Test et Validation du Logiciel
Test Structurel

Sami Taktak

sami.taktak@cnam.fr

Centre d'Étude et De Recherche en Informatique et Communications
Conservatoire National des Arts et Métiers

le cnam

Test Structurel ou Test boîte blanche

- Utilise la structure pour dériver des cas de tests
  - Au niveau unitaire : utilise le code (instructions, conditions, branchement)
  - Au niveau intégration : utilise le graphe d'appel entre modules
  - Au niveau système : utilise les menus, les processus
- Complète les tests boîte noire en étudiant la réalisation (et non la spécification)
- Est associée à des critères de couverture
  - Couverture de toutes les instructions
  - Couverture de toutes les conditions
  - Couverture de toutes les utilisations d'une variable

Méthodes de Test Structurel

2 méthodes utilisées :

- Dérivation de jeux de tests à partir du graphe de contrôle :
  - Suit l'enchaînement des opérations qui sont représentées par un graphe
  - Cherche à couvrir toutes les instructions, tous les chemins
- Dérivation de jeux de tests à partir du flot de données
  - Suit la vie des variables au cours de l'exécution du programme : définition, utilisation, destruction
  - Cherche à couvrir toutes les affectations, toutes les utilisations

Test à Partir du Graphe de Contrôle

- Technique de test structurel la plus ancienne
- Technique de base pour beaucoup d'autres tests
- Détaillée et complexe
- Utilisée principalement pour les tests unitaires et les tests de composants
- Basée sur les chemins d'exécutions
- Différents types de couverture peuvent être visés

Graphe de Contrôle

- Un graphe de contrôle est constitué d'arcs et de sommets
- Chaque arc représente **une ou plusieurs** instructions
- Chaque sommet représente :
  - Soit un départ conditionnel (sur 2 branches ou plus)
  - Soit la jonction de branches incidentes
- Un graphe de contrôle permet de construire les chemins
- Un chemin est une suite d'arcs reliant un point d'entrée du programme à un point de sortie
- Il y peut y avoir **beaucoup** de chemins possibles et certains chemins impossibles

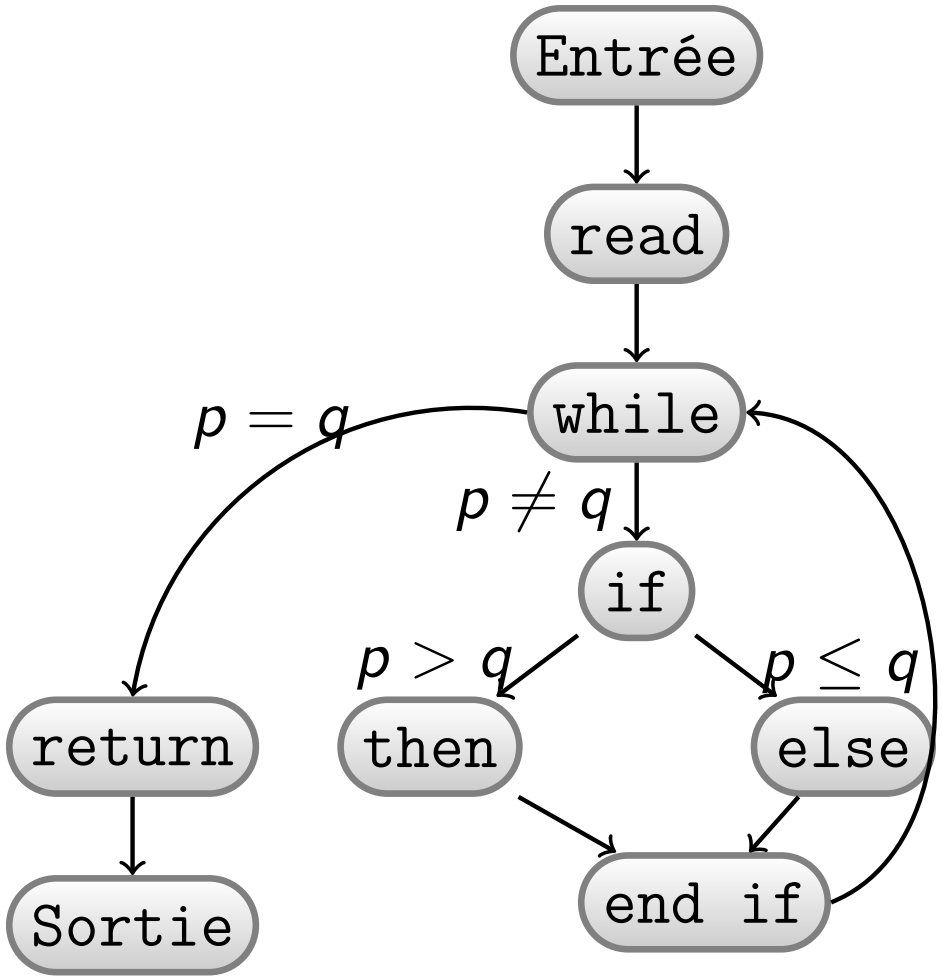
Exemple de Graphe de Contrôle

Calcul du PGCD de 2 entiers

Pré-condition : p et q entiers naturels positifs

PGCD(p, q) :

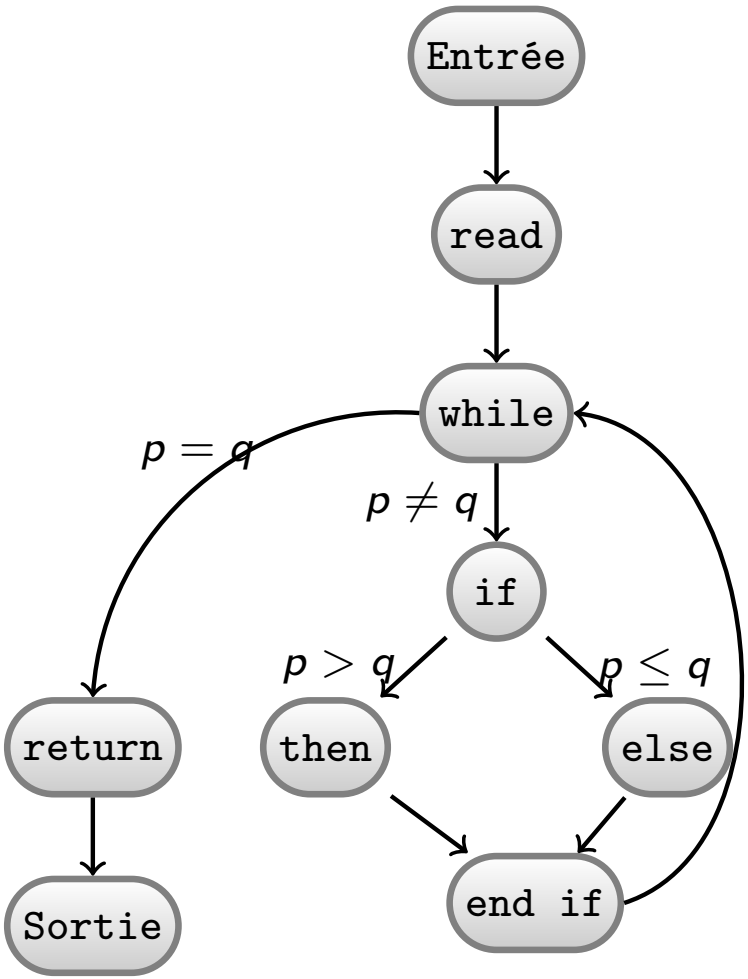
1: read(p, q)
2: **while** $p \neq q$ **do**
3: **if** $p > q$ **then**
4: $p = p - q$
5: **else**
6: $q = q - p$
7: **end if**
8: **end while**
9: return p



Objectifs de Couverture

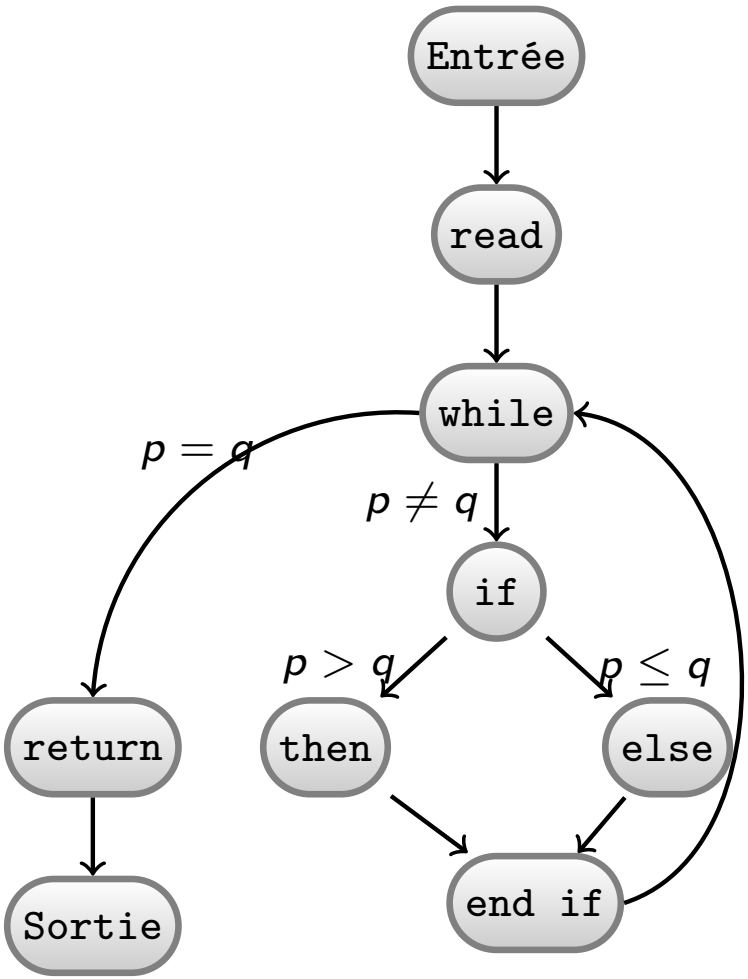
- On définit plusieurs niveaux (objectifs) de couverture
  - C0 : tester toutes les instructions (nœuds « fonction ») au moins une fois
  - C1 : tester toutes les décisions possibles au moins une fois (ou toutes les branches)
  - C-i : tester toutes les branches (C1) mais en passant i fois dans chaque boucle : (i- chemins)
  - C∞ : tester tous les chemins possibles
- C∞ est virtuellement impossible
- C1 et C0 sont différents
- En pratique, le test d'un programme consiste à satisfaire C0 et C1

Exemples de séquences



- Tous les noeuds (C0) :
  (Entrée, read, while, if, then, return, Sortie)
  (Entrée, read, while, if, else, return, Sortie)
- Toutes les décisions (C1) :
  idem + (Entrée, read, while, return, Sortie)

Exemples de séquences



- Tous les 2-chemins (C-2) :
  (Entrée, read, while, if, then, return, Sortie)
  (Entrée, read, while, if, else, return, Sortie)
  (Entrée, read, while, return, Sortie)

  (Entrée, read, while, if, then, while, if, then, return, Sortie)
  (Entrée, read, while, if, then, while, if, else, return, Sortie)
  (Entrée, read, while, if, else, while, if, then, return, Sortie)
  (Entrée, read, while, if, else, while, if, else, return, Sortie)

Graphe de Contrôle

Graphe orienté et connexe (N, A, E, S) où

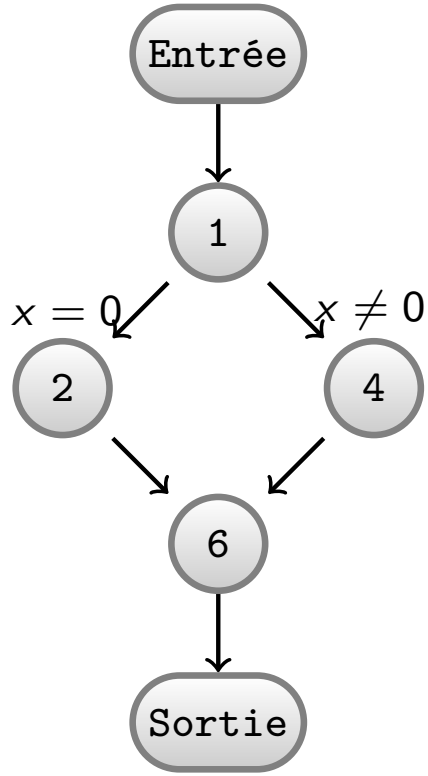
- N : ens. de sommets =
  bloc d'instructions exécutés en séquence
- A : relation de N x N =
  branchement possible du flot de contrôle
- E : sommet « d'entrée » du programme
- S : sommet « de sortie » du programme

Couverture de *Tous-Les-Nœuds*
(ou *Toutes-Les-Instructions*)

- Critère atteint lorsque tous les nœuds du graphe de contrôle sont parcourus
- Taux de couverture : $\frac{\text{nb de nœuds couverts}}{\text{nb total de nœuds}}$
- Exigence minimale pour la certification en aéronautique (norme RTCA/DO-178B ou EUROCAE ED-12B)
- Qualification niveau C :
  Un défaut peut provoquer un problème sérieux entraînant un dysfonctionnement des équipements vitaux de l'appareil

Couverture de *Tous-Les-Nœuds*
(ou *Toutes-Les-Instructions*)

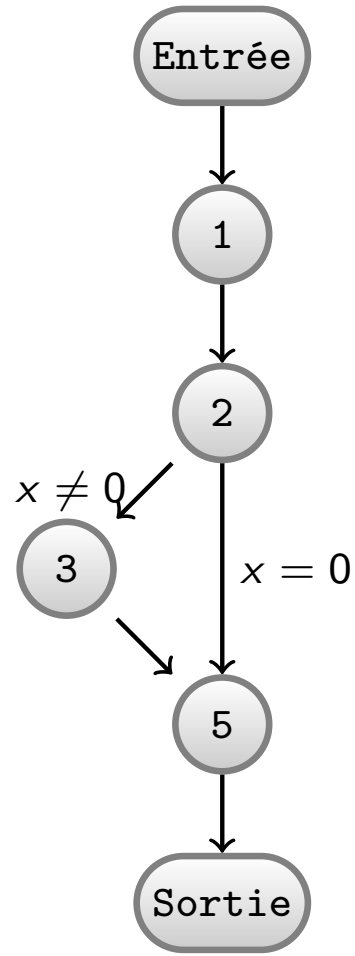
```
sum (x, y : entier):
 1: if x = 0 then
 2:     sum := x;
 3: else
 4:     sum := x + y;
 5: end if
 6: return sum;
```



⇒ L'erreur détectée par exécution du chemin $(E, 1, 2, 6, S)$

Limites du Critère *Tous-Les-Nœuds*

1: read(x) ;
2: **if** $x \neq 0$ **then**
3: $x := 1$;
4: **end if**
5: $y := 1/x$;



- Critère *tous-les-nœuds* satisfait par le chemin $(E, 1, 2, 3, 5, S)$
- Mais la division par zéro n'est pas détectée !

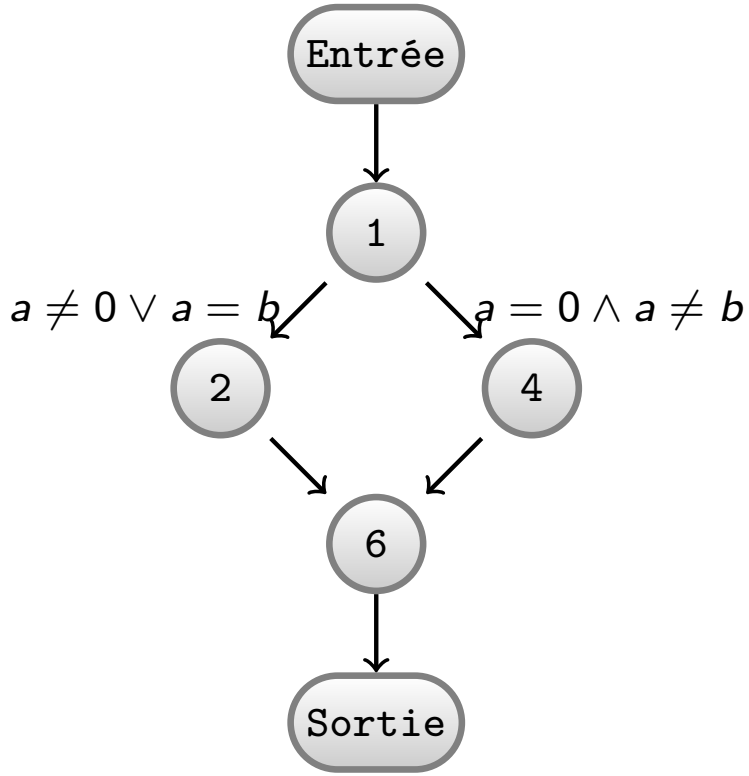
Couverture de *Toutes-Les-Décisions*
ou *Toutes-Les-Arcs*

- Pour chaque décision, un test rend la décision vrai, un test rend la décision fausse
- Taux de couverture : $\frac{\text{nb des arcs couverts}}{\text{nb total des arcs}}$
- Couverture de tous les arcs $\Leftrightarrow$ Couverture de toutes les valeurs de vérité pour chaque nœud de décision
- Norme DO 178B, qualification des systèmes embarqués au niveau B : un défaut peut provoquer un problème majeur entraînant des dégâts sérieux

critère *tous-les-arcs* est totalement réalisé
$\Rightarrow$ critère *tous-les-nœuds* est satisfait

Limites des Critères *Toutes-les-Décisions*

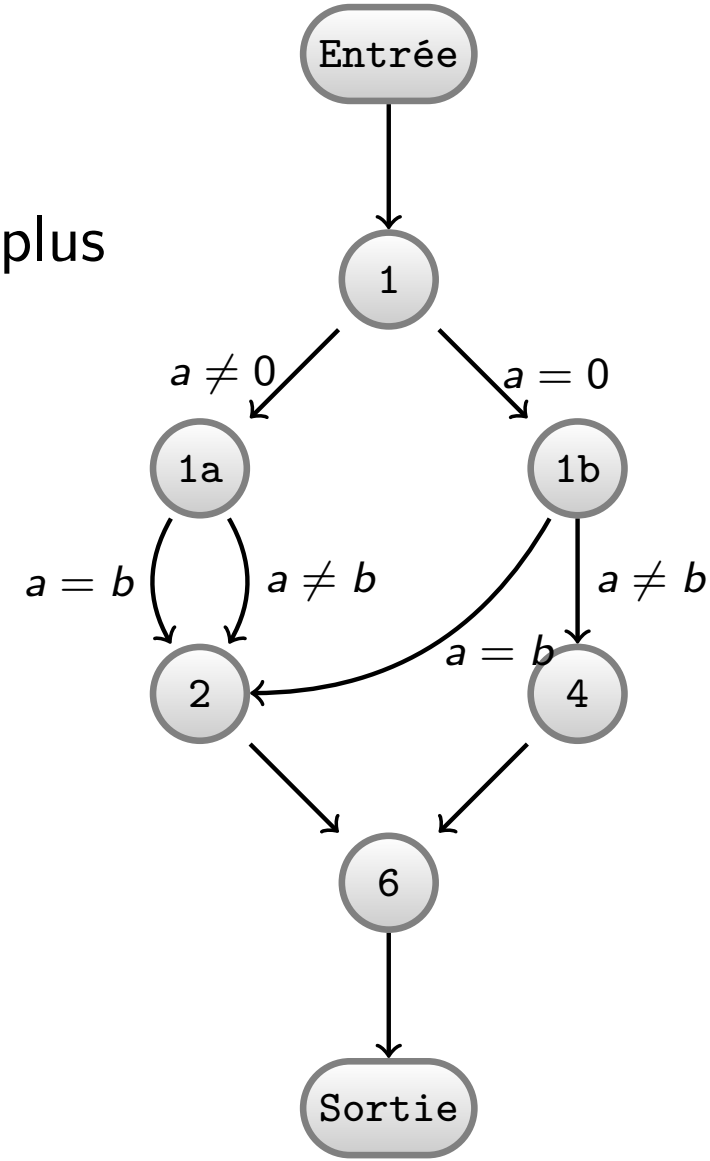
```
F (a, b : entier):
1: if a ≠ 0 ∨ a = b then
2:     x := 1 / a;
3: else
4:     x := 0;
5: end if
6: return x;
```



- Critère *Toutes-les-Décisions* satisfait avec $\{a = 0, b = 1\}$ et $\{a = 2, b = 1\}$
- Division par zéro non détectée lorsque $\{a = 0, b = 0\}$

Limites des Critères *Toutes-les-Décisions*

- Décomposer les conditionnelles dans le graphe de contrôle
- Critère *Toutes-les-Décisions* n'est plus satisfait avec $\{a = 0, b = 1\}$ et $\{a = 2, b = 1\}$
- Critère *Toutes-les-Décisions* satisfait avec $\{a = 0, b = 1\}$, $\{a = 2, b = 1\}$, $\{a = 2, b = 2\}$ et $\{a = 0, b = 0\}$



Couverture *Toutes-les-Conditions* Multiples

- Critère de *condition multiple* satisfait si :
  - Critère *tous-les-arcs* satisfait
  - Dans chaque expression, les conditions prennent toutes les combinaisons de valeurs possibles
  - **`Si A` $\wedge$ `B Alors`**... nécessite :
    - A = B = vrai
    - A = B = faux
    - A = vrai, B = faux
    - A = faux, B = faux

$\Rightarrow$ Problème de combinatoire : nombre de cas de test exponentielle en fonction du nombre de conditions

Couverture *Toutes-les-Conditions – Décisions* Modifié (MC/DC)

- Objectif : améliorer les critères de couverture basés sur les décisions tout en contrôlant la combinatoire
- Certification des logiciels pour l'avionique au niveau A : Situation catastrophiques encoures en cas d'erreurs du logiciel
- Critère MC/DC : *Modified Condition/Decision Coverage*
- Pour réduire la combinatoire : on ne s'intéresse à un jeu de tests faisant varier une condition que s'il influe sur la décision

Couverture *Toutes-les-Conditions – Décisions* Modifié (MC/DC)

Démontrer l'action de chaque condition sur la valeur de vérité de la décision :

```
if ( A && (B || C ))
```

Principe : pour chaque condition, trouvez 2 cas de test qui changent Dec lorsque toutes les autres conditions sont fixées

Exemple pour `A` :

- `A=0, B=1,C=1 - Dec=0`
- `A=1, B=1,C=1 - Dec=1`

Couverture *Toutes-les-Conditions – Décisions* Modifié (MC/DC)

Démontrer l'action de chaque condition sur la valeur de vérité de la décision :

```
if ( A && (B || C ))
```

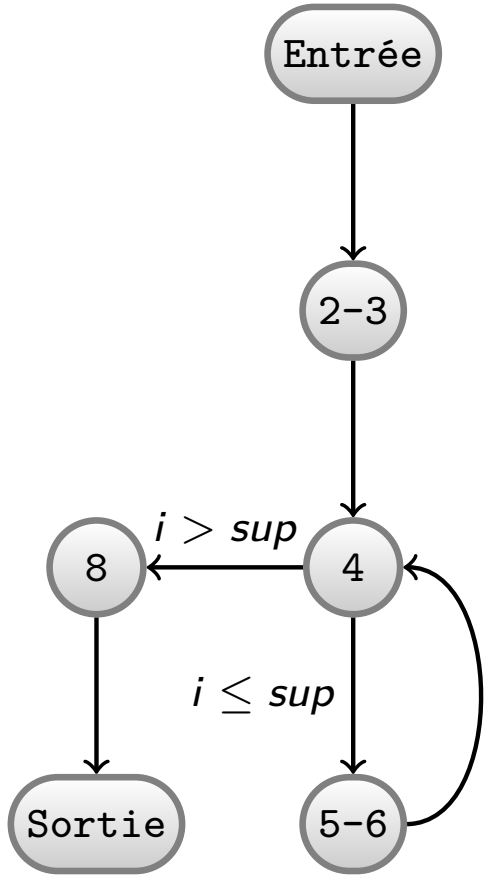
- pour `A` :
  - `A=0, B=1,C=1 - Dec=0`
  - `A=1, B=1,C=1 - Dec=1`
- pour `B` :
  - `A=1, B=1,C=0 - Dec=1`
  - `A=1, B=0,C=0 - Dec=0`
- pour `C` :
  - `A=1, B=0,C=1 - Dec=1`
  - ~~`A=1, B=0,C=0 - Dec=0`~~ $\Leftarrow$ déjà couvert

