Prouver des propriétés des programmes impératifs

Yann Salmon

Plan national de formation
Algorithmique et programmation au lycée
20 novembre 2017

(0/1)

« D'où parles-tu, camarade ? »

2009	Maitrise de mathématiques
2010	Agrégation de mathématiques option D (Informatique)
2011	M2 d'informatique
2011–2015	Thèse d'informatique
2014	Professeur d'informatique en CPGE

(1/1)

Les logiciels sont présents partout.

(2/1)

Les logiciels sont présents partout.

Comprend-on vraiment ce qu'ils font ?

(2/1)

Les logiciels sont présents partout.

Comprend-on vraiment ce qu'ils font ?

Ce qu'ils peuvent faire ?

(2/1)

Les logiciels sont présents partout.

Comprend-on vraiment ce qu'ils font ?

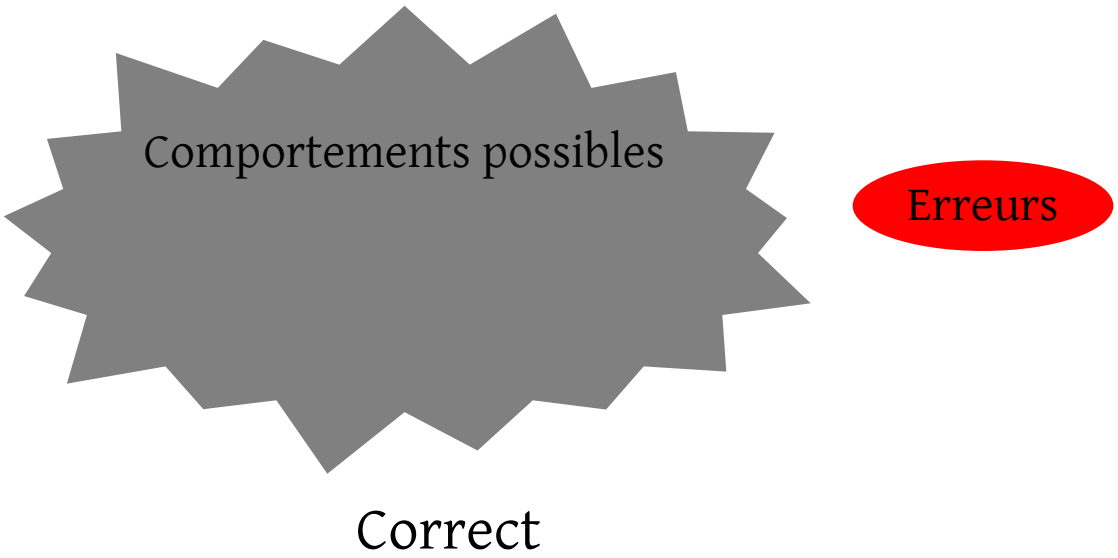
Ce qu'ils peuvent faire ?

Dans tous les cas ?

(2/1)

Vérification

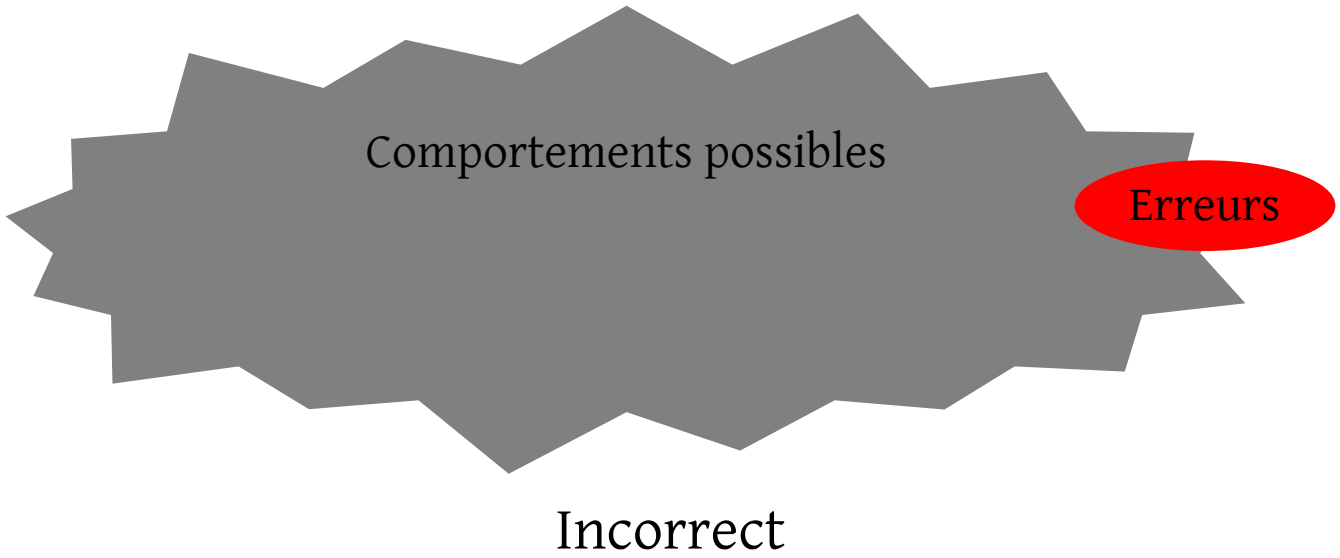
Prouver que le programme ne peut jamais avoir un comportement incorrect.



(3/1)

Vérification

Prouver que le programme ne peut jamais avoir un comportement incorrect.



(3/1)

En physique

Soit un moteur qui doit pouvoir fonctionner à 5000 tours par minute.
On le teste à 10 000 tours par minute. OK

(4/1)

En physique

Soit un moteur qui doit pouvoir fonctionner à 5000 tours par minute.
On le teste à 10 000 tours par minute. OK

En informatique

Un bug d'Excel 2007 provoque un affichage incorrect

$$850 \times 77.1 = 100\,000 \text{ au lieu de } 65\,535$$

et pour 11 autres valeurs *seulement*.
Indétectable par tests.

(4/1)

```
let rec exporap a n =
  if n = 0 then
    1
  else if n mod 2 = 0 then
    exporap (a*a) (n/2)
  else
    a * exporap (a*a) (n/2)
```

$$a^0 = 1$$
$$a^{2n} = (a \times a)^n$$
$$a^{2n+1} = a \times (a \times a)^n$$

- valeurs immuables
- pas d'affectation mais des définitions
- en fait, pas d'instruction

Programme = description de ce qu'**est** le résultat

(5/1)

```
def exporap(a, n) :
  r = 1
  f = a
  p = n
  while p > 0 :
    if p % 2 == 1 :
      r = r * f
    f = f * f
    p = p // 2
  return r
```

- les valeurs peuvent être mutables
- il y a des affectations
- plus généralement, des instructions

Programme = description de ce qu'on **fait** pour obtenir le résultat

(6/1)

Fonctionnel

- tout est immuable

Impératif

- mutations, instructions

(7/1)

Fonctionnel

- tout est immuable
- composition d'expressions

Impératif

- mutations, instructions
- séquences d'instructions

(7/1)

Fonctionnel

- tout est immuable
- composition d'expressions
- on peut abstraire la mémoire

Impératif

- mutations, instructions
- séquences d'instructions
- nécessité d'un modèle mémoire

(7/1)

Fonctionnel

- tout est immuable
- composition d'expressions
- on peut abstraire la mémoire
- preuves matheuses

Impératif

- mutations, instructions
- séquences d'instructions
- nécessité d'un modèle mémoire
- méthodes et formalismes spécifiques

(7/1)

Fonctionnel

- tout est immuable
- composition d'expressions
- on peut abstraire la mémoire
- preuves matheuses

Impératif

- mutations, instructions
- séquences d'instructions
- nécessité d'un modèle mémoire
- méthodes et formalismes spécifiques

Correction de `exporap` fonctionnelle

Pour $n \in \mathbb{N}$, soit $P(n)$ la propriété

$$\forall a \in \mathbb{N}, (\texttt{exporap}\ a\ n) \text{ renvoie } a^n.$$

On procède par récurrence, etc.

(7/1)

Fonctionnel

- tout est immuable
- composition d'expressions
- on peut abstraire la mémoire
- preuves matheuses

Impératif

- mutations, instructions
- séquences d'instructions
- nécessité d'un modèle mémoire
- méthodes et formalismes spécifiques

Correction de `exporap` fonctionnelle

Pour $n \in \mathbb{N}$, soit $P(n)$ la propriété

$$\forall a \in \mathbb{N}, (\texttt{exporap}\ a\ n) \text{ renvoie } a^n.$$

On procède par récurrence, etc.

(7/1)

Quatre instructions de base :

- affectation *var* `=` *expr*
- séquence
- alternative `if`
- boucle `while`

(8/1)

Quatre instructions de base :

- affectation *var* `=` *expr*
- séquence
- alternative `if`
- boucle `while`
- `return`

(8/1)

var = *expr* modifie l'environnement courant de sorte que *var* désigne la valeur de *expr*.
Il y a un *avant* et un *après*.

Propriétés concrètes et affectation

```
# ici v a une valeur inconnue, éventuellement aucune
v = 1900 + 17
# ici v vaut 1917
v = v + 19
# ici v vaut 1936
```


(9/1)

Séquence : faire une chose puis une autre

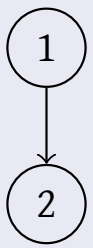
Définition

```
1 instr_1
2 instr_2
```

Exemple

```
1 v = 1900 + 17
2 v = v + 19
```

Graphe de flux de contrôle



(10/1)

Alternative : faire une chose ou bien une autre

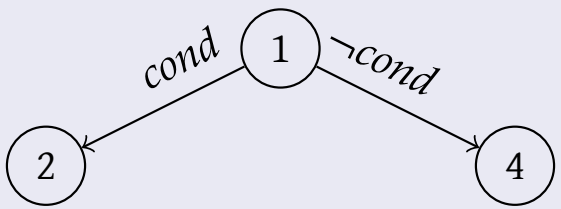
Définition

```
1  if cond :
2    instr_V
3  else :
4    instr_F
5  suite_du_programme
```

Exemple

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```

Graphe de flux de contrôle



(11/1)

Alternative : faire une chose ou bien une autre

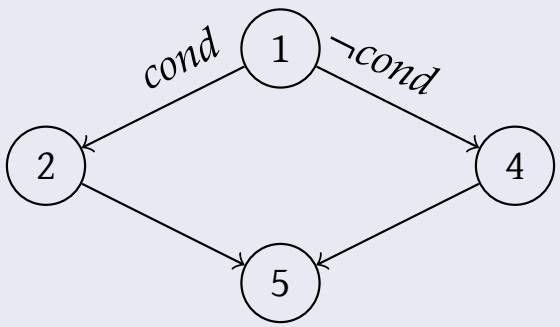
Définition

```
1  if cond :
2    instr_V
3  else :
4    instr_F
5  suite_du_programme
```

Exemple

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```

Graphe de flux de contrôle



(11/1)

Boucle : faire une chose répétitivement

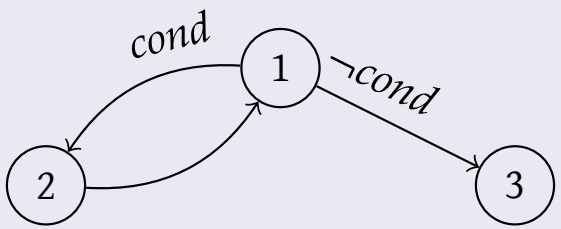
Définition

```
1 while cond :
2     corps
3 suite_du_programme
```

Exemple

```
1 while p > 0 :
2     p = p - 1
3 # etc.
```

Graphe de flux de contrôle



(12/1)

`return` : arrêter l'exécution en renvoyant une valeur

Définition

`return` *expr*

Exemple

```
1  def appartient(x, t) :
2    k = 0
3    while k < len(t) :
4      if t[k] == x :
5        return True
6      k = k + 1
7    return False
```

Graphe de flux de contrôle

Jamais de flèche sortante.

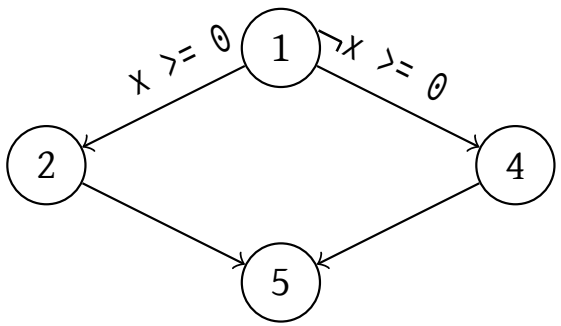
(13/1)

État concret de la machine :

- ligne en cours d'exécution (contrôle)
- valeur de chaque variable (environnement)

Exemple : partant d'un environnement où x vaut 2.

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



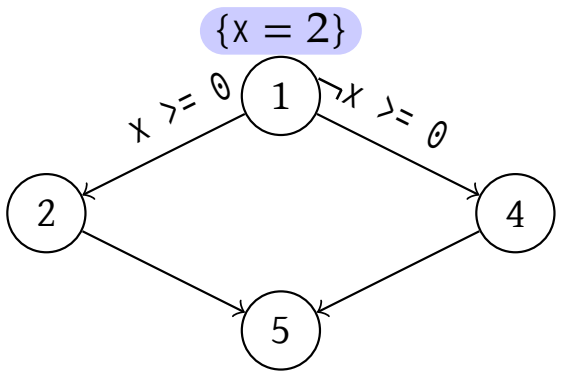
(14/1)

État concret de la machine :

- ligne en cours d'exécution (contrôle)
- valeur de chaque variable (environnement)

Exemple : partant d'un environnement où x vaut 2.

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



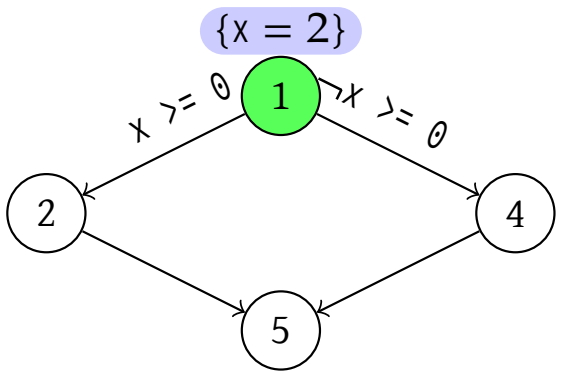
(14/1)

État concret de la machine :

- ligne en cours d'exécution (contrôle)
- valeur de chaque variable (environnement)

Exemple : partant d'un environnement où x vaut 2.

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



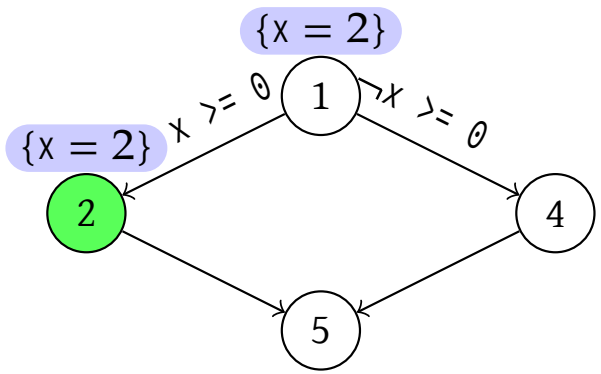
(14/1)

État concret de la machine :

- ligne en cours d'exécution (contrôle)
- valeur de chaque variable (environnement)

Exemple : partant d'un environnement où x vaut 2.

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



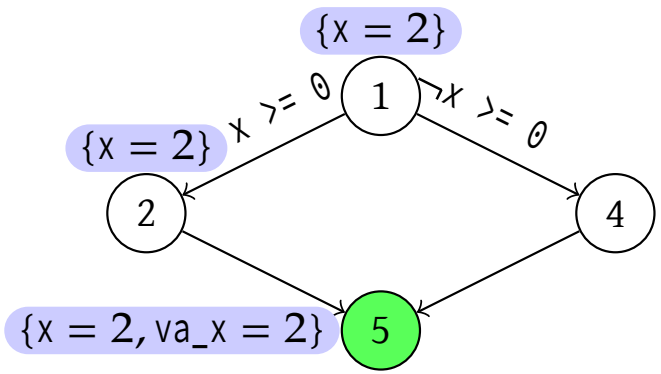
(14/1)

État concret de la machine :

- ligne en cours d'exécution (contrôle)
- valeur de chaque variable (environnement)

Exemple : partant d'un environnement où x vaut 2.

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



(14/1)

Propriété de programme

Énoncé qui met en jeu les (valeurs des) variables du programme.

Le retour de la valeur absolue

$$va\_x \geqslant 0$$

(15/1)

Propriété de programme

Énoncé qui met en jeu les (valeurs des) variables du programme.

