

Inf 431 – Cours 9

Assertions et Programmes

jeanjacqueslevy.net

secrétariat de l'enseignement:

Catherine Bensoussan

cb@lix.polytechnique.fr

Aile 00, LIX,

01 69 33 34 67

www.enseignement.polytechnique.fr/informatique/IF

Plan

1. Correction de programmes itératifs scalaires
2. Terminaison de programmes itératifs
3. Correction de programmes avec des tableaux
4. Logique de Hoare
5. Récursion et assertions
6. Ordres bien-fondés

La suite de Fibonacci (1/6)

- Récurrence linéaire d'ordre 2

$$u_0 = 0$$

$$u_1 = 1$$

$$u_n = u_{n-1} + u_{n-2}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, ...

- ```
static int fib (int n) {
 if (n < 2) return n;
 else return fib (n-1) + fib (n-2);
}
```

- ou en faisant un peu de programmation dynamique

```
static int fib1 (int n) {
 int[] res = new int[n+1];
 res[0] = 0; res[1] = 1;
 for (int i=2; i <= n; ++i)
 res[i] = res[i-1]+res[i-2];
 return res[n];
}
```

## La suite de Fibonacci (2/6)

- ou en ne gardant que les deux dernières valeurs  $x$  et  $y$

```
static int fibonacci (int n) {
 int x = 0;
 if (n != 0) {
 x = 1; int y = 0;
 for (int k=1; k != n; ++k) {
 int t = y;
 y = x;
 x = x + t;
 }
 }
 return x;
}
```

- On veut montrer que  $n \geq 0 \implies x = \text{fib}(n)$  à la fin.

# La suite de Fibonacci (3/6)

- ou en décomposant l'instruction `for`.

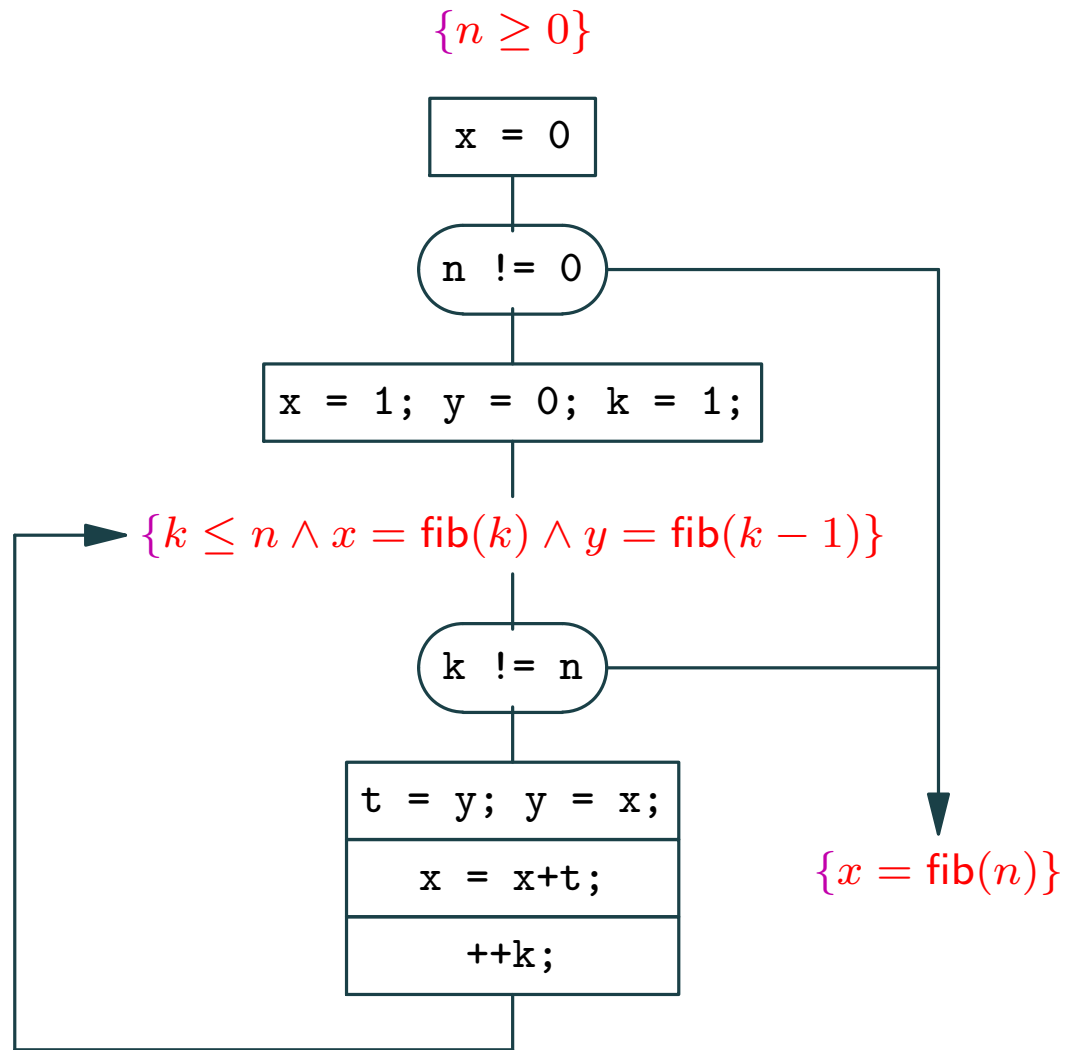
```
static int fibonacci (int n) {
 int x = 0;
 if (n != 0) {
 x = 1; int y = 0;
 int k = 1;
 while (k != n) {
 int t = y;
 y = x;
 x = x + t;
 ++k;
 }
 }
 return x;
}
```