Limites des Critères *Toutes-les-Conditions – Décisions*

- Programme calculant l'inverse de la somme des entiers d'un tableau :

```
1: Input : inf, sup, a[ ] ;
2: i := inf ;
3: sum := 0 ;

4: while i ≤ sup do
5:     sum := sum + a[i] ;
6:     i := i+1 ;
7: end while
8: return 1/sum ;
```



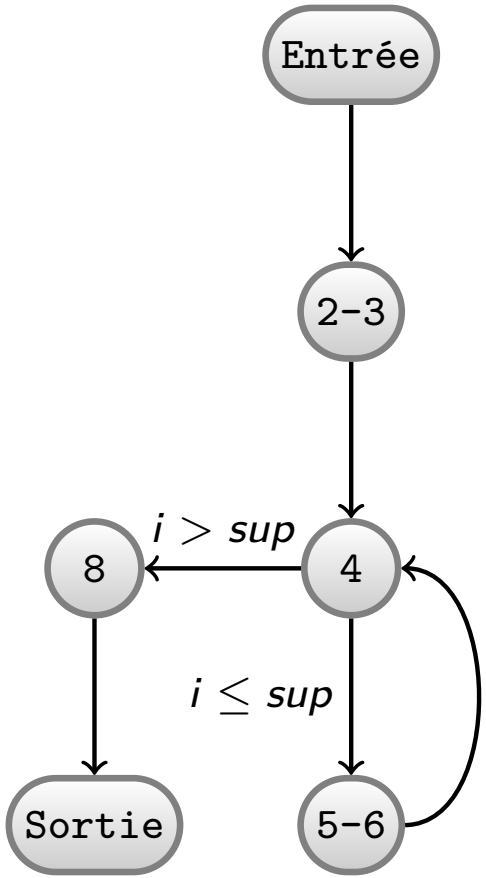
Limites des Critères *Toutes-les-Conditions – Décisions*

1: **Input :** inf, sup, a[ ] ;
2: i := inf ;
3: sum := 0 ;

4: **while** i $\leq$ sup **do**
5: sum := sum + a[i] ;
6: i := i+1 ;
7: **end while**
8: **return** $1/sum$;

- $\{a = [23, 45, 65], inf = 1, sup = 3\}$ couvre le critère *toutes-les-conditions*
- Problème non détecté :

  si $inf > sup$ erreur sur $1/sum$



Couverture de *Tous-Les-Chemins*

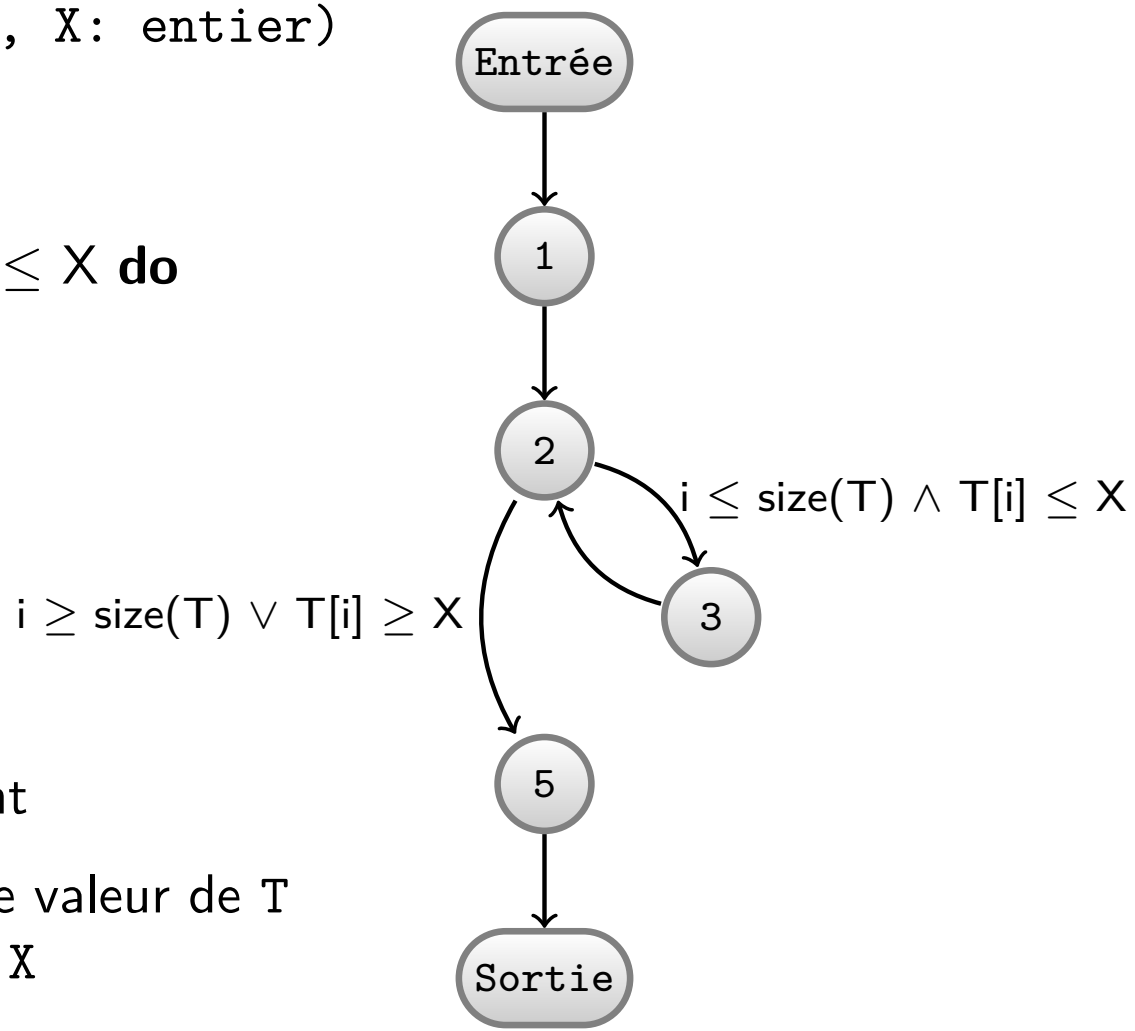
- Critère *tous-les-chemins* : parcourir tous les arcs dans chaque configuration possible (et non pas au moins une fois comme dans le critère tous-les-décisions)
- Si critère *tous-les-chemins* satisfait :
  - Critère *tous-les-décisions* satisfait
  - Critère *tous-les-nœuds* satisfait
- Mais impraticable car la présence d'une boucle produit un nombre infini de chemins
- On se limite aux chemins qui passent de 0 fois à i fois dans la boucle : les i-chemins

Couverture *Tous-Les-i-Chemins*

```
F (T: tableau d'entiers, X: entier)
```

1: i := 1 ;

2: **while** i ≤ size(T) ∧ T[i] ≤ X **do**
3: i := i+1 ;
4: **end while**
5: **return** $T[i]$;



- T trié par ordre croissant
- F retourne la plus petite valeur de T strictement supérieur à X
- Résultat faux si toutes les valeurs sont plus petites ou plus grandes que X

Couverture *Tous-Les-i-Chemins*

- Le critère *tous-les-i-chemins* impose de passer de zéro à i fois dans la boucle
- Permet de détecter les erreurs liées aux problèmes des boucles dans lesquelles ont ne rentre pas
- En pratique, on utilise des valeurs 1 et 2 pour i

Chemin Infaisable

- Chemin : Choix d'un ensemble de décision sur le graphe de contrôle
- Exécution : Chemin dont les conditions des décisions peuvent être satisfaites
  - S'il n'existe pas de valeur d'entrée pour satisfaire les conditions $\Rightarrow$ *chemin infaisable*
  - Pas de test possible pour couvrir un chemin infaisable
- Existence de chemins infaisables : problème indécidable
- Mais il est possible de calculer les conditions sur un chemin donné pour qu'il soit faisable

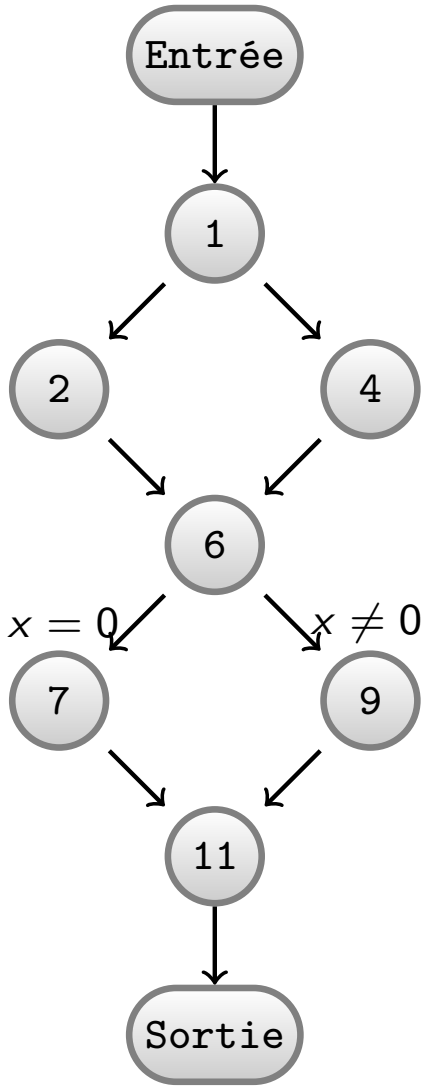
Exemple de Chemin Infaisable

Tous les chemins :

- (Entrée, 1, 2, 6, 7, 11, Sortie)
- (Entrée, 1, 4, 6, 9, 11, Sortie)
- (Entrée, 1, 2, 6, 9, 11, Sortie)
- (Entrée, 1, 4, 6, 7, 11, Sortie)

$x := 0$

Sont-ils tous faisables ?

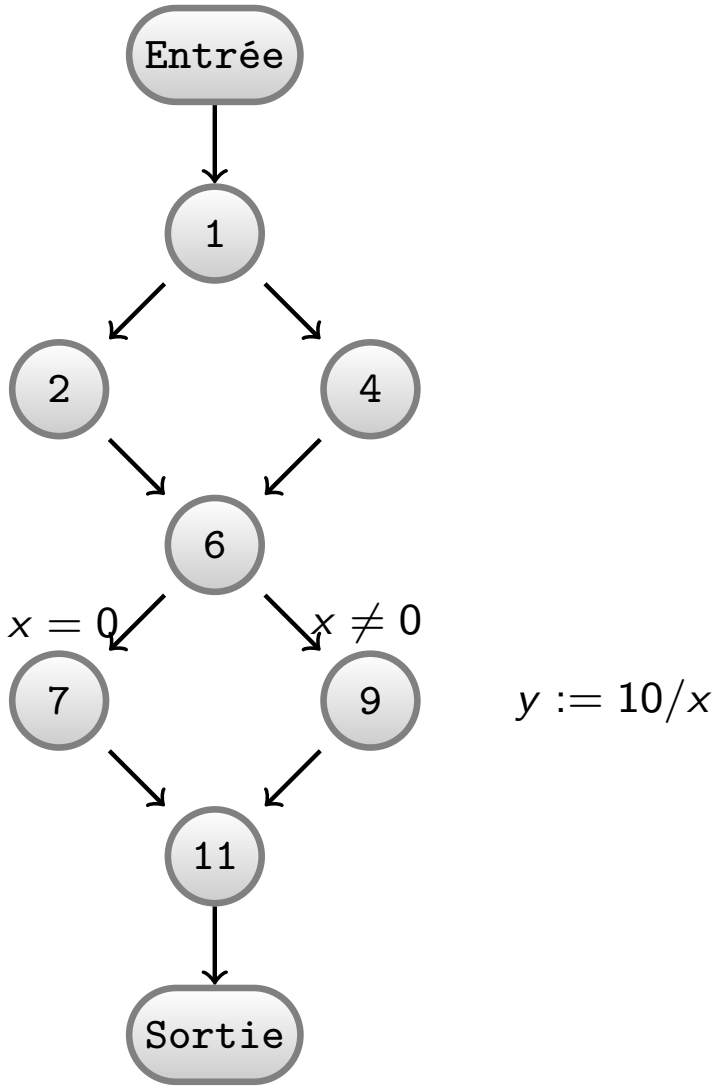


$y := 10/x$

Exemple de Chemin Infaisable

Tous les chemins :

- (Entrée, 1, 2, 6, 7, 11, Sortie)
- (Entrée, 1, 4, 6, 9, 11, Sortie)
- (Entrée, 1, 2, 6, 9, 11, Sortie)
- (Entrée, 1, 4, 6, 7, 11, Sortie)



$x := 0$

Sont-ils tous faisables ?

Non :

- (Entrée, 1, 2, 6, 9, 11, Sortie) n'est pas faisable

Test Structurel Statique

Font partie du *test structurel* ou *analyse statique* toutes les méthodes structurelles qui ne nécessitent pas l'exécution du code

Le code source est analysé en tant qu'entité indépendante et passive

Principal attrait : Complémentaire à l'analyse dynamique, permet la génération de jeux de test

Principal inconvénient : l'exécution réelle du code n'est pas considérée

Exécution Symbolique

L'*évaluation symbolique* ou *exécution symbolique* est une interprétation qui simule l'exécution de programme en l'absence de donnée en entrée

Principe : à chaque variable est associée un couple (garde, valeur symbolique) :

- La *garde* exprime au moyen de ces mêmes symboles une condition nécessaire pour attendre la valeur symbolique
- La *valeur symbolique* est une expression en fonction des paramètres d'entrée du programme, ces paramètres étant représentés par des symboles

Exécution Symbolique

- *Valeur d'une variable* (local ou global) :

  calculée en interprétant une par une les instructions rencontrées sur le chemin d'exécution

- *Garde* :

  associée à la valeur symbolique et composée de la conjonction de toutes les conditions d'exécution du chemin d'exécution

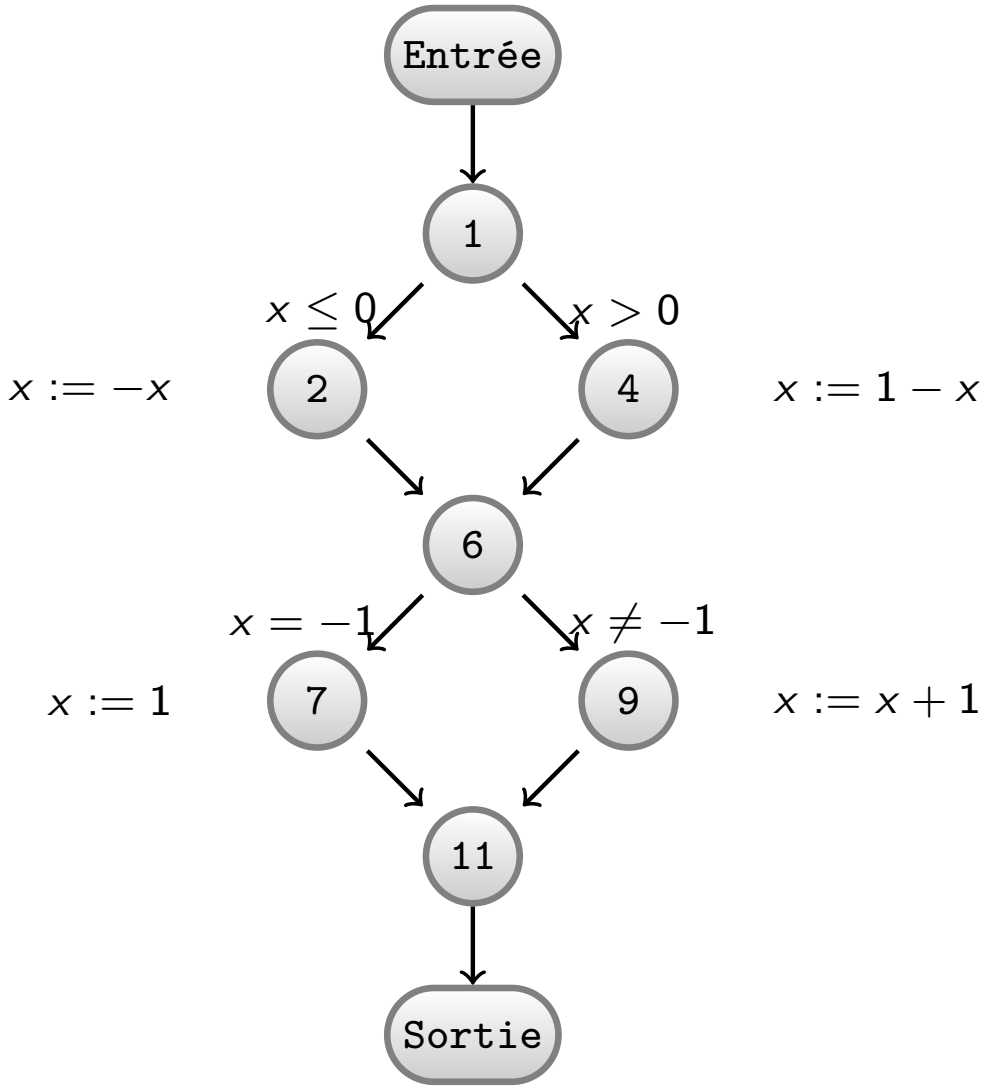
Exécution Symbolique

Soit `P` un chemin de `E` à `S` dans le graphe de contrôle

- On donne des valeurs symboliques aux variables x_0, y_0, $z_0 \ldots$
- On initialise la condition de chemin à `true`
- On suit le chemin `P` nœud par nœud depuis l'`Entrée` :
  - si le nœud est un bloc d'instructions, on exécute les instructions sur les valeurs symboliques
  - si le nœud est un bloc de décision étiqueté par une condition C :
    - si on suit la branche du then, on ajoute C à la condition de chemin
    - si on suit la branche du else, on ajoute la négation de C à la condition de chemin

Exécution Symbolique

```
1:  if x ≤ 0 then
2:      x := -x;
3:  else
4:      x := 1 - x;
5:  end if
6:  if x = −1 then
7:      x := 1;
8:  else
9:      x := x + 1;
10: end if
11: return x;
```

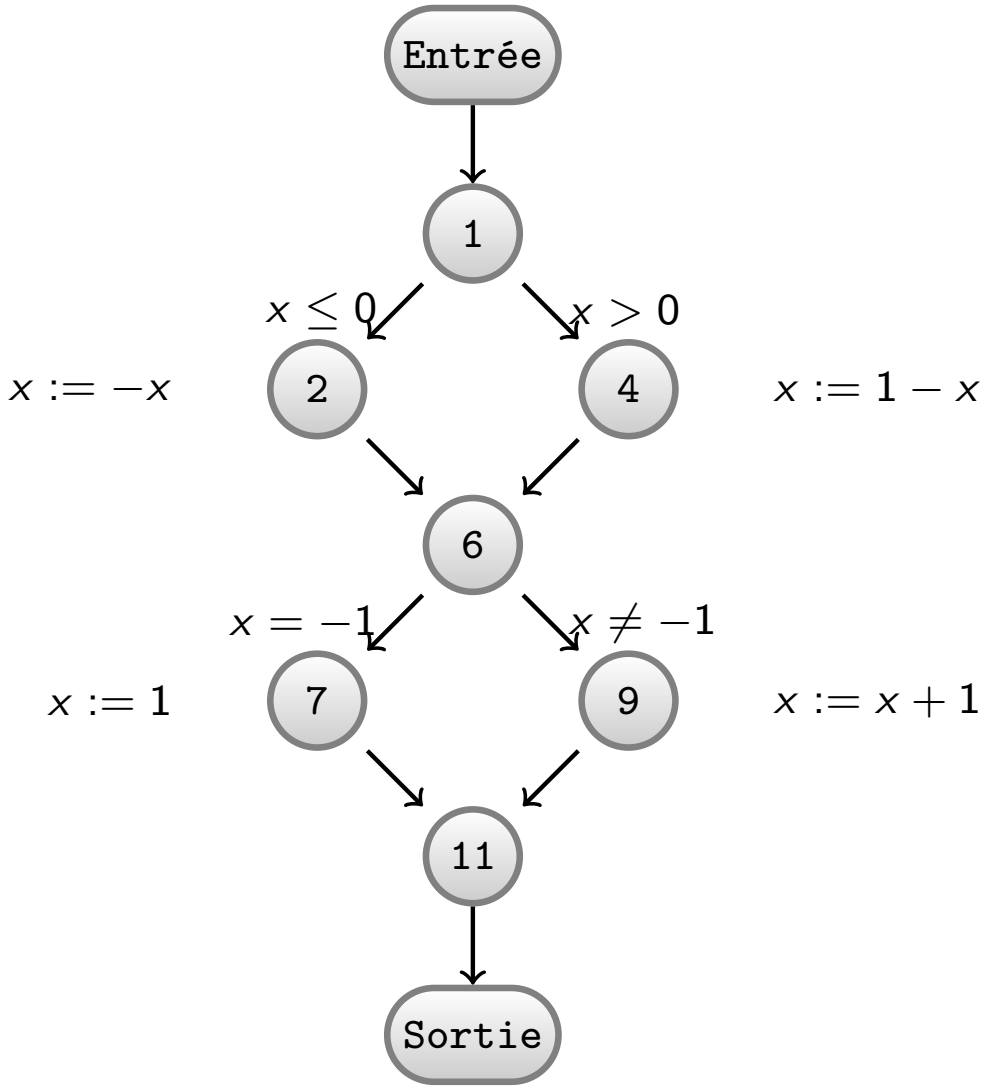


Exécution Symbolique

Valeur symbolique de x :

(garde, valeur symbolique de x)

1 (`true`, x_0)



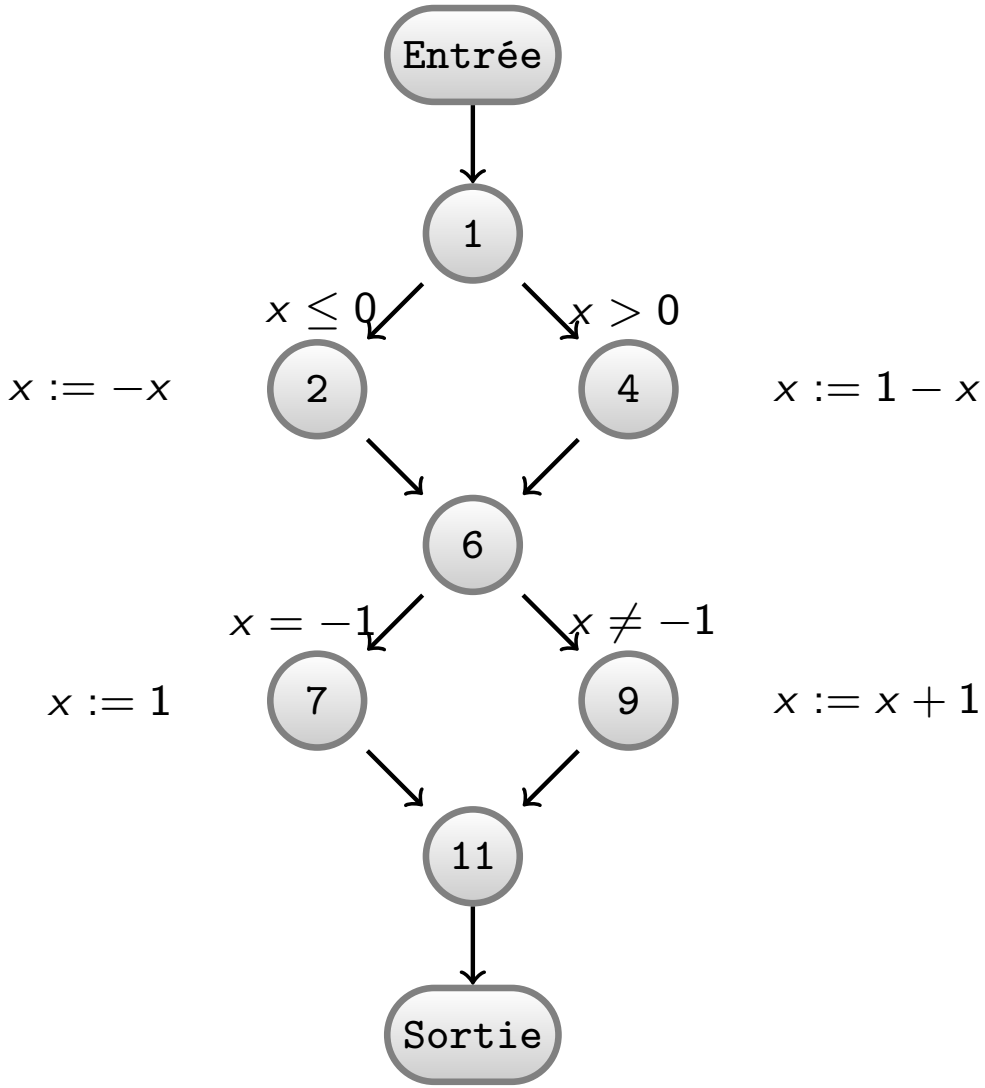
Exécution Symbolique

Valeur symbolique de x :

