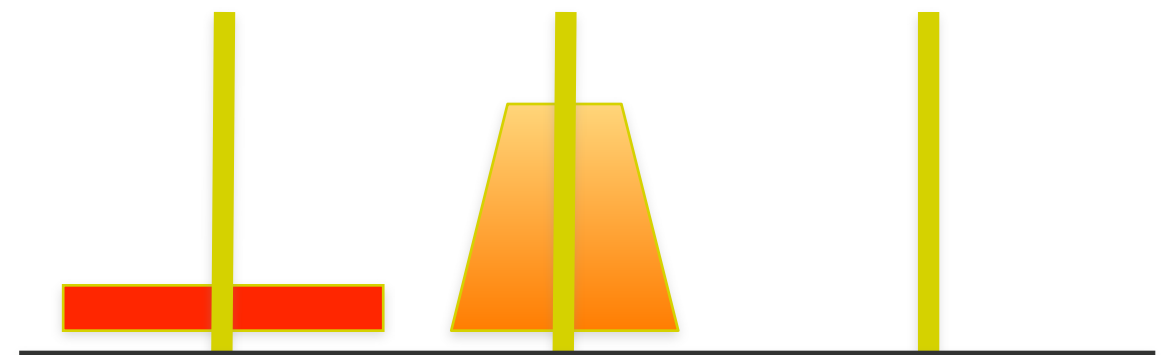


Appels récursifs : Puissance

En récursif: Pour déplacer 8 disques, il suffit d'en mettre 7 au milieu en attendant, déplacer le huitième à la fin et remettre les 7 disques dessus.