# La suite de Fibonacci (4/6)

```
• static int fibonacci (int n) {
 //P = {n ≥ 0}
 int x = 0;
 if (n != 0) {
 x = 1; int y = 0;
 int k = 1;
 //Q = {k ≤ n ∧ x = fib(k) ∧ y = fib(k - 1)}
 while (k != n) {
 int t = y;
 y = x;
 x = x + t;
 ++k;
 }
 }
 //R = {x = fib(n)}
 return x;
}
```

- $\mathcal{P}$  assertion d'entrée
- $\mathcal{Q}$  assertion invariant de boucle
- $\mathcal{R}$  assertion de fin
- Montrer que de  $\mathcal{P}$ , on peut dériver  $\mathcal{R}$ .

# La suite de Fibonacci (5/6)



# La suite de Fibonacci (6/6)

- ```
static int fibonacci (int n) {  
    {n ≥ 0}  
    int x = 0;  
    {n ≥ 0 ∧ x = 0}  
    if (n != 0) {  
        x = 1; int y = 0;  
        {n > 0 ∧ x = fib(1) ∧ y = fib(0)}  
        int k = 1;  
        {k ≤ n ∧ x = fib(k) ∧ y = fib(k - 1)}  
        while (k != n) {  
            {k < n ∧ x = fib(k) ∧ y = fib(k - 1)}  
            int t = y;  
            {k < n ∧ x = fib(k) ∧ y = fib(k - 1) ∧ t = fib(k - 1)}  
            y = x;  
            {k < n ∧ x = fib(k) ∧ y = fib(k) ∧ t = fib(k - 1)}  
            x = x + t;  
            {k < n ∧ x = fib(k + 1) ∧ y = fib(k) ∧ t = fib(k - 1)}  
            ++k;  
        }  
        {k = n ∧ x = fib(k) ∧ y = fib(k - 1)}  
    }  
    {x = fib(n)}  
    return x;  
}
```


Le PGCD (1/3)

- Un autre exemple itératif simple :

```
static int pgcd (int a, int b) {  
    int x = a, y = b;  
    while (x != y)  
        if (x > y)  
            x = x - y;  
        else  
            y = y - x;  
    return x;  
}
```

