

# Introduction à la vérification de programmes

Polytech Paris-Saclay, PEIP 2, Informatique Option S3

---

Thibaut Benjamin, Henri Saudubray, Philippe Volte-Vieira

18 Décembre 2025

Cours 2

# Format et date de l'Examen

- ▶ Examen de 1h30 (2h pour les tiers-temps)
- ▶ Le 9 Janvier 2026
- ▶ Au format TP, avec des exercices type

- ▶ La semaine dernière, nous avons vu comment utiliser la méthode du variant pour prouver la terminaison des boucles.
- ▶ **Rappel** : Un variant est une quantité entière positive associée à une boucle, et qui décroît strictement à chaque fois que l'on entre dans cette boucle.

- ▶ Les contrats de fonctions
  - Pré-conditions : ce que l'on doit vérifier à chaque appel de la fonction.
  - Les variants pour les fonctions récursives.
  - Post-conditions : ce que la fonction garantit.
- ▶ Contrats pour les fonctions de tri

## Les contrats de fonction

# Les contrats de fonction

## Les pré-conditions

## Un premier exemple

```
// renvoie le dernier element du tableau  
int renvoieDernier (int n, int tab[]){  
    return tab[n-1];  
}
```

Démo en Why3

## Un premier exemple

```
// renvoie le dernier element du tableau
int renvoieDernier (int n, int tab[]){
    return tab[n-1];
}
```

### Démo en Why3

- ▶ A quoi correspond l'obligation de preuve générée par Why3?
- ▶ Quelle information manque pour satisfaire la condition?



## Pré-conditions d'une fonction

On peut attacher une précondition à une fonction (mot-clé `@requires` en Why3)

```
// renvoie le dernier element du tableau
int renvoieDernier (int n, int tab[]){
    //@requires 1 <= n <= length (tab);
    return tab[n-1];
}
```

- ▶ A l'intérieur de la fonction on suppose que les préconditions sont vraies

Démo

- ▶ A chaque appel de la fonction, il faut prouver les préconditions

Démo

## Pré-conditions pour prouver la correction

- ▶ Dans l'exemple précédent, on a donné une précondition pour vérifier l'absence de bugs : pas d'accès à des zones du tableau non définies.
- ▶ Si on veut garantir que la fonction renvoie effectivement le dernier élément du tableau, il faut une précondition plus précise.

Démo

## Exercice : affichage d'un tableau

Donner les préconditions pour la fonction suivante

```
void afficherTableau(int n, int tab []){  
    //Precondition ?  
    for (int i = 0; i<n; i++){  
        //@variant n - i;  
        printf("%i ", tab[i]);  
    }  
}
```

Démo

**Attention** : Même avec la bonne précondition, why3 ne saura pas nécessairement montrer l'absence de bugs!

## Exercice : Addition bête

Donner le variant et les préconditions pour la fonction suivante

```
int additionBete (int x, int y){  
    // Contrat?  
    while (y != 0){  
        // Variant ?  
        x = x + 1;  
        y = y - 1;  
    }  
    return x;  
}
```

Démo

## Exercice : Afficher de k en k

Donner le variant et les préconditions pour la fonction suivante

```
void afficherEnK (int n, int tab[], int k){  
    //Precondition?  
    for (int i = 0; i < n; i = i + k){  
        //Variant?  
        printf("%i ", tab[i]);  
    }  
}
```

Démo

## Exercice : Afficher de k en k

Donner le variant et les préconditions pour la fonction suivante

```
void afficherEnK (int n, int tab[], int k){  
    //Precondition?  
    for (int i = 0; i < n; i = i + k){  
        //Variant?  
        printf("%i ", tab[i]);  
    }  
}
```

Démo

**Remarque** : Dans cet exemple, c'est la précondition qui rend le variant correct.  
Comment interpréter cela ?

## Exercice : pgcd

Donner les préconditions pour la fonction suivante

```
int pgcd (int a, int b){  
    //Preconditions  
    while(a != b){  
        //@variant a + b;  
        if (a > b) { a = a - b; }  
        else{ b = b - a; }  
    }  
    return a;  
}
```

Démo

## Les contrats de fonction

Les variants de fonctions récursives



- ▶ Pour une fonction récursive, on peut ajouter un variant au contrat de la fonction.
- ▶ Dans ce contexte, un variant est un entier positif qui décroît à chaque appel de la fonction

## Exemple

```
int factorielle (int n){  
    // @requires n >= 0;  
    // @variant n;  
    if (n == 0) { return 1; }  
    return n * factorielle (n-1);  
}
```

Combiner la précondition et le variant permet de prouver l'arrêt du programme.

