

Exécution Symbolique

D'abord un **mini-langage** :

- déclaration de procédure : **nom**: **PROCEDURE** (paramètre₁, paramètre₂, ... paramètre_n)
 <suite-d' instructions>
 END;
- deux types de procédures : fonctions et sous-routines (**CALL**)
- toute variable a des valeurs entières : **DECLARE** var₁, var₂, ... var_n **INTEGER**
- opérations de base : + * - ...
- affectation : **var** := <expression>
- composition : **DO** <suite-d' instructions> **END**
- opérations booléenne : true, false, <, >, =, ≥, ≤, ≠, & (et), | (ou), → (implique), ¬ (not)
- conditionnelle : **IF** <booléen> **THEN** instruction₁ **ELSE** instruction₂
- itérative : **DO WHILE** <booléen>; <suite-d' instructions> **END**;
- paramètres par référence, **RETURN**; (subroutine), **RETURN**(<expr>); (fonction)

Assertions

des contraintes sur les entrées et sorties peuvent être exprimées par des **assertions** :

assertion d'entrée : **ASSUME** (<bool éen>);

exemple: **ASSUME** (P1 > 0);

assertion de sortie : **PROVE** (<bool éen>);

exemple: **PROVE** ((X = Y) & (Y = X));

Un programme est (partiellement) correct si la vérification de l'entrée implique la vérification de l'assertion de sortie

Un programme exemple (simple)

1 **ABSOLUTE:**

PROCEDURE(X) ;

2 **ASSUME(true) ;**

3 **DECLARE X, Y INTEGER;**

4 **IF X < 0**

5 **THEN Y := -X**

6 **ELSE Y := X**

7 **PROVE((Y = X | Y = -X) & Y ≥ 0 & X = X);**

8 **RETURN(Y) ;**

9 **END;**

Exécution symbolique de ABSOLUTE

Un appel typique : **ABSOLUTE**(a) ;

Cas ❶ : $a < 0$ test-IF = true \Rightarrow THEN $Y := -a$

$$Y = -a \quad \mathbf{X} = \underline{\mathbf{X}} = a$$

$$\Rightarrow (-a = a \mid -a = -a) \ \& \ -a \geq 0 \ \& \ a = a$$



démontrer: $-a \geq 0$ ou $a < 0$

Cas ❷ : $a \geq 0$ test-IF = false \Rightarrow ELSE $Y := a$

$$Y = a \quad \mathbf{X} = \underline{\mathbf{X}} = a$$

$$\Rightarrow (a = a \mid a = -a) \ \& \ a \geq 0 \ \& \ a = a$$



démontrer: $a \geq 0$

équivalent exécution réelle = exécution algébrique

- Affectation change

exemple: $X=a$ $Y=\beta$

$X:=Y+X \mapsto X:= a+\beta$

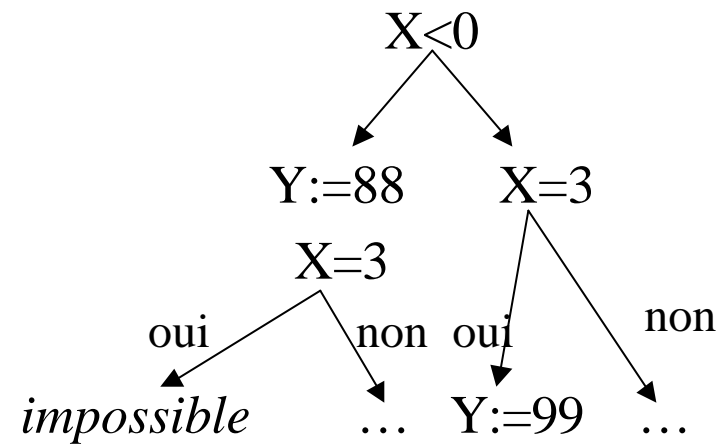
$Y:=3*X-Y \mapsto Y:=3a+2\beta$

- **CALL** et **RETURN** ne changent pas
- **IF** change : soit **true**, soit **false** soit *indéterminé*
si *indéterminé*, alors analyse de cas (pour la preuve et/ou pour trouver des cas impossibles)

exemple : **IF X<0 THEN Y: =88**

IF X=3 THEN Y: =99

Exemple de cas impossible

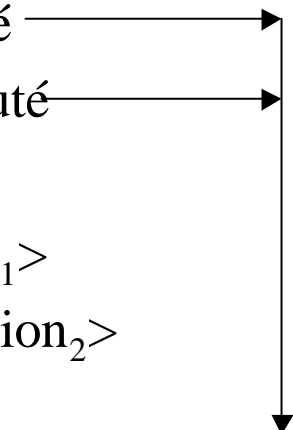


Conditions de chemin (*path-condition*)