Le retour de la valeur absolue

$$va\_x \geqslant 0$$

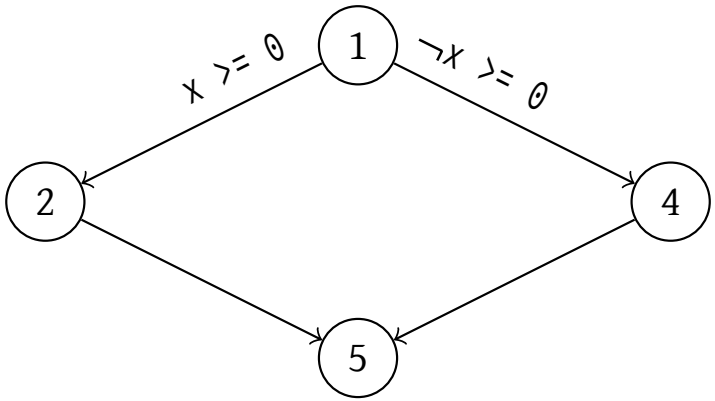
Environnement abstrait

Ensemble de propriétés connues à un moment donné.

$$\{x \in \mathbb{Z}\}$$

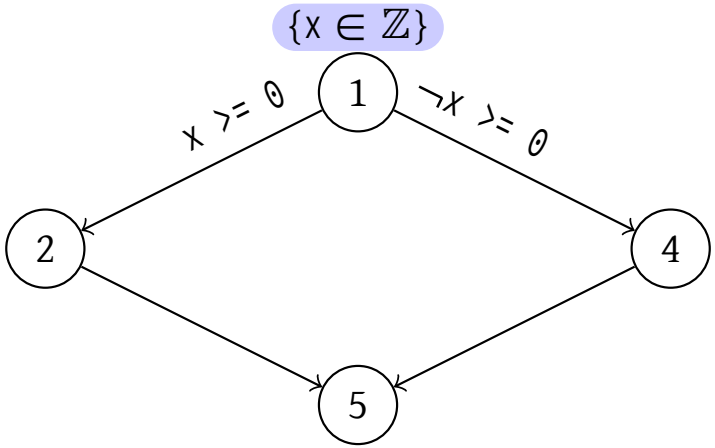
(15/1)

```
1 if x >= 0 :
2   va_x = x
3 else :
4   va_x = -x
5 # etc.
```



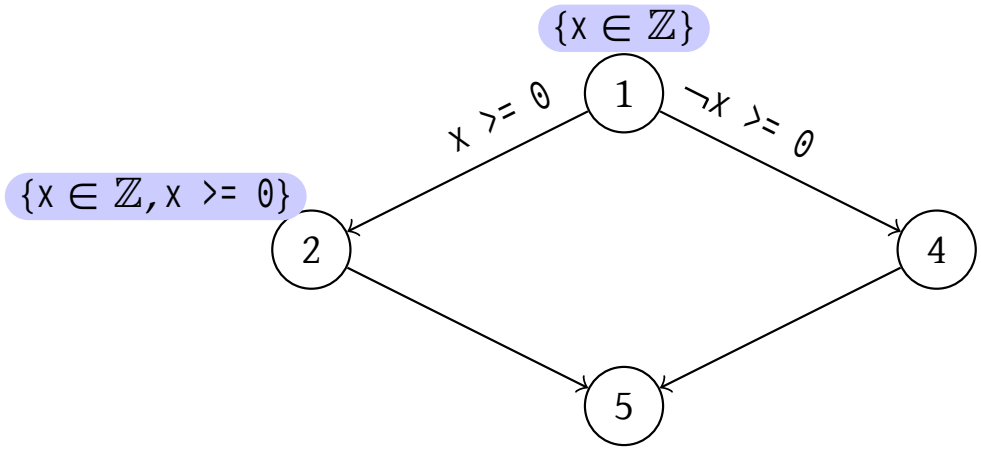
(16/1)

```
1 if x >= 0 :
2   va_x = x
3 else :
4   va_x = -x
5 # etc.
```



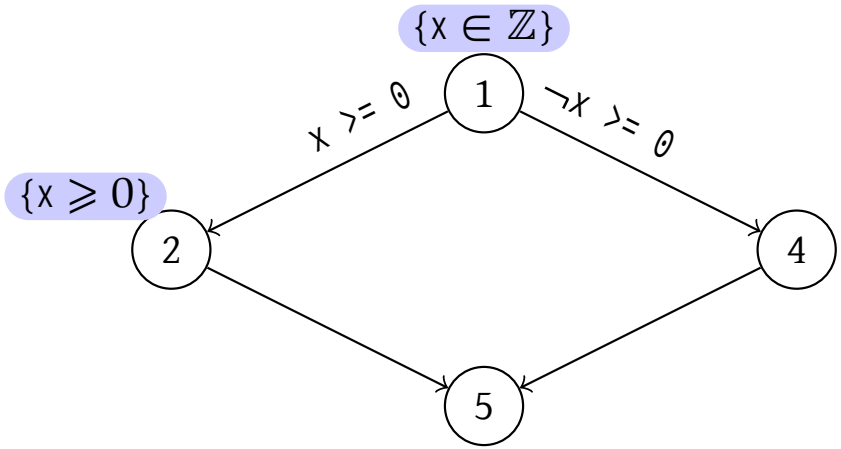
(16/1)

```
1 if x >= 0 :
2   va_x = x
3 else :
4   va_x = -x
5 # etc.
```



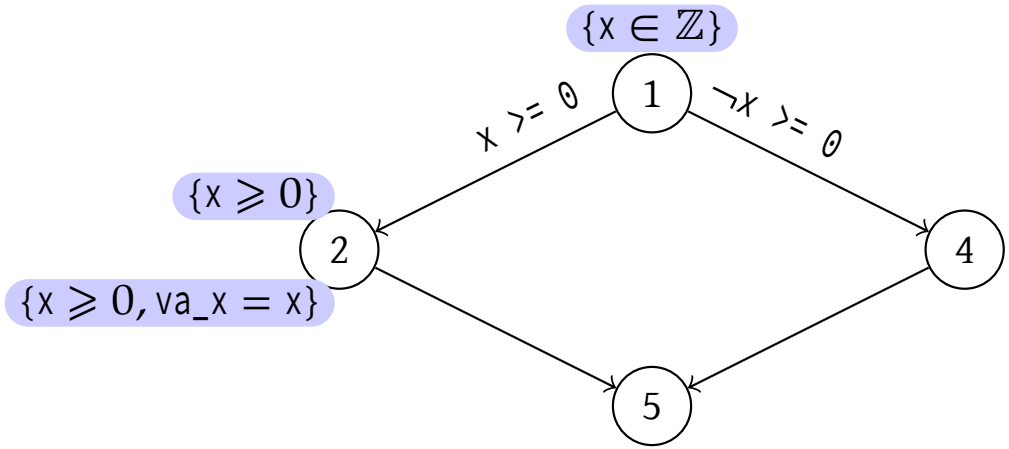
(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



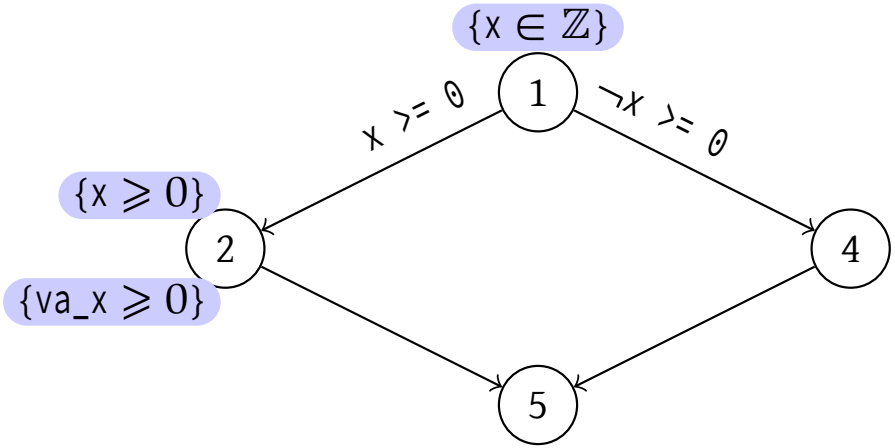
(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



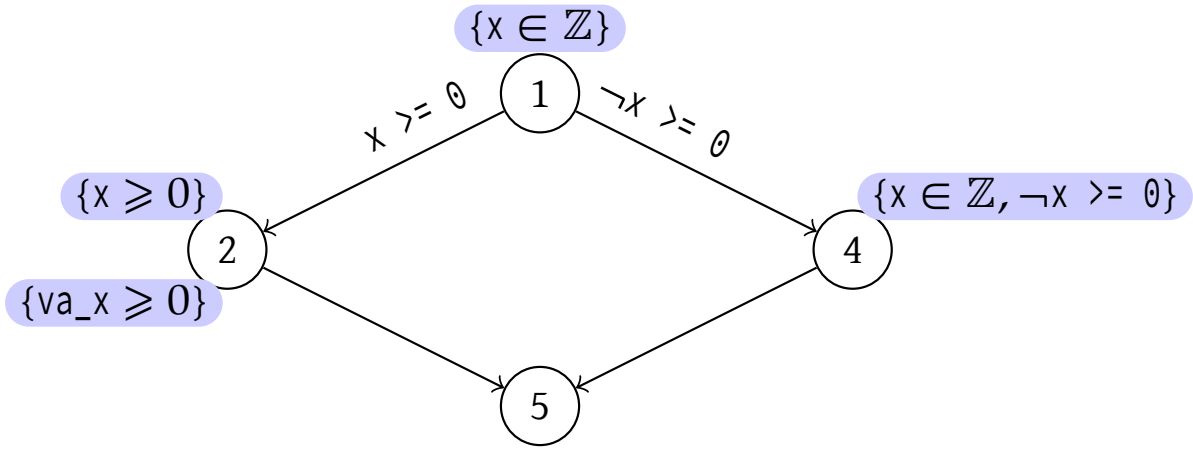
(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



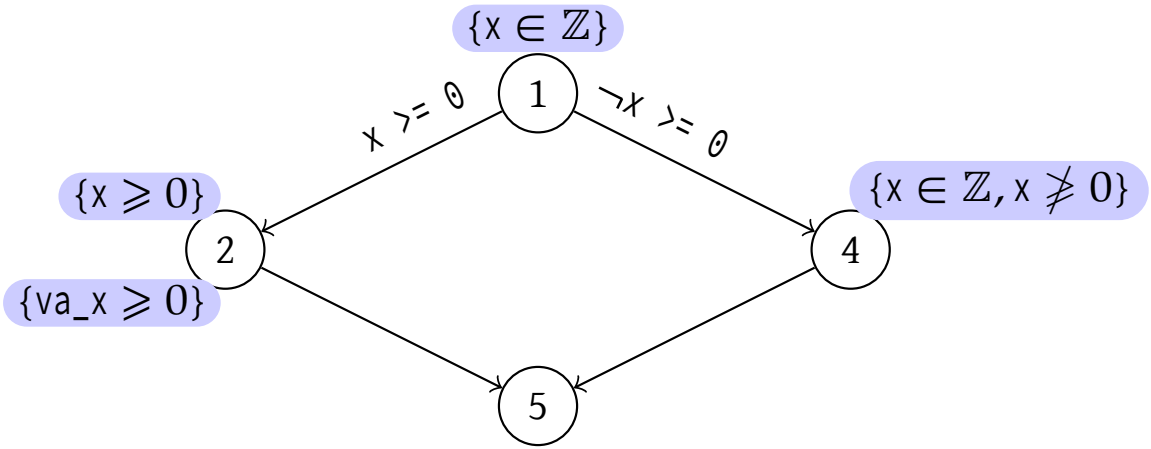
(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



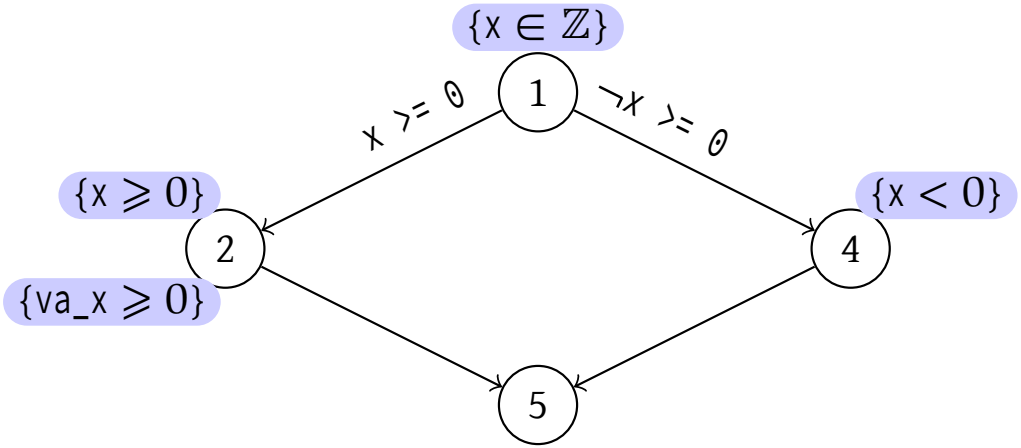
(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



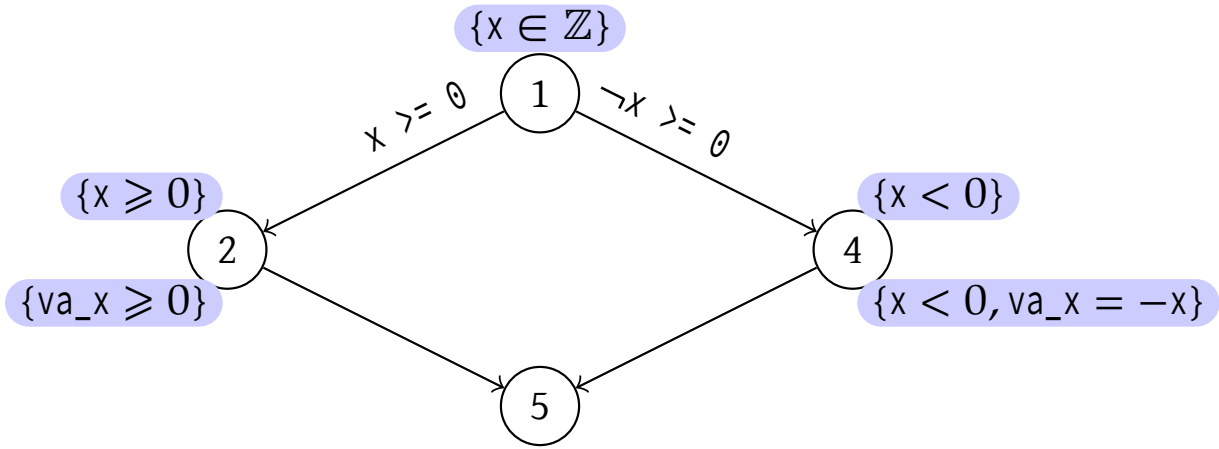
(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



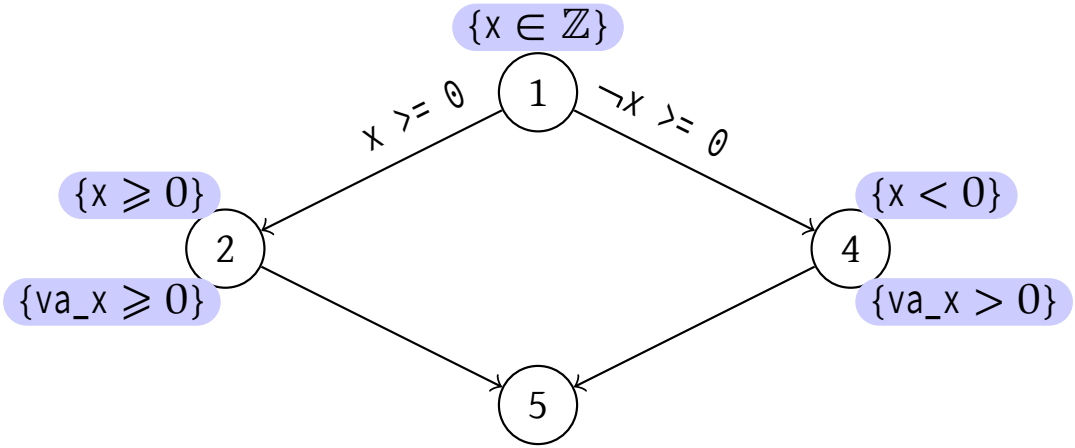
(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



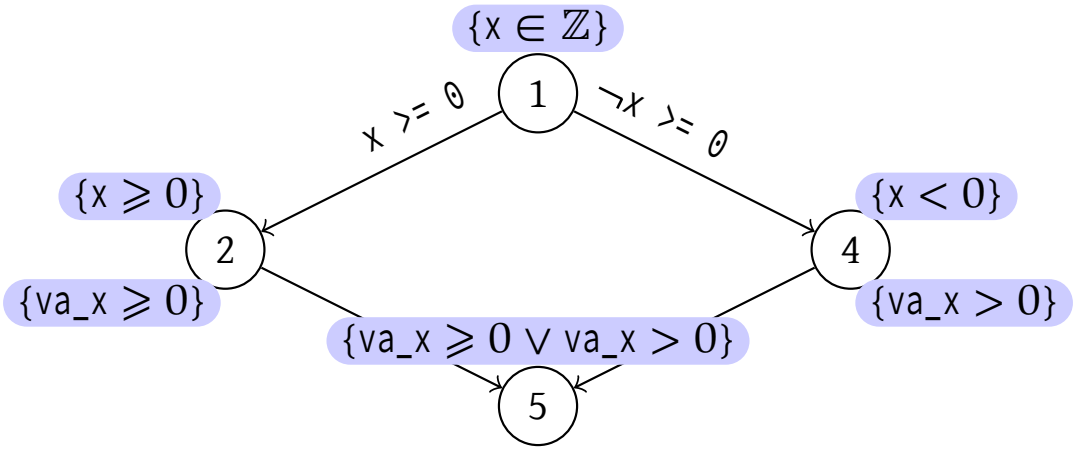
(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



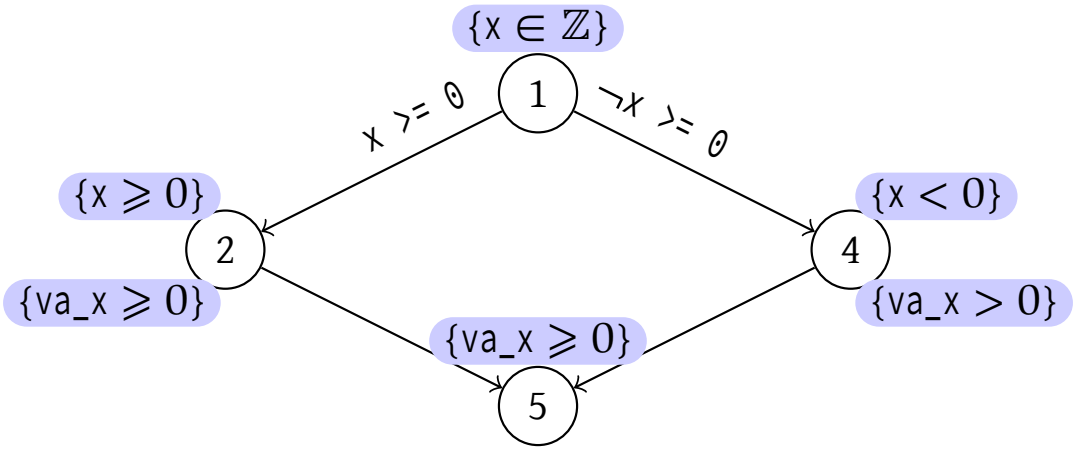
(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