(garde, valeur symbolique de x)

1 (`true`, x_0)

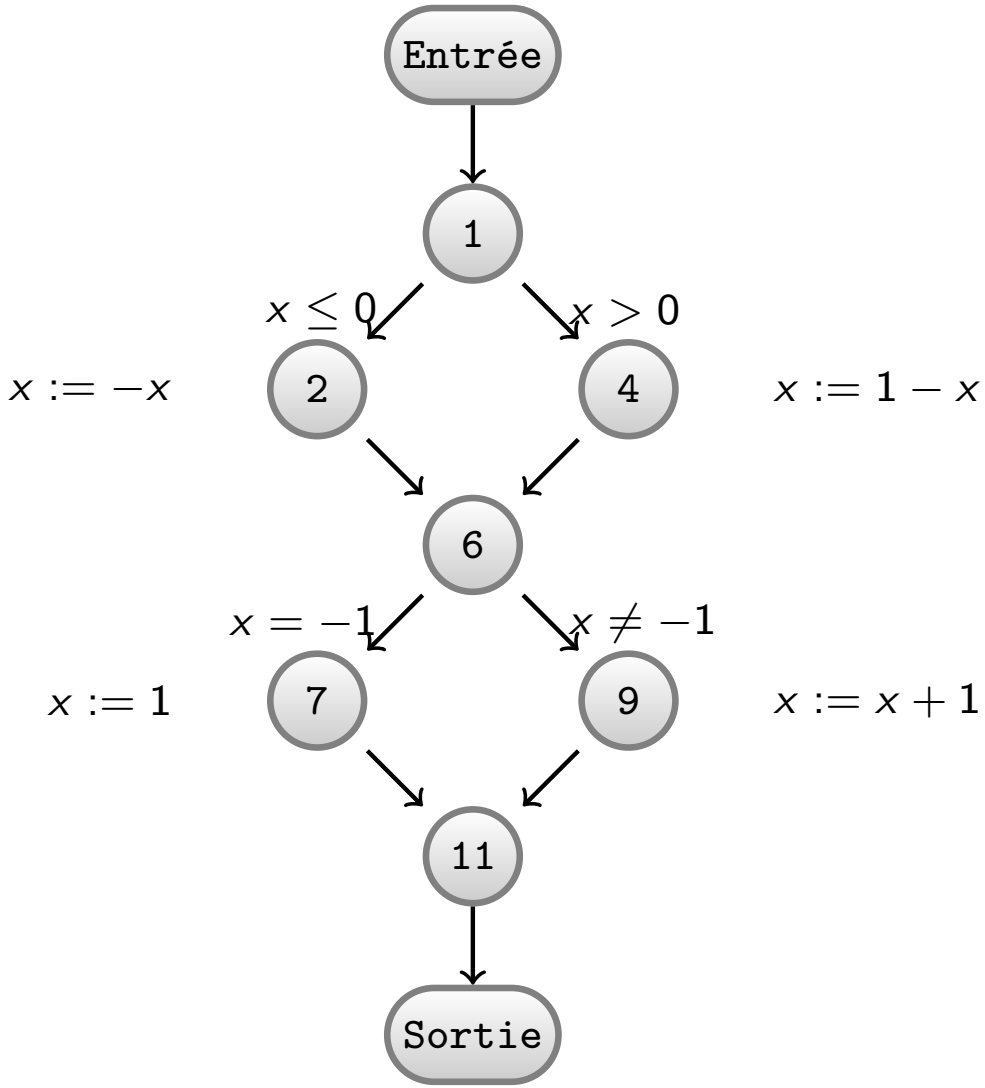
2 ($x_0 \leq 0, -x_0$)



Exécution Symbolique

Valeur symbolique de x :

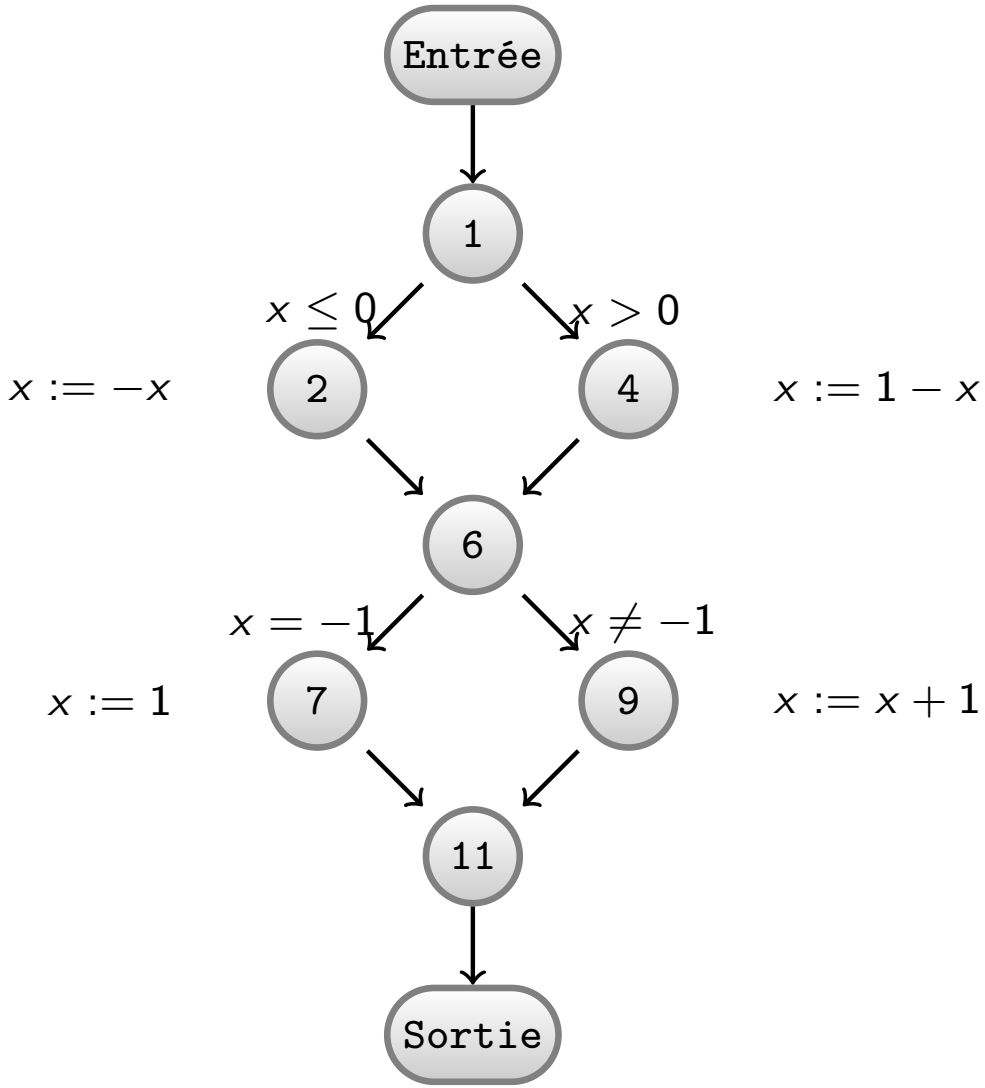
	(garde, valeur symbolique de x)
1	(`true`, x_0)
2	$(x_0 \leq 0, -x_0)$
4	$(x_0 > 0, 1 - x_0)$



Exécution Symbolique

Valeur symbolique de x :

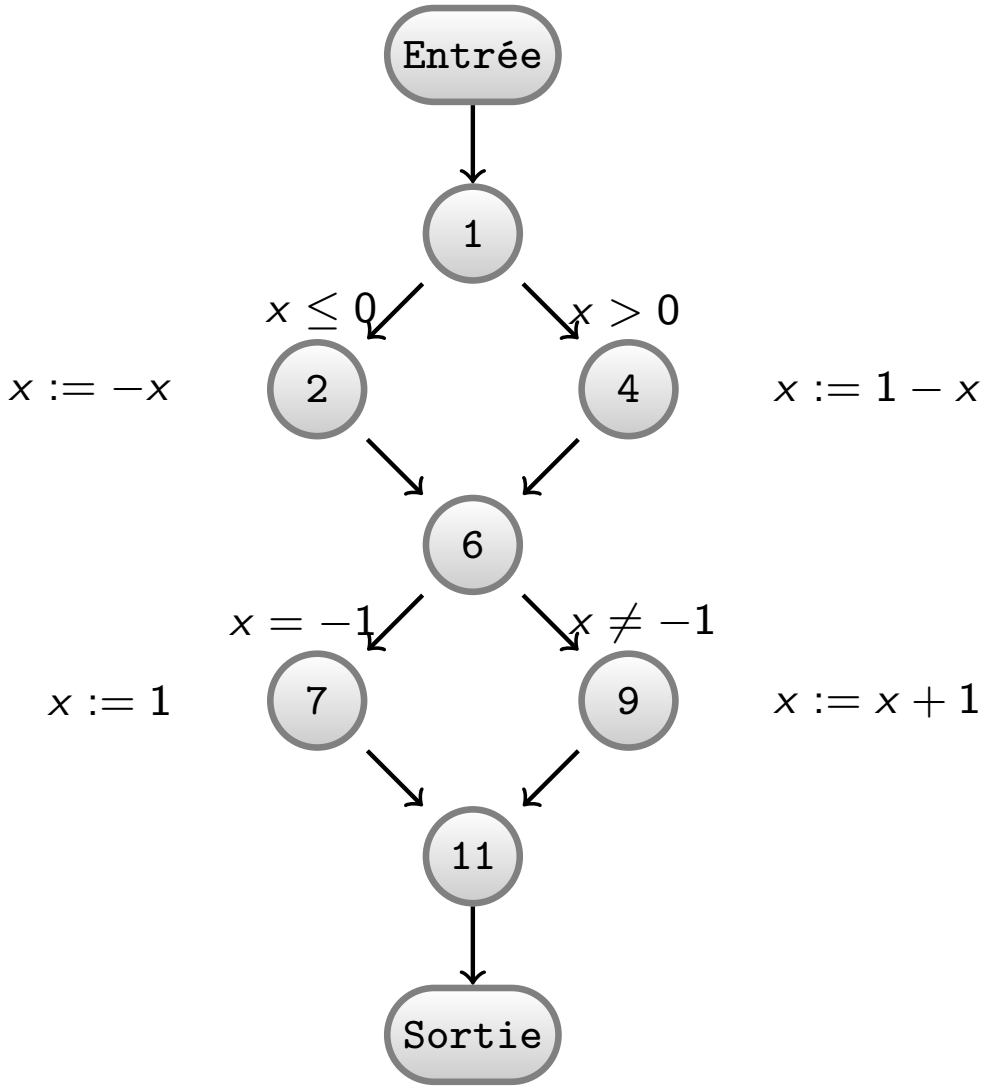
	(garde, valeur symbolique de x)
1	$(\texttt{true},\ x_0)$
2	$(x_0 \leq 0, -x_0)$
4	$(x_0 > 0, 1 - x_0)$
6	$((x_0 \leq 0, -x_0),\ (x_0 > 0, 1 - x_0))$



Exécution Symbolique

Valeur symbolique de x :

	(garde, valeur symbolique de x)
1	(`true`, x_0)
2	$(x_0 \leq 0, -x_0)$
4	$(x_0 > 0, 1 - x_0)$
6	$((x_0 \leq 0, -x_0), (x_0 > 0, 1 - x_0))$
7	$(((x_0 \leq 0) \wedge (-x_0 = -1), 1), ((x_0 > 0) \wedge (1 - x_0 = -1), 1))$



Exécution Symbolique

Valeur symbolique de x :

(garde, valeur symbolique de x)

1 (`true`, x_0)

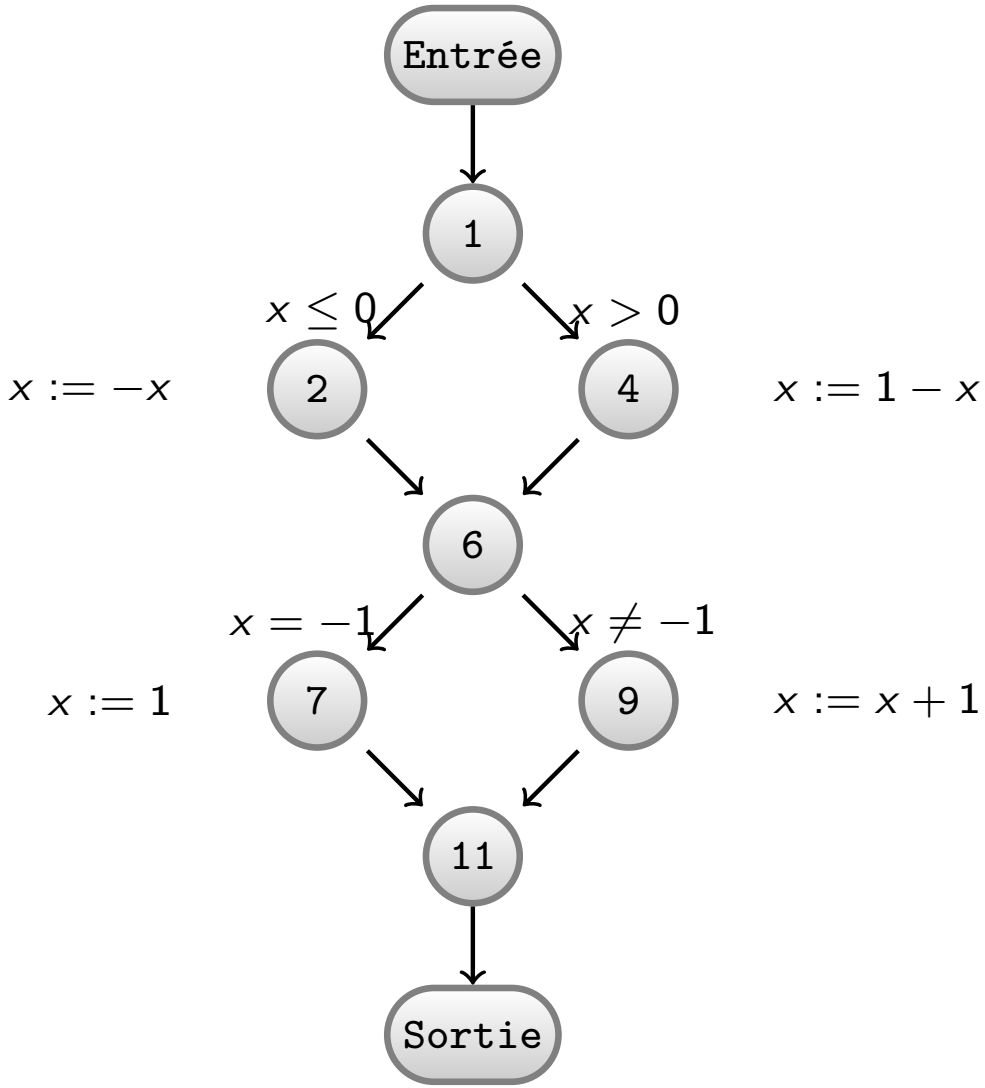
2 $(x_0 \leq 0, -x_0)$

4 $(x_0 > 0, 1 - x_0)$

6 $((x_0 \leq 0, -x_0),\ (x_0 > 0, 1 - x_0))$

7 $(((x_0 \leq 0) \wedge (-x_0 = -1), 1),\ ((x_0 > 0) \wedge (1 - x_0 = -1), 1))$

7 $(((x_0 \leq 0) \wedge (x_0 = 1), 1),\ ((x_0 > 0) \wedge (x_0 = 2), 1))$



Exécution Symbolique

Valeur symbolique de x :

(garde, valeur symbolique de x)

1 (`true`, x_0)

2 $(x_0 \leq 0, -x_0)$

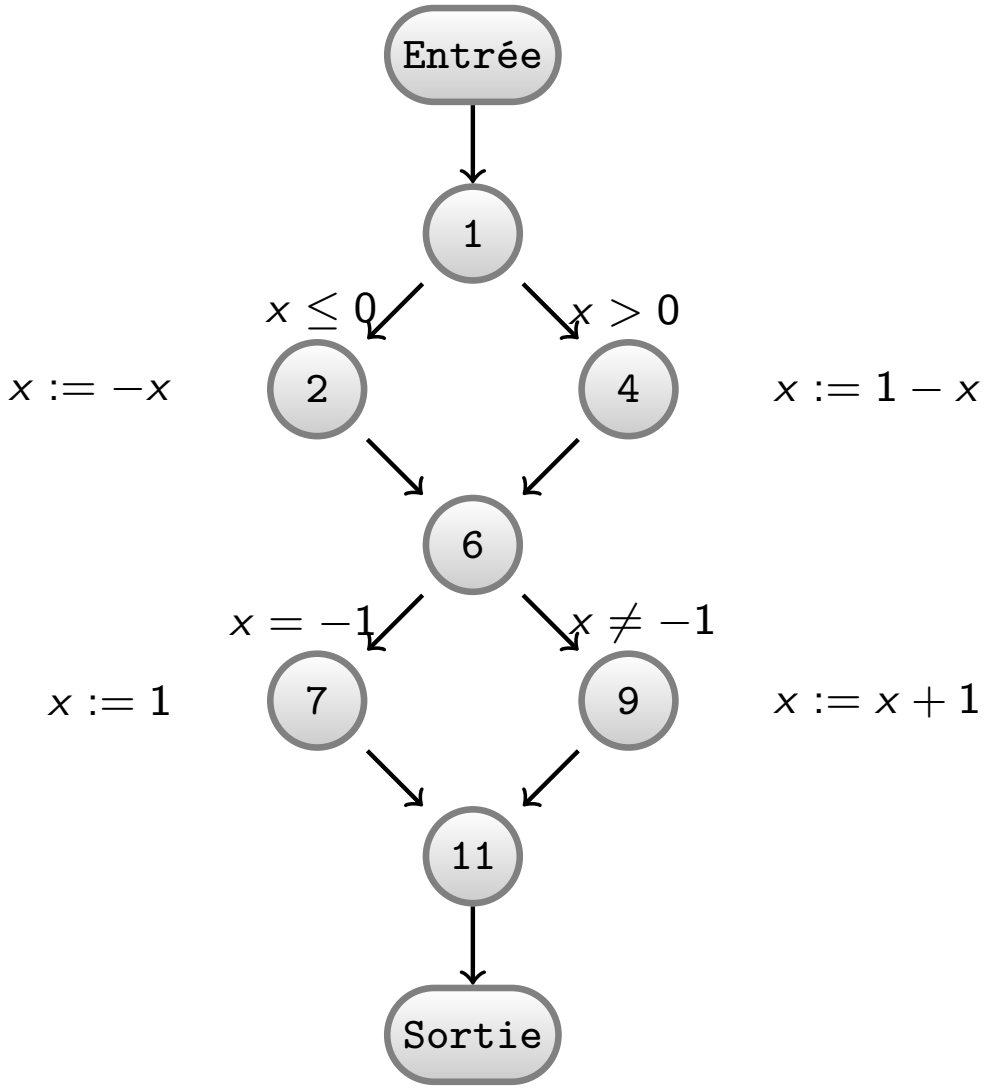
4 $(x_0 > 0, 1 - x_0)$

6 $((x_0 \leq 0, -x_0),\ (x_0 > 0, 1 - x_0))$

7 $(((x_0 \leq 0) \wedge (-x_0 = -1), 1),\ ((x_0 > 0) \wedge (1 - x_0 = -1), 1))$

7 $(((x_0 \leq 0) \wedge (x_0 = 1), 1),\ ((x_0 > 0) \wedge (x_0 = 2), 1))$

7 (`(false, 1)`, $(x_0 = 2, 1)$)



Exécution Symbolique

Valeur symbolique de x :

(garde, valeur symbolique de x)

1 (`true`, x_0)

2 $(x_0 \leq 0, -x_0)$

4 $(x_0 > 0, 1 - x_0)$

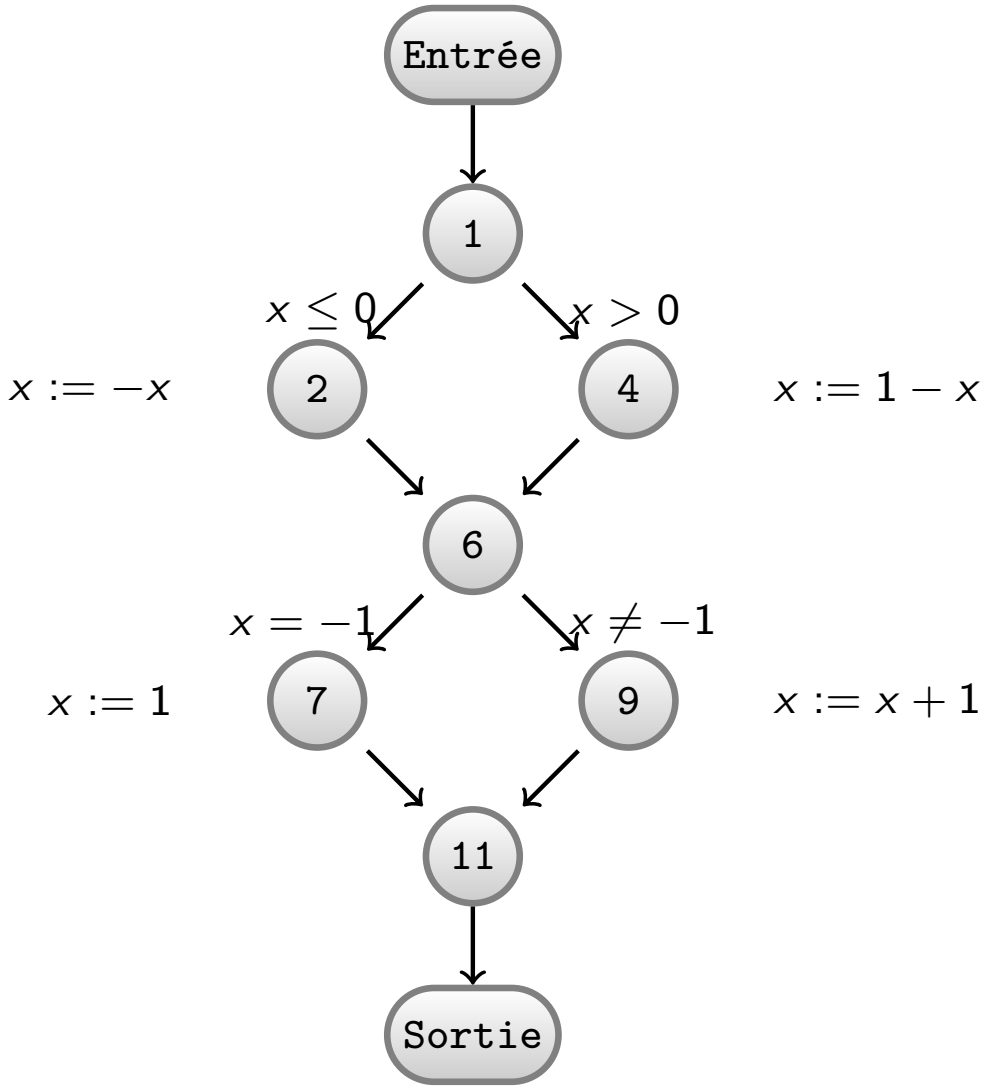
6 $((x_0 \leq 0, -x_0),\ (x_0 > 0, 1 - x_0))$