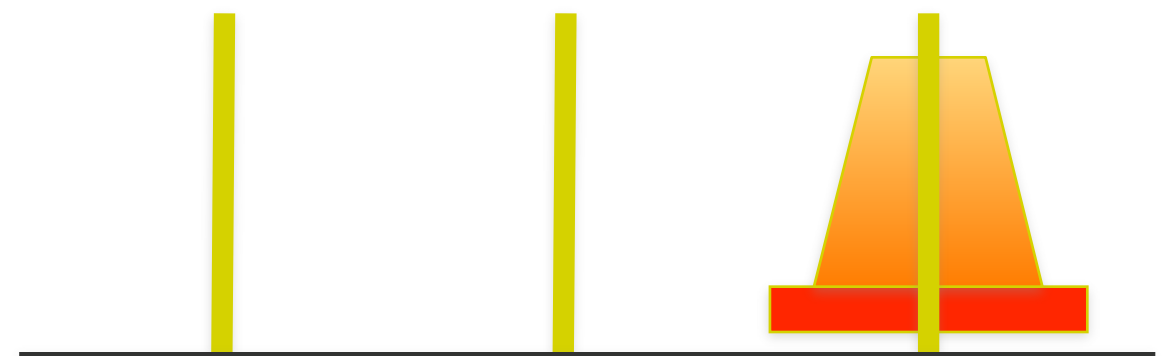
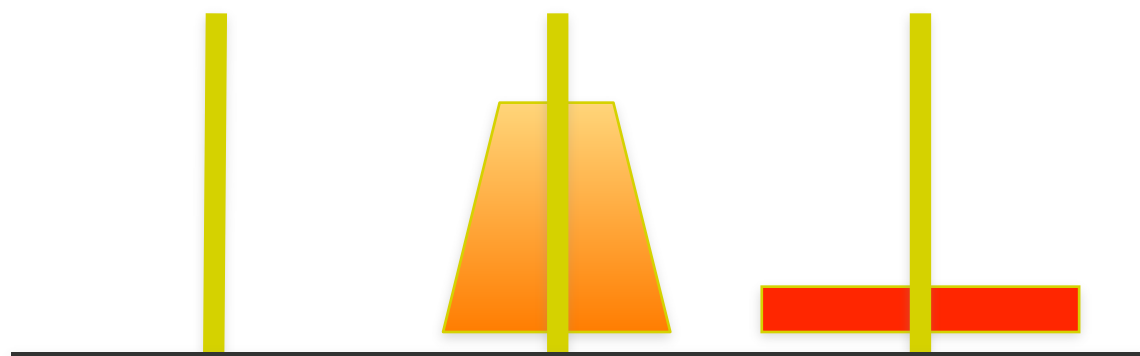
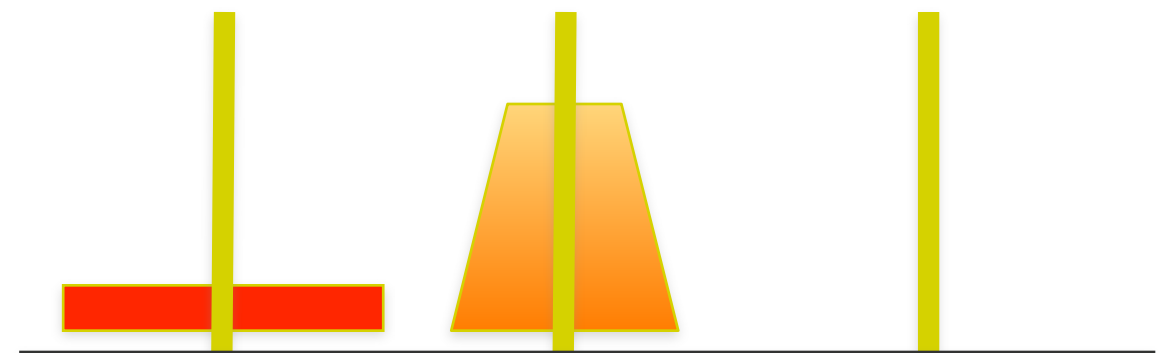
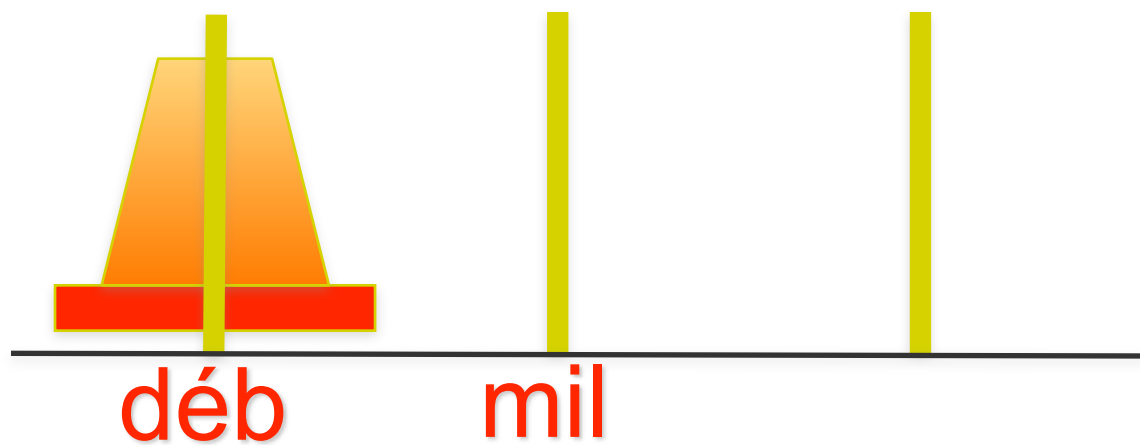
déb