Le PGCD (2/3)

- ```
static int pgcd (int a, int b) {
 {a > 0 ∧ b > 0}
 int x = a, y = b;
 {x > 0 ∧ y > 0 ∧ pgcd(x, y) = pgcd(a, b)}
 while (x != y) {
 {x > 0 ∧ y > 0 ∧ x ≠ y ∧ pgcd(x, y) = pgcd(a, b)}
 if (x > y) {
 {x > 0 ∧ y > 0 ∧ x > y ∧ pgcd(x - y, y) = pgcd(a, b)}
 x = x - y;
 {x > 0 ∧ y > 0 ∧ pgcd(x, y) = pgcd(a, b)}
 } else {
 {x > 0 ∧ y > 0 ∧ x < y ∧ pgcd(x, y - x) = pgcd(a, b)}
 y = y - x;
 {x > 0 ∧ y > 0 ∧ pgcd(x, y) = pgcd(a, b)}
 }
 }
 {x > 0 ∧ x = y = pgcd(x, y) = pgcd(a, b)}
 return x;
}
```

**Exercice 1** Montrer que le raisonnement n'est plus valide avec  $\{a \geq 0 \wedge b \geq 0\}$ . Comment corriger le programme ?

# Le PGCD (3/3)

**Exercice 2** Faire le raisonnement avec l'algorithme d'Euclide suivant avec  $\{a \geq 0 \wedge b \geq 0\}$  comme assertion d'entrée.

```
static int pgcd (int a, int b) {
 int x = a, y = b;
 while (y != 0) {
 int r = x % y;
 x = y;
 y = r;
 }
 return x;
}
```

**Exercice 3** Modifier les programmes pour qu'ils acceptent comme assertion d'entrée  $\{a \in \mathbf{Z} \wedge b \in \mathbf{Z}\}$ .

# Assertions

- Les variables d'un programme **itératif** ont des valeurs **modifiables**.
- Une assertion est une proposition **logique**, décrivant une propriété d'un état mémoire des variables.
- Une assertion est attachée à un **point** d'un programme.
- Pour montrer l'implication de l'assertion de fin à partir de l'assertion d'entrée, on procède par implications **successives** grâce à des assertions intermédiaires.

# Terminaison (1/2)

```
static int fibonacci (int n) {
 { $n \geq 0$ }
 int x = 0;
 if (n != 0) {
 x = 1; int y = 0;
 int k = 1;
 { $\Omega(n, k) = n - k$ }
 while (k != n) {
 int t = y;
 y = x;
 x = x + t;
 ++k;
 }
 }
 { $x = \text{fib}(n)$ }
 return x;
}
```

En un tour de boucle comme  $(n, k)$  devient  $(n, k + 1)$ , on a  $\Omega(n, k) = n - k > n - k - 1 = \Omega(n, k + 1)$ .

L'instruction `while` s'arrête donc.

## Terminaison (2/2)

```
static int pgcd (int a, int b) {
 {a > 0 ∧ b > 0}
 int x = a, y = b;
 {x > 0 ∧ y > 0 ∧ Ω(x, y) = max(x, y)}
 while (x != y)
 if (x > y)
 x = x - y;
 else
 y = y - x;
 return x;
}
```

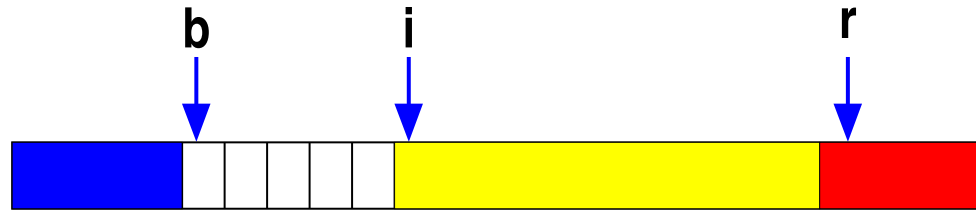
Ici encore en un tour de boucle, si  $x > y$ , on a  
 $\Omega(x, y) = \max(x, y) > \max(x - y, y) = \Omega(x - y, y)$

De même, si  $x < y$ , on a  
 $\Omega(x, y) = \max(x, y) > \max(x, y - x) = \Omega(x, y - x)$

L'instruction `while` s'arrête donc.