7 $(((x_0 \leq 0) \wedge (-x_0 = -1), 1),\ ((x_0 > 0) \wedge (1 - x_0 = -1), 1))$

7 $(((x_0 \leq 0) \wedge (x_0 = 1), 1),\ ((x_0 > 0) \wedge (x_0 = 2), 1))$

7 (`false`, 1), $(x_0 = 2, 1))$

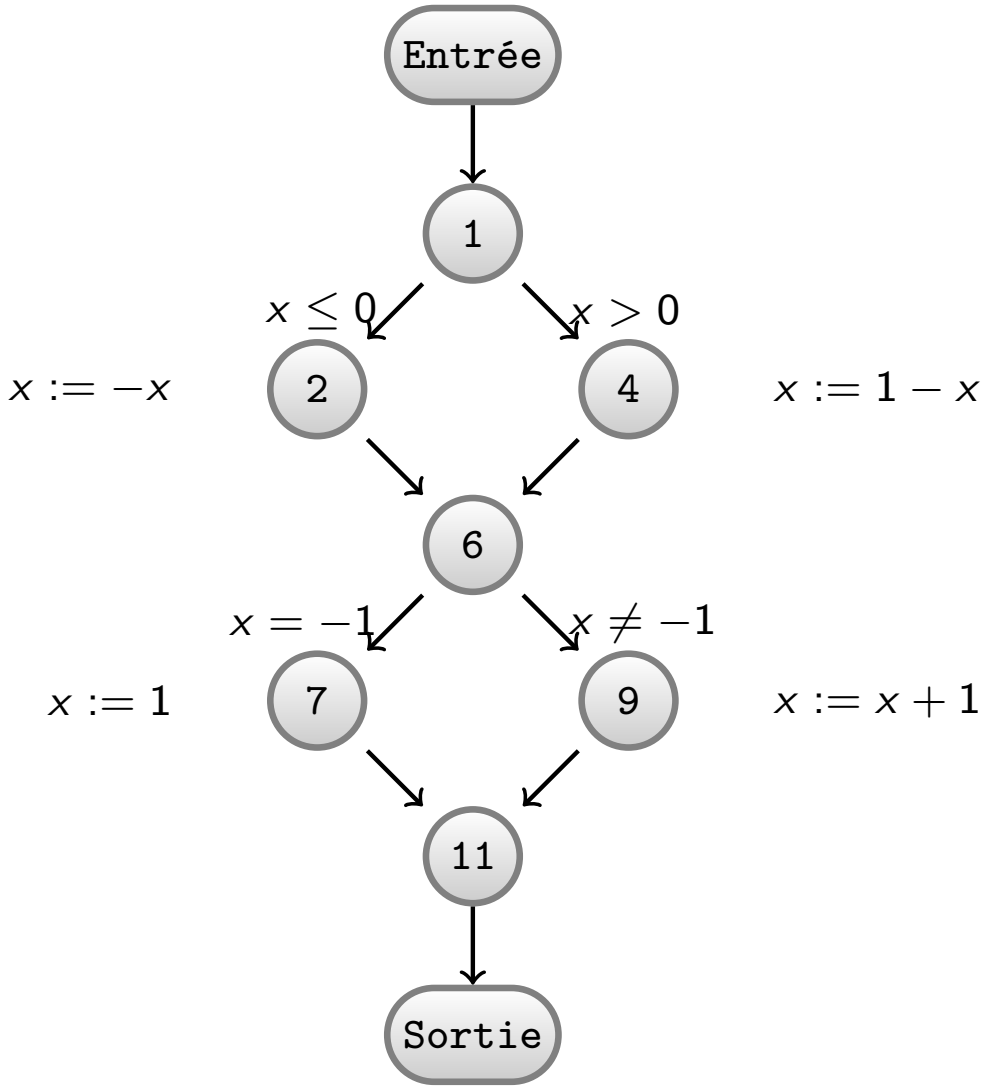
7 $(x_0 = 2, 1)$



Exécution Symbolique

Valeur symbolique de x :

	(garde, valeur symbolique de x)
1	(`true`, x_0)
2	$(x_0 \leq 0, -x_0)$
4	$(x_0 > 0, 1 - x_0)$
6	$((x_0 \leq 0, -x_0), (x_0 > 0, 1 - x_0))$
7	$(x_0 = 2, 1)$



Exécution Symbolique

Valeur symbolique de x :

(garde, valeur symbolique de x)

1 (`true`, x_0)

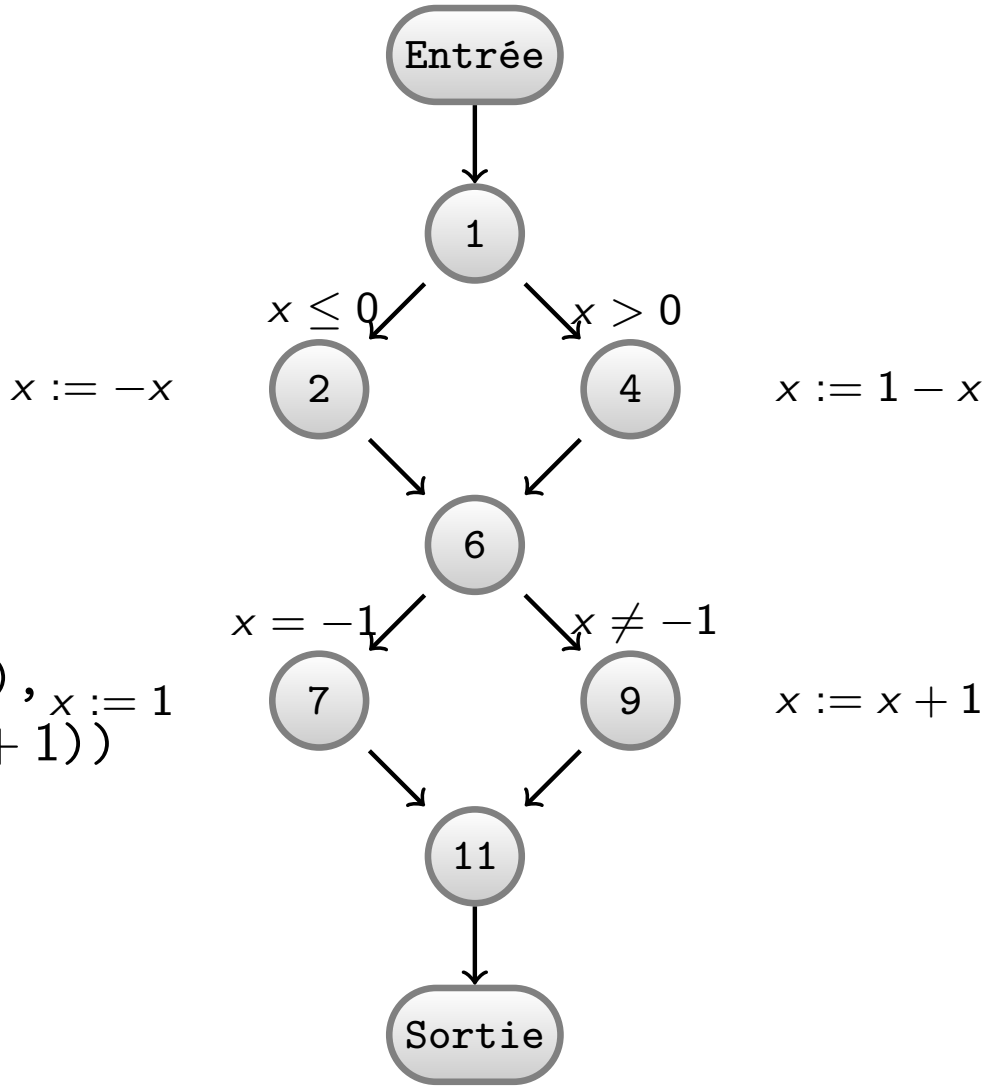
2 $(x_0 \leq 0, -x_0)$

4 $(x_0 > 0, 1 - x_0)$

6 $((x_0 \leq 0, -x_0)$, $(x_0 > 0, 1 - x_0))$

7 $(x_0 = 2, 1)$

9 $(((x_0 \leq 0) \wedge (-x_0 \neq -1), -x_0 + 1)$, $x := 1$
$((x_0 > 0) \wedge (1 - x_0 \neq -1), 1 - x_0 + 1))$



Exécution Symbolique

Valeur symbolique de x :

(garde, valeur symbolique de x)

1 (`true`, x_0)

2 $(x_0 \leq 0, -x_0)$

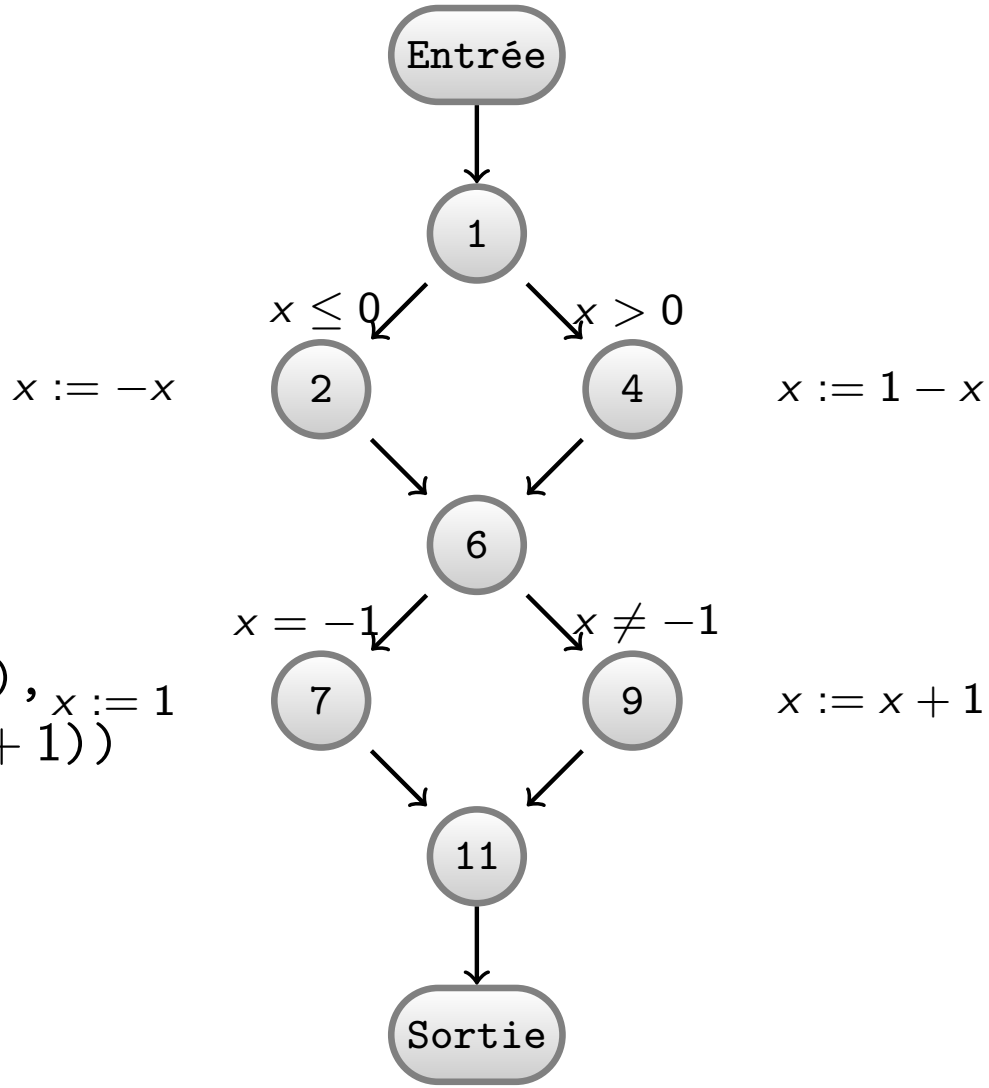
4 $(x_0 > 0, 1 - x_0)$

6 $((x_0 \leq 0, -x_0)$, $(x_0 > 0, 1 - x_0))$

7 $(x_0 = 2, 1)$

9 $(((x_0 \leq 0) \wedge (-x_0 \neq -1), -x_0 + 1)$, $x := 1$
$((x_0 > 0) \wedge (1 - x_0 \neq -1), 1 - x_0 + 1))$

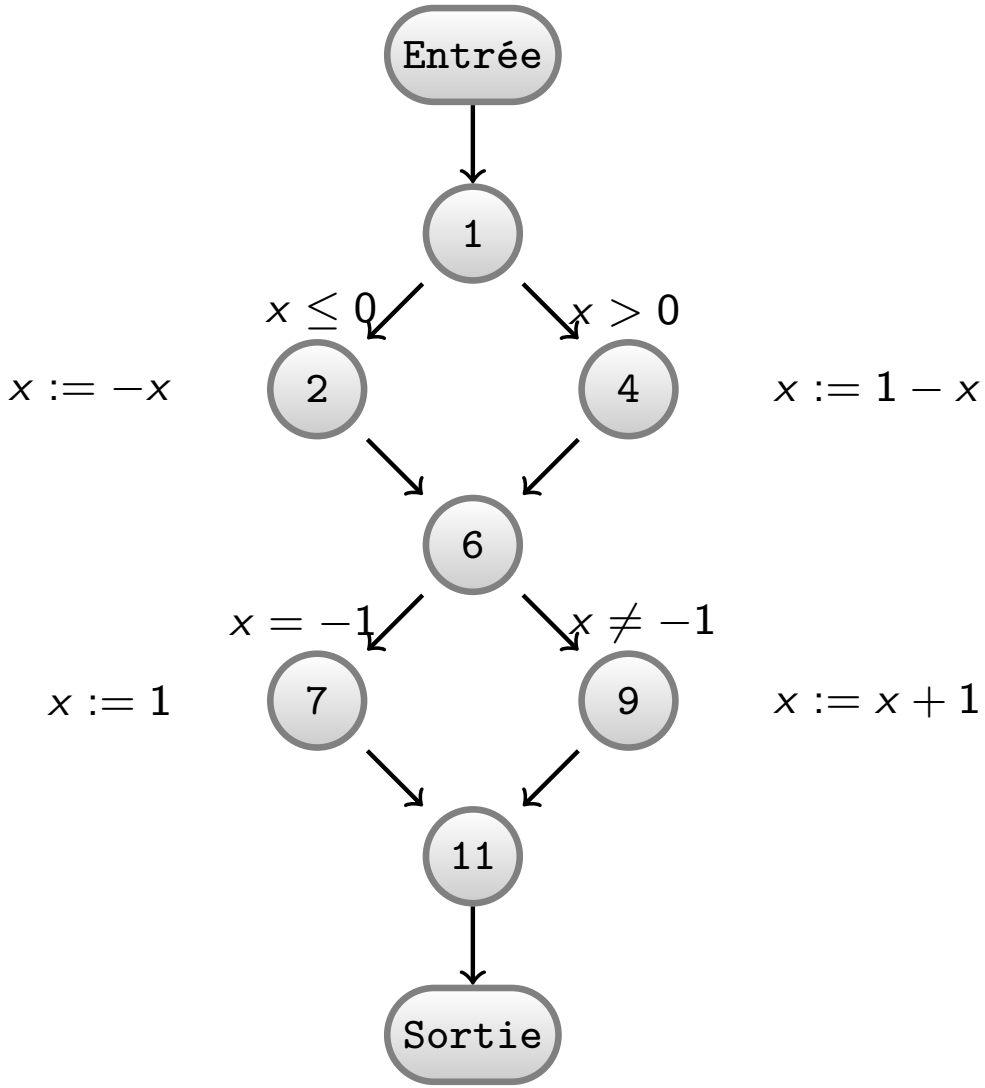
9 $(((x_0 \leq 0), -x_0 + 1)$,
$((x_0 > 0) \wedge (x_0 \neq 2), 2 - x_0))$



Exécution Symbolique

Valeur symbolique de x :

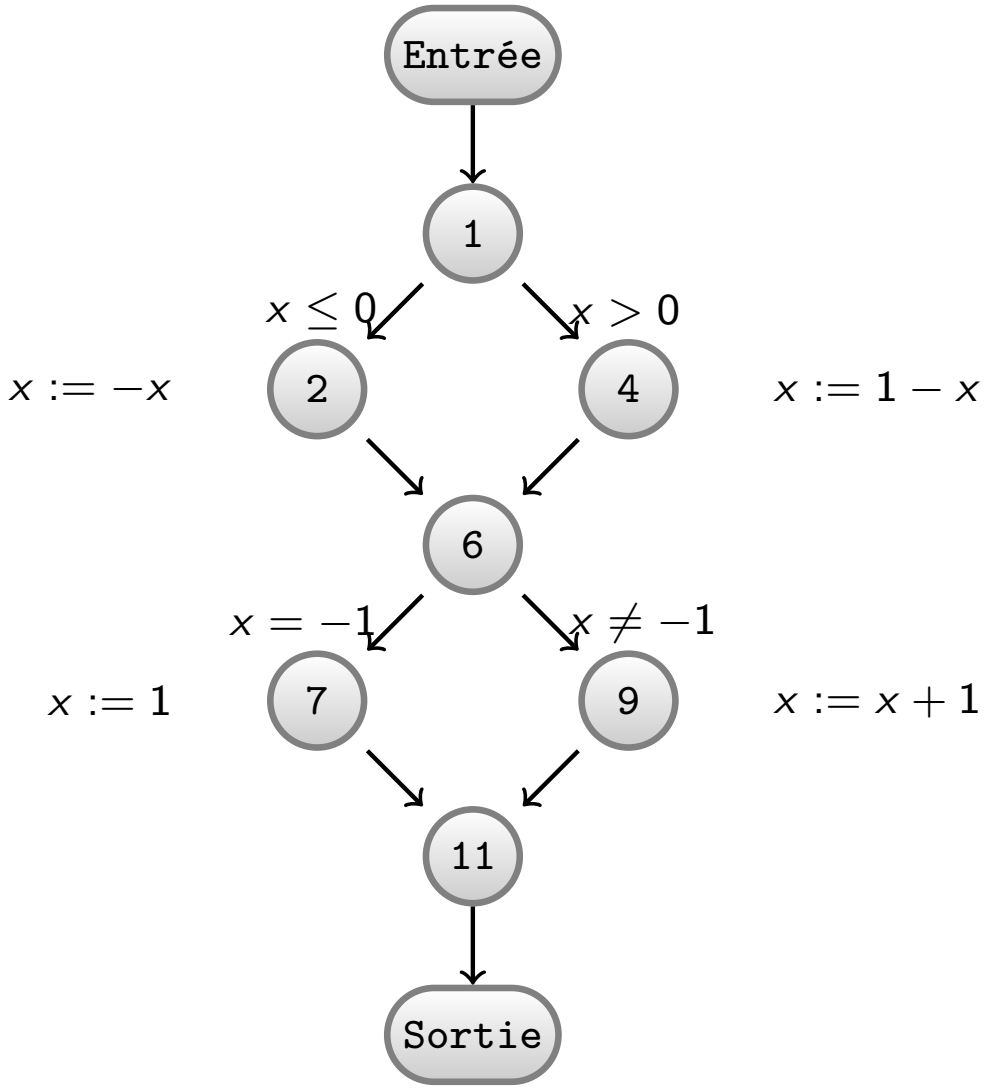
	(garde, valeur symbolique de x)
1	$(\texttt{true},\ x_0)$
2	$(x_0 \leq 0, -x_0)$
4	$(x_0 > 0, 1 - x_0)$
6	$((x_0 \leq 0, -x_0),\ (x_0 > 0, 1 - x_0))$
7	$(x_0 = 2, 1)$
9	$(((x_0 \leq 0), -x_0 + 1),\ ((x_0 > 0) \wedge (x_0 \neq 2), 2 - x_0))$



Exécution Symbolique

Valeur symbolique de x :

	(garde, valeur symbolique de x)
1	$(\texttt{true},\ x_0)$
2	$(x_0 \leq 0, -x_0)$
4	$(x_0 > 0, 1 - x_0)$
6	$((x_0 \leq 0, -x_0),\ (x_0 > 0, 1 - x_0))$
7	$(x_0 = 2, 1)$
9	$(((x_0 \leq 0), -x_0 + 1),$ $((x_0 > 0) \wedge (x_0 \neq 2), 2 - x_0))$
11	$((x_0 = 2, 1),$ $((x_0 \leq 0), -x_0 + 1),$ $((x_0 > 0) \wedge (x_0 \neq 2), 2 - x_0))$



Exécution Symbolique

Problèmes lors de la mise en œuvre de l'exécution symbolique :

- Expressions symboliques croissent très vite si non simplifiables
- Traitement des pointeurs et des indices de tableaux complexe
- Perte des informations sur les variables modifiées dans les boucles à nombre variable d'itérations

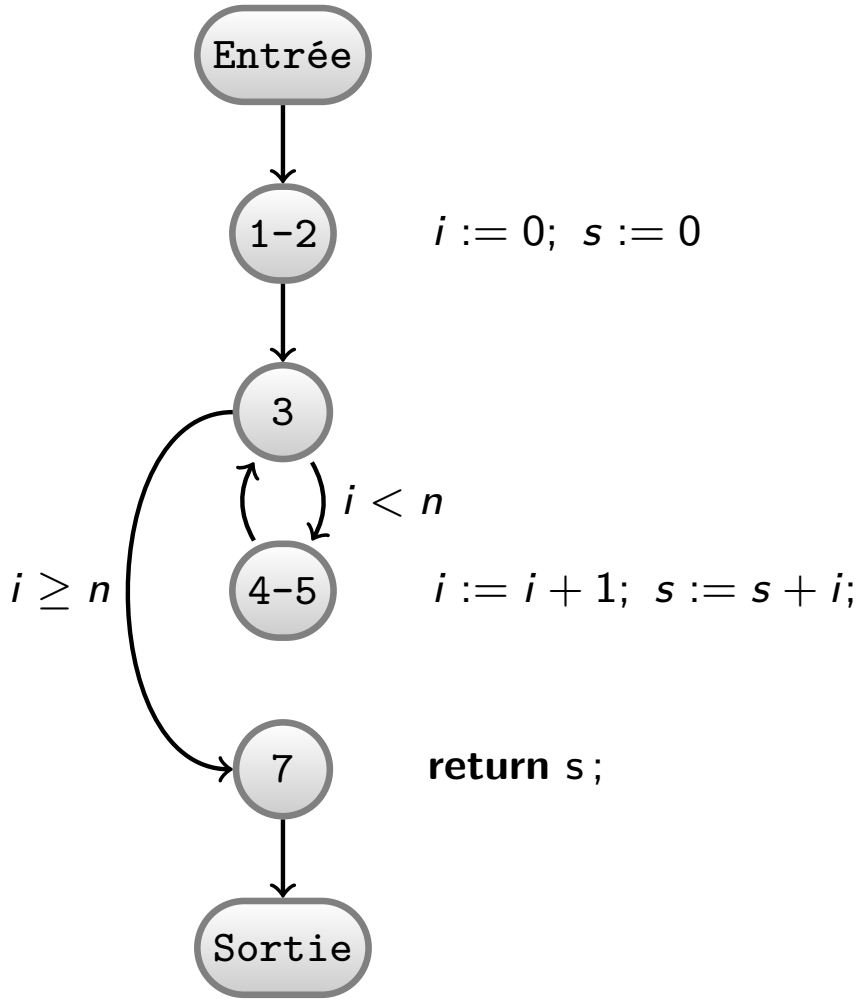
Traitements spécifiques des boucles :

- Exécution symbolique des boucles un nombre arbitraire de fois
- Calcul automatique d'invariant de boucle
- Mode interactif pour permettre l'entrée de renseignements supplémentaires