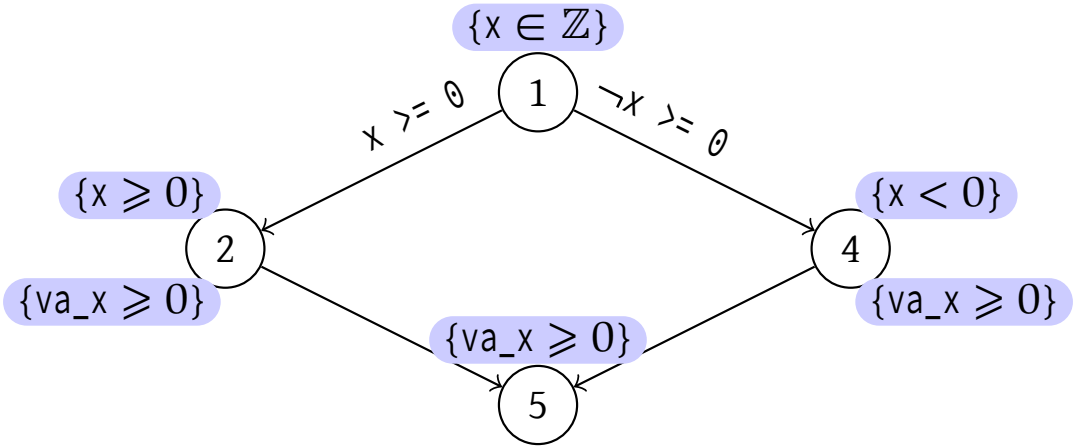
(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



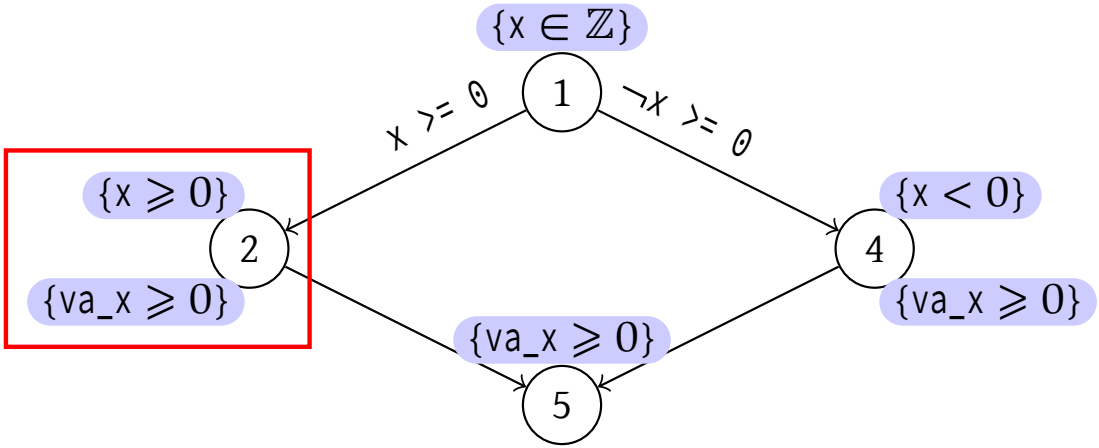
(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



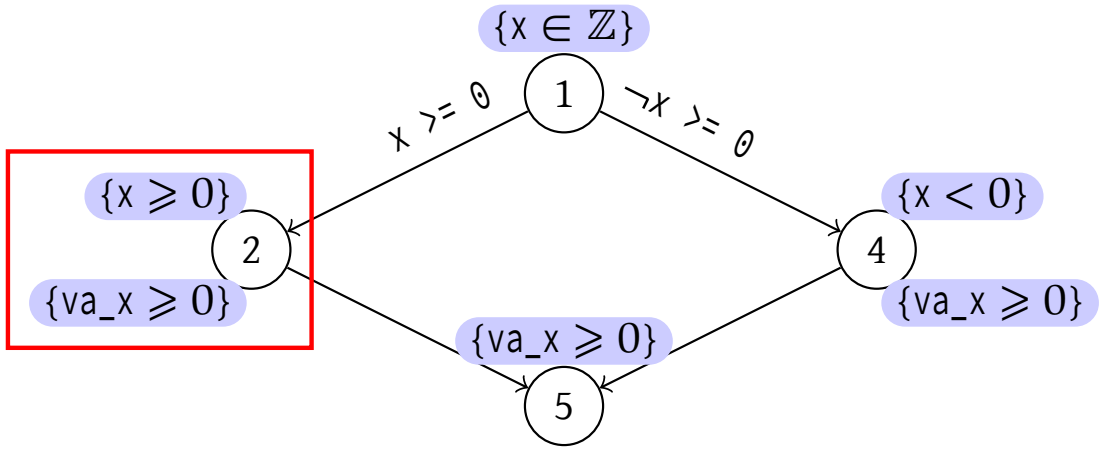
(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```

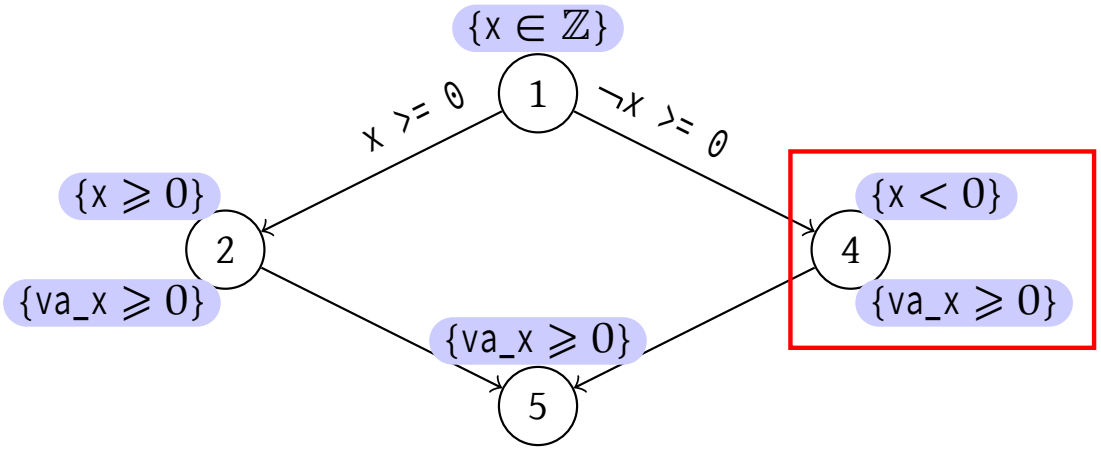


Triplets de Hoare

$\{x \geqslant 0\}$ va_x = x $\{va\_x \geqslant 0\}$

(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```

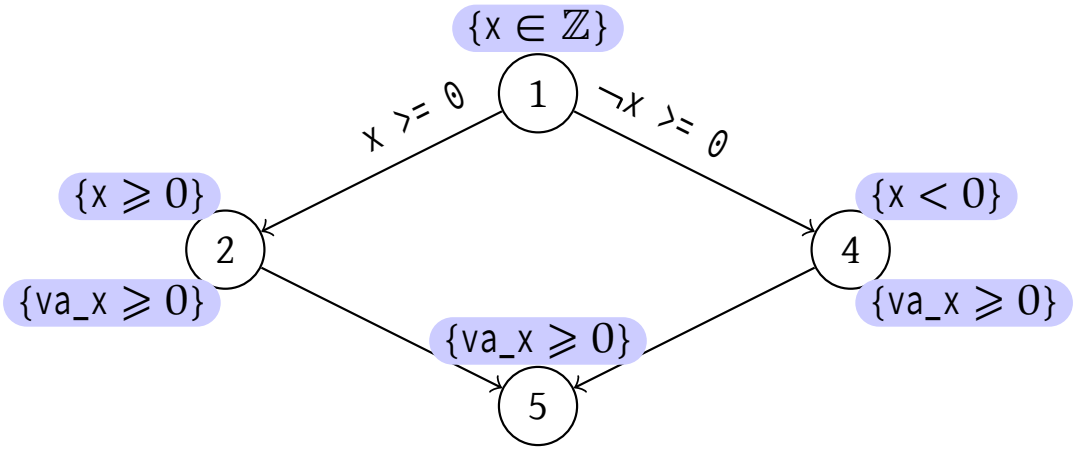


Triplets de Hoare

$\{x \geqslant 0\}$ `va_x = x` $\{va\_x \geqslant 0\}$ $\{x < 0\}$ `va_x = -x` $\{va\_x \geqslant 0\}$

(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



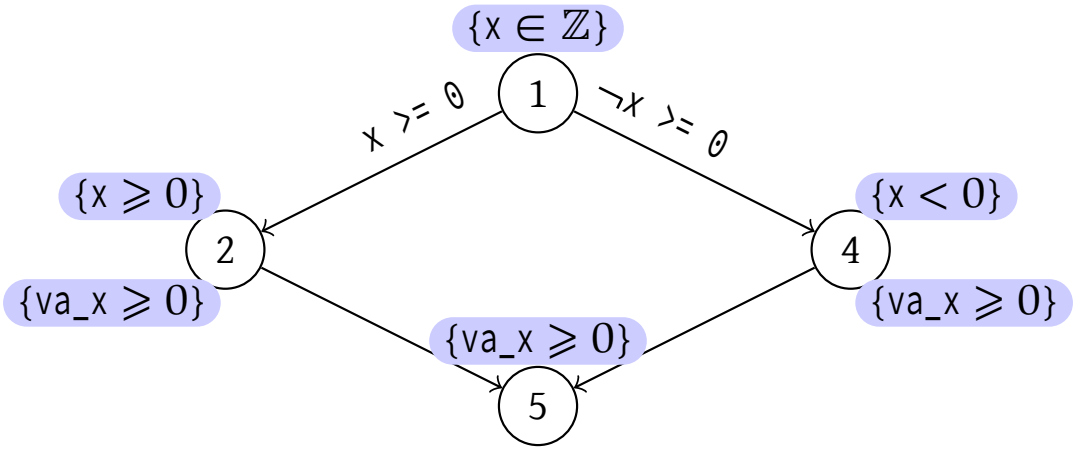
Triplets de Hoare

On observe :

$\{x \geqslant 0\}$ `va_x = x` $\{va\_x \geqslant 0\}$ et $\{x < 0\}$ `va_x = -x` $\{va\_x \geqslant 0\}$.

(16/1)

```
1  if x >= 0 :
2    va_x = x
3  else :
4    va_x = -x
5  # etc.
```



Triplets de Hoare

On observe :

$\{x \geqslant 0\}$ `va_x = x` $\{va\_x \geqslant 0\}$ et $\{x < 0\}$ `va_x = -x` $\{va\_x \geqslant 0\}$.

On en déduit :

$$\{x \in \mathbb{Z}\} \quad \begin{array}{l}\texttt{if x >= 0 :}\\ \quad\texttt{va\_x = x}\\ \texttt{else :}\\ \quad\texttt{va\_x = -x}\end{array} \quad \{va\_x \geqslant 0\}.$$

(16/1)

Règle de déduction pour l'alternative

Ayant prouvé

$$\{P, C\} \quad instr_V \quad \{Q\}$$

et

$$\{P, \neg C\} \quad instr_F \quad \{Q\},$$

on peut déduire

$$\left\{P\right\} \quad \begin{array}{l}\texttt{if } C \texttt{ :} \\ \quad instr_V \\ \texttt{else :} \\ \quad instr_F\end{array} \quad \left\{Q\right\}.$$

(17/1)

Formulation en termes de but

Pour prouver

$$\left\{P\right\} \quad \begin{array}{l}\texttt{if } C\texttt{ :}\\ \quad instr_V\\ \texttt{else :}\\ \quad instr_F\end{array} \quad \left\{Q\right\},$$

on prouve

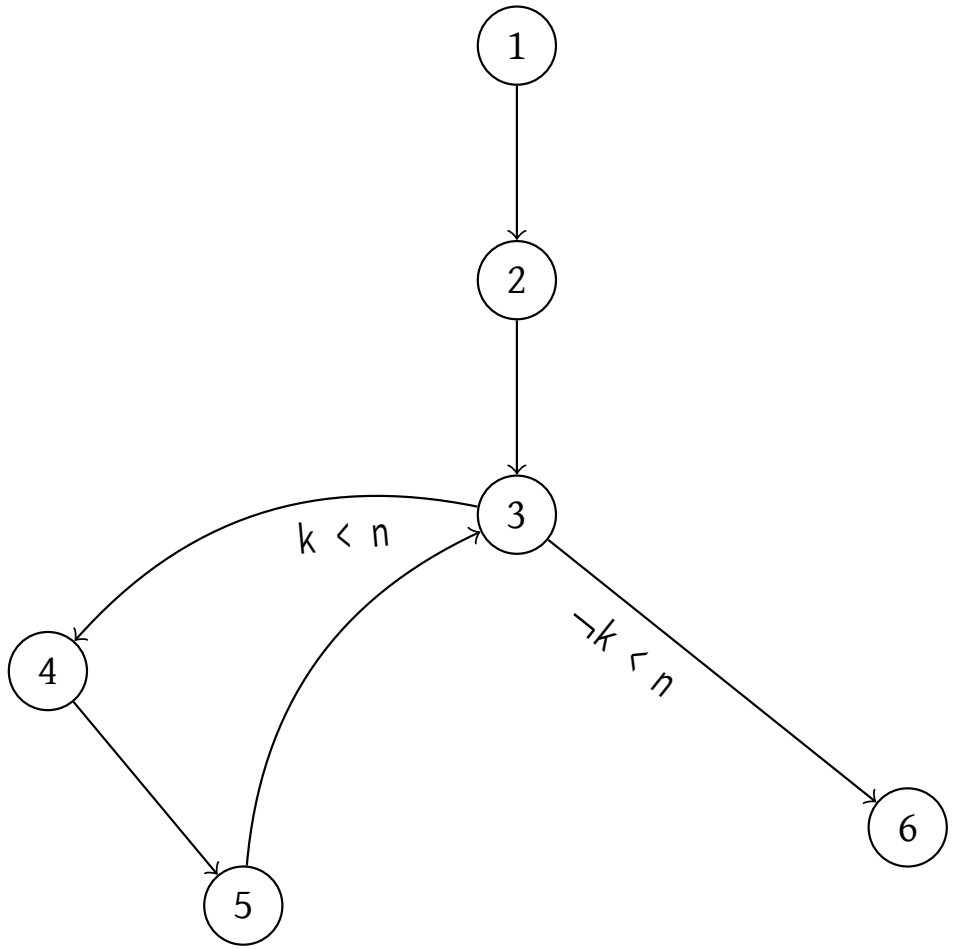
$$\{P, C\} \quad instr_V \quad \{Q\}$$

et

$$\{P, \neg C\} \quad instr_F \quad \{Q\}.$$

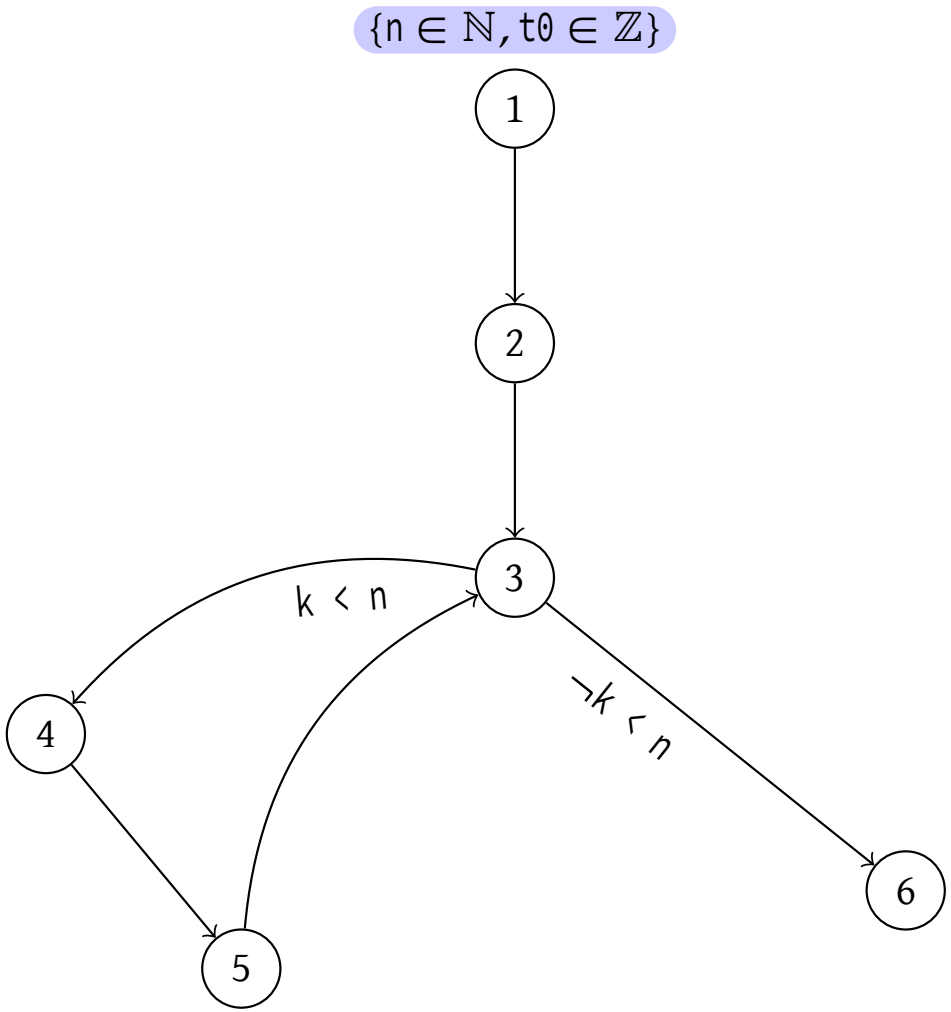
(18/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