**Exercice 4** Montrer la terminaison de l'algorithme d'Euclide.

# Assertions et tableaux (1/3)



```
static void drapeauHollandais (int[] a) { [Dijkstra]
 int n = a.length; int b = 0, i = 0, r = n;
 while (i < r) {
 switch (a[i]) {
 case BLEU:
 int t = a[b]; a[b] = a[i]; a[i] = t;
 ++b; ++i;
 break;
 case BLANC:
 ++i;
 break;
 case ROUGE:
 --r;
 int u = a[r]; a[r] = a[i]; a[i] = u;
 break;
 }
 }
}
```

## Assertions et tableaux (2/3)

```
static void drapeauHollandais (int[] a) {
 int n = a.length; int b = 0, i = 0, r = n;
 { $\phi(0, b, \text{BLEU}) \wedge \phi(b, i, \text{BLANC}) \wedge \phi(r, n, \text{ROUGE}) \wedge i \leq r$ }
 while (i < r) {
 switch (a[i]) {
 case BLEU:
 { $\phi(0, b, \text{BLEU}) \wedge \phi(b, i, \text{BLANC}) \wedge \phi(r, n, \text{ROUGE}) \wedge a[i] = \text{BLEU} \wedge i < r$ }
 int t = a[b]; a[b] = a[i]; a[i] = t;
 { $\phi(0, b + 1, \text{BLEU}) \wedge \phi(b + 1, i + 1, \text{BLANC}) \wedge \phi(r, n, \text{ROUGE}) \wedge i < r$ }
 ++b; ++i; break;
 case BLANC:
 { $\phi(0, b, \text{BLEU}) \wedge \phi(b, i + 1, \text{BLANC}) \wedge \phi(r, n, \text{ROUGE}) \wedge i < r$ }
 ++i; break;
 case ROUGE:
 { $\phi(0, b, \text{BLEU}) \wedge \phi(b, i, \text{BLANC}) \wedge \phi(r, n, \text{ROUGE}) \wedge a[i] = \text{ROUGE} \wedge i < r$ }
 --r;
 { $\phi(0, b, \text{BLEU}) \wedge \phi(b, i, \text{BLANC}) \wedge \phi(r + 1, n, \text{ROUGE}) \wedge a[i] = \text{ROUGE} \wedge i \leq r$ }
 int u = a[r]; a[r] = a[i]; a[i] = u;
 break;
 } }
 { $\phi(0, b, \text{BLEU}) \wedge \phi(b, r, \text{BLANC}) \wedge \phi(r, n, \text{ROUGE})$ }
}
```

où  $\phi(i, j, c) = \forall k. i \leq k < j \Rightarrow a[k] = c$



# Assertions et tableaux (3/3)

La terminaison du drapeau hollandais se montre en considérant l'ordinal  $\Omega(b, i, r) = r - i$ .

**Exercice 5** Montrer la correction du tri par sélection suivant :

```
static void triSelection (int[] a) {
 int n = a.length;
 for (int i = 0; i < n - 1; ++i) {
 int min = i;
 for (int j = i+1; j < n; ++j)
 if (a[j] < a[min])
 min = j;
 int t = a[min]; a[min] = a[i]; a[i] = t;
 }
}
```

**Exercice 6** Montrer la correction du tri par insertion.

**Exercice 7** Montrer la correction du tri par bulles.  
(cf. cours 1ère année)

# Quelques principes logiques

- On a toujours  
 $\{P(E)\} \text{ x} = E; \{P(x)\}$
- Pour montrer  
 $\{P\} \text{ if } (E) \text{ S else S'} \{Q\}$   
il faut montrer  
 $\{P \wedge E\} \text{ S } \{Q\}$   
et  
 $\{P \wedge \neg E\} \text{ S'} \{Q\}$
- Pour montrer  
 $\{P\} \text{ while } (E) \text{ S } \{Q\}$   
il faut deviner l'**invariant**  $I$ , et montrer  $P \Rightarrow I$  et  $I \wedge \neg E \Rightarrow Q$  et  
 $\{I \wedge E\} \text{ S } \{I\}$

cf. logique de **Floyd-Hoare**.

Dans le triplet  $\{P\} \text{ S } \{Q\}$ , on appelle  $P$  et  $Q$  des pré-condition et post-condition de  $S$ .