La path-condition (PC) donne à *tout* instant de l'exécution symbolique l'ensemble des valeurs symboliques en vigueur.

Exemple : exécution symbolique de

IF <bool> **THEN** <expression₁> **ELSE** <expression₂>

- ❶ eval de <bool> donne B (symbolique)
 - ❷ si $PC \supset B$ alors <expression₁> est toujours exécuté
si $PC \supset \neg B$ alors <expression₂> est toujours exécuté
si ni B ni $\neg B$ alors continuer en ❸
 - ❸ assumer B et $PC := PC_{old} \& B$ exécuter <expression₁>
 - ❹ assumer $\neg B$ et $PC := PC_{old} \& \neg B$ exécuter <expression₂>
- 

exécution symbolique des assertions

- Exécution symbolique de **ASSUME**

ASSUME(<bool>) ;

1 eval de <bool> donne \mathbb{B} (symbolique)

2 $PC := PC_{old} \& \mathbb{B}$

- Exécution symbolique de **PROVE**

PROVE(<bool>) ;

1 eval de <bool> donne \mathbb{B} (symbolique)

2 si $PC \Rightarrow \mathbb{B}$ alors le programme est vérifié
sinon le programme est \neg vérifié

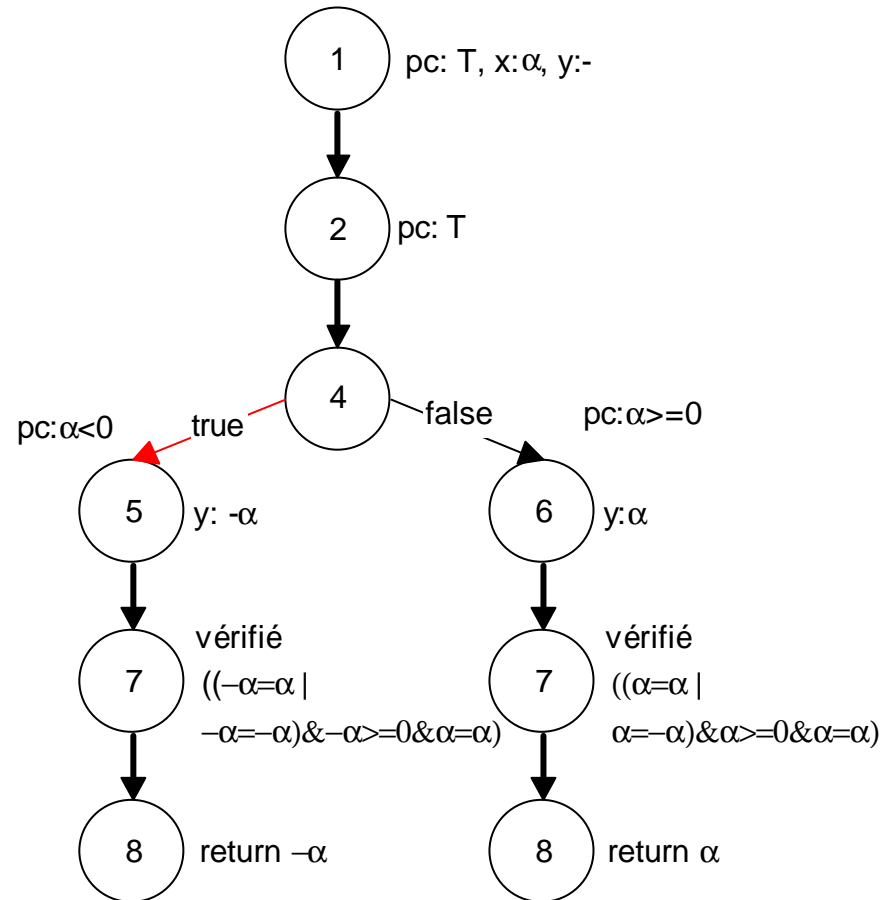
Arbre d'exécution symbolique

1 ABSOLUTE:

```

    PROCEDURE(X);
2  ASSUME(true);
3  DECLARE X, Y INTEGER;
4  IF X < 0
5      THEN Y := -X
6      ELSE Y := X
7  PROVE((Y = X | Y = -X) & Y ≥
8      0 & X = X);
8  RETURN(Y);
9  END;

```



Un 2^{ème} programme exemple (moins simple)

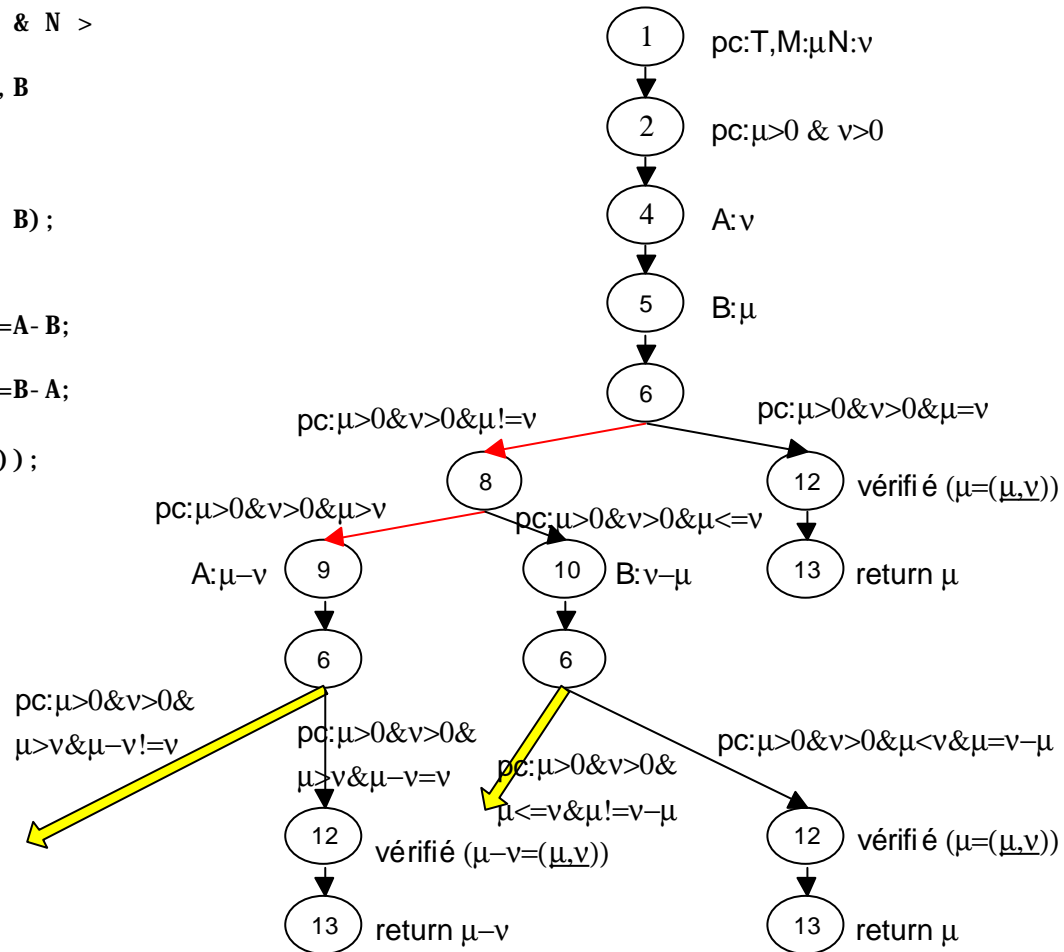
```
1 GCD: PROCEDURE (M, N)
2   ASSUME (M > 0 & N > 0);
3   DECLARE M, N, A, B INTEGER;
4   A: =M;
5   B: =N;
6   DO WHILE (A = B);
7
8       IF A > B
9           THEN A: =A- B;
10          ELSE B: =B- A;
11      END
12  PROVE (A=(M, N));
13  RETURN(A);
14 END
```