Exécution Symbolique : Boucle `While`

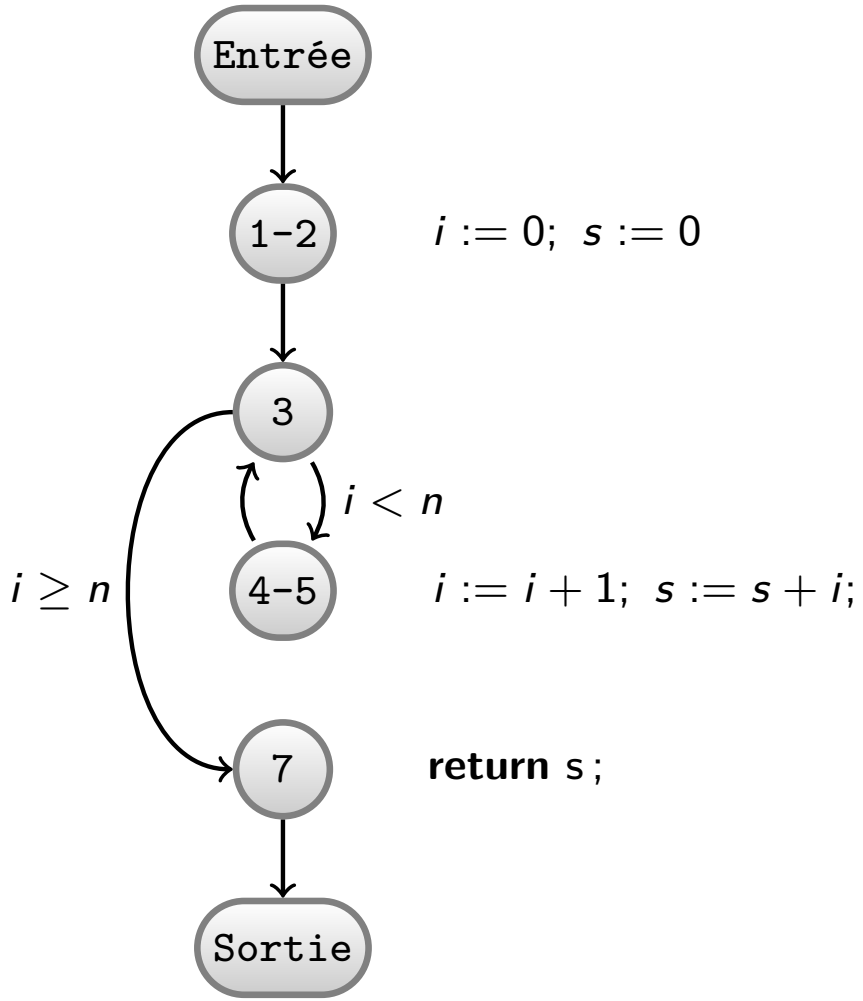
Exemple d'une boucle `WHILE` :

```
1: i := 0;
2: s := 0;
3: while i < n do
4:     i := i + 1;
5:     s := s + i;
6: end while
7: return x;
```



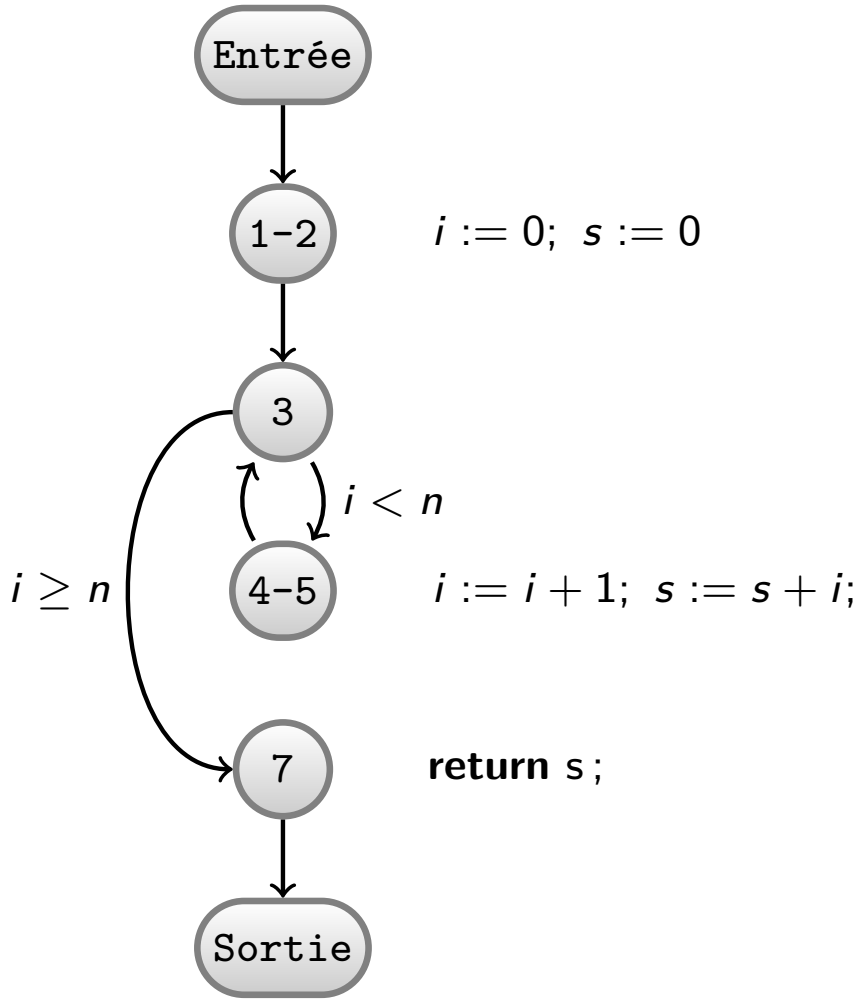
Exécution Symbolique : Boucle `While`

	(garde, n, i, s)
$1-2$	(`true`, $(n_0, 0, 0)$)



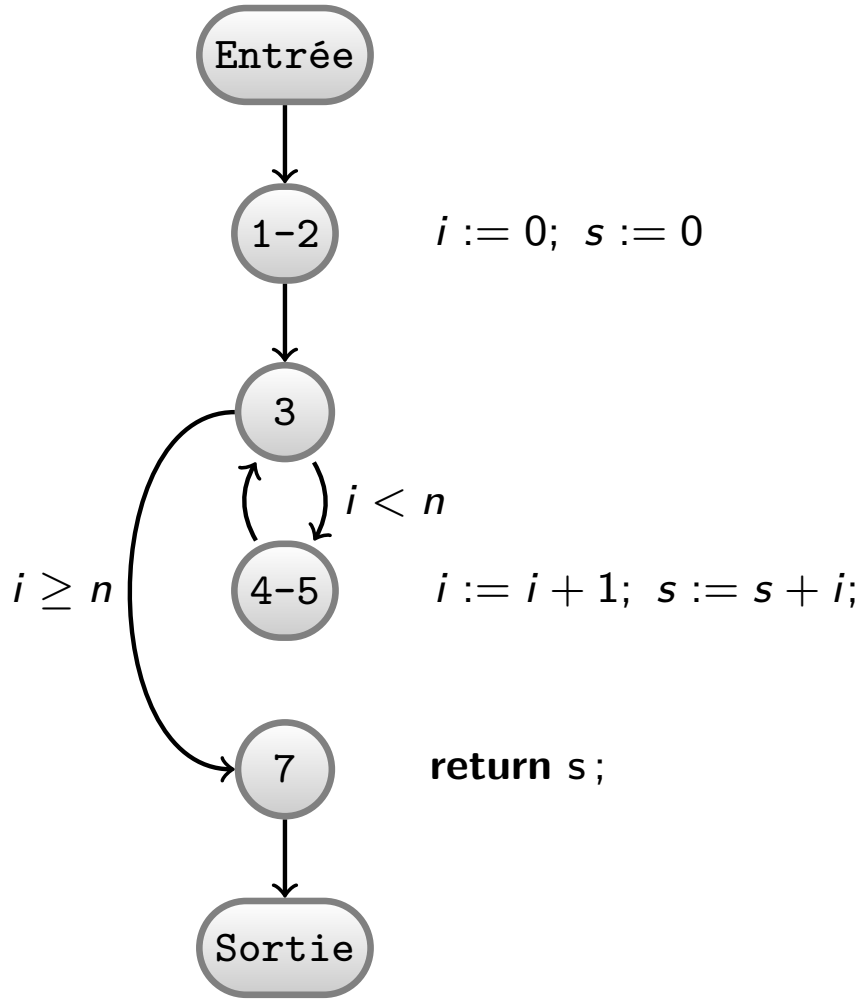
Exécution Symbolique : Boucle `While`

	(garde, n, i, s)	
1 – 2	(`true`, $(n_0, 0, 0)$)	
3	(`true`, $(n_0, 0, 0)$)	1^{er} passage



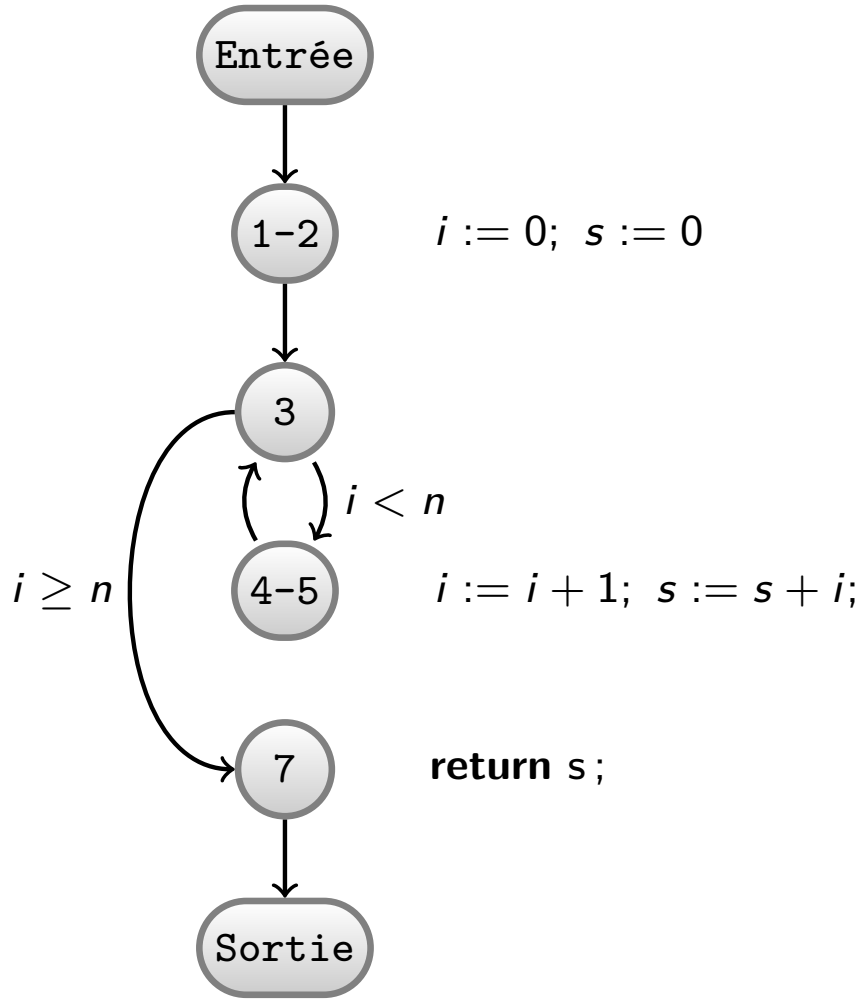
Exécution Symbolique : Boucle `While`

	(garde, n, i, s)
$1-2$	$(\texttt{true},\ (n_0, 0, 0))$
3	$(\texttt{true},\ (n_0, 0, 0))$ 1^{er} passage
$4-5$	$(0 < n_0,\ (n_0, 1, 1))$ 1^{er} passage



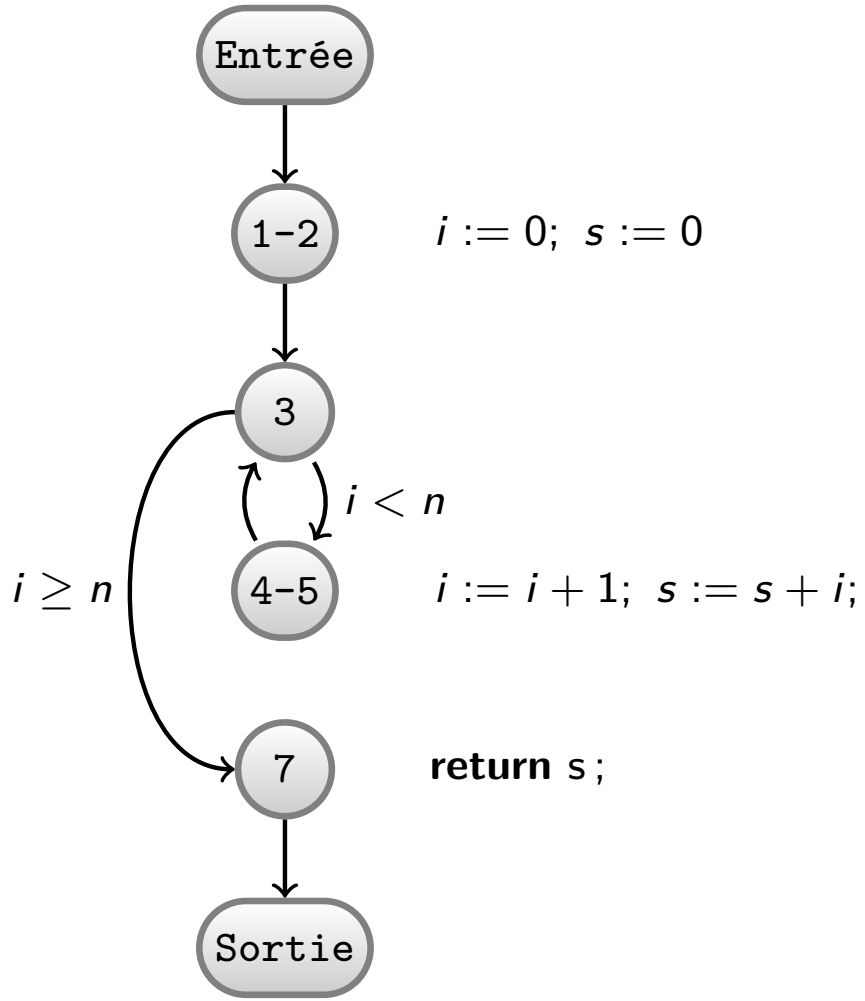
Exécution Symbolique : Boucle `While`

	(garde, n, i, s)
$1-2$	$(\texttt{true},\ (n_0, 0, 0))$
3	$(\texttt{true},\ (n_0, 0, 0))$ 1^{er} passage
$4-5$	$(0 < n_0,\ (n_0, 1, 1))$ 1^{er} passage
3	$(0 < n_0,\ (n_0, 1, 1))$ 2^{e} passage



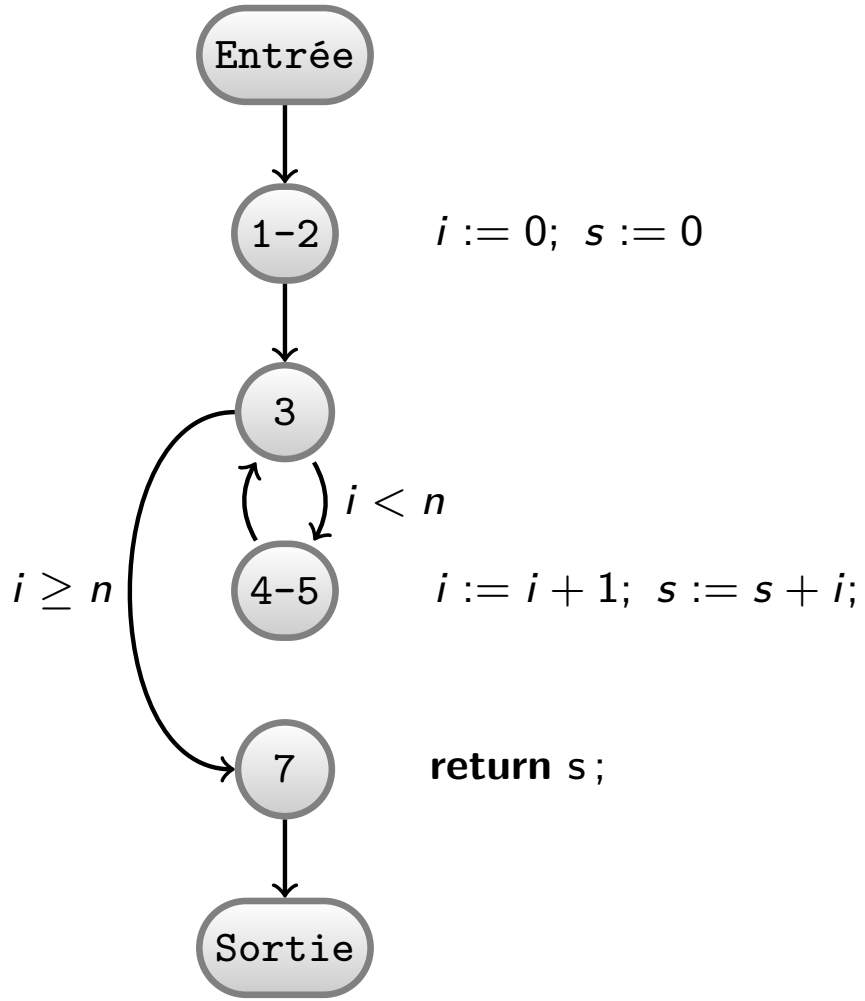
Exécution Symbolique : Boucle `While`

	(garde, n, i, s)	
$1-2$	$(\texttt{true},\ (n_0, 0, 0))$	
3	$(\texttt{true},\ (n_0, 0, 0))$	1^{er} passage
$4-5$	$(0 < n_0,\ (n_0, 1, 1))$	1^{er} passage
3	$(0 < n_0,\ (n_0, 1, 1))$	2^{e} passage
$4-5$	$(1 < n_0,\ (n_0, 2, 3))$	2^{e} passage



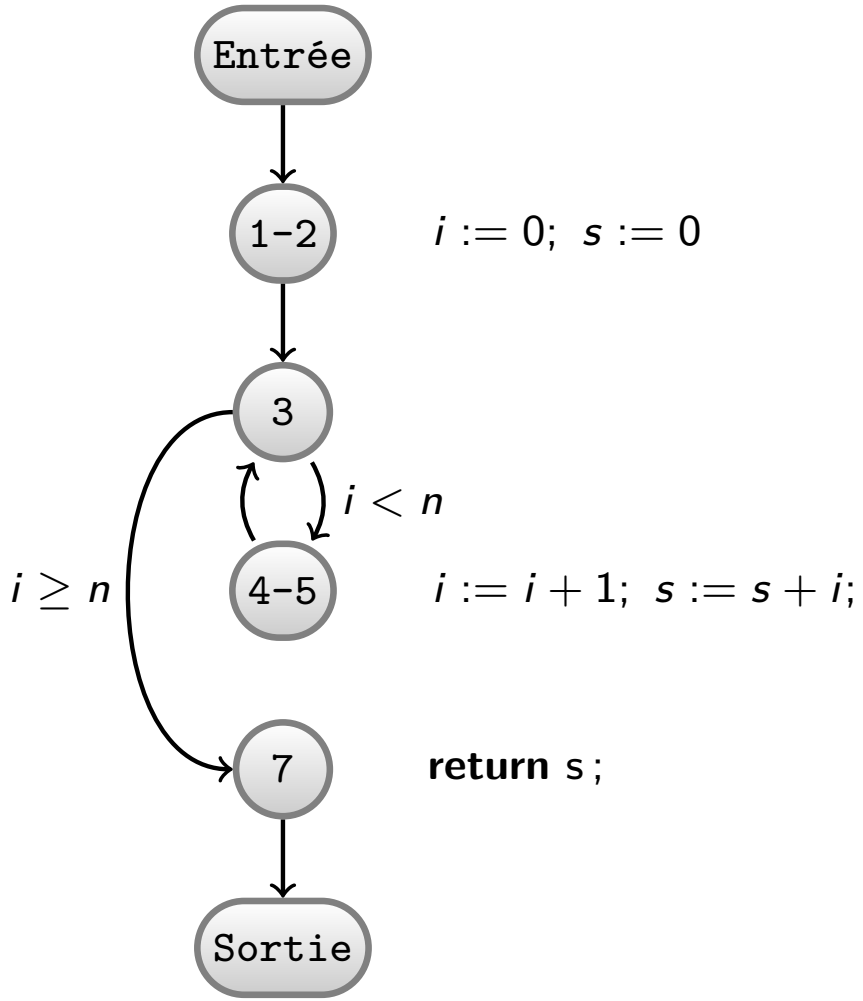
Exécution Symbolique : Boucle `While`

	(garde, n, i, s)	
$1-2$	$(\texttt{true},\ (n_0, 0, 0))$	
3	$(\texttt{true},\ (n_0, 0, 0))$	1^{er} passage
$4-5$	$(0 < n_0,\ (n_0, 1, 1))$	1^{er} passage
3	$(0 < n_0,\ (n_0, 1, 1))$	2^{e} passage
$4-5$	$(1 < n_0,\ (n_0, 2, 3))$	2^{e} passage
	$\vdots$	



Exécution Symbolique : Boucle `While`

Valeur symbolique en 7 ?

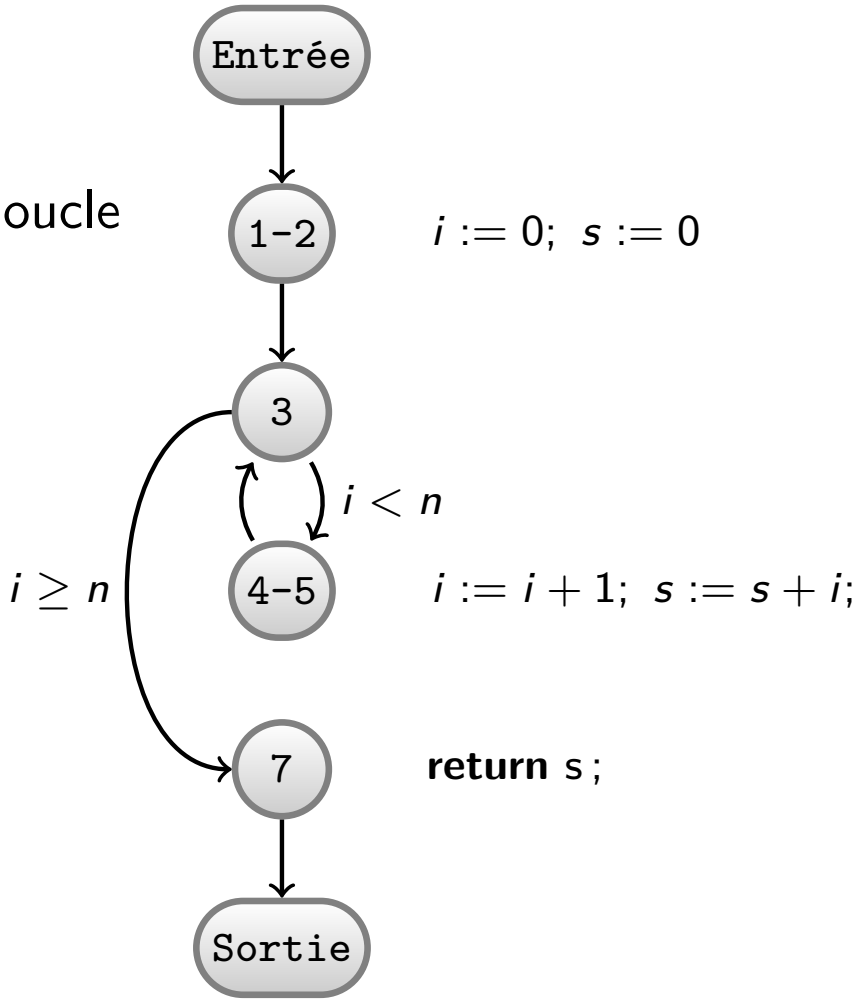


Exécution Symbolique : Boucle `While`

Valeur symbolique en 7 ?