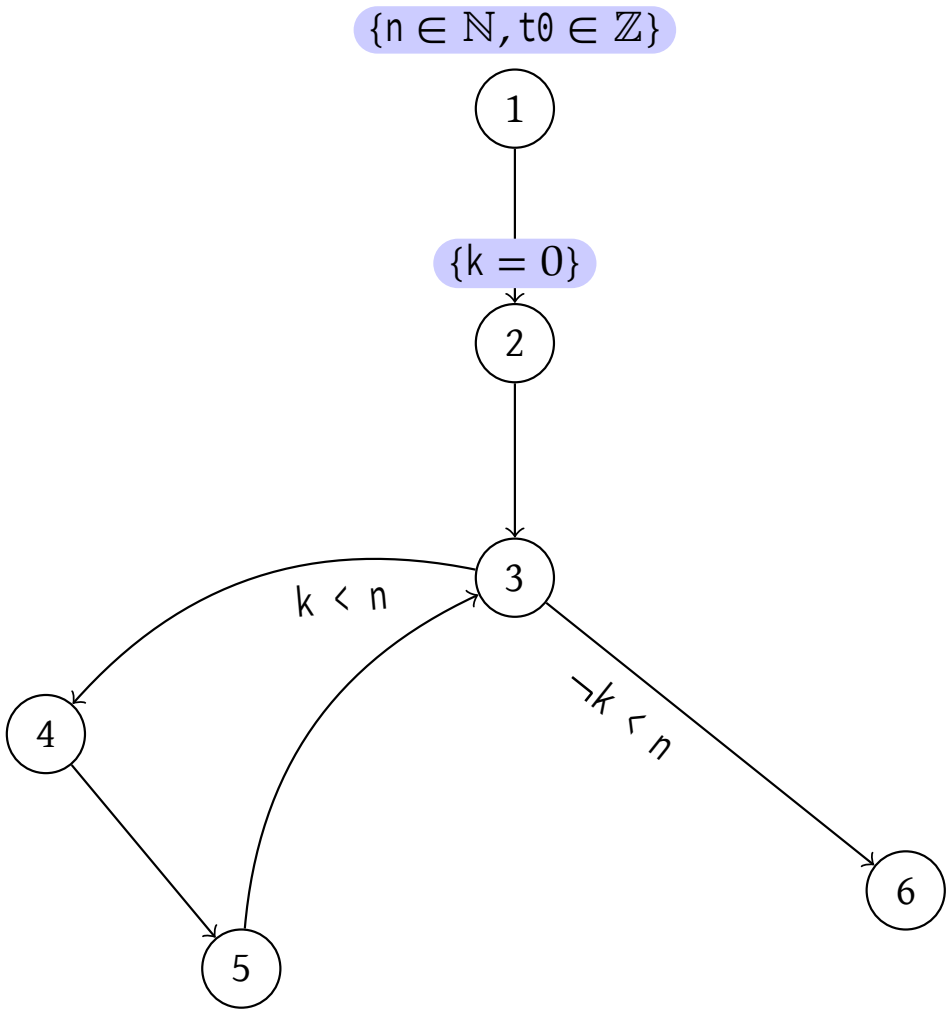
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



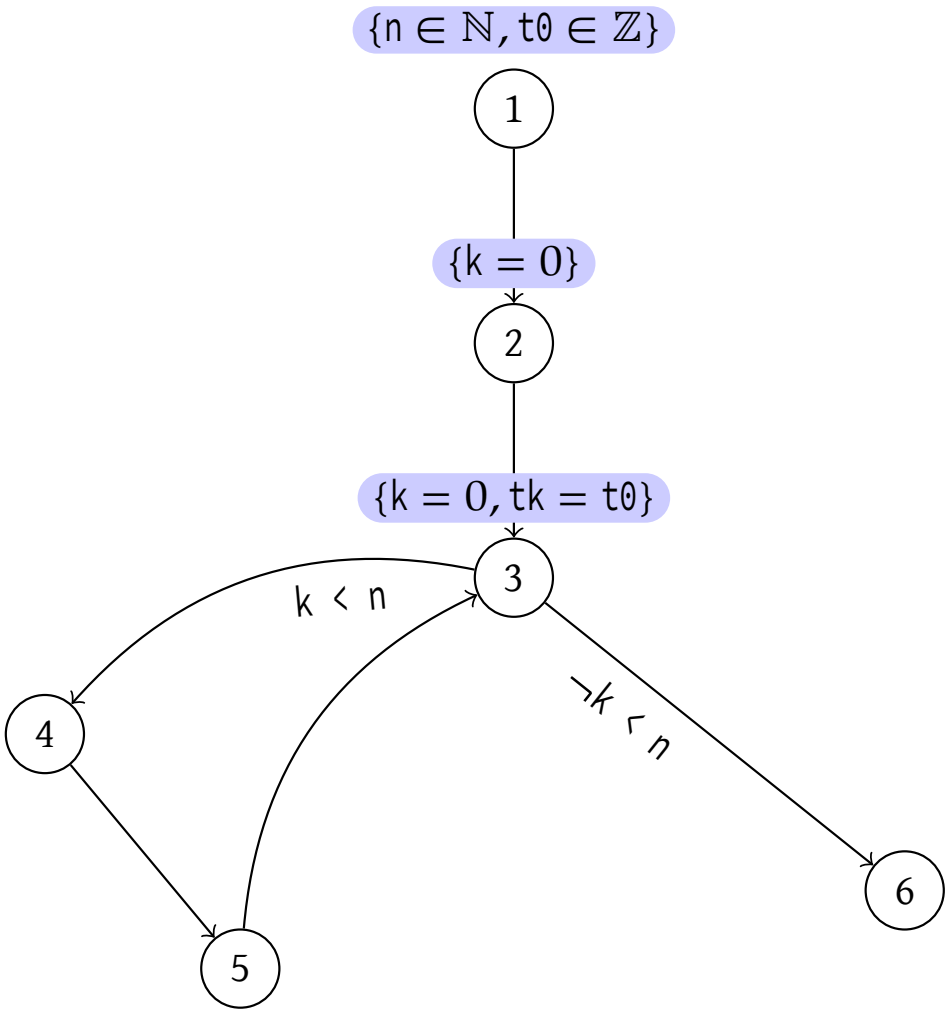
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



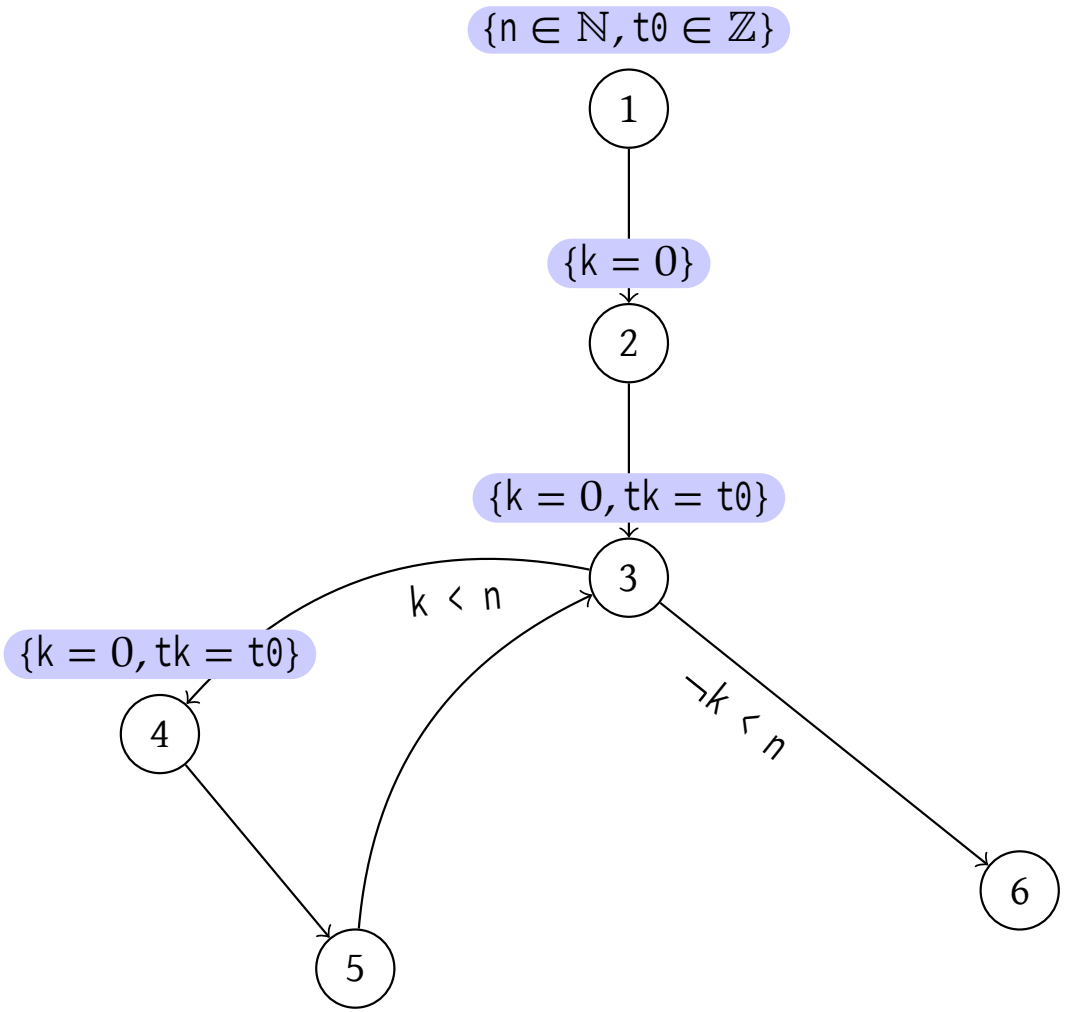
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



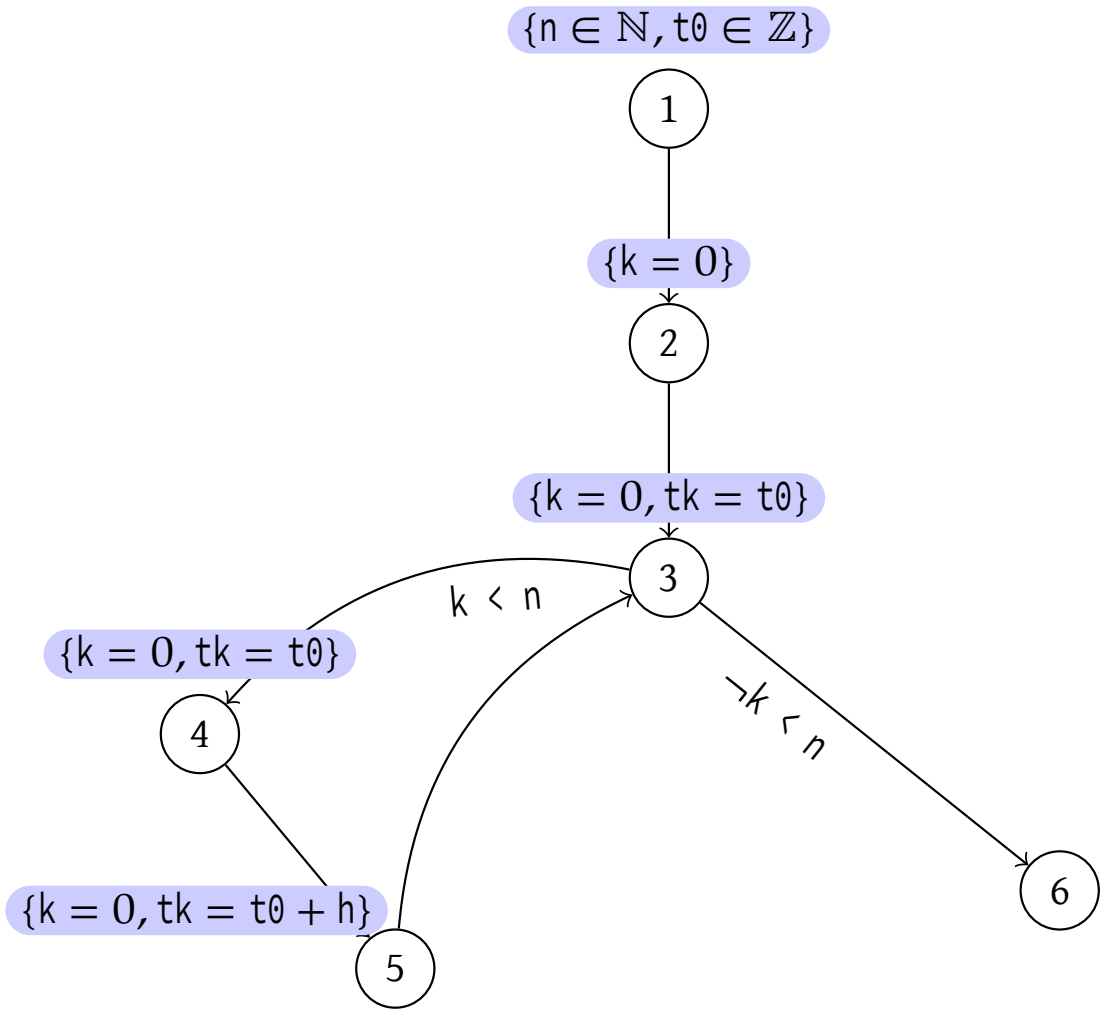
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



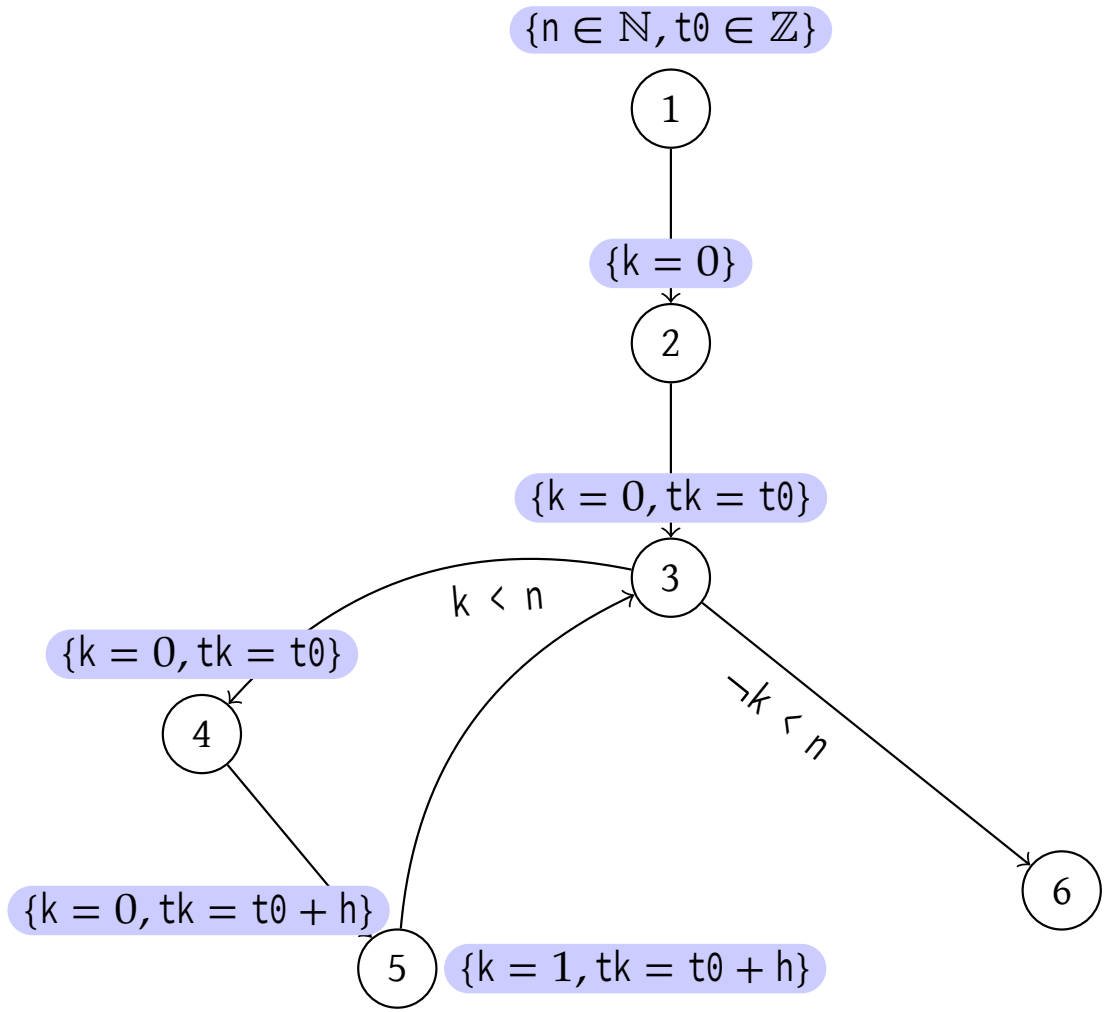
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