# Logique de Hoare

|                                                                                     |                                                                                                                           |
|-------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| $\frac{}{\{P(E)\} \ x = E; \ {P(x)}}$                                               | $\frac{\{P \wedge E\} \ S \ {Q} \quad \{P \wedge \neg E\} \ S' \ {Q}}{\{P\} \ \text{if (E)} \ S \ \text{else } S' \ {Q}}$ |
| $\frac{\{P\} \ S \ {Q} \quad \{Q\} \ S' \ {R}}{\{P\} \ S \ S' \ {R}}$               | $\frac{\{P\} \ S \ {Q}}{\{P\} \ \{S\} \ {Q}}$                                                                             |
| $\frac{\{P \wedge E\} \ S \ {P}}{\{P\} \ \text{while (E)} \ S \ {P \wedge \neg E}}$ | $\frac{P \Rightarrow P' \quad \{P'\} \ S \ {Q'} \quad Q' \Rightarrow Q}{\{P\} \ S \ {Q}}$                                 |

Les **formules** sont des triplets  $\{P\}S\{Q\}$ .

Les **prémisses** d'une règle d'inférence sont au dessus de la barre.

La **conclusion** d'une règle d'inférence est en dessous.

Un **axiome** est une règle sans prémisse.

# Correction partielle – Correction totale

- $\{P\} S \{Q\}$  vrai montre que si  $P$  est vrai, alors  $Q$  l'est aussi. Tout dépend donc de ce qu'on veut prouver avec  $Q$ . Par exemple, on peut ne s'intéresser qu'à la valeur d'une seule variable, sans se soucier du reste.
- $\{P\} S \{Q\}$  vrai ne donne aucune indication de terminaison, puisque  $Q$  n'est vrai que si  $S$  termine. On dit qu'il y a **correction partielle**.
- seuls des ordinaux  $\Omega$  qui décroissent dans les boucles assurent la terminaison. Alors on a **correction totale**.
- assertions  $\neq$  **spécifications**. Faire la correspondance entre spécifications (« le cahier des charges ») et programmes est plus complexe et souvent imprécis.
- la correspondance entre spécifications et programmes peut se faire progressivement : programmation par raffinements successifs (*stepwise refinement*).
- les spécifications sont parfois formelles (B, TLA, Coq, Isabelle, HOL, PVS, etc) ; on peut alors formaliser la correspondance entre spécifications et programmes (« méthodes formelles »).

# Implémentation des assertions (1/2)

Java 1.4 : l'instruction `assert` lève `AssertionError` si l'assertion n'est pas vérifiée. Beaucoup de langages ont cette facilité : Caml, C, C++.

- ```
static void drapeauHollandais (int[ ] a) {
    int n = a.length; int b = 0, i = 0, r = n;
    assert phi(a,0,b,BLEU) && phi(a,b,i,BLANC) && phi(a,r,n,ROUGE)
        && i <= r;
    while (i < r) {
        switch (a[i]) {
            ...
        }
        assert phi(a,0,b,BLEU) && phi(a,b,i,BLANC) && phi(a,r,n,ROUGE);
    }
    assert phi(a,0,b,BLEU) && phi(a,b,r,BLANC) && phi(a,r,n,ROUGE);
}

static boolean phi (int[ ] a, int i, int j, int c) {
    return i >= j || a[i] == c && phi(a, i+1, j, c);
}
```
- ```
javac -source 1.4 DutchFlag.java
java -enableassertions DutchFlag 0 1 2
```

# Implémentation des assertions (2/2)

- On peut l'implémenter en Java 1.1.8 par :

```
class AssertionError extends Error { }
```

```
public class Assertion {
 public static void check (boolean e) {
 if (!e) throw new AssertionError();
 }
}
```

- On peut passer en argument de `Assertion.check` une expression booléenne quelconque correspondant à l'assertion à tester.
- Bien sûr, les assertions avec des quantificateurs  $\forall$  ou  $\exists$  sont plus dures à tester !

# Récursion et assertions (1/3)

- On veut montrer  $\{n \geq 0\} \text{ r} = \text{fact}(n); \{fact(n) = n!\}$   
Pour cela, on suppose (comme pour les boucles) que les appels (récursifs) utilisés vérifient déjà cette formule.
- ```
static int fact (int n) {  
    int r;  
    {n ≥ 0 ∧ ∀n fact(n) = n! }  
    if (n == 0) {  
        {n = 0}  
        r = 1;  
        {n = 0 ∧ r = 1 = 0!}  
    } else {  
        {n - 1 ≥ 0 ∧ fact(n - 1) = (n - 1)!}  
        r = n * fact(n-1);  
        {r = n(n - 1)! = n!}  
    }  
    {r = n!}  
    return r;  
}
```
- A nouveau, il s'agit de correction partielle.
Un autre argument montre la terminaison.