Démo

## Exercice : Fibonacci

Donner les préconditions et le variant pour prouver l'arrêt de la fonction de calcul des nombres de Fibonacci.

```
int fibo (int n){  
    // Contrat  
    if (n <= 1){ return n; }  
    return fibo(n-1) + fibo (n-2);  
}
```

Démo

## Exercice : PGCD récursif

Écrire les préconditions et le variant pour prouver la terminaison de l'algorithme du pgcd récursif.

```
int pgcd (int a, int b){  
    // Contrat  
    if (a == b){ return a; }  
    else if (a > b){ return pgcd(b, a-b); }  
    else { return pgcd (a, b-a); }  
}
```

Démo

# Les contrats de fonction

## Les post-conditions

## Une question pour le moment insoluble

```
int abs (int x){  
    if (x >= 0) { return x; }  
    else { return -x; }  
}  
  
void fonction (int n){  
    //@requires n >= 0;  
    ...  
}  
  
int main (){  
    fonction (abs (-5)); //Comment verifier la precondition?  
    return 0;  
}
```

## Solution : une post-condition

- ▶ Pour pouvoir prouver la précondition, il faut un moyen de dire que la fonction **abs** renvoie toujours un entier positif
- ▶ C'est le rôle d'une post condition (mot-clé **@ensures** en Why3)

```
// valeur absolue
int abs (int x){
    //@ensures  result >= 0;
    if (x >= 0) { return x; }
    else { return -x; }
}
```

Démo

## Exercice : Addition bête

Donner les post-conditions pour la fonction suivante

```
int additionBete (int x, int y){  
    // Contrat?  
    while (y != 0){  
        // Variant ?  
        x = x + 1;  
        y = y - 1;  
    }  
    return x;  
}
```

Démo



## Exercice : PGCD

```
int pgcd (int a, int b){  
    //Postconditions  
    while(a != b){  
        if (a > b) { a = a - b; }  
        else{ b = b - a; }  
    }  
    return a;  
}
```

### Démo

**Indice** : Commencer par donner les conditions disant que c'est un diviseur commun, puis la condition disant que c'est le plus grand.

Pour la dernière post-condition, il faudra utiliser le mot-clé **forall**.

## Définition de fonctions et prédicats

- Pour s'aider à donner les conditions, on peut définir des fonctions et prédicats auxiliaires.

```
//@predicate positif (int x) = x >= 0;  
//@function int carre (int x) = x * x;
```

- On peut même utiliser des quantificateurs (**forall**, **exists**)

```
/*@predicate tousPositifs (int a[], int n) =  
    forall i. 0 <= n < length(a) -> positif (a[i]);*/  
/*@predicate existeUnPositif (int a[], int n) =  
    exists i. 0 <= n < length(a) -> positif (a[i]);*/
```

Contrats pour le tri par insertion récursif

## Rappel : Le tri par insertion

- ▶ Pour procéder au tri par insertion, on commence par trier le premier élément du tableau, puis les deux premiers, puis les trois premiers, etc
- ▶ A l'étape, on performe une insertion, permettant d'insérer l'élément à la position  $n$  dans la bonne position parmi les  $n$  premiers éléments déjà triés du tableau.

## Exercice : Prédicat de tri

- ▶ Définir un prédicat **trieJusque** prenant un argument un tableau d'entier et un entier **n** et qui détermine si les  $n$  premières entrées du tableau sont triées.
- ▶ Définir un prédicat **trie** prenant un argument un tableau d'entier et qui détermine si le tableau est trié ou non.

## Exercice : La fonction d'insertion

Donner le contrat associé à la fonction d'insertion dans le tableau.

```
void insertion (int tab[], int n){  
    // Contrat  
    // CODE DE LA FONCTION  
}
```

**Remarque** : Le code de la fonction n'est pas nécessaire pour donner le contrat : modularité.

Démo

## Exercice : La fonction de tri par insertion récursive

Donner le contrat associé à la fonction récursive de tri par insertion

```
void triInsertionRec (int tab[], int n){  
    // Contrat  
    if (n == 0) {}  
    else{  
        triInsertionRec (tab, n-1);  
        insertion(tab, n);  
    }  
}
```

Démo

## Exercice : La fonction de tri par insertion récursive

Donner le contrat associé à la fonction récursive de tri par insertion

```
void triInsertion(int tab[], int n){  
    // Contrat  
    triInsertionRec(tab, n);  
}
```

Démo



- ▶ La logique de la récursivité est correcte.
- ▶ Des lors que l'on fournit une fonction d'insertion correcte, notre tri par insertion est correct.

## Une spécification incomplète ?

- ▶ Trouver une fonction qui satisfait le contrat précédent, mais qui n'est pas le tri d'un tableau.
- ▶ Comment affiner notre contrat pour complètement spécifier le tri ?  
Ne pas le faire avec Why3.