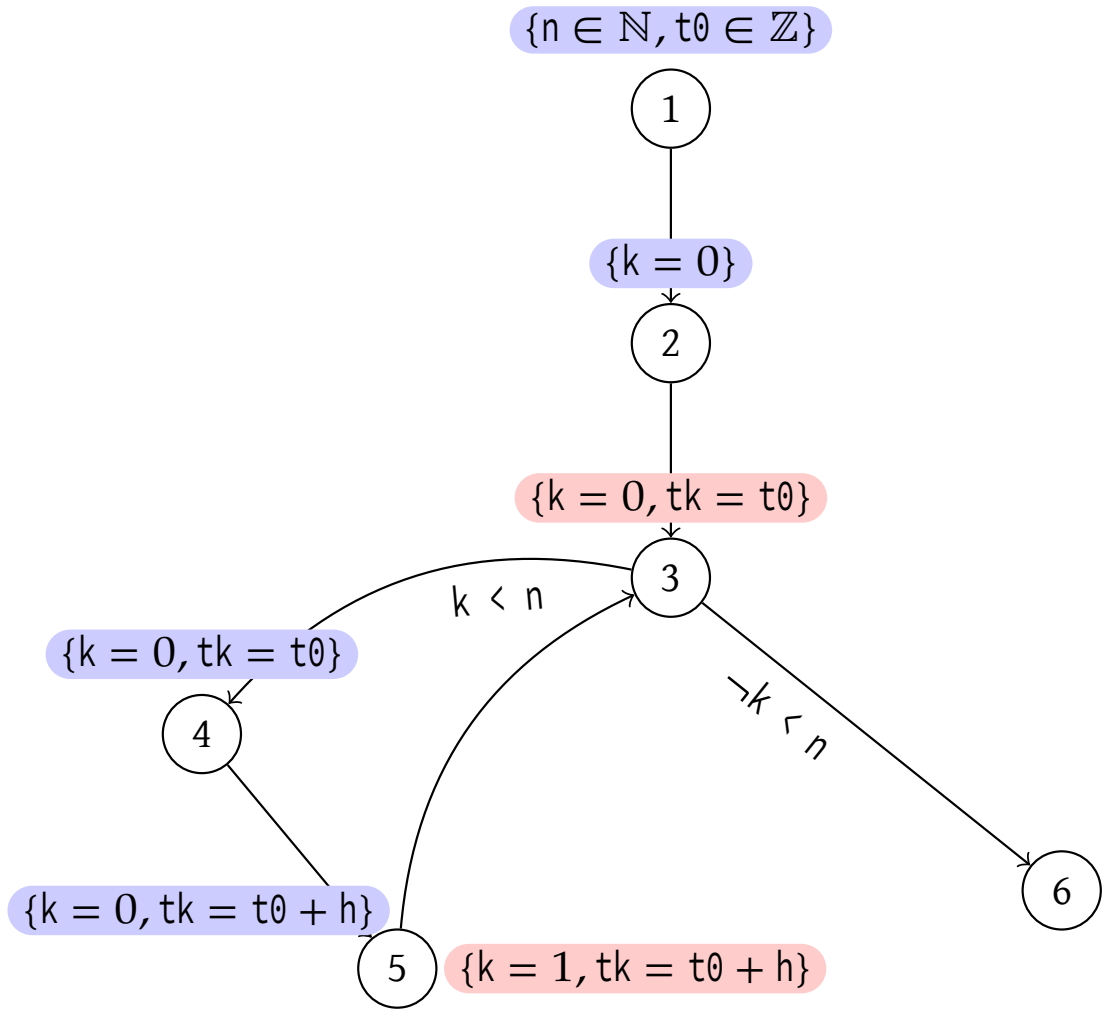
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



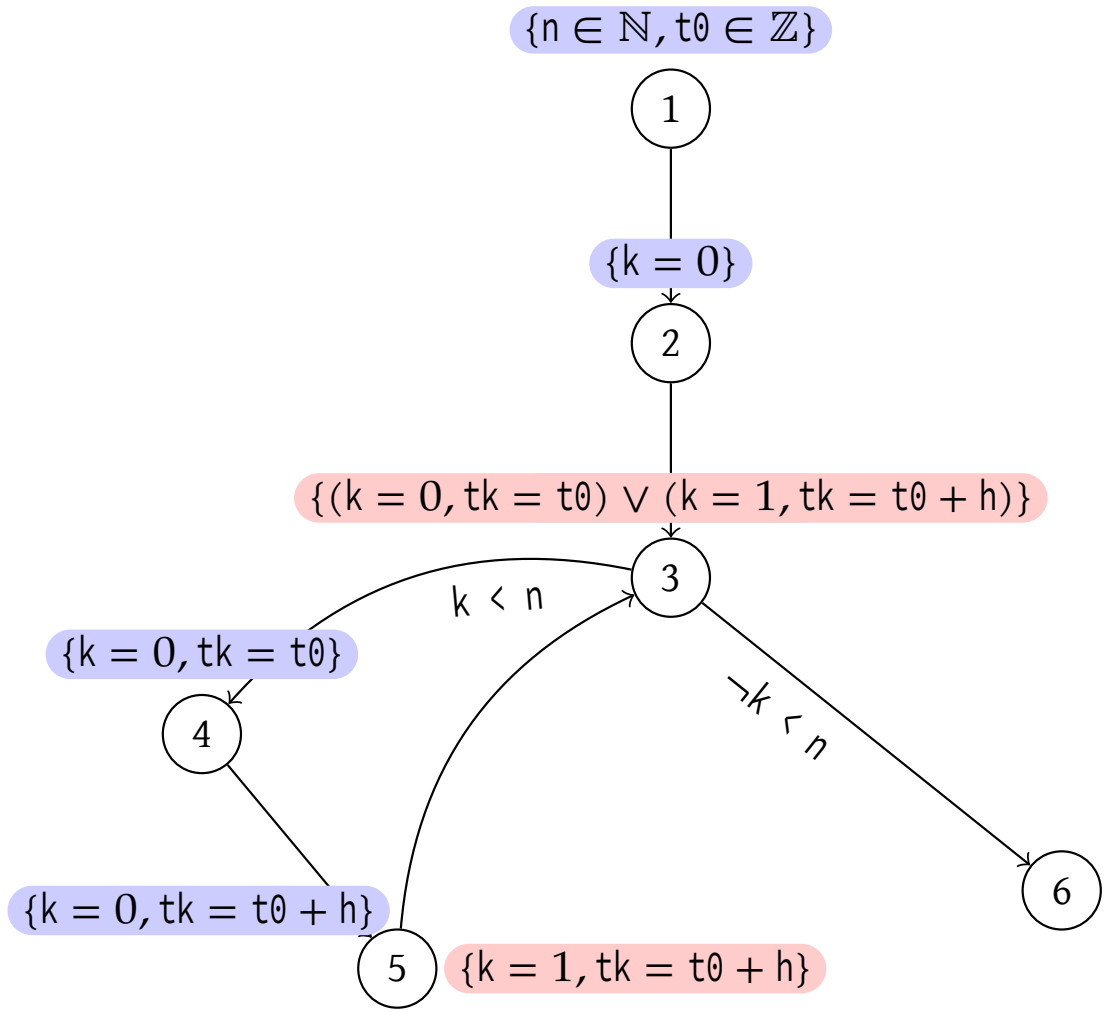
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



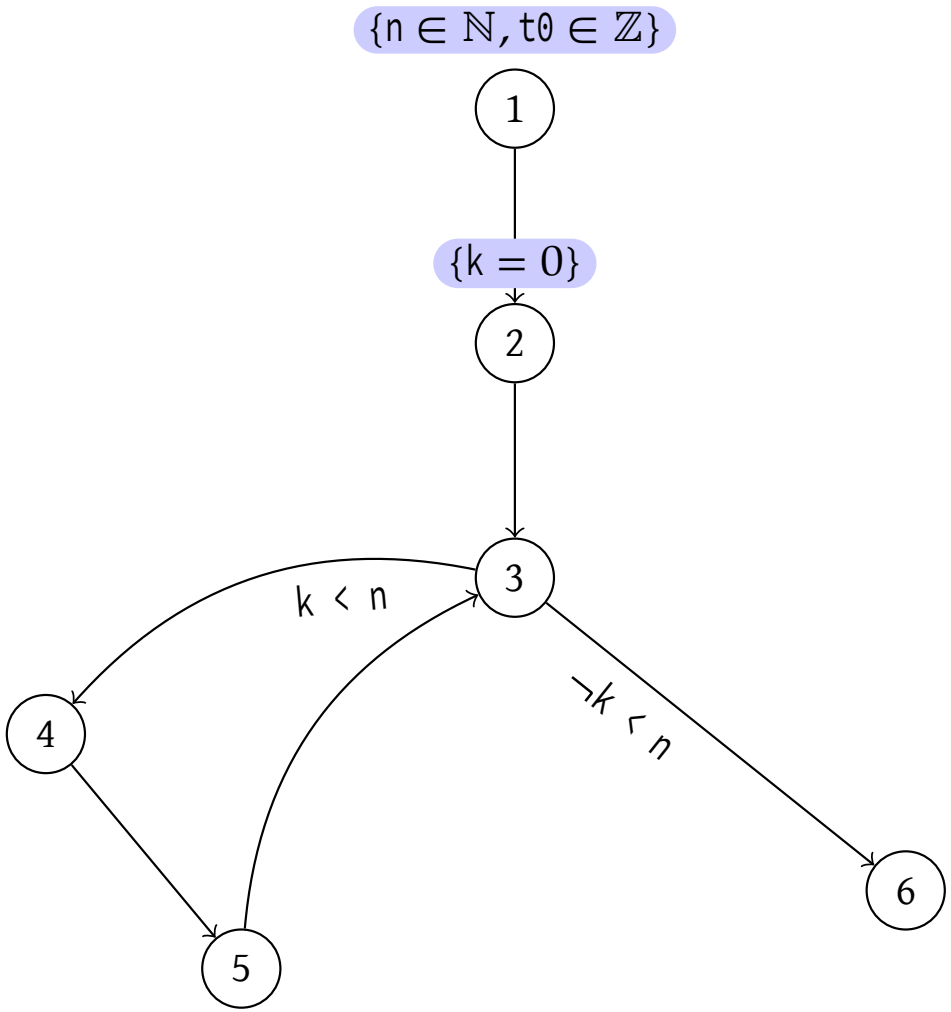
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



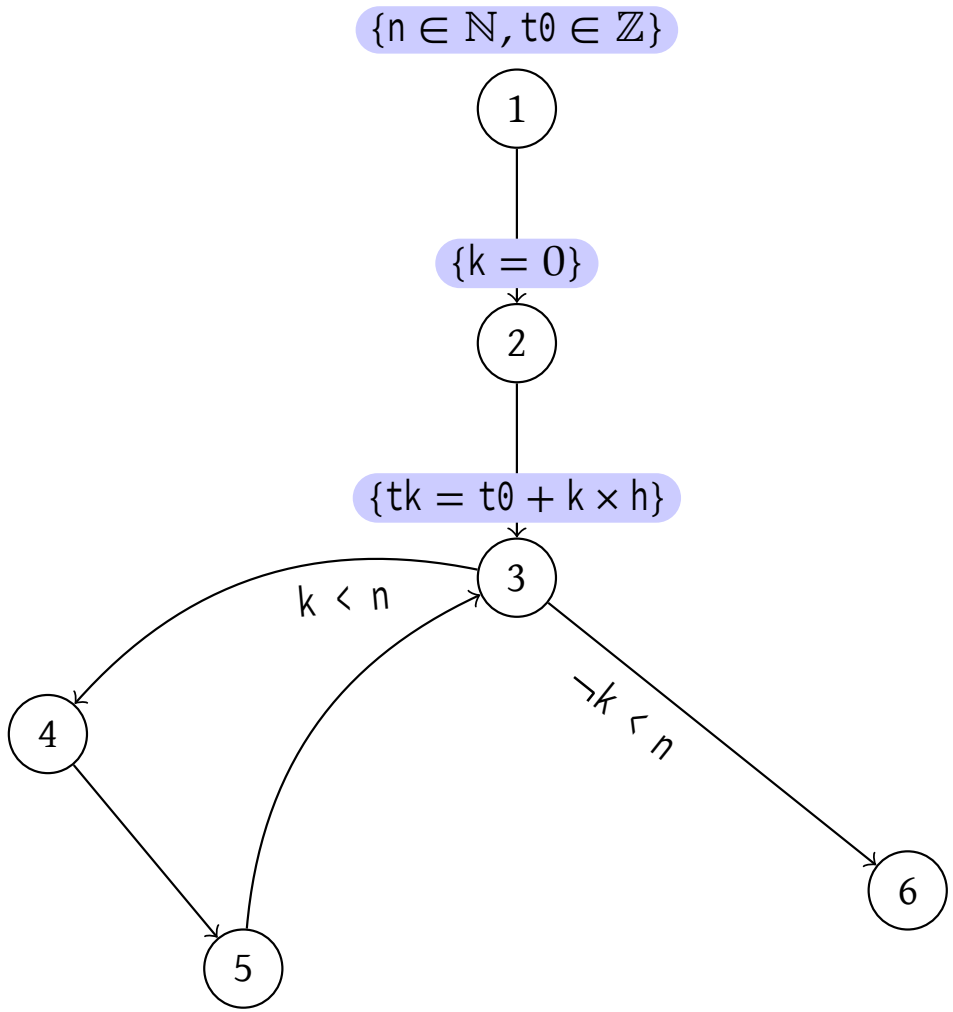
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



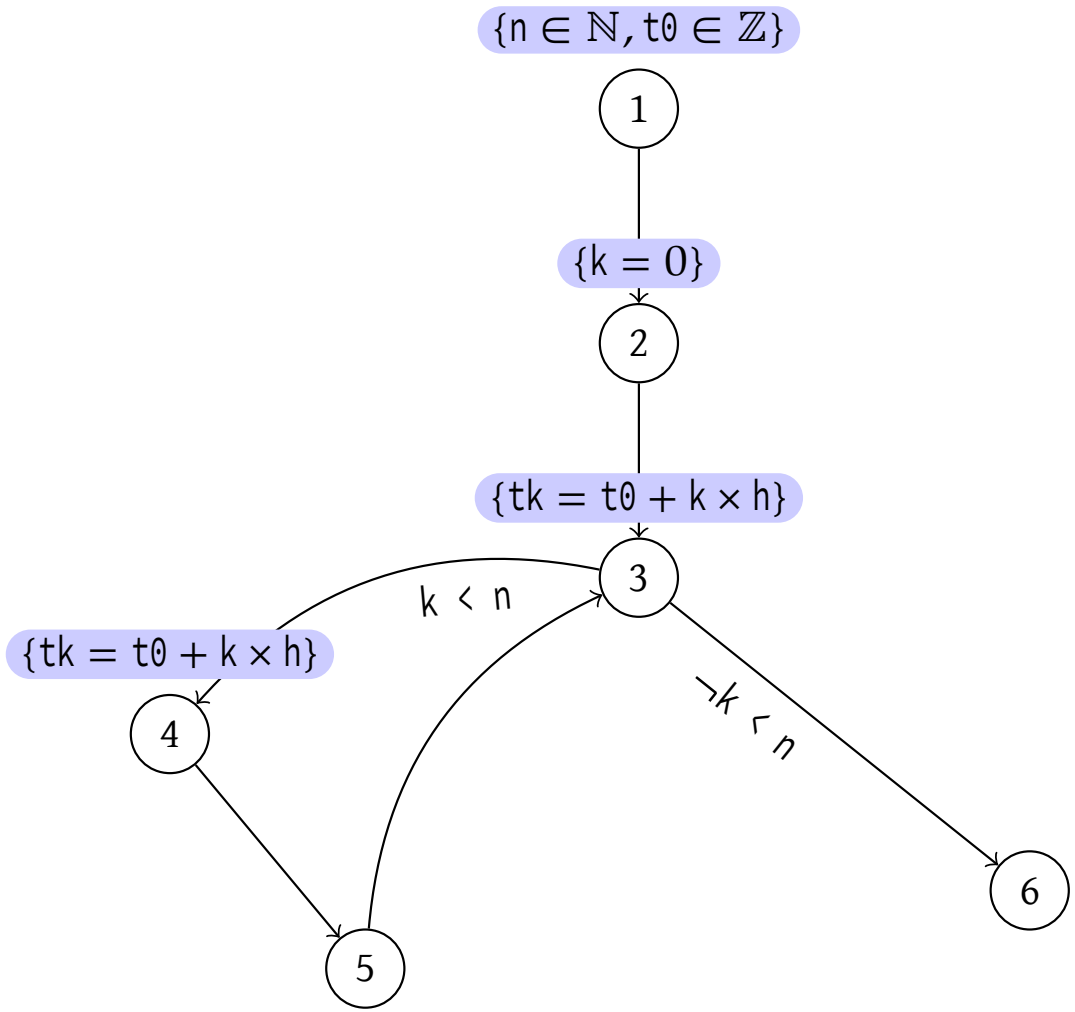
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



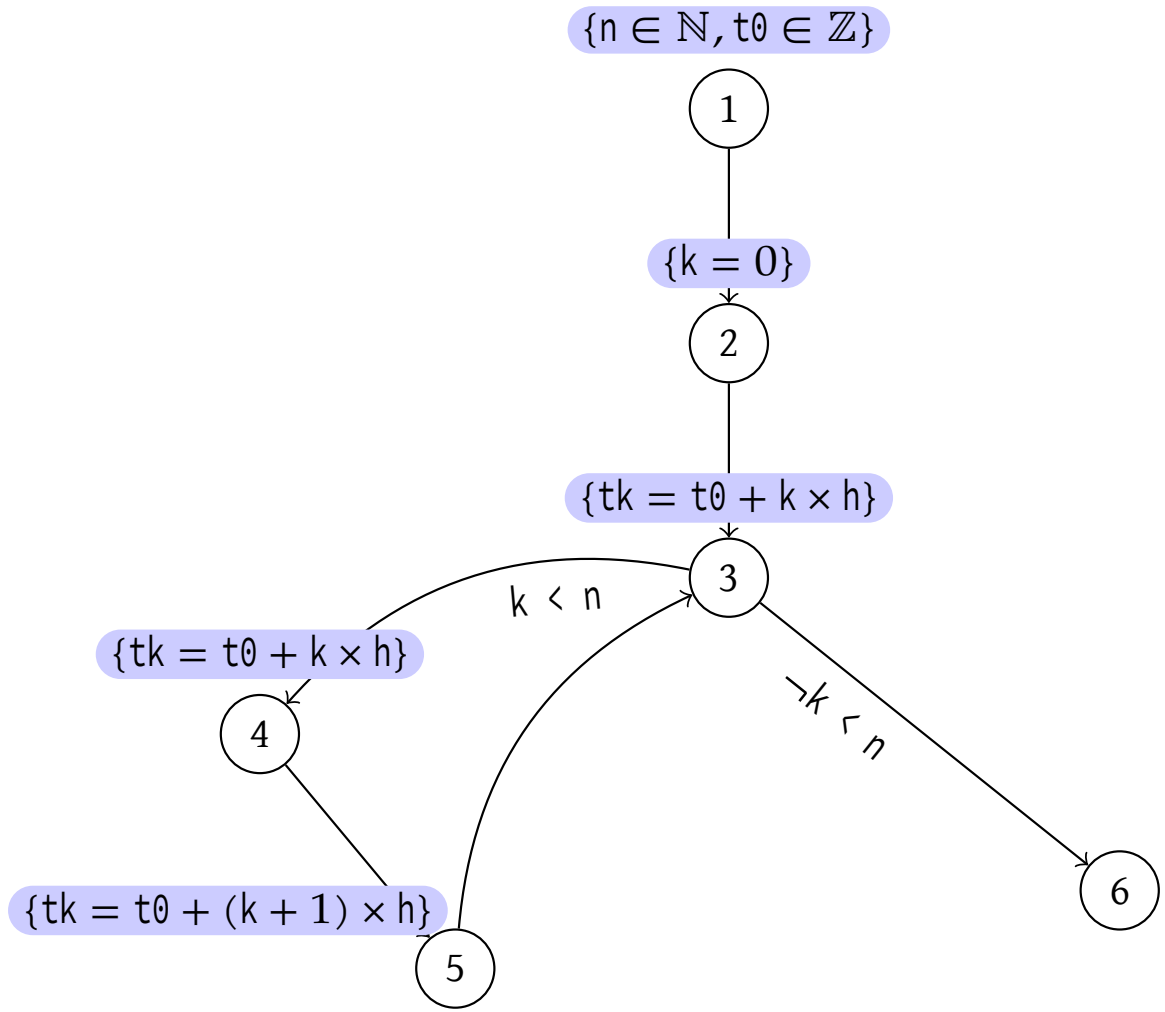
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