Récursion et assertions (2/3)

- On veut montrer $\{n \geq 0\} r = f(n); \{f(n) = \text{if } n > 100 \text{ then } n - 10 \text{ else } 91\}$

- static int f (int n) { // _____ fonction 91 [McCarthy]

int r;

$\{n \geq 0 \wedge \forall n' f(n') = \text{if } n' > 100 \text{ then } n' - 10 \text{ else } 91\}$

if (n > 100) {

$\{n > 100\}$

r = n-10;

$\{n > 100 \wedge r = n - 10\}$

} else {

$\{n \leq 100 \wedge \forall n' f(n') = \text{if } n' > 100 \text{ then } n' - 10 \text{ else } 91\}$

r = f(f(n+11));

$\{n \leq 100 \wedge r = \Phi(n)\} \leftrightarrow \{n \leq 100 \wedge r = 91\}$

}

$\{r = \text{if } n > 100 \text{ then } n - 10 \text{ else } 91\}$

return r;

} Exécution

$$\Phi(n) = f(\text{if } n + 11 > 100 \text{ then } n + 1 \text{ else } 91)$$

$$= \text{if } n \geq 90 \text{ then } f(n + 1) \text{ else } f(91)$$

$$= \text{if } n \geq 90 \text{ then if } n \geq 100 \text{ then } n - 9 \text{ else } 91 \text{ else } 91$$

$$= \text{if } n \geq 100 \text{ then } n - 9 \text{ else } 91$$

Réursion et assertions (3/3)

- Considérons l'insertion dans une liste triée

```
static Liste inserer (int x, Liste a) {  
    int r;  
    if (a == null || x < a.val) {  
        r = new Liste(x, a);  
    } else if (a.val < x) {  
        r = new Liste(a.val, inserer(x, a.suiv));  
    } else  
        r = a;  
    return r;  
}
```

- Considérons les propositions

$\{ord(a) = (a = null \vee a.suiv = null \vee a.val < a.suiv.val \wedge ord(a.suiv))\}$

- On cherche à montrer

$\{ord(a)\} r = inserer(x, a); \{ord(inserer(x, a))\}$

Exercice 8 Le démontrer... Indication : se servir de la formule

$\{prem(x, a) = (a = null \vee inserer(x, a).val = x \vee inserer(x, a).val = a.val)\}$

Ordres bien-fondés et terminaison (1/3)

- Une **relation d'ordre** (strict) \prec est une relation :

irréflexive $x \not\prec x$

transitive $x \prec y \prec z \Rightarrow x \prec z$

- Un ordre **bien fondé** est une relation d'ordre qui n'admet pas de chaîne infinie descendante $x_0 \succ x_1 \succ x_2 \succ \dots x_n \succ \dots$

Exemples d'ordre bien fondés

- $\langle \mathbf{N}, < \rangle$
- ordre lexicographique : $\langle \mathbf{N} \times \mathbf{N}, <_{lex} \rangle$

$$(x, y) <_{lex} (x', y') \quad \text{ssi} \quad x < x' \vee (x = x' \wedge y < y')$$

$(4, 3) >_{lex} (4, 2) >_{lex} (3, 15) >_{lex} (3, 6) >_{lex} (3, 4) >_{lex} (3, 2) >_{lex}$
 $(2, 21) >_{lex} (2, 19) >_{lex} (2, 12) >_{lex} (1, 90) \dots$

- ordre des multi-ensembles : $\langle \mathcal{P}(\mathbf{N}), <_{mul} \rangle$

$$E <_{mul} E' \quad \text{ssi} \quad E = F \uplus \{y_1, y_2, \dots, y_n\}, E' = F \uplus \{x\} \quad (y_i < x, n \geq 0)$$