(garde, n, i, s)

$(0 \geq n_0\,,\ (n_0, 0, 0))$ pas d'exécution de la boucle



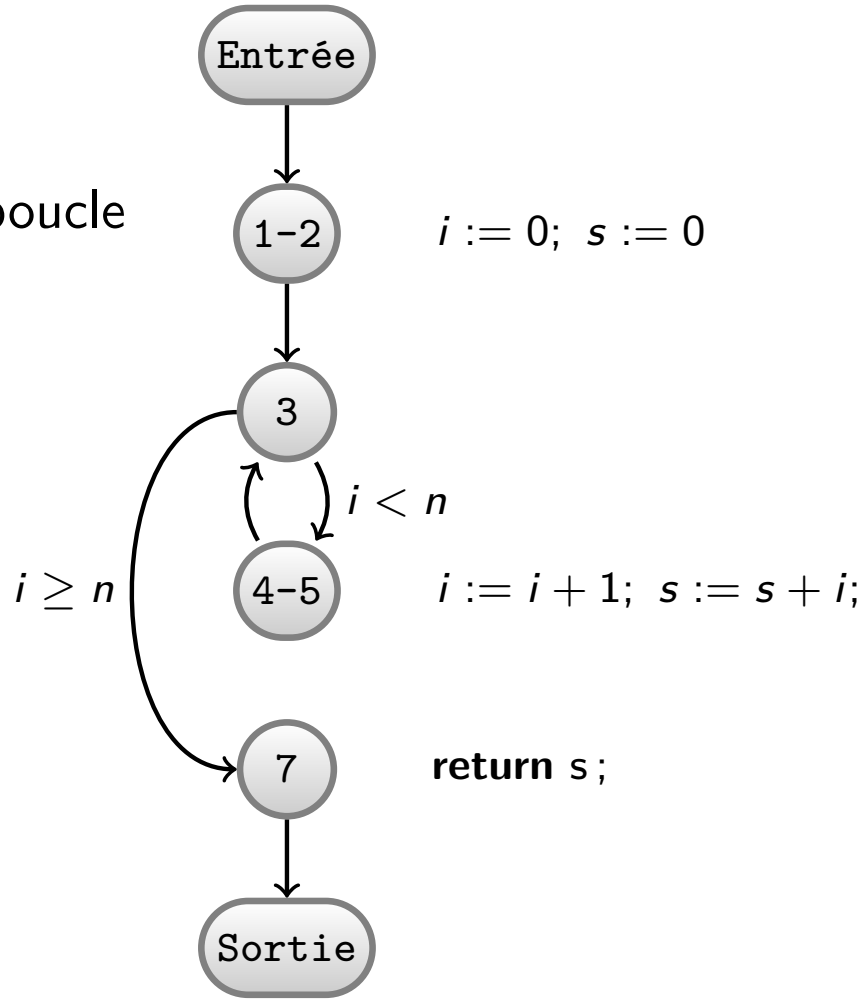
Exécution Symbolique : Boucle `While`

Valeur symbolique en 7 ?

(garde, n, i, s)

$(0 \geq n_0\,,\ (n_0, 0, 0))$ pas d'exécution de la boucle

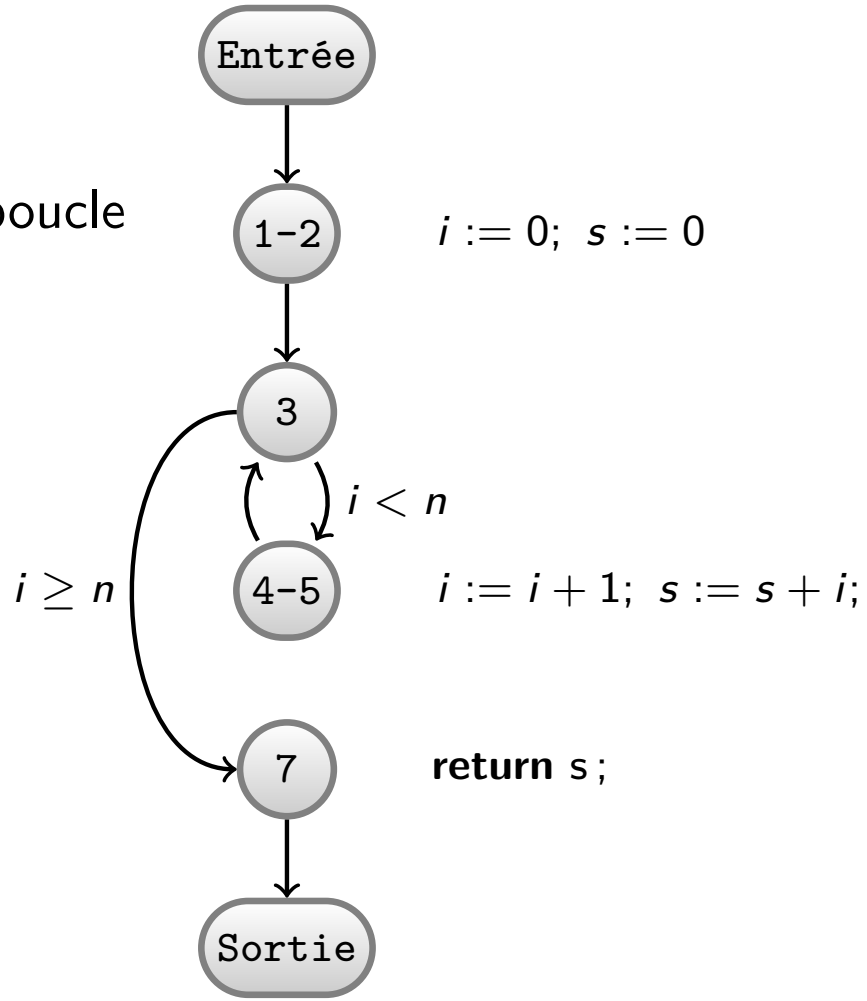
$(1 \geq n_0\,,\ (n_0, 1, 1))$ 1 exécution



Exécution Symbolique : Boucle `While`

Valeur symbolique en 7 ?

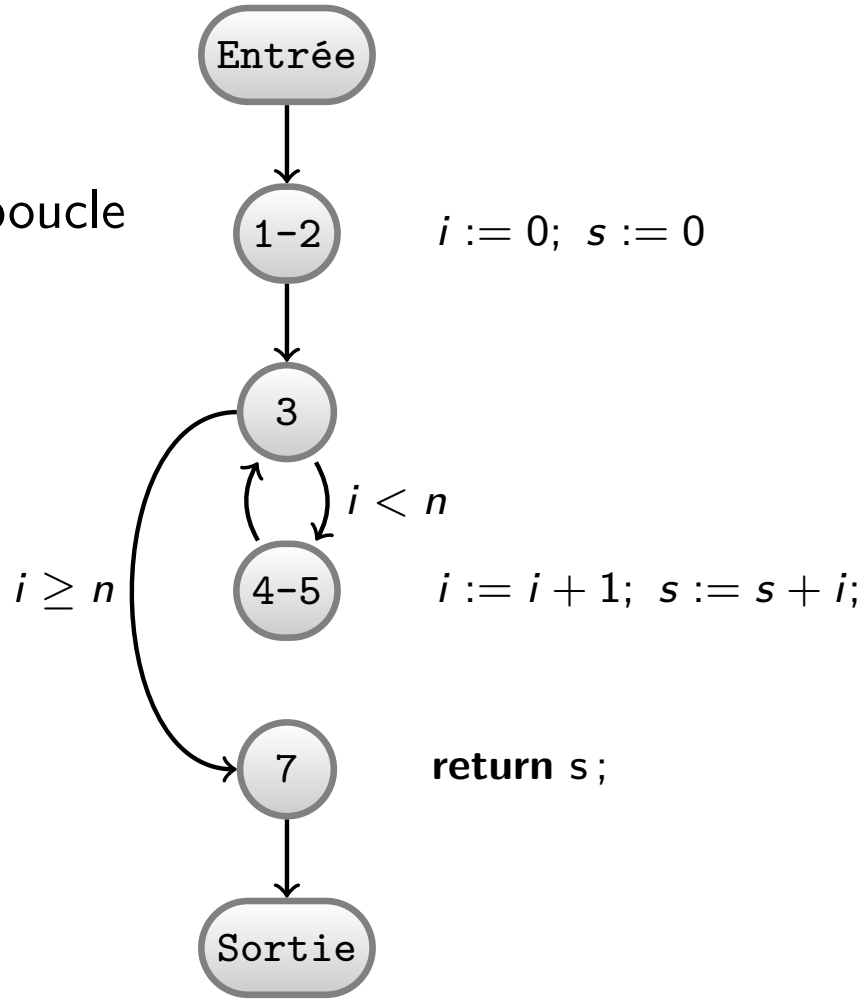
(garde, n, i, s)		
$(0 \geq n_0\,,$	$(n_0, 0, 0))$	pas d'exécution de la boucle
$(1 \geq n_0\,,$	$(n_0, 1, 1))$	1 exécution
$(2 \geq n_0\,,$	$(n_0, 2, 3))$	2 exécutions



Exécution Symbolique : Boucle `While`

Valeur symbolique en 7 ?

(garde, n, i, s)		
$(0 \geq n_0\,,$	$(n_0, 0, 0))$	pas d'exécution de la boucle
$(1 \geq n_0\,,$	$(n_0, 1, 1))$	1 exécution
$(2 \geq n_0\,,$	$(n_0, 2, 3))$	2 exécutions
	$\vdots$	



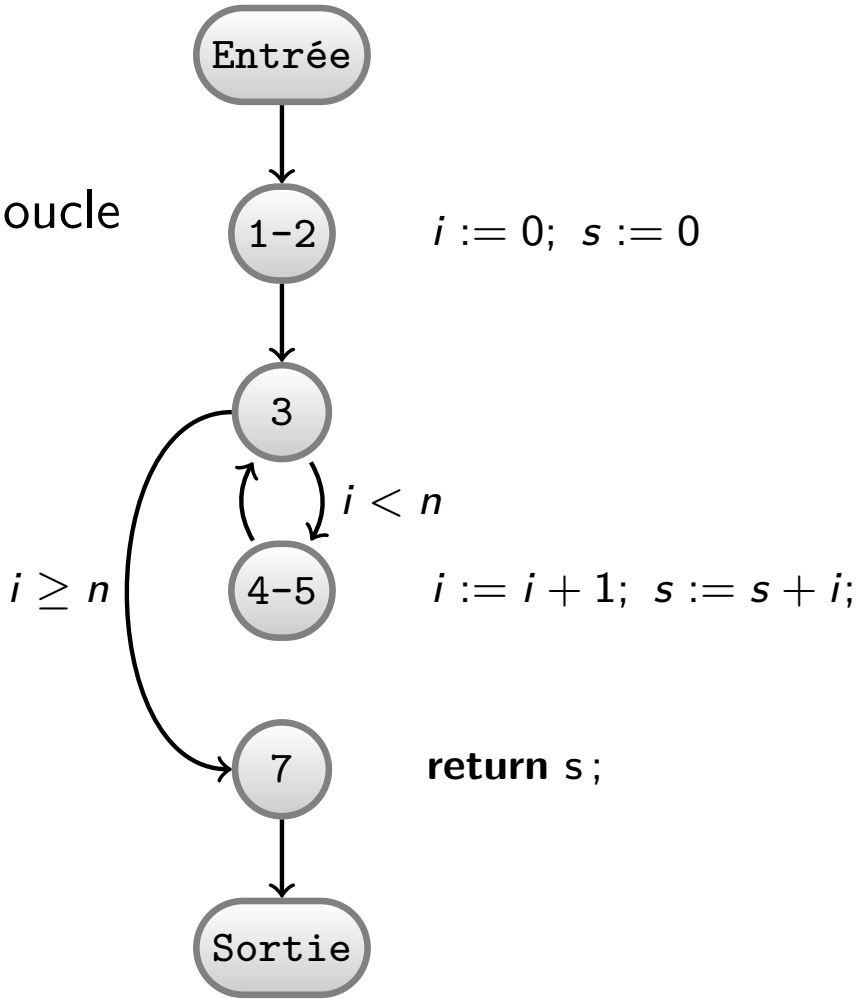
Exécution Symbolique : Boucle `While`

Valeur symbolique en 7 ?

(garde, n, i, s)		
$(0 \geq n_0\,,$	$(n_0, 0, 0))$	pas d'exécution de la boucle
$(1 \geq n_0\,,$	$(n_0, 1, 1))$	1 exécution
$(2 \geq n_0\,,$	$(n_0, 2, 3))$	2 exécutions
	$\vdots$	

Invariant de boucle :

$$s = \frac{i(i+1)}{2}$$



Exécution Symbolique : Boucle `While`

Valeur symbolique en 7 ?

(garde, n, i, s)

$(0 \geq n_0\,,\ (n_0, 0, 0))$ pas d'exécution de la boucle

$(1 \geq n_0\,,\ (n_0, 1, 1))$ 1 exécution

$(2 \geq n_0\,,\ (n_0, 2, 3))$ 2 exécutions

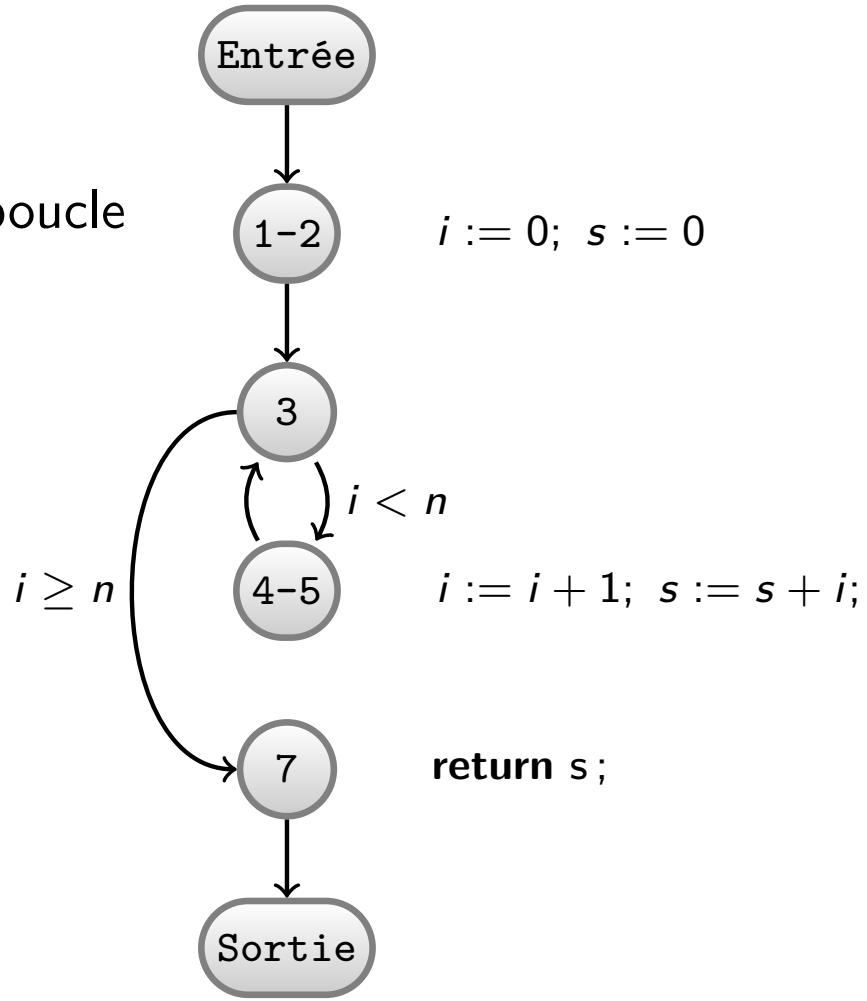
$\vdots$

Invariant de boucle :

$$s = \frac{i(i+1)}{2}$$

Valeur symbolique en 7 :

$(\texttt{true}, (n_0, n_0, n_0(n_0+1)/2))$



Exécution Symbolique

- Rarement utilisée seule
- Efficace en analyse statique pour :
  - Détection des chemins non exécutables
  - Débordement de tableau
  - Preuve de propriétés définies par le testeur :
    - la variable x peut-elle valoir 0 en tel point du programme ?
    - la variable x garde-t-elle une valeur dans l'intervalle $[a, b]$ quelque soit l'exécution du programme
    - peut-on avoir simultanément $x = 0$ et $y = -1$?
- Couramment utilisée dans les outils de génération de tests structurels

Expression des Chemins sous Forme d'Expression Régulières

Description des chemins à l'aide d'expressions régulières :

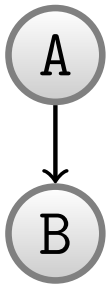
- Définir un langage
- L'alphabet : l'ensemble des nœuds du graphe de contrôle
- '$*$' : répétition d'un mot zéro ou plusieurs fois
- '+' : répétition d'un mot un nombre de fois non nul
- Utilisation des parenthèses pour regrouper les lettres est former un mot

Expression des Chemins sous Forme d'Expression Régulières

3 règles de construction d'un mot du langage :

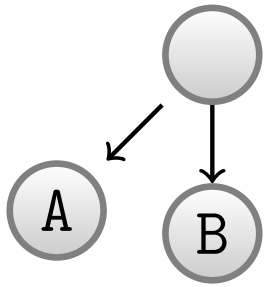
- Transcription d'une séquence
- Transcription d'un choix
- Transcription d'une boucle

Séquence



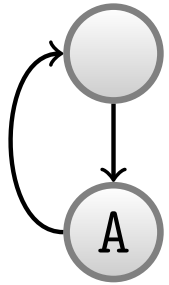
AB

Choix



A | B

Boucle



A* ou A+

Chemins sous Forme d'Expression Régulières

Ensemble de chemin sous forme d'expression régulière pour ce graphe de contrôle :

(Entrée.1-2.3.(4-5.3)*.7.Sortie)

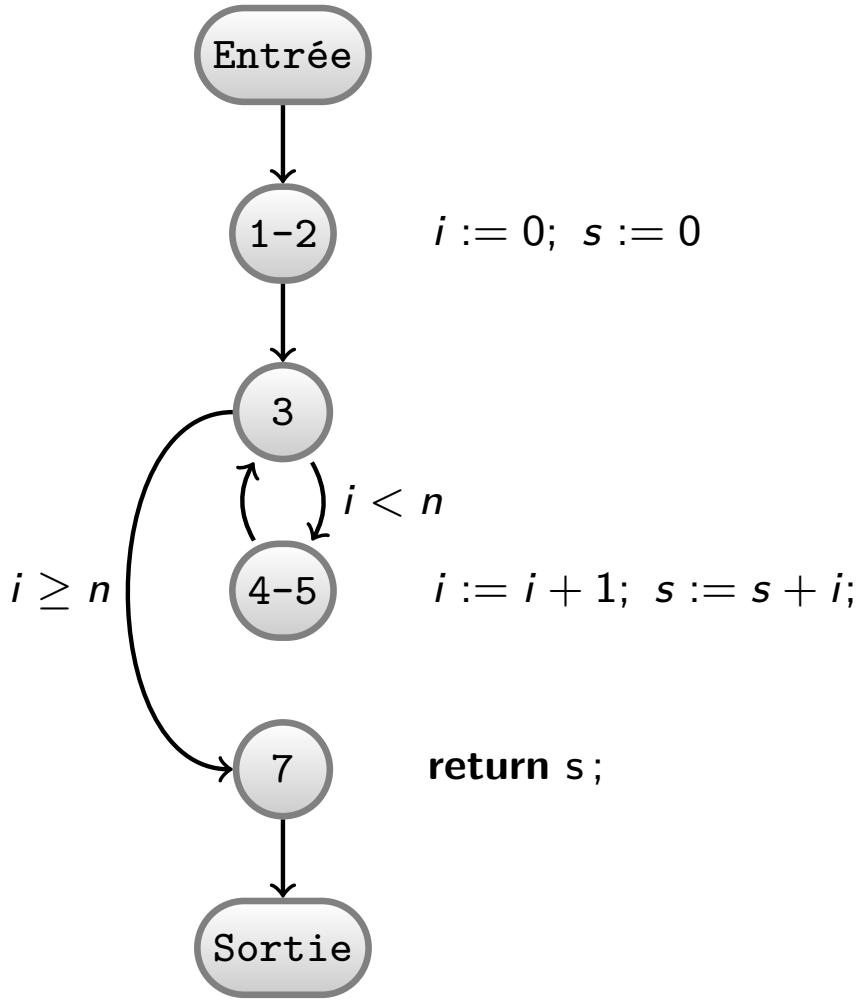
Représente tout les chemins :

(Entrée.1-2.3.7.Sortie)

(Entrée.1-2.3.4-5.3.7.Sortie)

(Entrée.1-2.3.4-5.3.4-5.3.7.Sortie)

⋮



Revues de Code

Principes : les *revues de codes* ou *relectures de code* consistent à faire relire le code source par une ou plusieurs personnes autres que celles qui l'ont codé

- Réalisées par des membres d'équipe de programmation
- Éventuellement assistés de personnes issues des équipes Qualité, Sûreté de Fonctionnement, ...
- Éventuellement uniquement par relectures croisées entre développeurs (pour des raisons de coût)

Revues de Code

Objectifs :

- Vérifier le respect de certains standards de codage (généraux, propres à l'industriel, issus de contraintes sur le système, ...).
- Activité généralement considérée comme faisant partie du contrôle de la qualité
- Identifier certaines pratiques de programmation suspectes

Remarque :

Si le relecteur connaît la spécification correspondant au code source, il pourra également détecter des erreurs fonctionnelles

Revues de Code

Par exemple :

- Nombre suffisant de commentaires *utiles*
- Code structuré (éviter l'utilisation de `goto`, `exit`,...)
- Limiter l'utilisation de littéraux. Utiliser plutôt des constante pour la compréhension et la maintenabilité
- Taille de procédure et fonction acceptables
- Décision exprimée de façon simple
- Absence de boucles utilisant l'opérateur $\neq$ dans la condition de terminaison
  - utiliser `while i < max`
  - plutôt que `while i` $\neq$ `max`

Revues de Code

Par exemple (suite) :

- Variable non initialisée
- Division par zéro
- Mauvaise utilisation d'indice de tableau
- Mauvaise manipulation de fichier (fichier non fermé)
- Mauvaise gestion de la mémoire (fuite mémoire)
- Comparaison entre réels (erreur de précision)
- Pas d'effet de bord sur les paramètres de la fonction (ne pas les modifier à l'intérieur de la fonction

Revues de Code

Conclusion :

- La vérification de tous ces critères permet d'avoir une bonne idée de la qualité du code source
- Ne montre pas pour autant que le code est correct
- Relectures de code efficaces mais coûteuses en ressources humaines
- Mais ne nécessite pas d'outillage particulier
- Permettent de garantir un bon niveau de maintenabilité

Flot de Données

Critères de couverture basés sur le flot de contrôle se focalisent sur les instructions et les séquences d'instructions en termes de chemin d'exécution

$\Rightarrow$ ne considère pas la causalité entre différentes instructions

```
I1 : X=1
⋮
I2 : If X = 1 then
```

Il peut être intéressant de couvrir tous les chemins passant par `I1` puis `I2` sans couvrir tous les chemins d'exécution possibles

Flot de Données

Critères de couverture basés sur le flot de données :

- Déclaration de variable
- Initialisation de variable
- Affectation de valeur à une variable
- Lecture de la valeur d'une variable (opération arithmétique ou prédicat)
- Permet de combler l'écart qui existe entre le critère « toutes les branches » et le critère « tous-les-chemins »

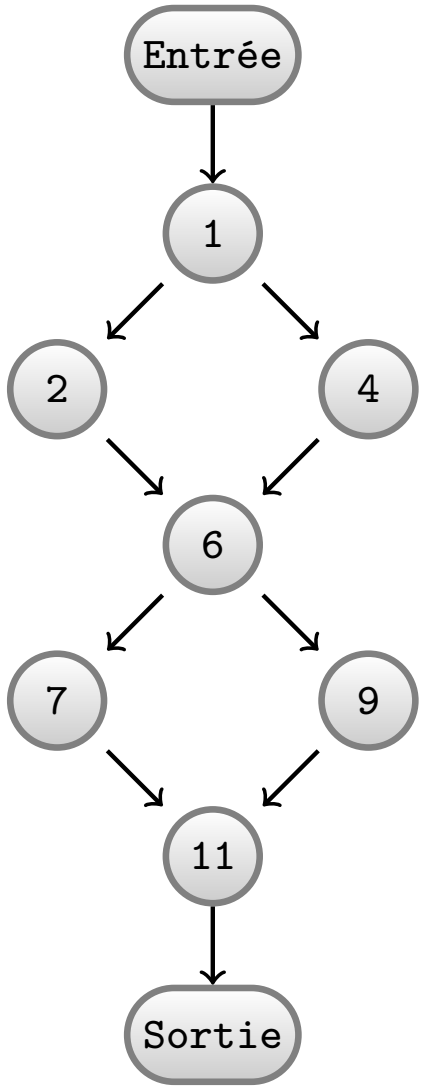
$\Rightarrow$ Prend en compte l'utilisation des variables au lieu des décisions