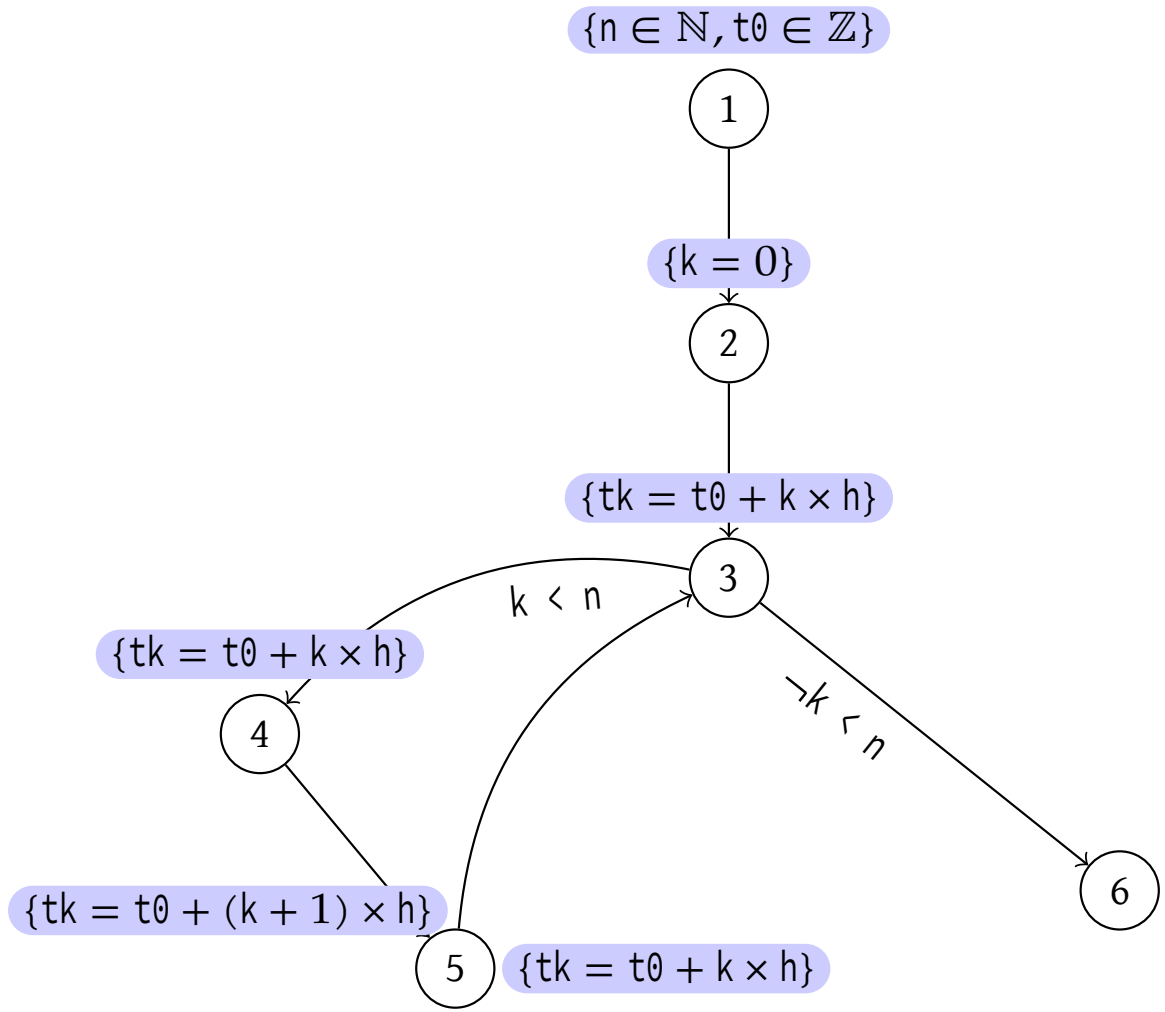
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



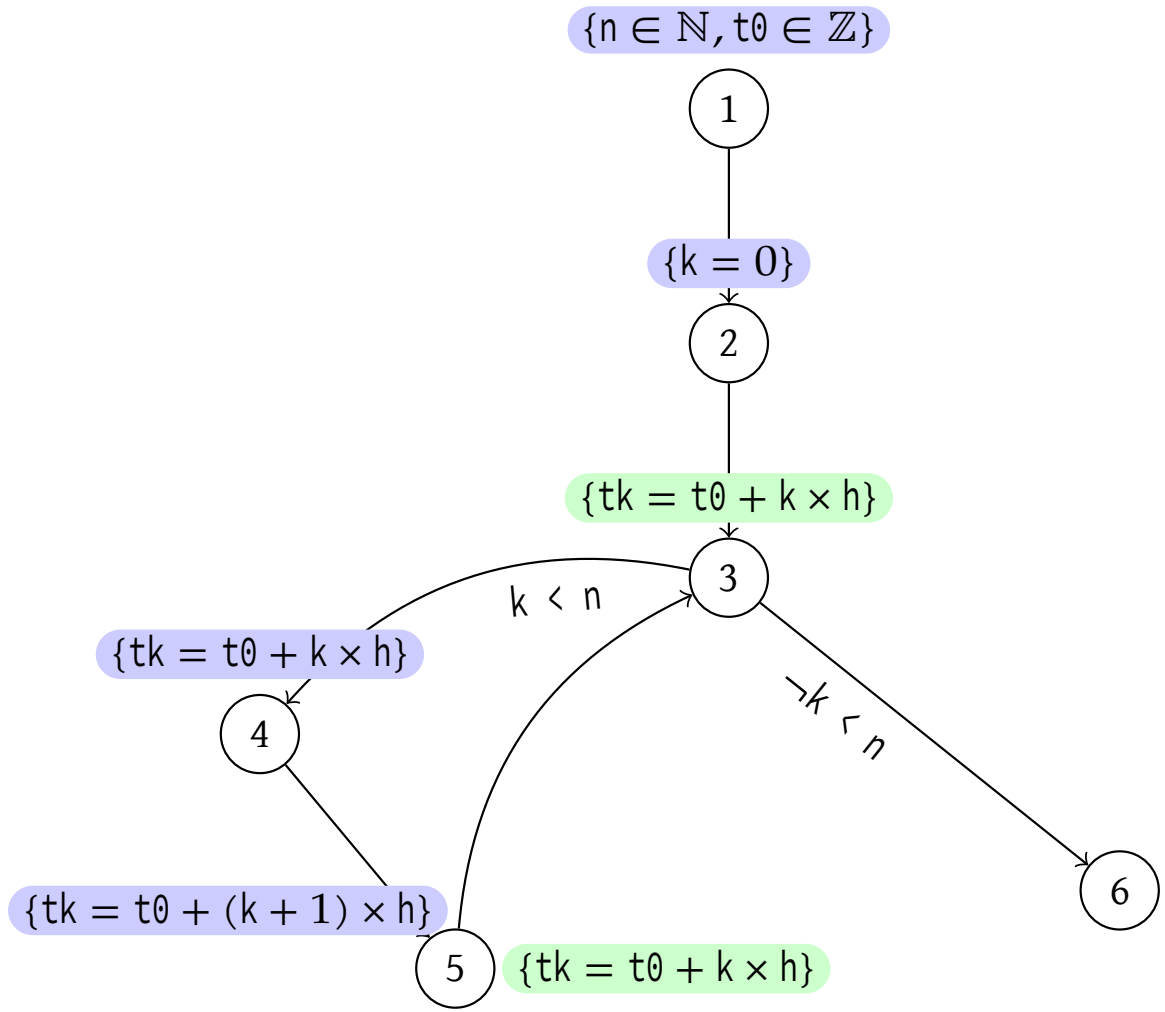
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



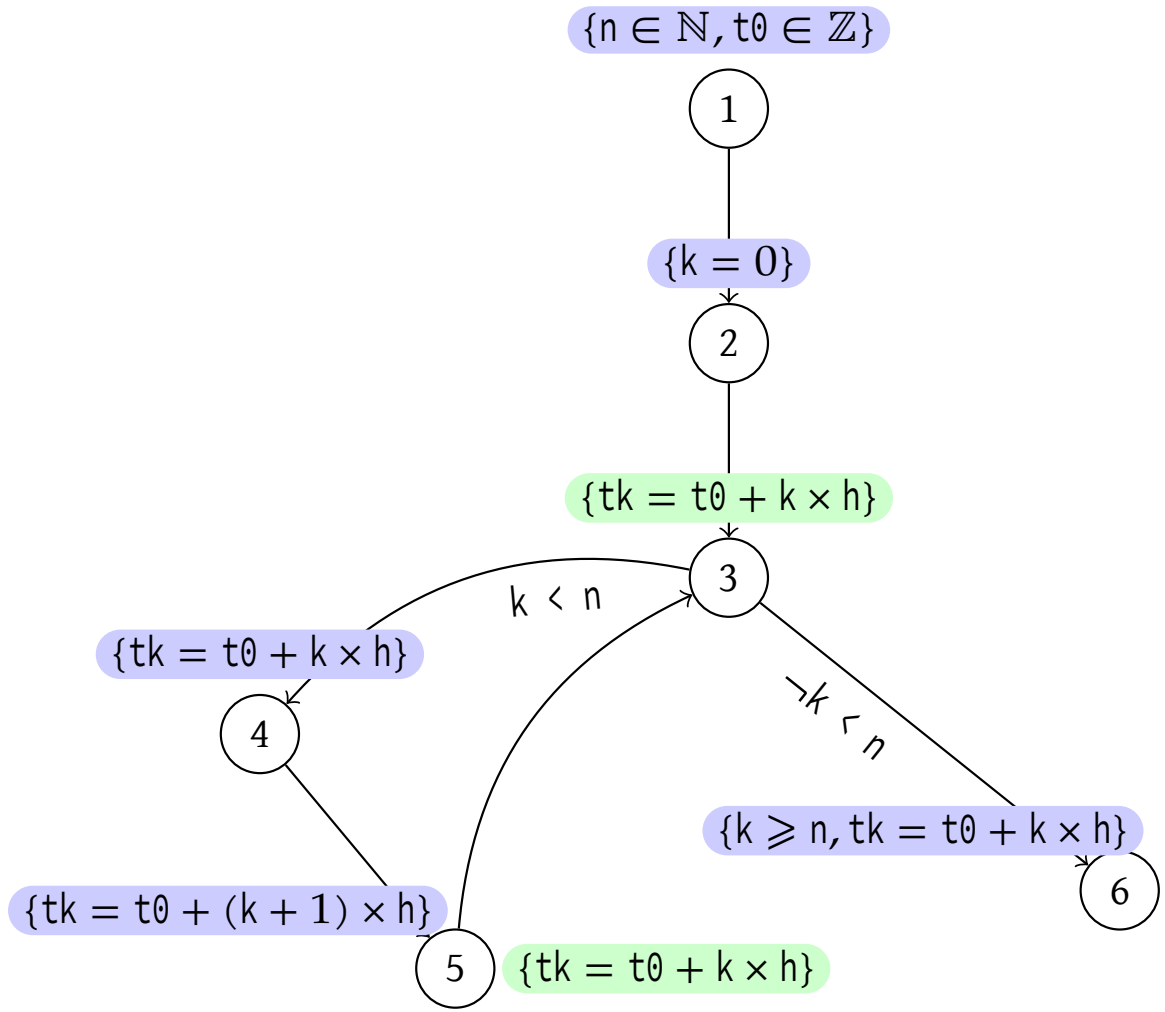
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



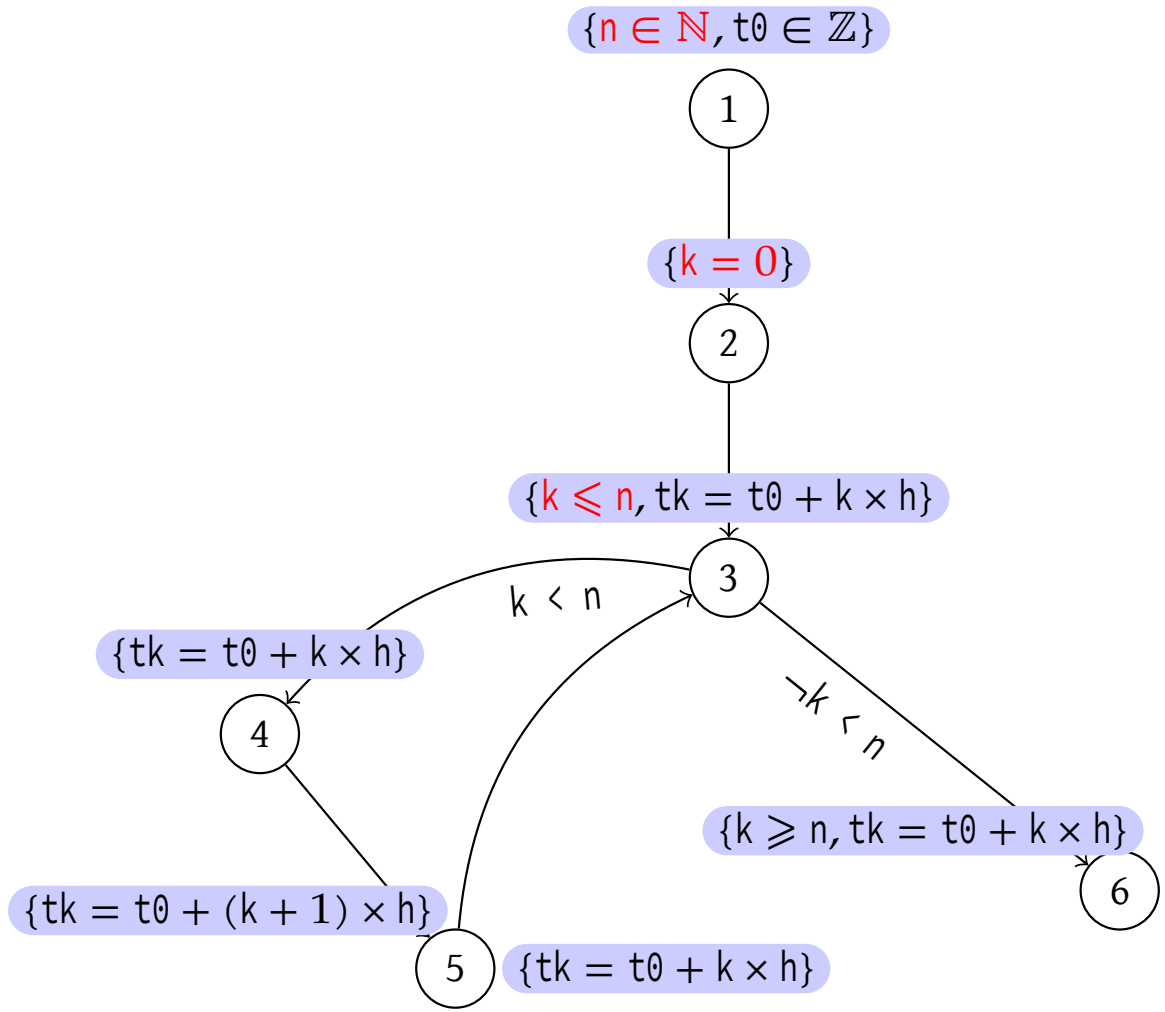
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



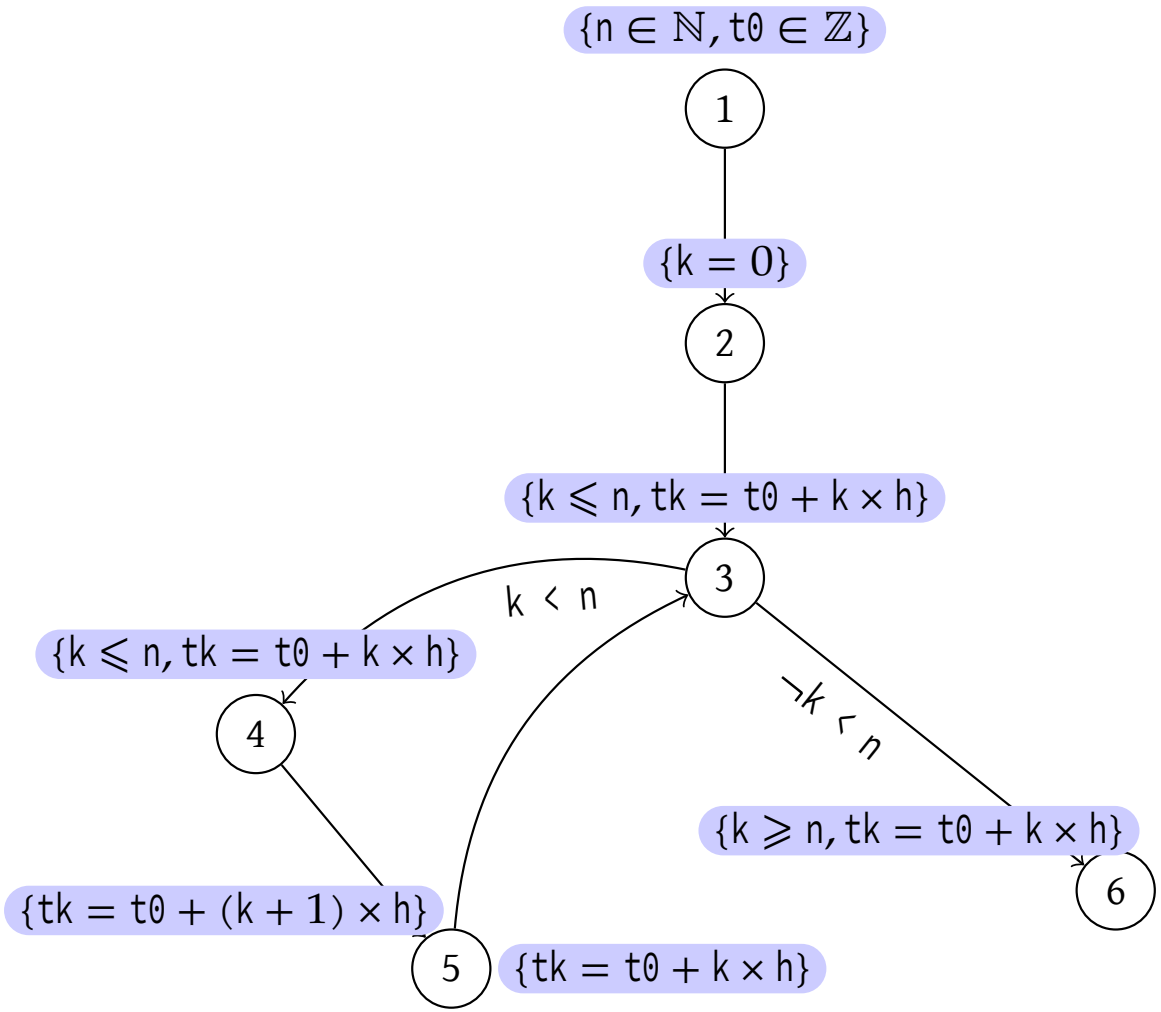
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