Arbre d'exécution pour GCD

```

1 GCD: PROCEDURE (M, N)
2   ASSUME (M > 0 & N > 0);
3   DECLARE M, N, A, B
4   INTEGER;
5   A: =M;
6   B: =N;
7   DO WHILE (A = B);
8     IF A > B
9       THEN A: =A- B;
10      ELSE B: =B- A;
11     END
12   PROVE (A=(M, N));
13   RETURN(A);
14 END

```



Comment éviter les arbres infinis

chaque traversée d'une itération peut être marquée par une *coupure* au moins à *un* endroit de l'itération :

1. commencer par une *coupure* et arrêter au **RETURN** ou à la *coupure* suivante
2. Associé à chaque *coupure* un **ASSUME** et à sa fin un **PROVE** \mapsto on peut démontrer les coupures
3. Preuve du programme devient somme des preuves des coupures

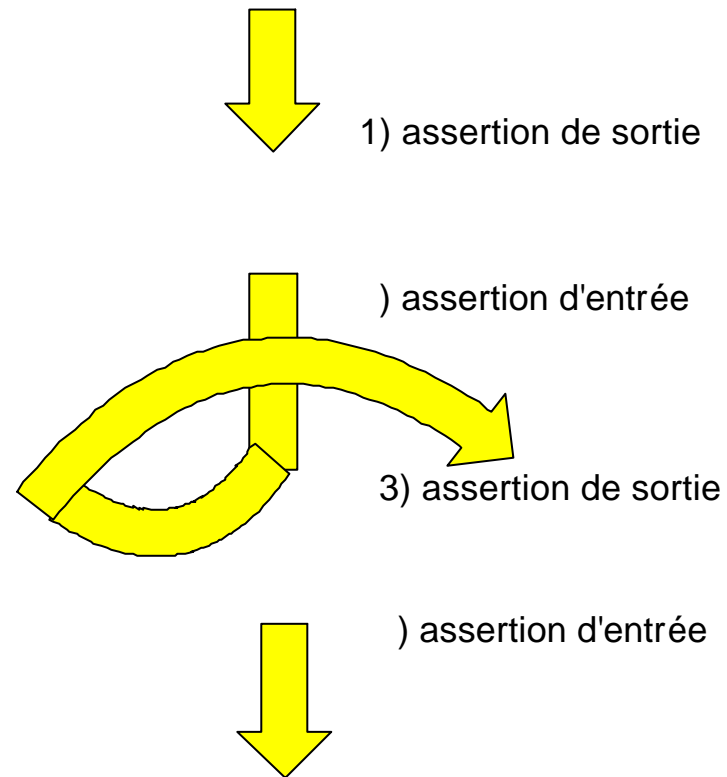
ASSERT

ASSERT (<bool>);

- au premier rencontre, **ASSERT** joue le rôle d'un **PROVE** c'est alors une assertion de sortie pour l'exécution symbolique arrivant à la coupure. C'est également une assertion d'entrée, un **ASSUME**, pour l'exécution symbolique commençant à la coupure.
- Sinon c'est la dernière instruction d'une coupure, c'est alors une assertion de sortie, un **PROVE**, qui devient une assertion d'entrée, un **ASSUME**, pour l'exécution symbolique suivant la coupure

ASSERT (suite)