Flot de Données

Chemins toutes les branches :

- (Entrée, 1, 2, 6, 7, 11, Sortie),
  (Entrée, 1, 4, 6, 9, 11, Sortie)
- (Entrée, 1, 2, 6, 9, 11, Sortie),
  (Entrée, 1, 4, 6, 7, 11, Sortie)

- Le premier ensemble de chemins ne permet pas de découvrir l'erreur de division par zéro
- Le deuxième ensemble de chemins permet de découvrir :
  - soit l'erreur
  - soit que ce chemin n'est pas faisable



$x := 0$

$y := 10/x$

Construction du Graphe de Flots de Données

Le graphe de Flot de données est :

- Un graphe de flot de contrôle
- Annoté avec les informations sur la vie des variables

Au niveau d'une instruction, une variable peut :

- Être définie (ou déclarée) : `int` x
- Recevoir une valeur (affectation) : $x = 5$
- Être utilisée dans un calcul : $y = x + 5$
- Être utilisée dans un prédicat : `if` $x = 5$
- Être détruite (*Killed*) : sortie du domaine de définition de la variable

Graphe de Flots de Données

Critères basés sur le flot de données :

- Sélectionnent les données de test en fonction :
  - des définitions des variables du programme
  - des utilisations des variables du programme

Définitions sur les occurrences de variables :

- une variable est dite *définie* lors d'une instruction si la valeur de la variable est modifiée (affectations)
- Une variable est dite *référencée* si la valeur de la variable est utilisée

Annotation du Graphe de Flots de Données

Variable référencée :

- Dans le prédicat d'une instruction de décision (`if`, `while`, ...) : il s'agit d'une *p-utilisation*
- Dans les autres cas (par exemple dans un calcul), il s'agit d'une *c-utilisation*

Annotation du graphe de contrôle :

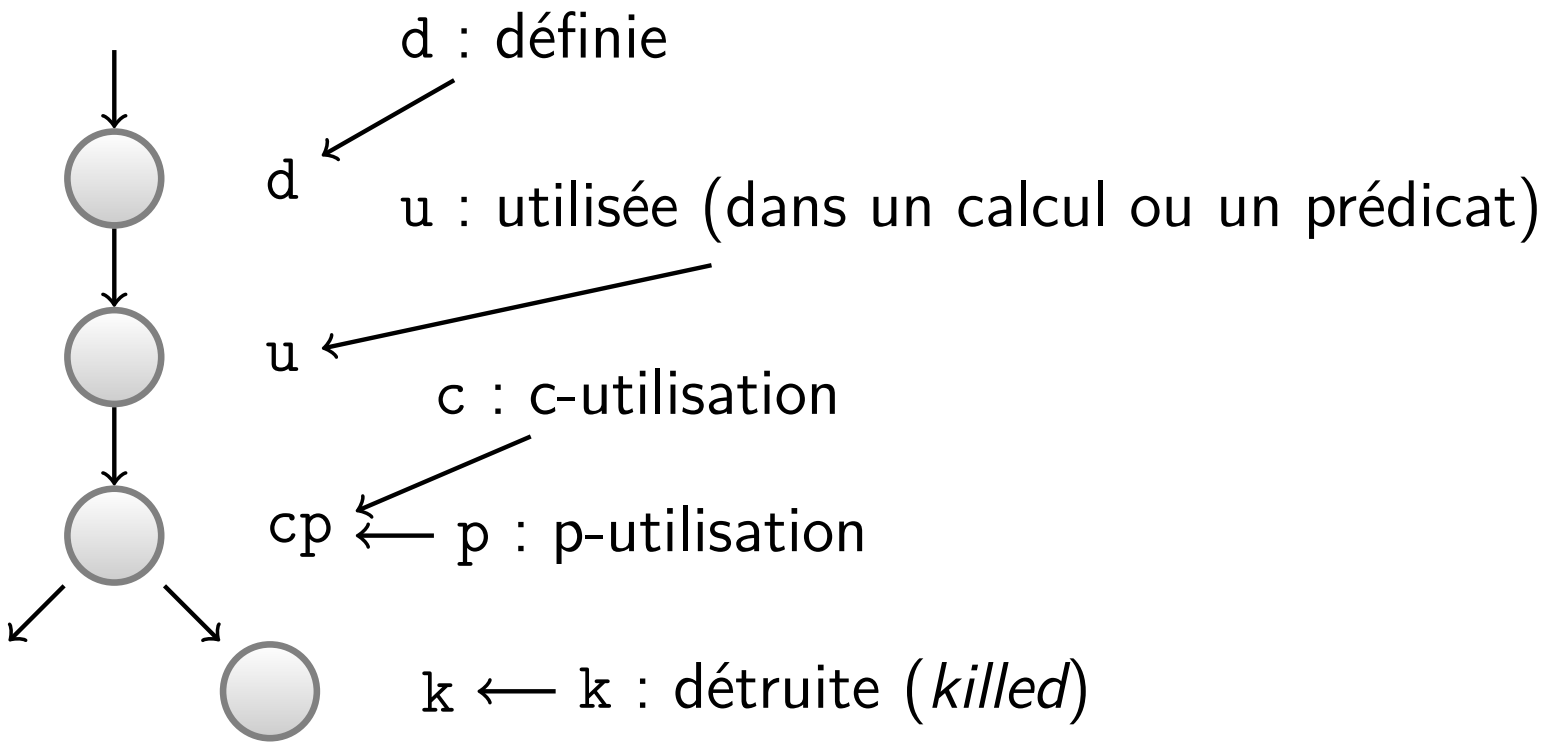
- Définition : `d`
- Utilisation : `u`
- Utilisation dans un calcul (*c-utilisation*) : `c`
- Utilisation dans un prédicat (*p-utilisation*) : `p`
- Destruction (*Killed*) : `k`

Graphe de Flots de Données

Cas d'utilisations multiples d'une variable :

- Instruction utilisant une variable dans un calcul et dans un prédicat :
  - cas de *c-utilisation* et de *p-utilisation* conjointe
  - Exemple : `if` $(x + 5) > 0$
  - Nœud étiqueté par 2 lettres : `cp`
- Instruction utilisant une variable dans un calcul et dans une affectation :
  - cas de *c-utilisation* et de déclaration
  - Exemple : $x = x + 5$
  - Nœud étiqueté par 2 lettres : `dc`

Exemple de Graphe Étiqueté



Graphe de Flot de Données

Une fois construit :

- Utilisé statiquement pour :
  - La recherche de séquences particulières
  - La mise en évidence des erreurs de réalisation
  - Exemple : oublie de l'affectation d'une variable
- Utilisé dynamiquement :
  - Définition de critères de couverture liés à la vie des variables

Utilisation Statique de l'Étiquetage des Nœuds

- L'exécution d'un logiciel correspond à un parcours du graphe de contrôle associé
- En observant l'utilisation d'une variable, l'annotation du graphe de contrôle va produire un mot sur l'alphabet $\{\mathtt{d}, \mathtt{u}, \mathtt{c}, \mathtt{p}, \mathtt{k}\}$
- Exemple de mot : « `dupcduuk` »
- L'ensemble des mots pouvant être engendrés forme un langage
- Analyser l'exécution du logiciel revient à analyser ce langage
- Analyse faite *à priori* et de façon statique

Utilisation Statique de l'Étiquetage des Nœuds

Description du langage à l'aide d'expressions régulières :

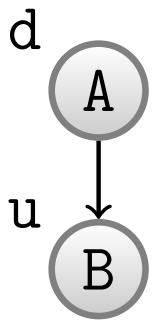
- Utilisant l'alphabet : $\{\mathtt{d}, \mathtt{u}, \mathtt{c}, \mathtt{p}, \mathtt{k}\}$
- '*' : répétition d'un mot zéro ou plusieurs fois
- '+' : répétition d'un mot un nombre de fois non nul
- Utilisation des parenthèses pour regrouper les lettres et former un mot
- Exemple : dans « `kdup(pcd)+u*k` »
  - `(pcd)+` : `pcd` est répété un nombre de fois non nul
  - `u*` : `u` est répété zéro ou plusieurs fois
- Avant toute utilisation une variable est considérée comme détruite

Graphe de Flot de Données

3 règles de construction d'un mot du langage :

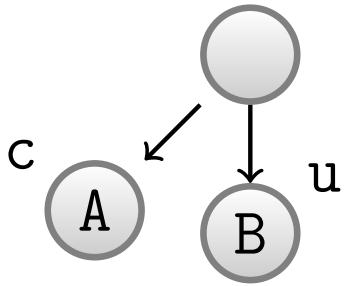
- Transcription d'une séquence
- Transcription d'un choix
- Transcription d'une boucle

Séquence



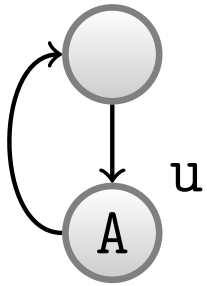
du

Choix



c | u

Boucle



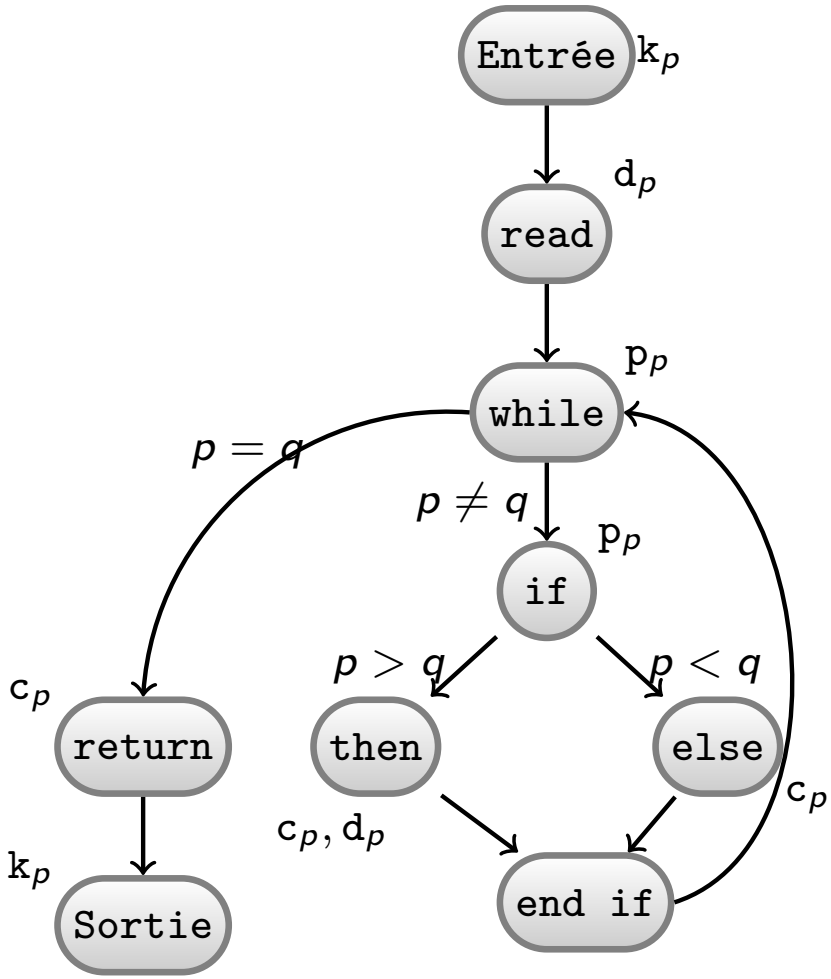
u* ou u+

Calcul du PGCD de 2 entiers

Pré-condition : p et q entiers naturels positifs

PGCD(p, q) :

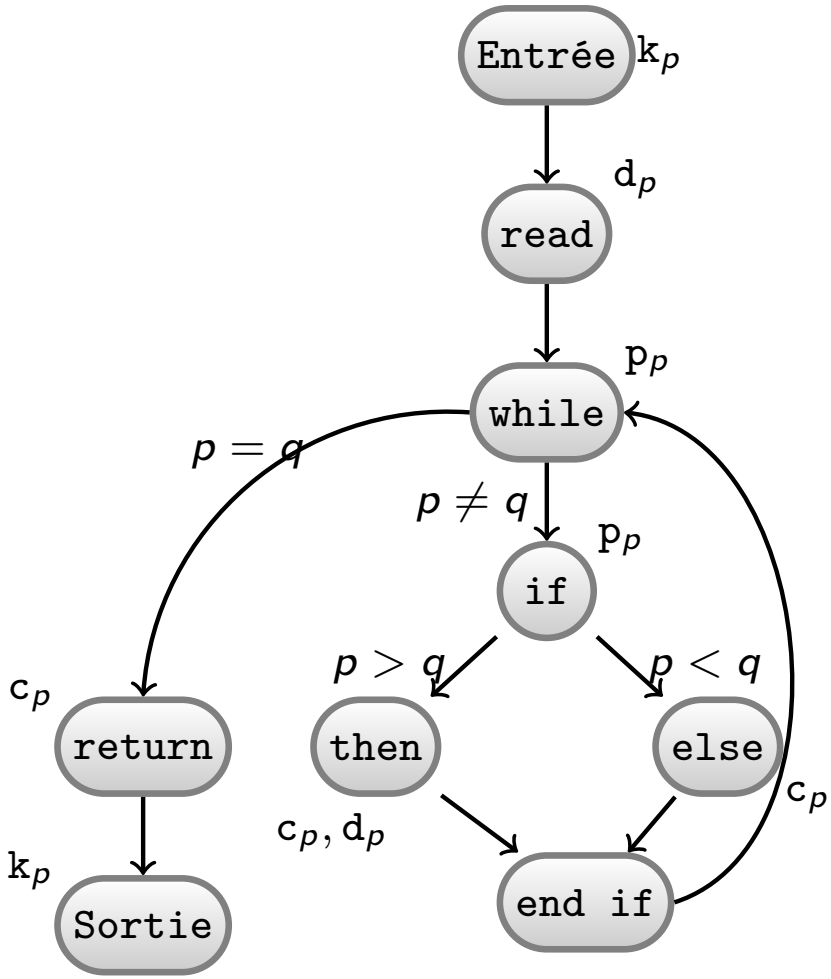
1: read(p, q)
2: **while** $p \neq q$ **do**
3: **if** $p > q$ **then**
4: $p = p - q$
5: **else**
6: $q = q - p$
7: **end if**
8: **end while**
9: return p



Calcul du PGCD de 2 entiers

Langage engendré par les exécutions possibles vis-à-vis de la variable p :

- `kd(pp(cd|c))*pck`



Graphe de Flot de Données

Langage engendré par les exécutions possibles vis-à-vis de la variable p :

- `kd(pp(cd|c))*pck)`
- mot commençant par « `kd` »
- qui se terminent par « `pck` »
- qui comprennent en leur milieu zéro ou plusieurs fois un mot commençant par « `pp` » suivi de « `cd` » ou de « `c` »

Exemple :

- « `kdpck` » correspond au chemin : (Entrée,read,While,Return,Sortie)
- « `kdppcdpck` » correspond au chemin : (Entrée,read,While,If,Then,EndIf,Return,Sortie)

Graphe de Flot de Données

Détection des erreurs potentielles :

- Par analyse du langage
- Recherche des séquences d'erreur à 2 caractères :
  - Exemple : « `ku` » utilisation d'une variable après qu'elle ai été détruite
- Recherche des séquences d'alarme à 2 caractères

Graphe de Flot de Données

Séquences caractéristiques et interprétation en terme de comportement :

`dd`	Double définition
`du` (`dc`, `dp`)	Définition puis utilisation ; séquence normale
`dk`	Définition puis destruction ; probablement une *erreur* (variable inutilisée)
`ud`	Utilisation avant définition ; si la variable a été définie précédemment, cas *normal*, sinon c'est une *erreur*

Graphe de Flot de Données

Séquences caractéristiques et interprétation en terme de comportement (suite) :

`uu`	Succession d'utilisation ; cas *normal*
`uk`	Utilisation puis destruction ; cas *normal*
`kd`	Définition après destruction ; cas *normal*
`ku` (`kc`,`kp`)	Destruction puis utilisation ; probablement une *erreur* (variable non initialisée)
`kk`	Succession de destructions ; probablement une *erreur*

Graphe de Flot de Données

Analyse Automatique de séquences :

- Nécessité de simplifier les expressions définissant les langages
- En particulier, simplifier les expressions contentant '$*$' et '+'

Theorem (Théorème de Huang (1979))

Soit un alphabet $\{A, B, C\}$ et S une chaîne de 2 caractères sur cet alphabet.
Si S est une sous-chaîne des mots $AB^nC(n > 0)$ alors S est également une sous-chaîne de AB^2C

Graphe de Flot de Données

Conséquences pour la recherche de séquence d'erreur ou d'alarme de 2 caractères :

- $X+$ peut être remplacé par X^2
- $X*$ peut être remplacé par $1|X^2$

Par itération de ces substitutions :

La recherche de séquences d'erreur ou d'alarme dans une expression régulière peut se ramener à la recherche dans une expression régulière sans '+' et sans '$*$'

Graphe de Flot de Données

1: **Input :** $A[\,], z$
2: $i := 0\,;$
3: **while** $i = 0 \wedge i \leq z$ **do**
4: $\quad x := A[i]\,;$
5: $\quad i := i + 1$
6: **end while**
7: **If** $x = 0$ **Then** ...

Expression régulière concernant la variable x :

- $\mathtt{kd*p}$
- substitution de $\mathtt{d*}$ par $\mathtt{(1|d^2)}$: $\mathtt{k(1|d^2)p}$
- $\mathtt{kp|kd^2p}$

Graphe de Flot de Données

1: **Input :** $A[\,], z$
2: $i := 0\,;$
3: **while** $i = 0 \wedge i \leq z$ **do**
4: $\quad x := A[i]\,;$
5: $\quad i := i + 1$
6: **end while**
7: **If** $x = 0$ **Then** ...

Expression régulière concernant la variable x :

- $\mathtt{kd*p}$
- substitution de $\mathtt{d*}$ par $\mathtt{(1|d^2)}$: $\mathtt{k(1|d^2)p}$
- $\mathtt{kp|kd^2p}$

Séquence d'erreur « $\mathtt{kp}$ » :

$\Rightarrow$ utilisation d'une variable non initialisée

Critères de Couverture Associés aux Flots de Données

- Approche précédente : analyse statique du graphe de donnée
- Graphe de flot de données peut aussi être utilisé pour définir des objectifs de couverture
- Objectifs de couverture basés sur le flux de données
- « suivre » les chemins qui vont de la définition des variables à leur utilisation

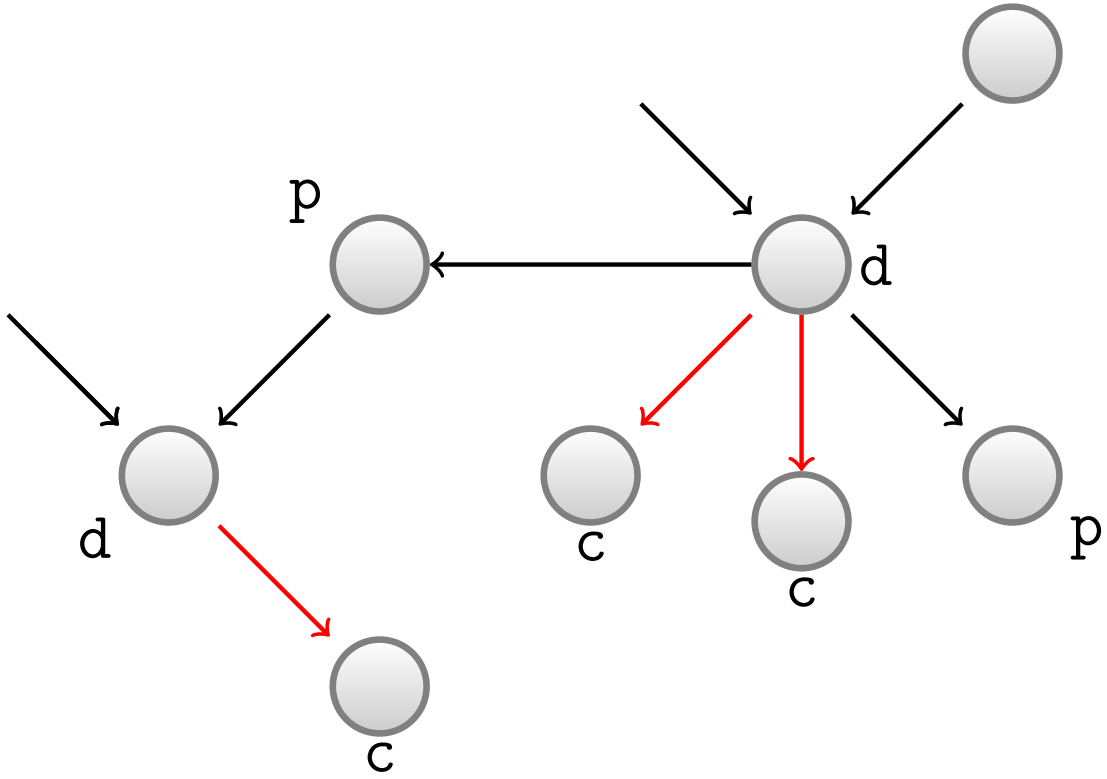
Critères *Toutes-Les-Utilisations* dans un Calcul

ACU – *All Computation Uses*

- Vérifier que la valeur donnée à une variable est utilisée correctement dans les calculs
- Ils s'agit de couvrir toutes les utilisations dans un calcul
- Pour chaque variable du logiciel, parcours de toutes les portions de chemins qui vont d'une définition d'une variable vers une utilisation de celle-ci dans un calcul sans repasser par une nouvelle définition de cette variable

Critères ACU – *All Computation Uses*

Portion d'un graphe de contrôle annoté vis-à-vis d'une variable :



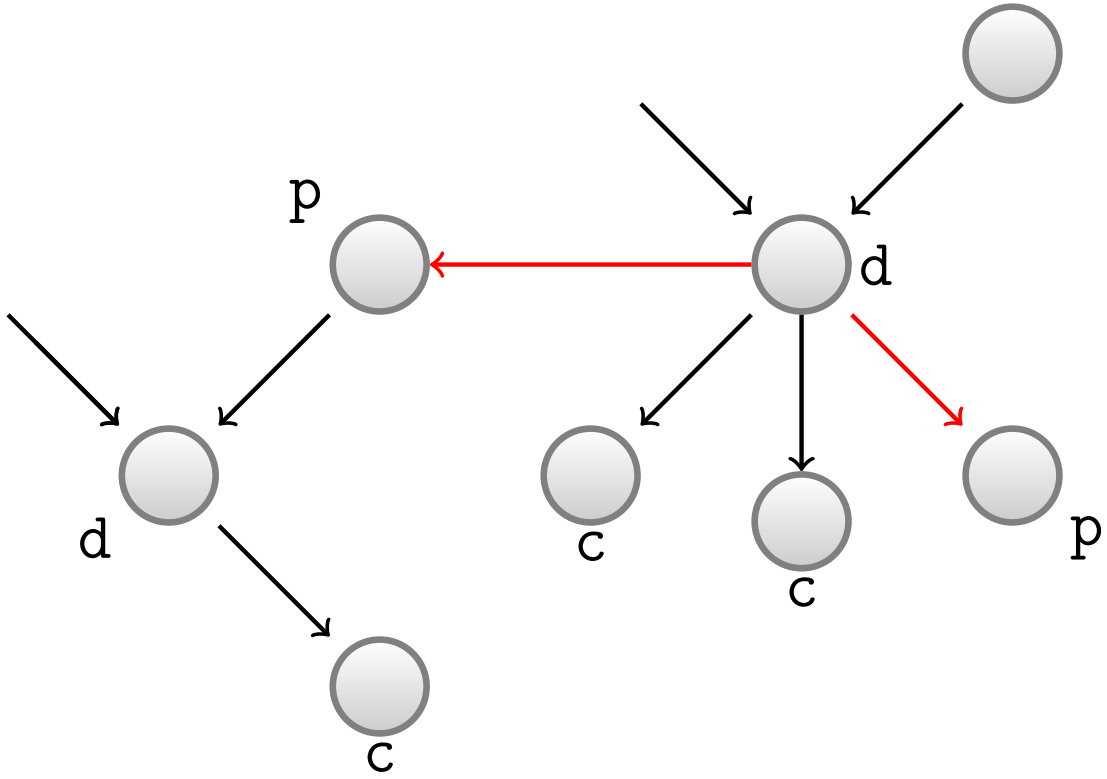
Critères *Toutes-Les-Utilisations* dans un Prédicat