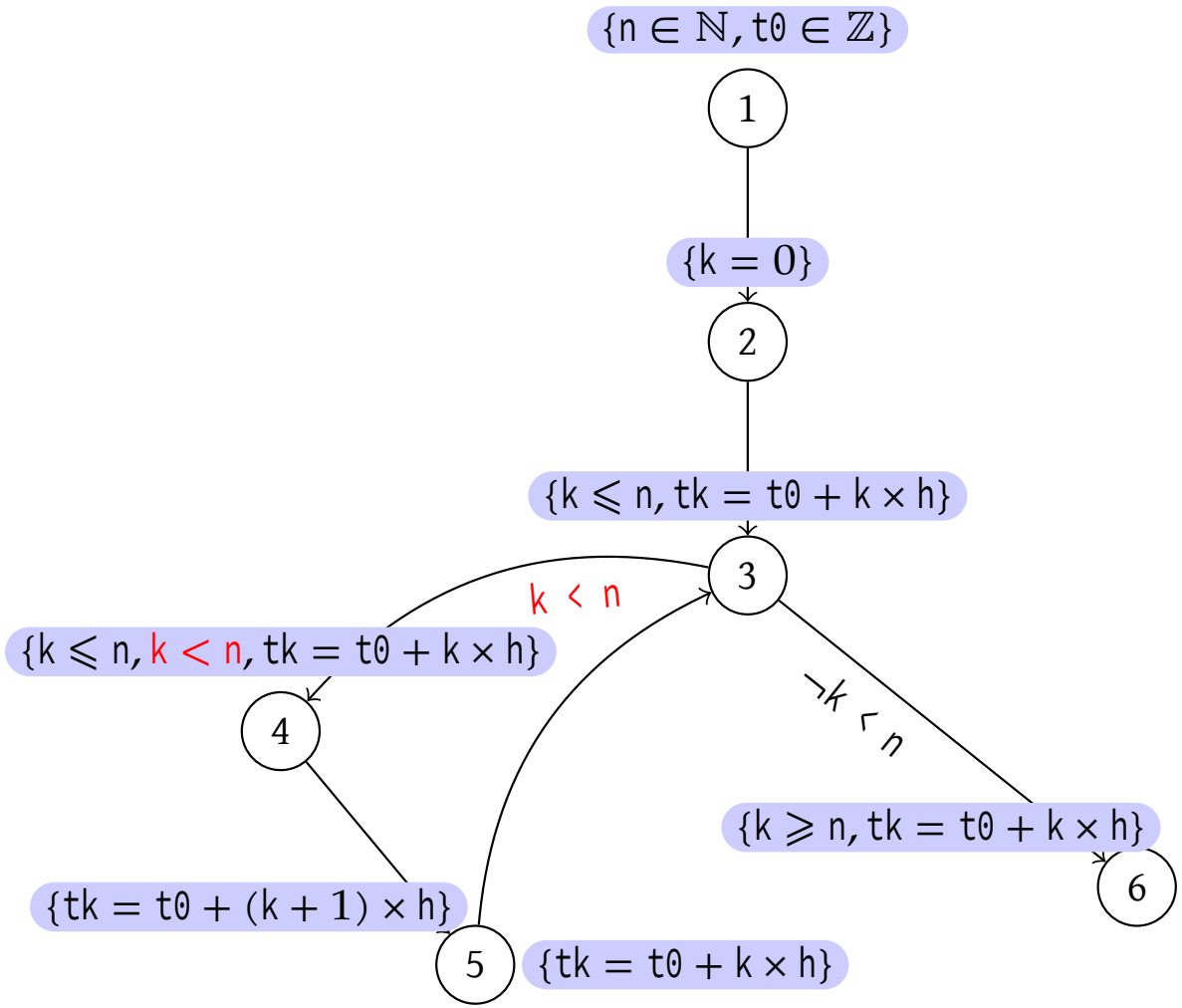
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



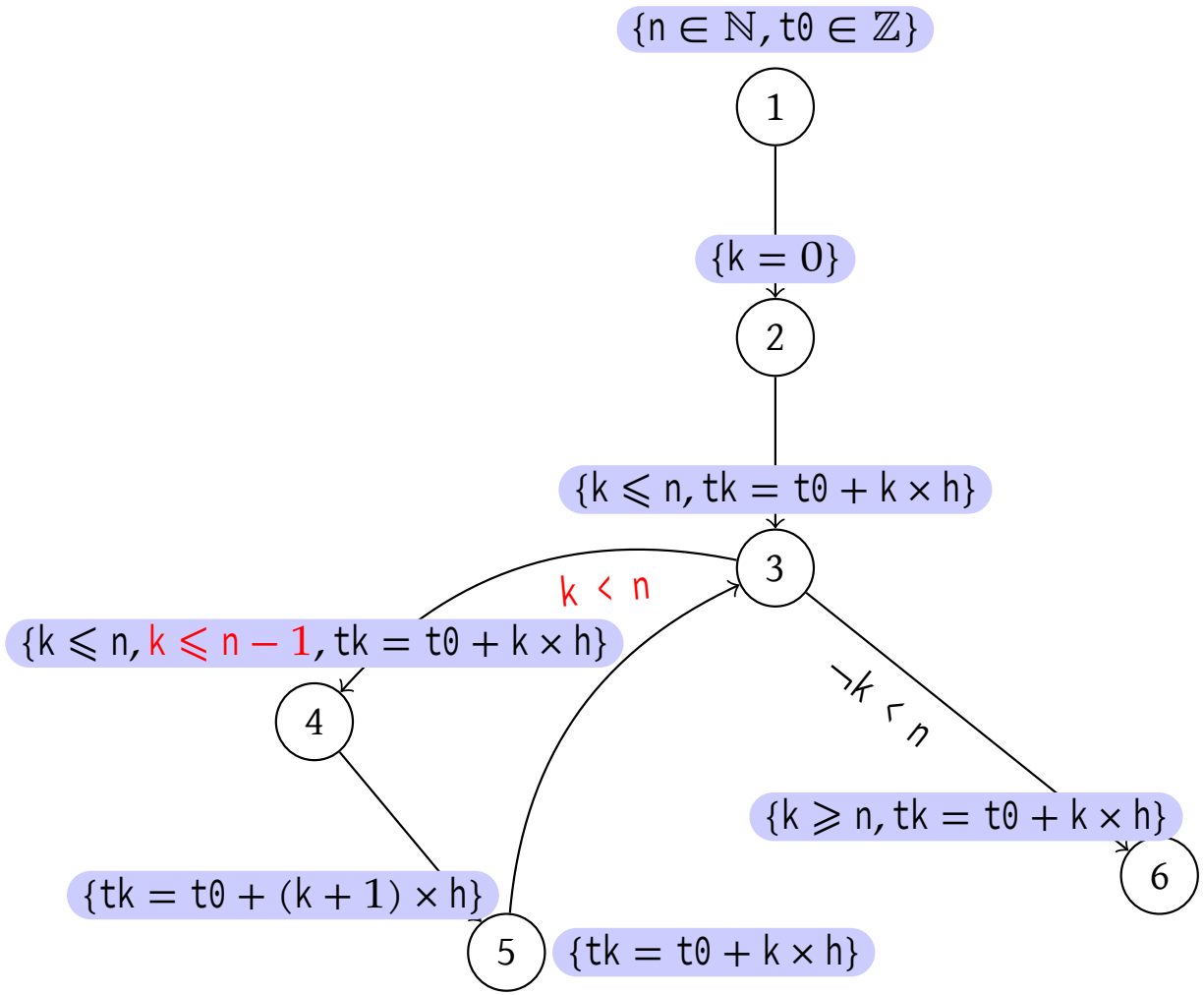
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



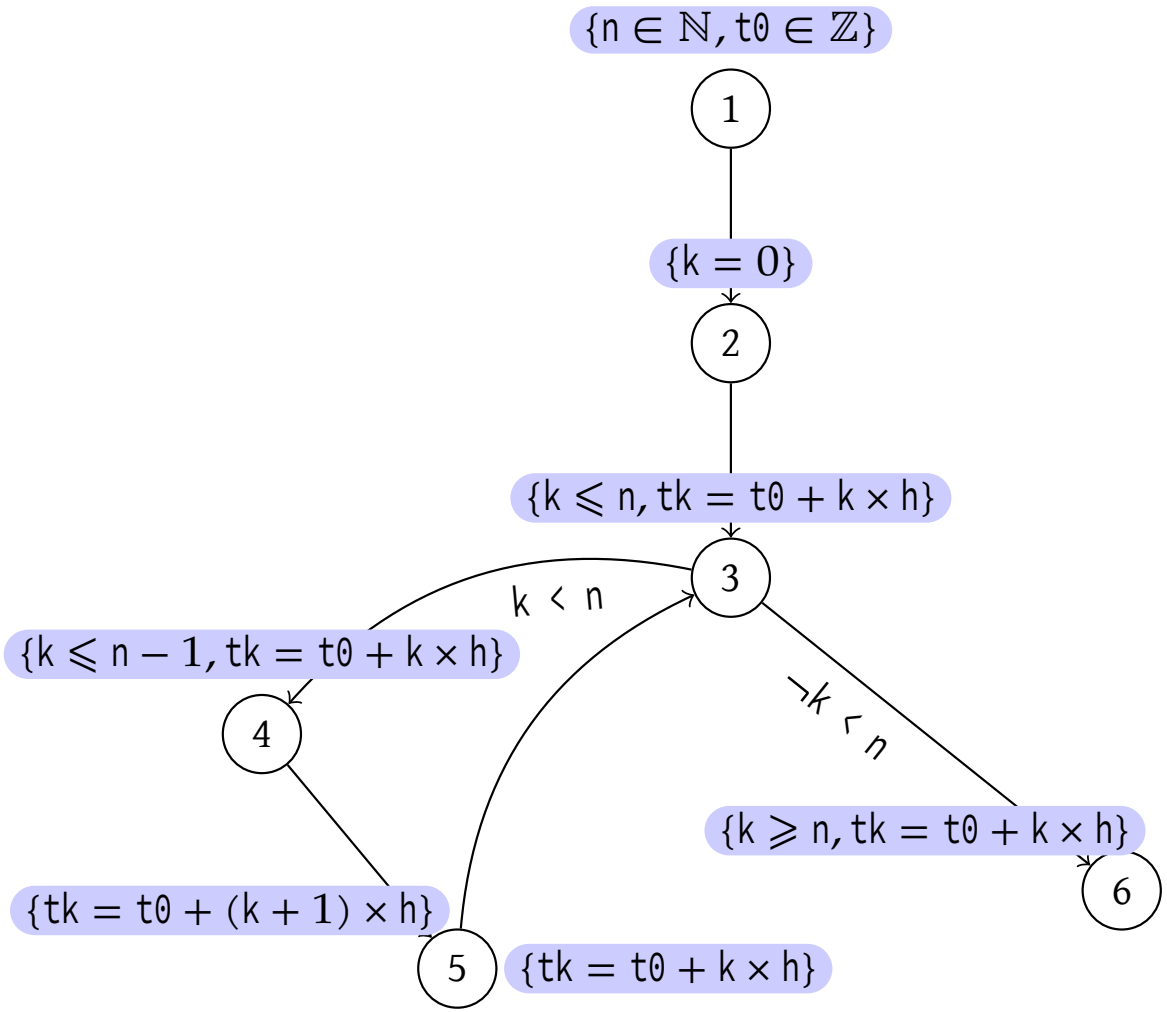
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



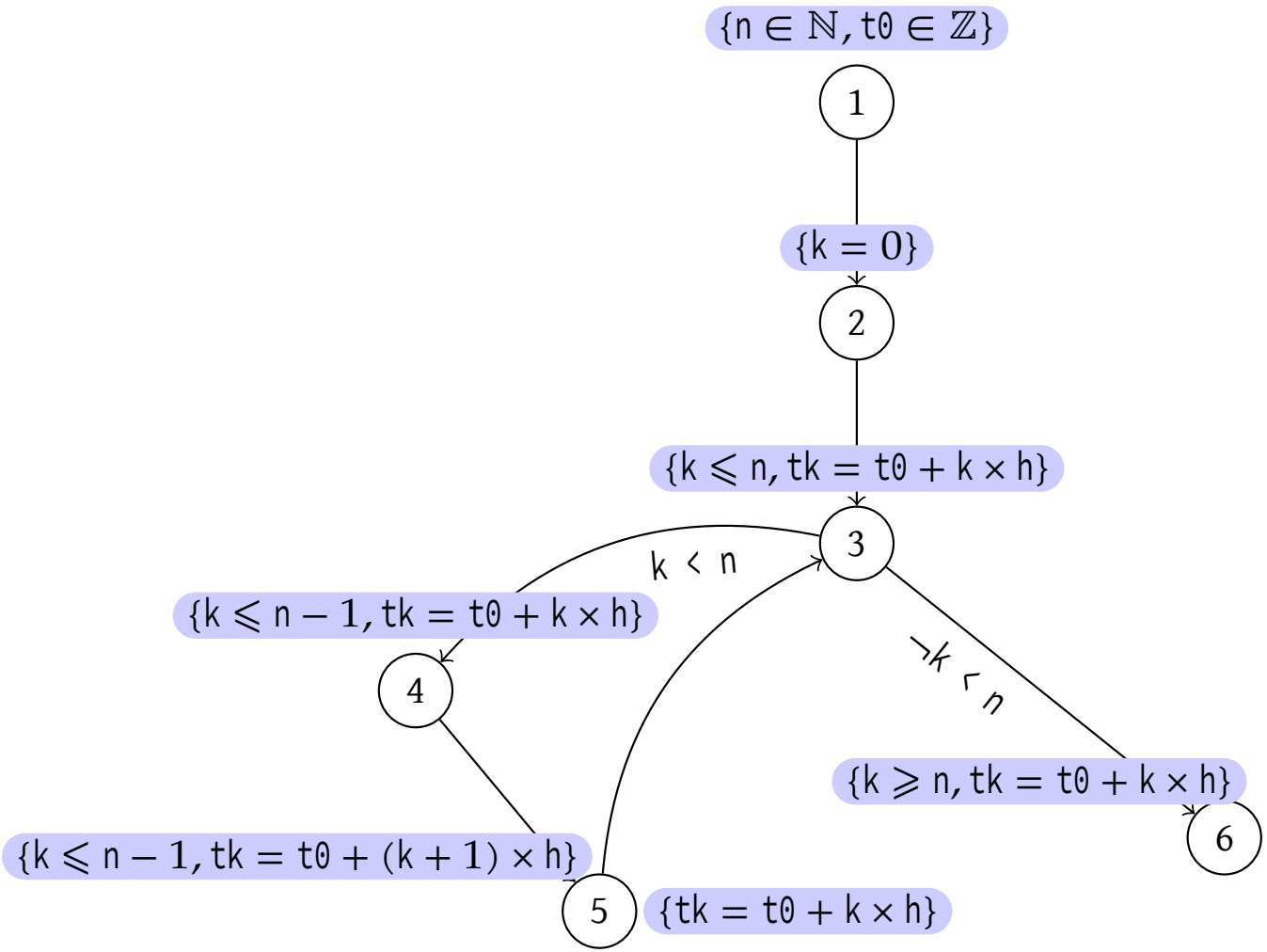
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