Appels récursifs : Puissance

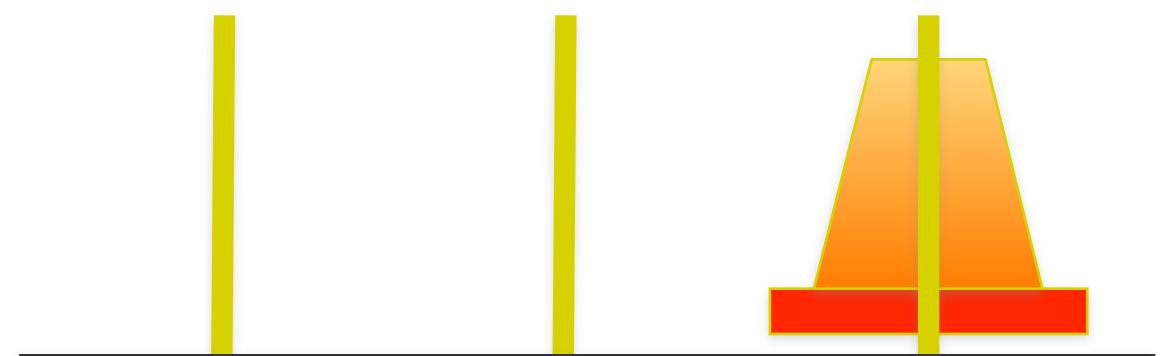
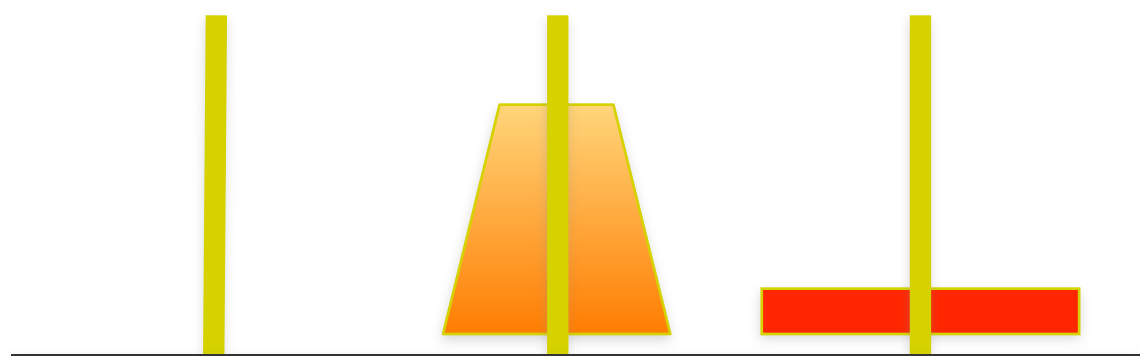
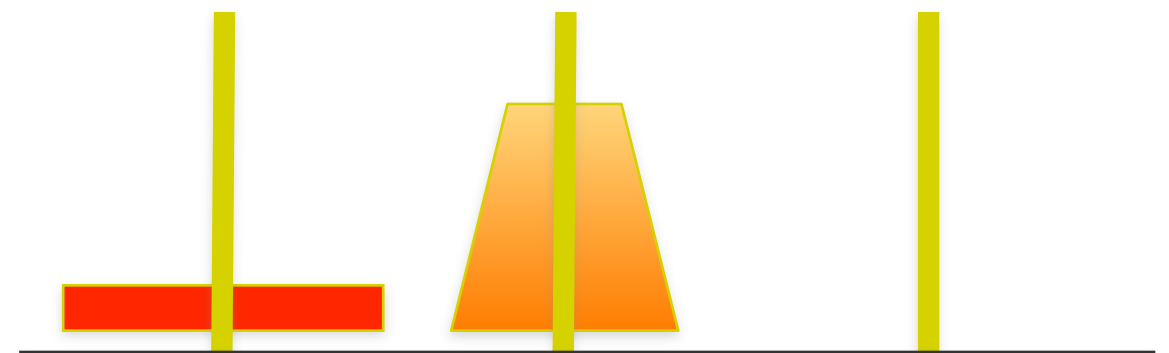
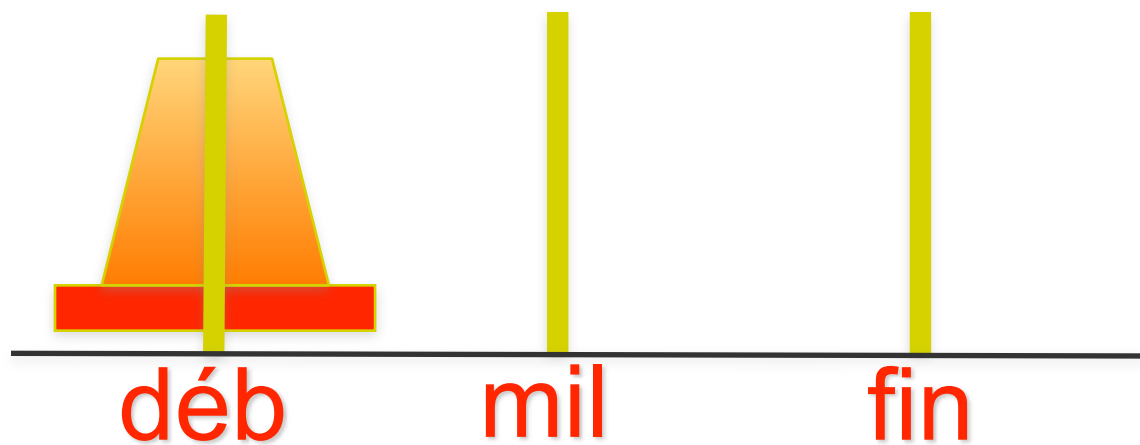
En récursif: Pour déplacer 8 disques, il suffit d'en mettre 7 au milieu en attendant, déplacer le huitième à la fin et remettre les 7 disques dessus.





Appels récursifs : Puissance

En récursif: Pour déplacer 8 disques, il suffit d'en mettre 7 au milieu en attendant, déplacer le huitième à la fin et remettre les 7 disques dessus.