$\{4, 5, 5\} >_{mul} \{3, 5, 5\} >_{mul} \{3, 4, 4, 4, 5\} >_{mul} \{3, 4, 4, 5\} >_{mul}$
 $\{3, 2, 2, 2, 1, 4, 5\} >_{mul} \{3, 1, 0, 2, 2, 1, 4, 5\} >_{mul} \{3, 1, 4, 5\} >_{mul} \dots$

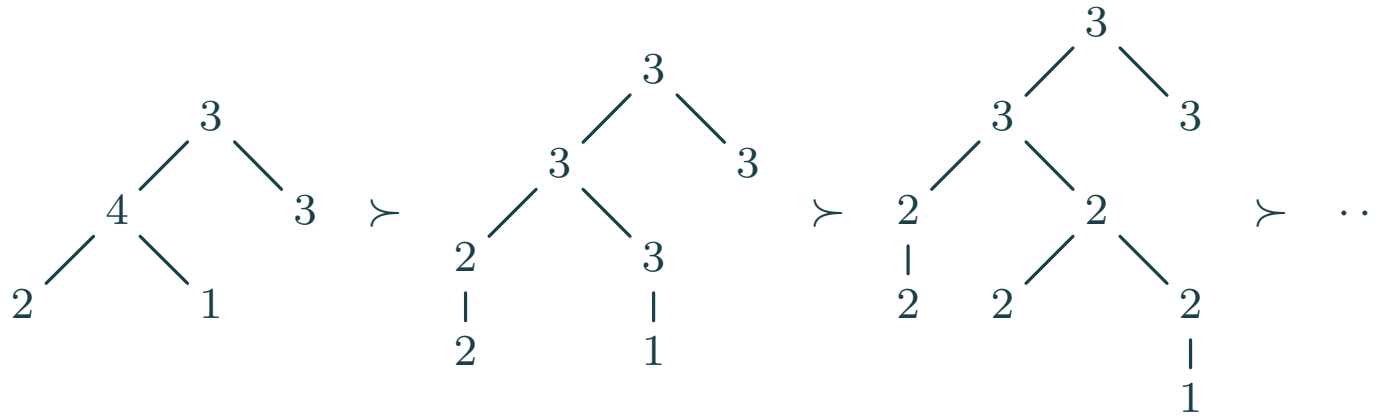
Ordres bien-fondés et terminaison (2/3)

- ordre de simplication : $\langle \mathcal{A}(\mathbf{N}), \prec \rangle$

$\mathcal{A}(\mathbf{N})$ arbres étiquetés par des entiers naturels

$t = n(t_1, t_2, \dots, t_n) \succ u = m(u_1, u_2, \dots, u_p)$ si $n > m$ et $t \succ u_1, t \succ u_2, \dots, t \succ u_p$

$t = n(\dots, t_i, \dots) \succ u = n(\dots, \dots)$



Théorie des w.q.o. (*well quasi orderings*) [Kruskal, Nash-Williams, Plaisted, Dershowitz]

Théorie des systèmes de réécriture

Démonstration automatique.

En fait tout ordre \succ sur les arbres étiquetés par un ensemble fini $E \subset \mathbf{N}$ vérifiant :

$t = n(t_1, t_2, \dots, t_n) \succ u = n(u_1, u_2, \dots, u_p)$ si $t_i \succ u_i$ pour un i

$t = n(\dots, t_i, \dots) \succ u = n(\dots, \dots)$

est bien fondé.

Ordres bien-fondés et terminaison (3/3)

Exercice 9 Trouver les ordres bien-fondés qui permettent de conclure à la terminaison de ces deux fonctions récursives (fonction 91 de [McCarthy] ou de [Ackermann]).

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

Exécution

Indications : considérer l'ordre lexicographique pour Ackermann et l'ordre partiel suivant pour 91 :

$$0 \succ 1 \succ 2 \succ \dots \succ 98 \succ 99 \succ 100 \succ x$$

où x quelconque tel que $x > 100$.

Exercices

Exercice 10 Montrer la correction de *Quicksort*.

Exercice 11 Montrer la correction de la fusion de listes triées.

Exercice 12 Montrer la correction du test d'acyclicité des graphes par *depth-first-search*.

Exercice 13 Montrer que la terminaison peut être une propriété aussi dure à montrer que la correction.