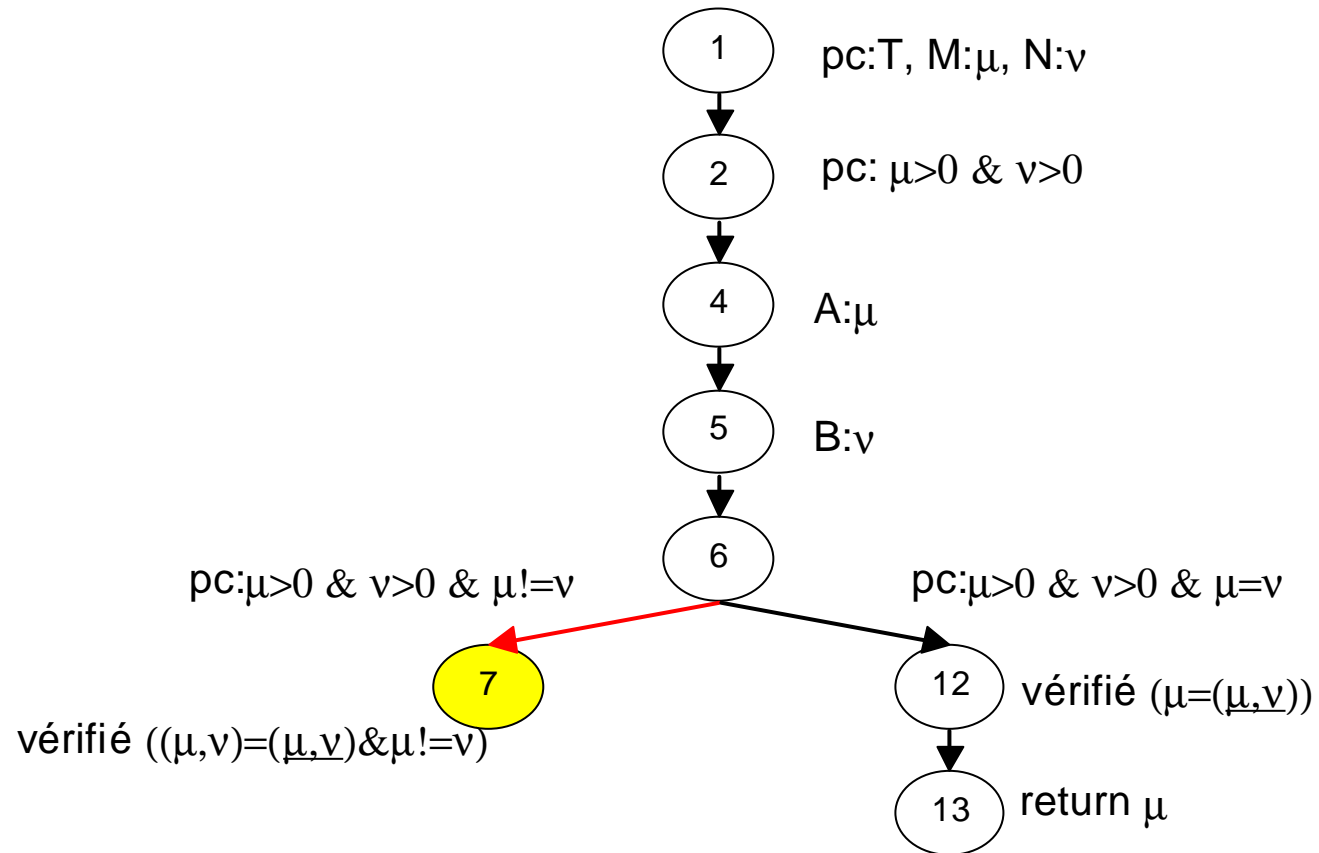
ASSERT



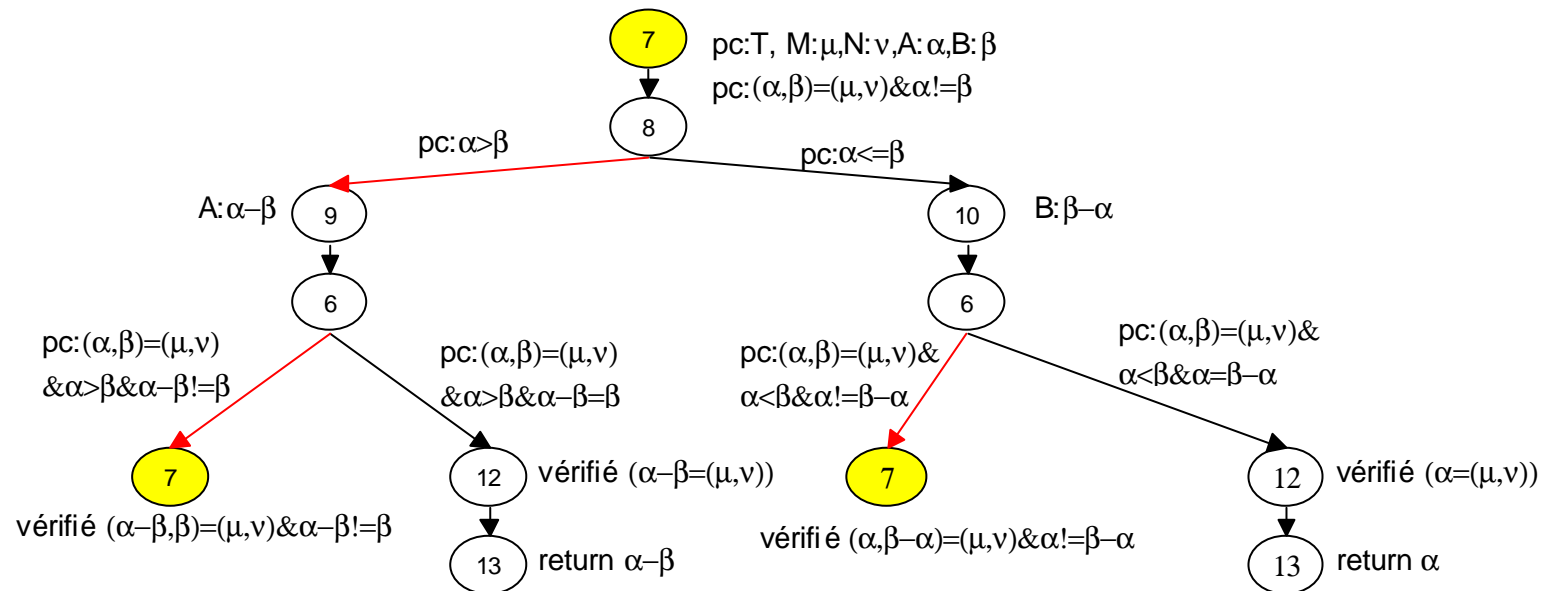
Retour vers GCD

```
1 GCD: PROCEDURE (M, N)
2   ASSUME (M > 0 & N > 0); ----- coupure2
3   DECLARE M, N, A, B INTEGER;
4   A: =M;
5   B: =N;
6   DO WHILE (A = B);
7   ASSERT ((A, B)=(M, N) & A ≠ B); ----- coupure7
8       IF A > B
9           THEN A: =A- B;
10          ELSE B: =B- A;
11      END
12  PROVE (A=(M, N));
13  RETURN(A); ----- return
14 END
```