Appels récursifs : Puissance

Fonction Hanoi(deb,mil,fin: chaines, n : entier) : rien

Début

Si ($n > 0$)

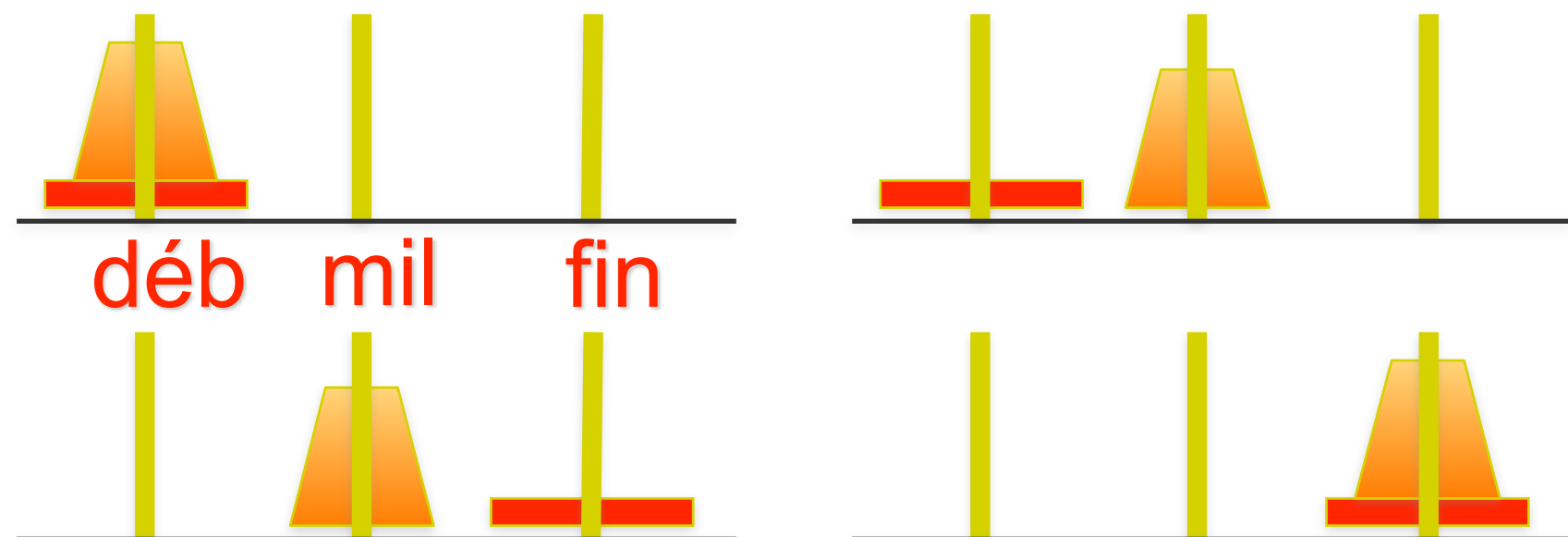
Alors Hanoi(deb,fin,mil,n-1);

Ecrire('Déplacer '+n+' de '+deb+' à '+fin);

Hanoi(mil,deb,fin,n-1);

fin si

Fin



Réursion terminale

Pour minimiser l'espace mémoire nécessaire au stockage des appels récurifs, on s'arrange pour que l'appel soit la dernière instruction de la fonction.

On parle alors de **réursion terminale**

On peut toujours passer d'une réursion non terminale à une réursion terminale

On peut toujours passer d'un fonction réursive à une fonction non réursive aussi !!!

Réursion terminale

Fonction fact_iteratif(n)

Début

res ← 1;

pour i de allant de 1 à n faire

res ← res * n;

fin pour

retourner(res);

Fin

Fonction fact_recuratif(n)

Début

res ← 1;

si (n>0)

Alors res ← fact_recuratif(n-1) * n;

fin si

retourner(res);

Fin

Fonction fact_recNT(n,res)

Début

si (n<1)

Alors retourner(res);

Sinon fact_recNT(n-1,res*n);

fin si

Fin

Appel : fact_recNT(15,1)

Bilan

- Une fonction est définie par un nom(/identifiant), un ou des paramètres formels et un type de valeur de retour.
- Il existe de nombreuses fonctions prédéfinies.
- La syntaxe de déclaration d'une nouvelle fonction est proche de celle d'un algorithme.
- La portée des variables déclarées dans une fonction est locale à la fonction (par opposition aux variables de l'algorithme qui ont une portée globale).
- La récursivité est particulièrement adaptée pour résoudre des problèmes qui se décrivent mieux de manière récursive.