APU – *All Predicat Uses*

- Analyser l'impact d'une définition sur les branchements qui suivent cette définition
- Ils s'agit de couvrir toutes les utilisations dans un prédicat
- Pour chaque variable du logiciel, parcours de toutes les portions de chemins qui vont d'une définition d'une variable vers une utilisation de celle-ci dans un branchement sans repasser par une nouvelle définition de cette variable

Critères APU – *All Predicat Uses*

Portion d'un graphe de contrôle annoté vis-à-vis d'une variable :

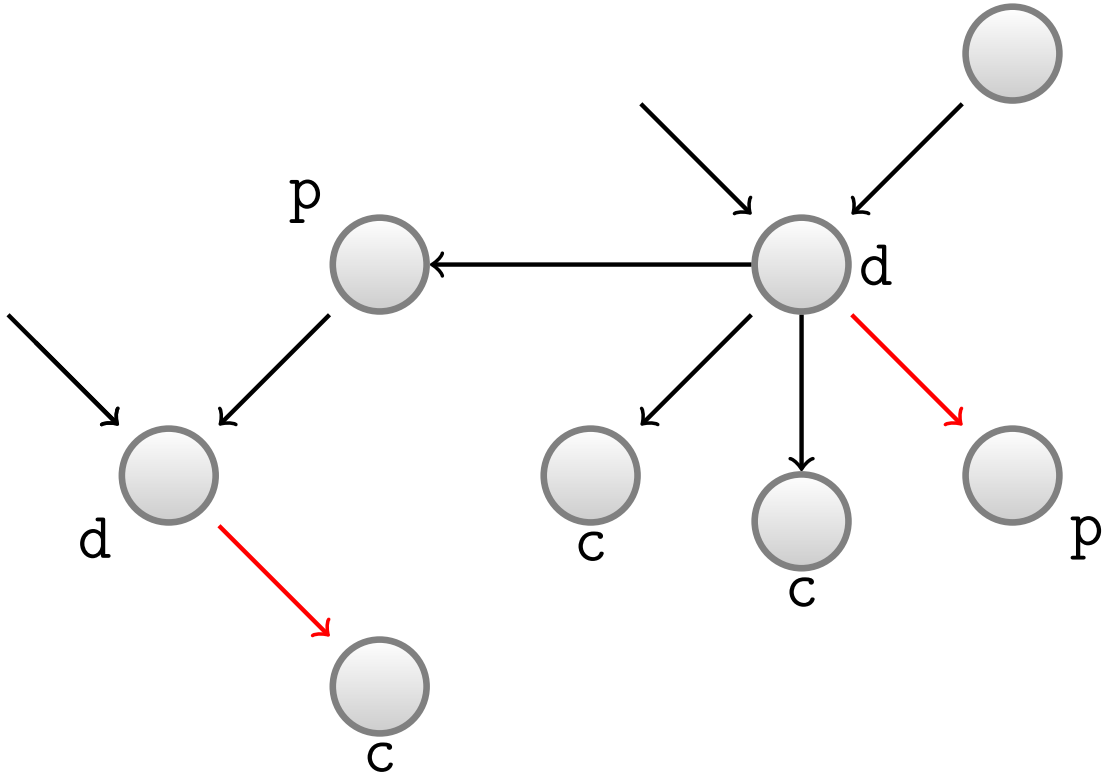


Critères *Toutes-Les-Définitions*
AD – *All Definitions*

- Analyser l'impact d'une définition sur une utilisation de celle-ci (p-utilisation ou c-utilisation)
- Pour chaque variable du logiciel, parcours d'une portion de chemin qui va d'une définition d'une variable vers une utilisation (p-utilisation ou c-utilisation) de celle-ci sans repasser par une nouvelle définition de cette variable

Critères AD – *All Definitions*

Portion d'un graphe de contrôle annoté vis-à-vis d'une variable :



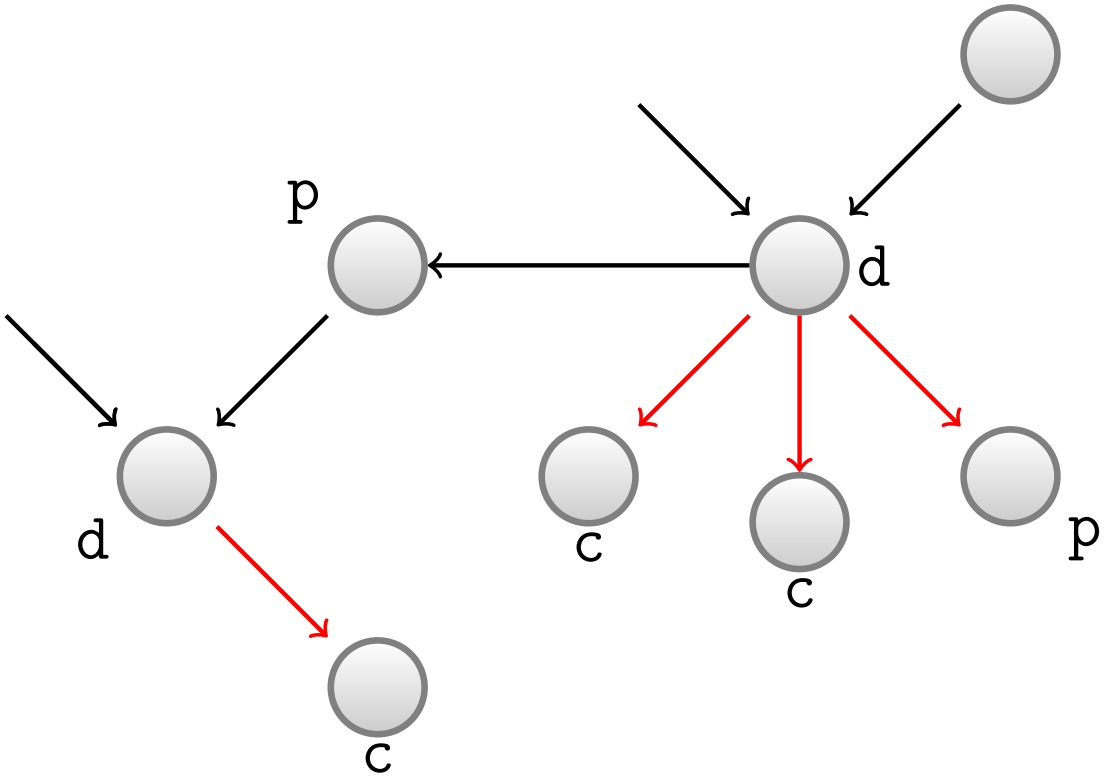
Critères *Toutes-Les-Utilisations* dans un Calcul et Au Moins une Utilisation dans un Prédicat

ACU + P – *All Computation Uses and one Predicat*

- Rapprocher les précédents critères afin d'améliorer la couverture
- Chercher à couvrir toutes les c-utilisations d'une définition et au moins une p-utilisation
- Pour chaque variable du logiciel, parcours de toutes les portions de chemins qui vont d'une définition d'une variable vers une utilisation de celle-ci dans un calcul et d'une portion de chemin vers une utilisation dans un prédicat sans repasser par une nouvelle définition de cette variable

Critères ACU + P – *All Computation Uses and one Predicat*

Portion d'un graphe de contrôle annoté vis-à-vis d'une variable :



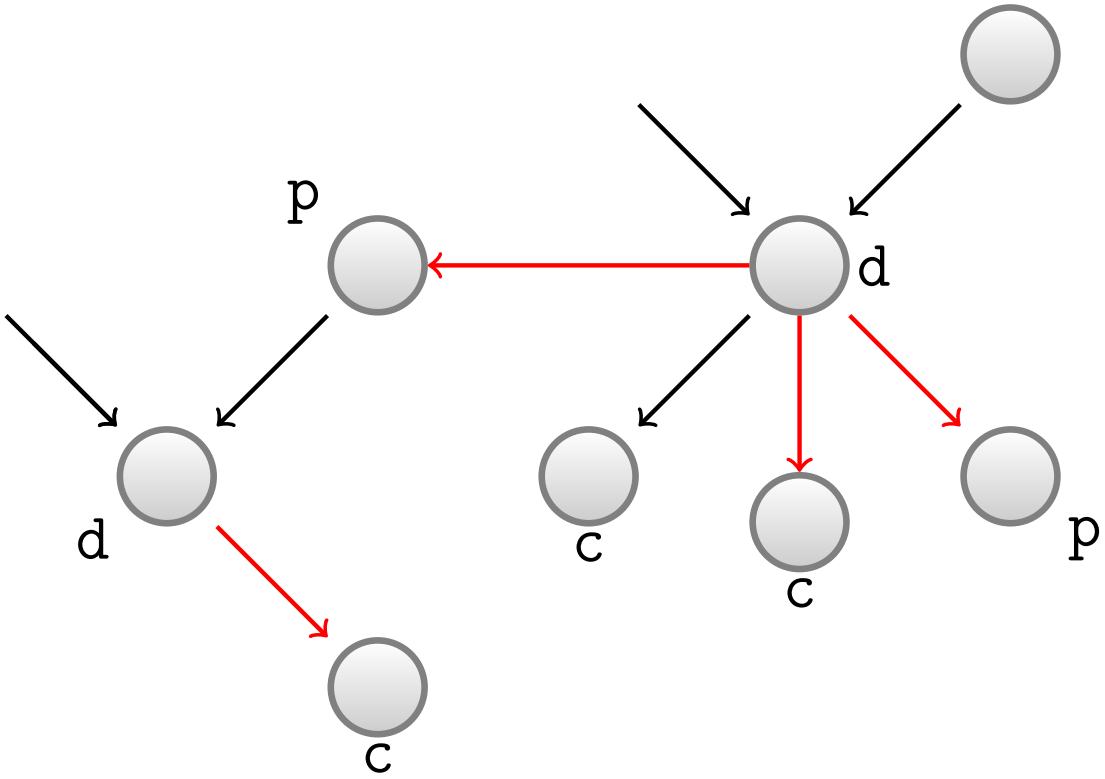
Critères *Toutes-Les-Utilisations* dans un Prédicat et Au Moins une Utilisation dans un Calcul

APU + C – *All Predicat Uses and one Computation*

- Critère de couverture « symétrique » du précédent
- Chercher à couvrir toutes les p-utilisations d'une définition et au moins une c-utilisation
- Pour chaque variable du logiciel, parcours de toutes les portions de chemins qui vont d'une définition d'une variable vers une utilisation de celle-ci dans un prédicat et d'une portion de chemin vers une utilisation dans un calcul sans repasser par une nouvelle définition de cette variable

Critères APU + C – *All Predicat Uses and one Computation*

Portion d'un graphe de contrôle annoté vis-à-vis d'une variable :



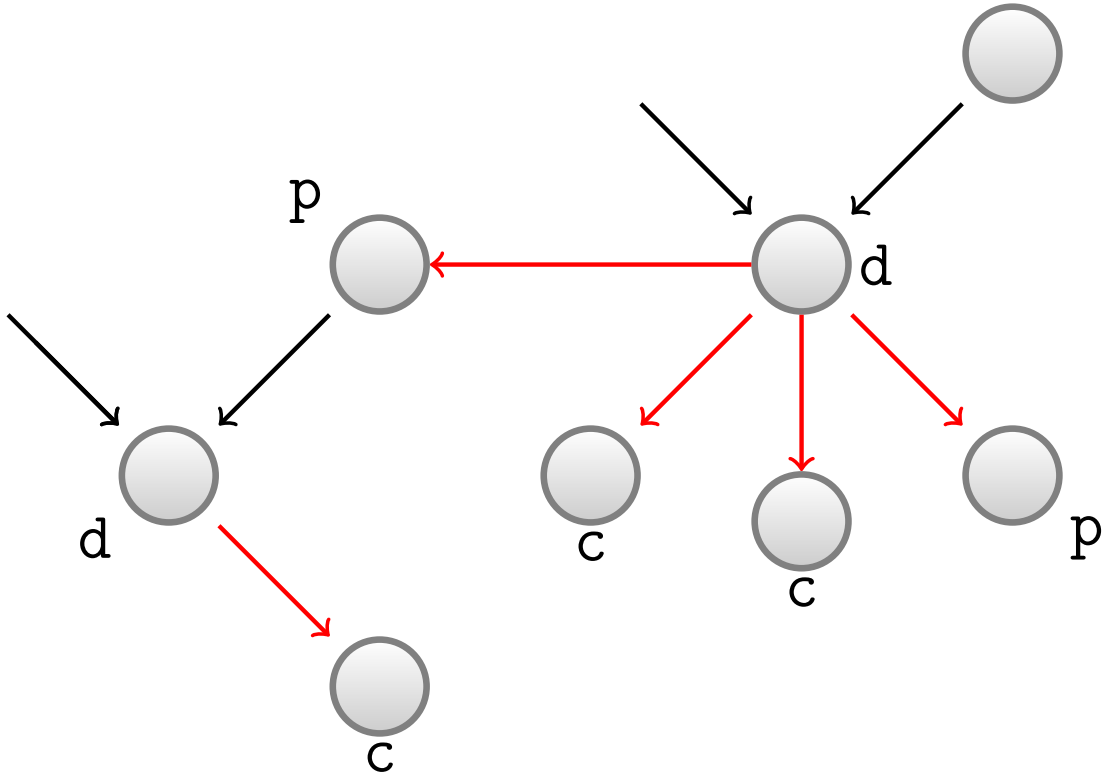
Critères *Toutes-Les-Utilisations*

AU – *All Uses*

- Critère de couverture plus exigent que les précédents
- Chercher à couvrir toutes p-utilisations et toutes c-utilisation
- Pour chaque variable du logiciel, parcours de toutes les portions de chemins qui vont d'une définition d'une variable vers une utilisation de celle-ci dans un prédicat et de toutes les portions de chemins qui vont d'une définition d'une variable vers une utilisation de celle-ci dans un calcul sans repasser par une nouvelle définition de cette variable
- Critère exigent mais réaliste quant à sa mise en œuvre

Critères AU – *All Uses*

Portion d'un graphe de contrôle annoté vis-à-vis d'une variable :



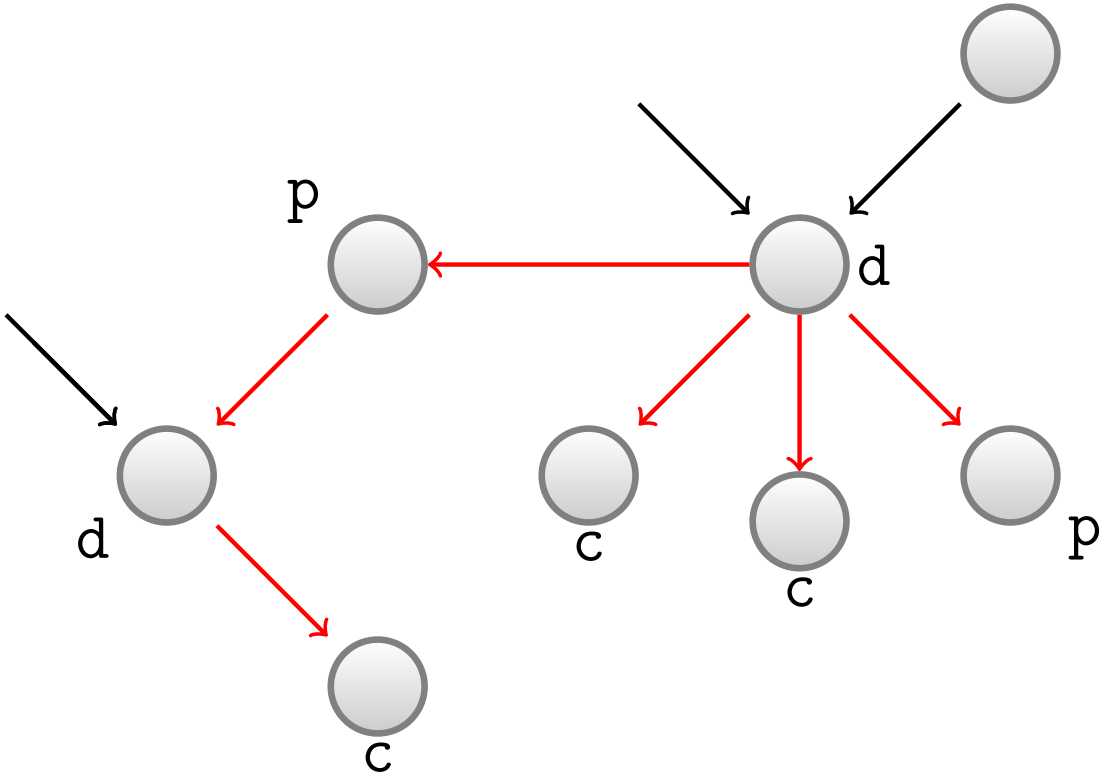
Critère *Tous-Les-Chemins-d'Utilisation* des Définitions

ADUP – *All Definition Uses Path*

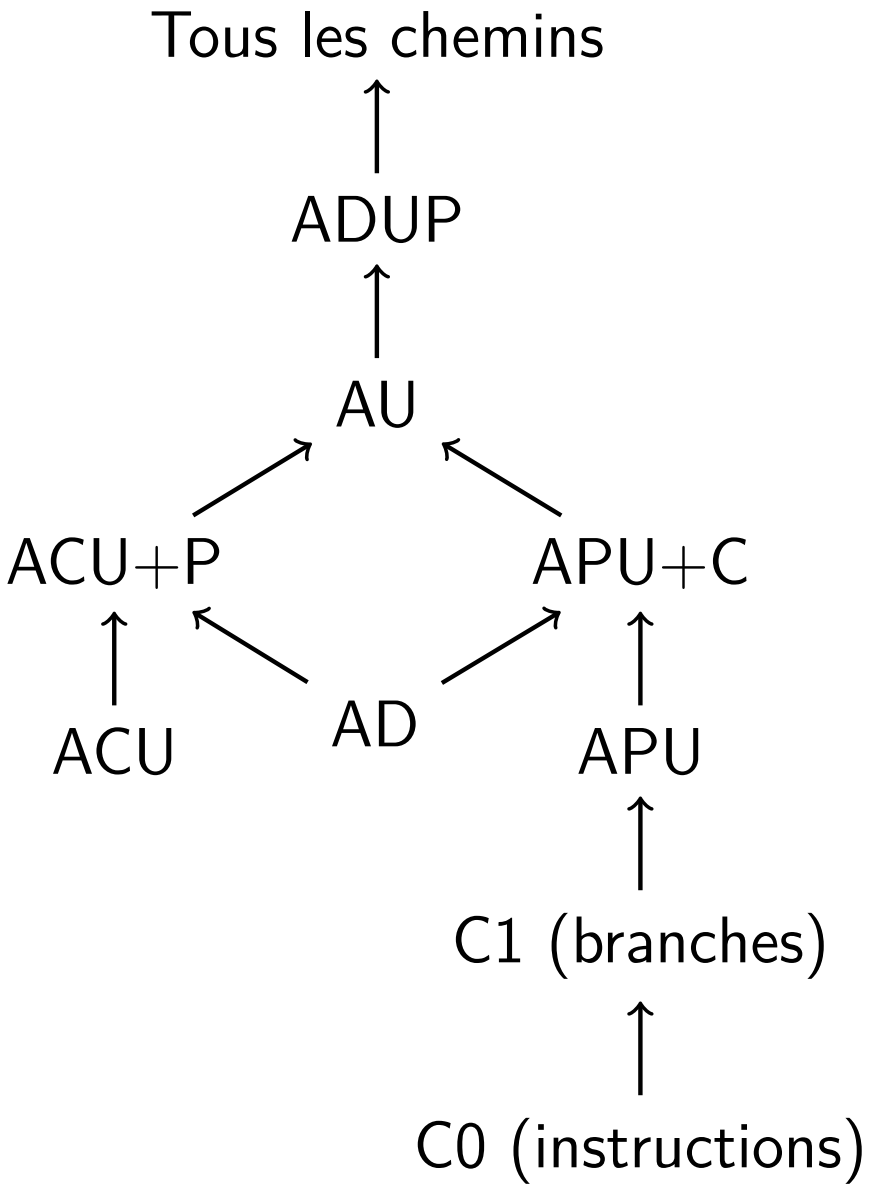
- Tous les critères de couverture vu précédemment se limitent aux chemins sans redéfinition
- Dans le cas des boucles ne fait que une passe
- Pour atteindre une couverture plus complète des boucles, chercher à couvrir des chemins allant d'une définition à une utilisation sans se limiter à la dernière définition
- Critère difficile à obtenir ; se rapproche de la complexité du critère *Tous-Les-Chemins*

Critères ADUP – *All Definition Uses Path*

Portion d'un graphe de contrôle annoté vis-à-vis d'une variable :



Sévérité des Critères de Couvertures



Test Mutationnel

- Technique visant à évaluer des jeux de tests vis-à-vis de la liste des fautes les plus probables qui ont été envisagées (modèles de fautes)
- Mutations : transformations syntaxiques élémentaires de programme
- Étant donné un programme initial à tester, on produit un certain nombre de mutants (programmes identiques au précédent à une instruction près)
- Objectif : sélectionner un jeux de tests efficace pour tester le programme original
- Principal critère de sélection : capacité à différencier le programme initial d'un de ses mutants

Score de Mutation

Pour un programme P :

- Soit $M(P)$ l'ensemble de tous les mutants obtenus par le modèle de faute
- Certains mutants peuvent avoir le même comportement que P, noter E(P)
- Exécution des tests sur chacun des mutants de $M(P)$. On note $DM(P)$ l'ensemble de ceux qui ne fournissent pas le même résultat que P
- Score de mutation $MS(P, T)$:

$$MS(P, T) = \frac{DM(P)}{M(P) - E(P)}$$

Exemple de Test Mutationnel

Pour l'instruction :

$$\texttt{if } a > 0 \texttt{ then } x := y;$$

On peut produire les mutants suivants :

- $\texttt{if } a < 0 \texttt{ then } x := y;$
- $\texttt{if } a \geq 0 \texttt{ then } x := y;$
- $\texttt{if } a > 10 \texttt{ then } x := y;$
- $\texttt{if } a > 0 \texttt{ then } x := y + 1;$
- $\texttt{if } a > 0 \texttt{ then } x := x;$

Règles de Mutations d'Opérateurs

- Suppression d'une expression
- Négation d'une expression booléenne
- Décalage dans l'association de terme
- Échange d'un opérateur arithmétique par un autre
- Échange d'un opérateur relationnel par un autre
- Échange d'un opérateur logique par un autre
- Négation logique
- Échange d'une variable par une autre
- Échange d'une variable par une constante

Opérateurs de mutations

- Mauvais nœud de départ
- Branchement manquant
- Évènement manquant
- Évènements échangés
- Destination échangée
- Sortie manquante
- Sorties échangées

Conclusion :

- Méthode permettant d'avoir une image précise de la fiabilité des tests
- Une condition nécessaire d'arrêt des test peut être, par exemple, que tous les mutants soient tués par le jeux de tests (ou plus d'un certain %)
- Utilisé pour évaluer les jeux de tests produit par d'autre méthodes
- Inconvénient majeur : temps de calcul nécessaire pour l'exécution de tous les mutants