et l'arbre arrivant à la coupure₇



Ensuite l'arbre à partir de la coupure₇



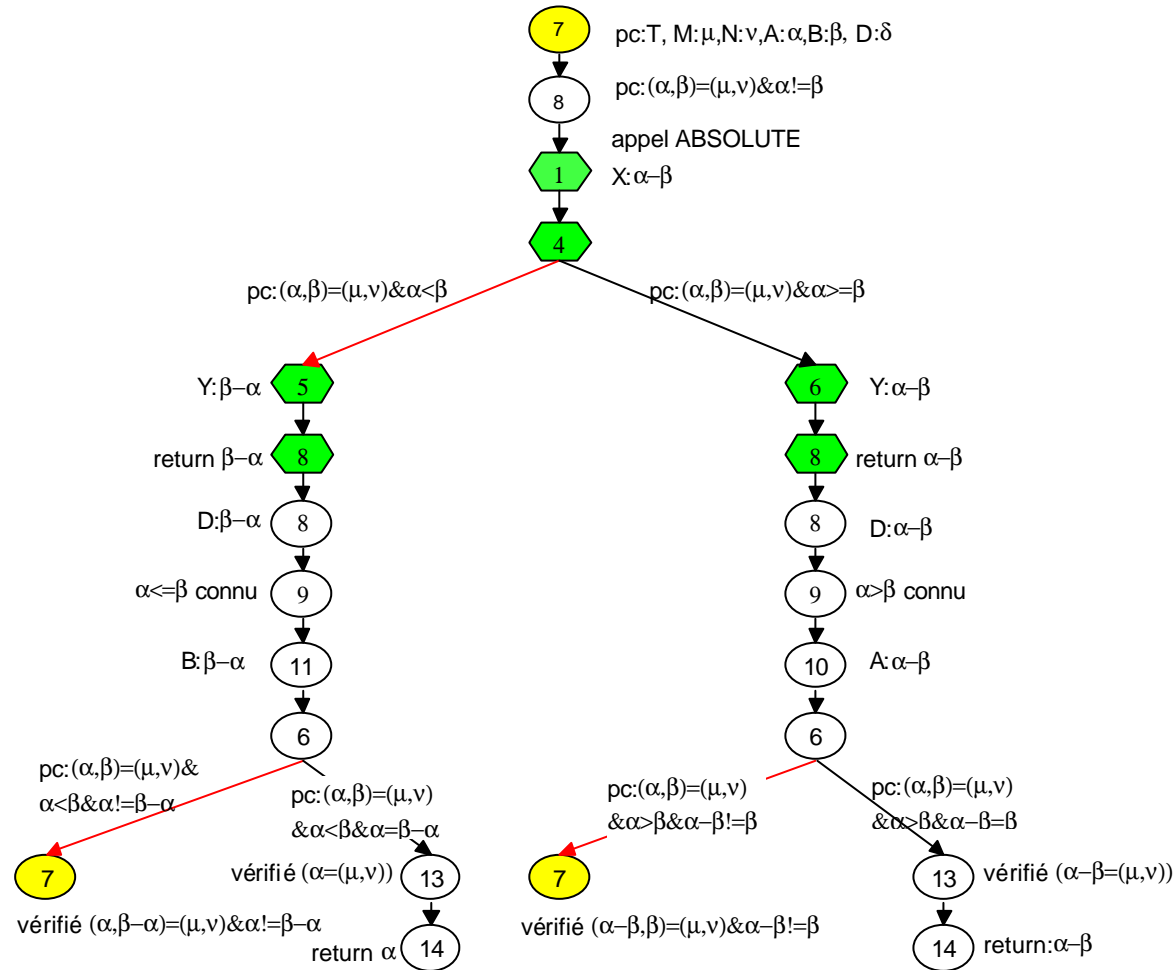
deuxième version de GCD

```
1 GCD2: PROCEDURE (M, N);
2   ASSUME (M>0 & N>0); _____ coupure2
3   DECLARE M, N, A, B, D INTEGER;
4   A := M;
5   B := N;
6   DO WHILE (A ≠ B);
7     ASSERT ((A, B) = (M, N) & A ≠ B); _____ coupure7
8     D := ABSOLUTE(A - B);
9     if A > B
10        THEN A := D;
11        ELSE B := D;
12    END;
13    PROVE (A = (M, N));
14    RETURN (A); _____ return
15 END;
```

Exécution symbolique

- première partie (**1** → **7** et **1** → **return**) ne change pas
- L'exécution de **7** → **7** et **7** → **return** doit refaire l'exécution de **ABSOLUTE** chaque fois

Arbre de GCD2



Sous-procédures

1. On invente des symboles pour chaque variable de la procédure appelante qui risque de changer de valeur par l'appel
2. Au lieu d'exécuter symboliquement le corps de la sous-procédure, les valeurs des variables potentiellement changées sont remplacées par les nouvelles symboles
3. Si la sous-procédure a été démontrée correcte, son assertion de sortie est valide pour ces nouvelles valeurs et donne les informations nécessaires sur ces valeurs pour démontrer le programme

Utiliser les preuves comme des lemmes

On crée des sous-procédures *abrégées* en :

1. Changeant le **ASSUME** initial en **PROVE** (sans changer les arguments)
2. Changeant le **PROVE** final en un **ASSUME** (sans changer les arguments)
3. Remplaçant le corps de la sous-procédure par une séquence d'affectations (de nouveaux symboles), une pour chaque variable qui peut être altérée par la procédure

ABSOLUTE abrégée

```
1 ABSOLUTE: PROCEDURE (X);  
2   PROVE (T)  
3   DECLARE X, Y INTEGER;  
4   X := newsymbol();  
5   Y := newsymbol();  
6   ASSUME ((Y=X | Y=-X) & Y≥0 & X=X);  
7   RETURN (Y)  
8 END;
```

Nouvel arbre de GCD2