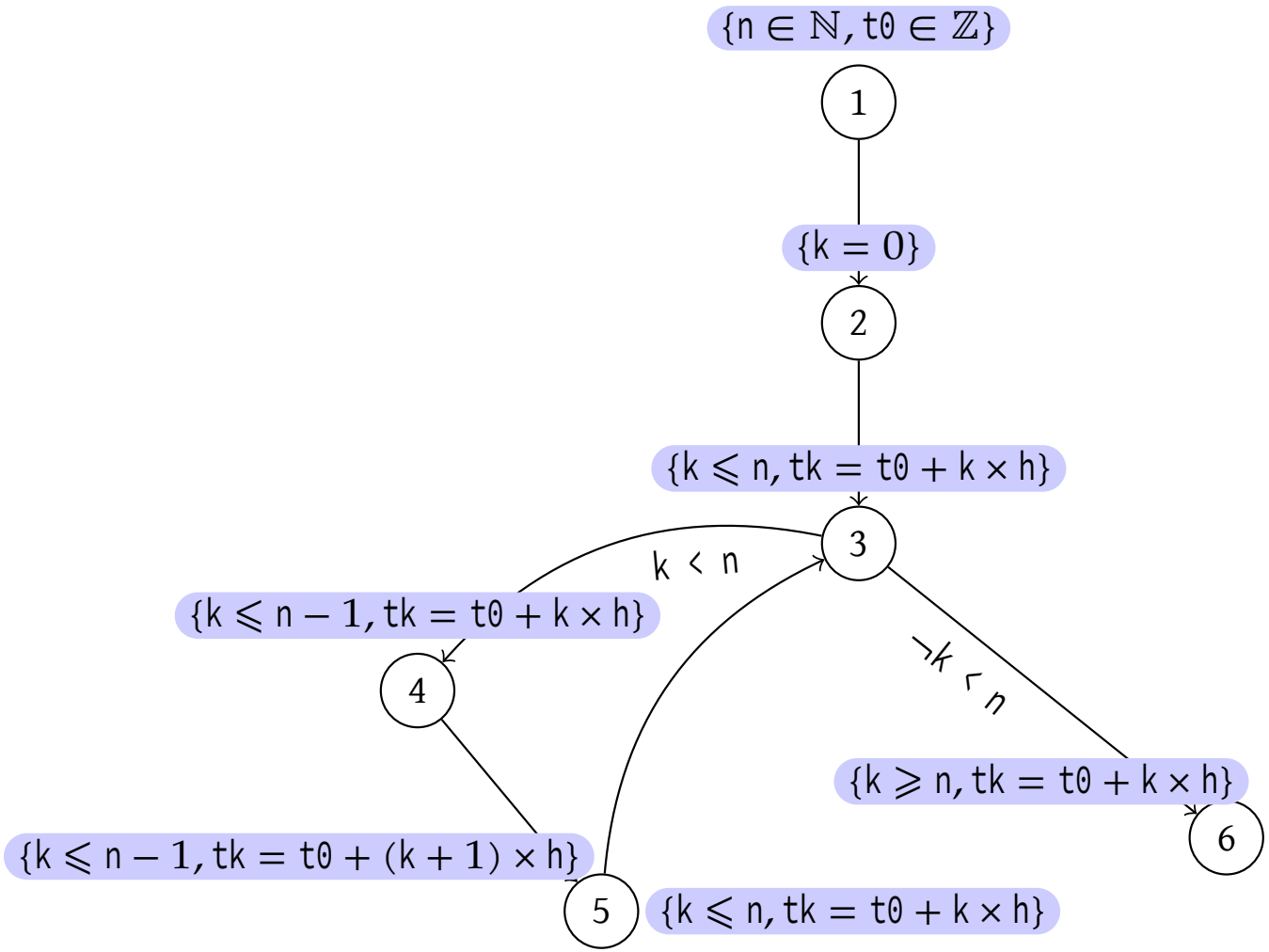
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



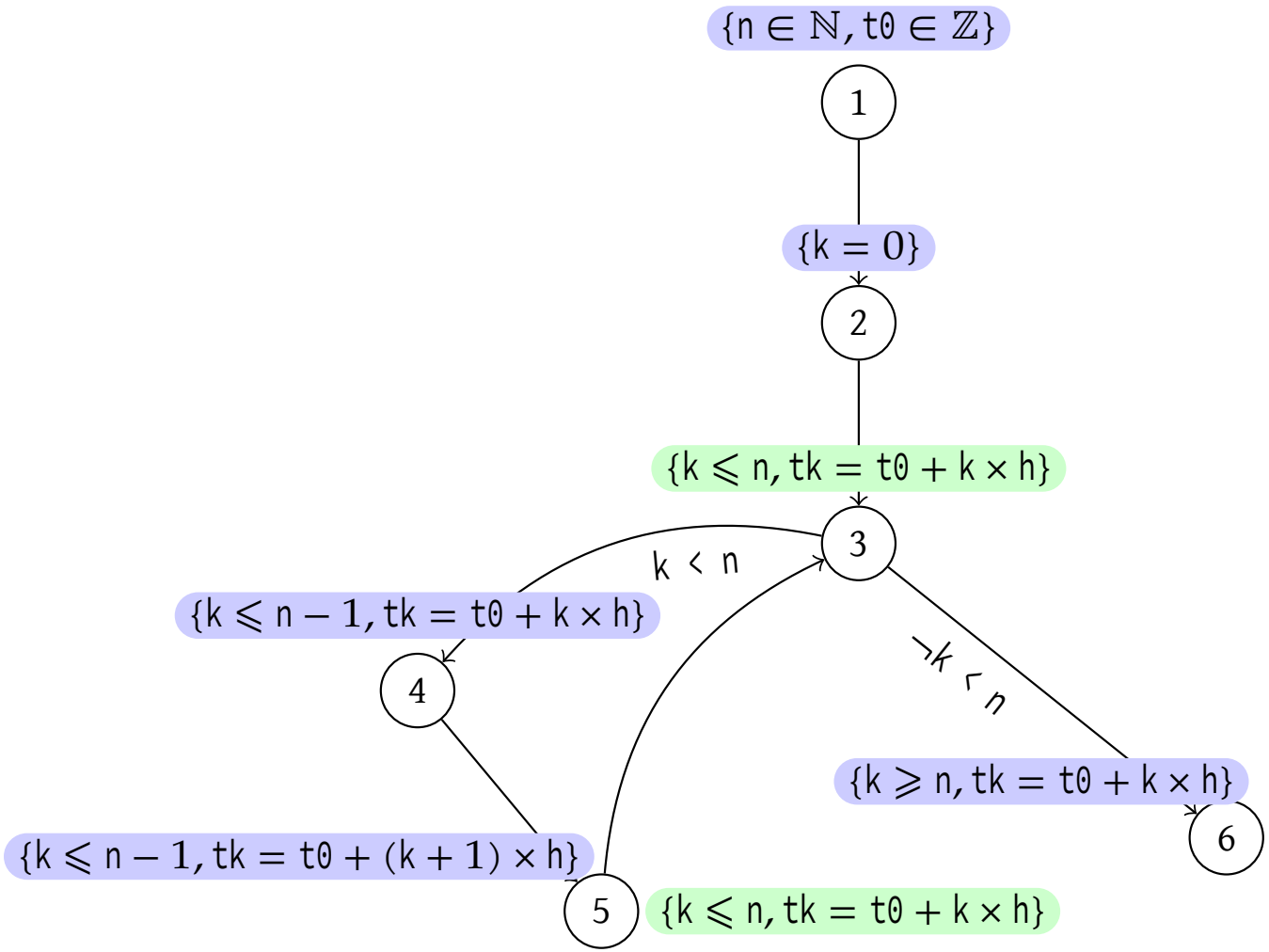
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



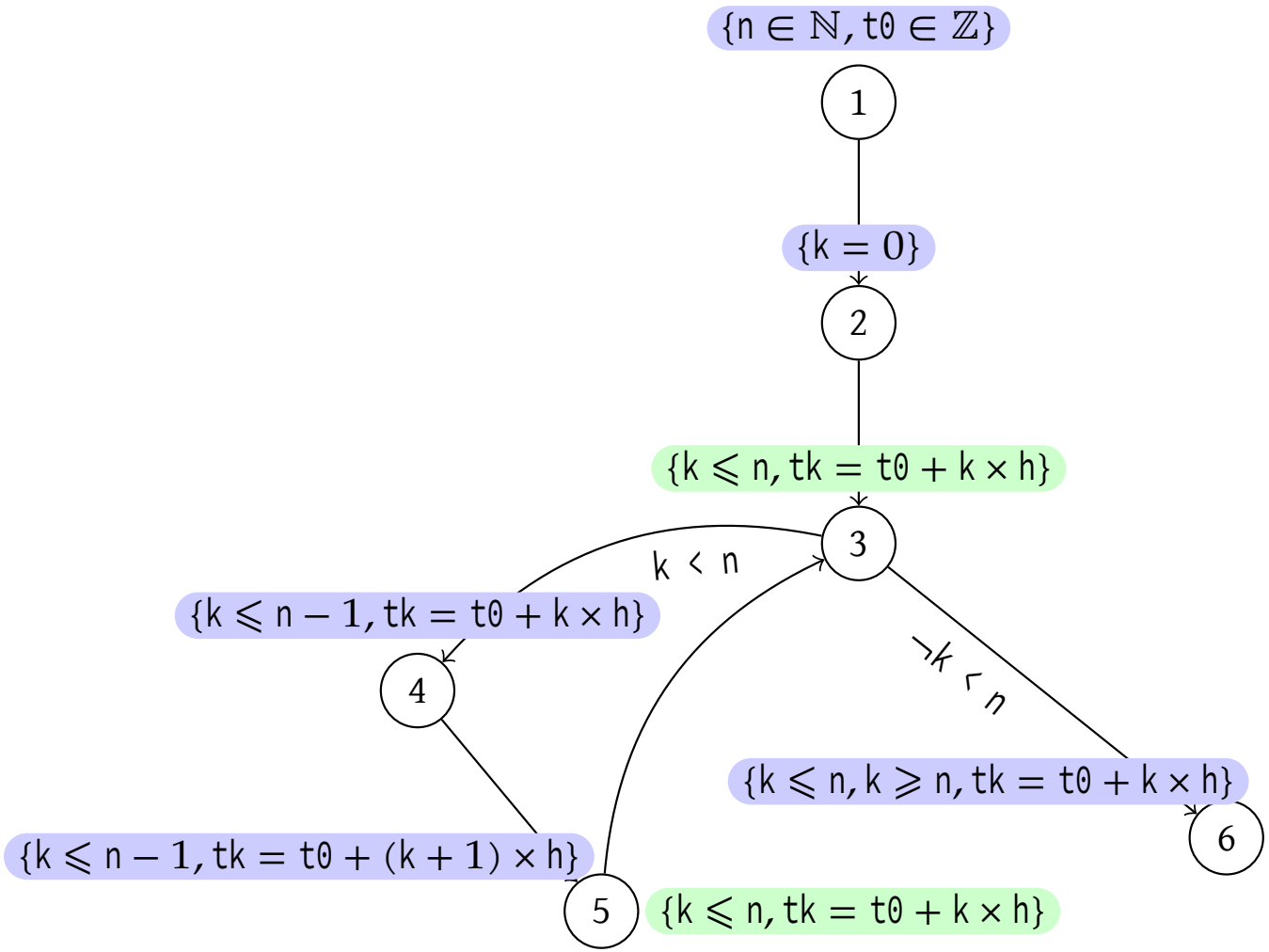
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



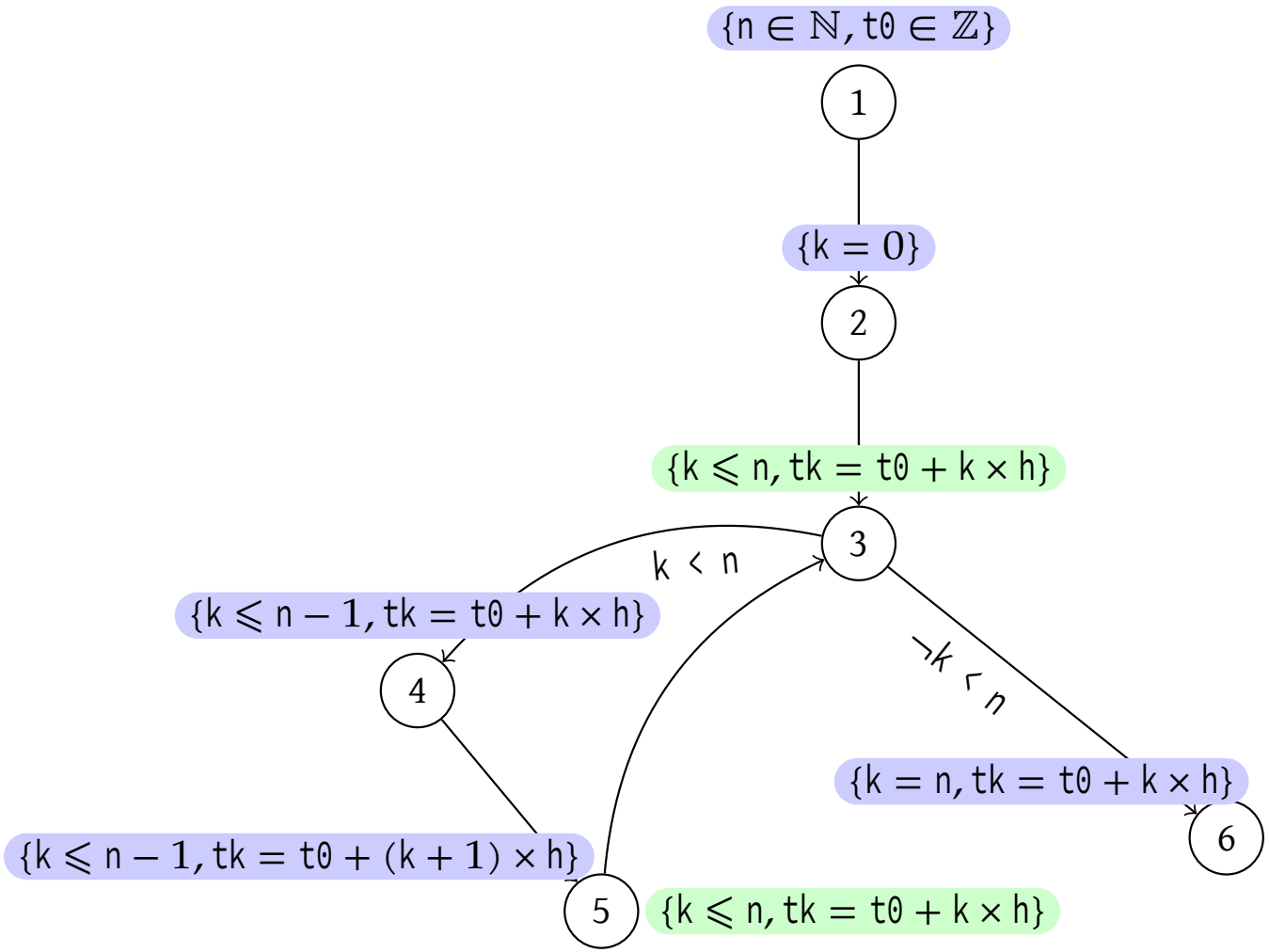
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



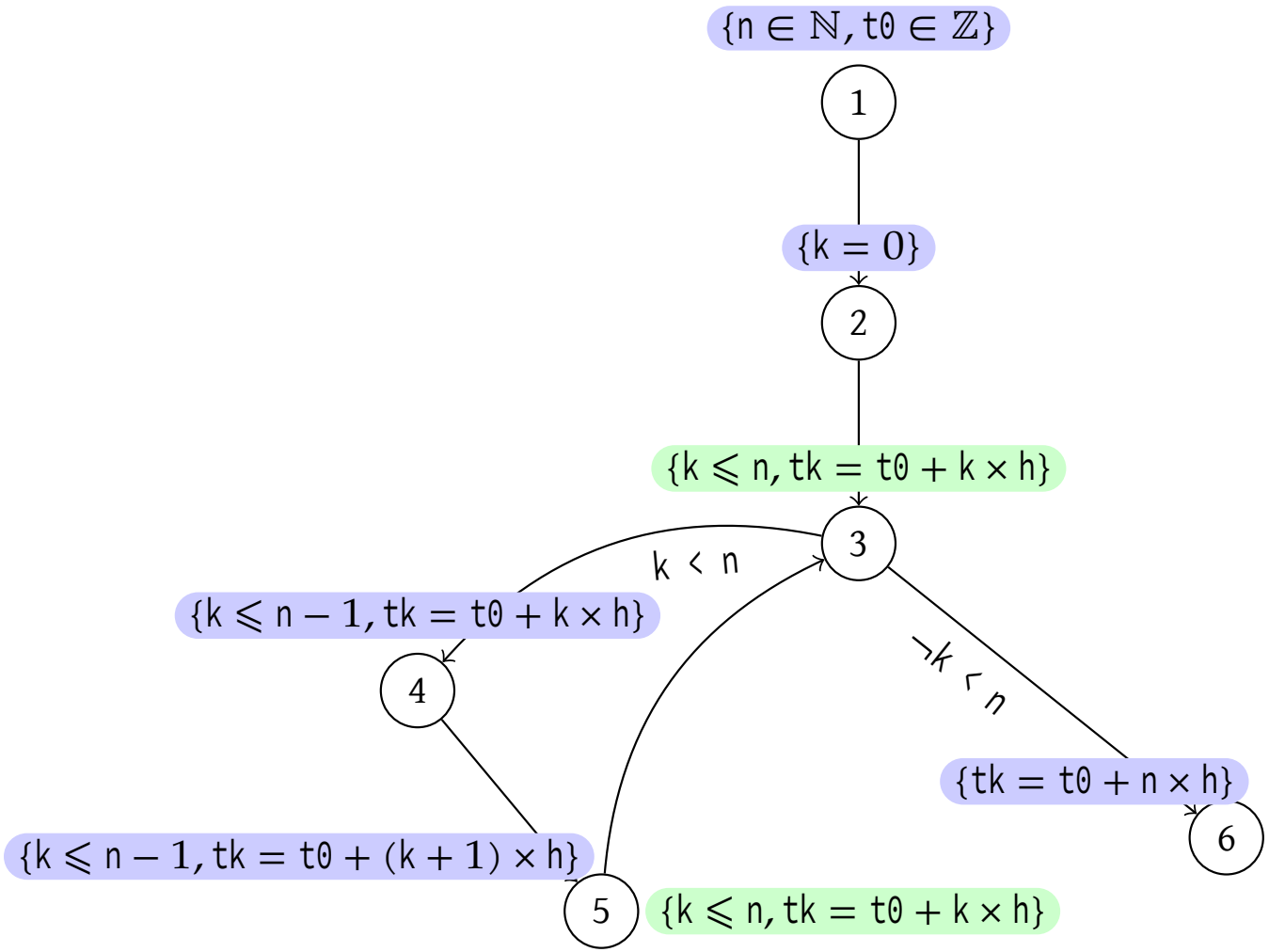
(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



(19/1)

```
1  k = 0
2  tk = t0
3  while k < n :
4    tk = tk + h
5    k = k + 1
6  # etc.
```



(19/1)

Règle de déduction pour la boucle

Ayant prouvé

$$\{I, C\} \quad corps \quad \{I\}$$

on peut déduire

$$\left\{I\right\} \quad \begin{array}{l}\texttt{while } C : \\ \quad corps\end{array} \quad \left\{I, \neg C\right\}.$$

I est un **invariant** de la boucle.

(20/1)

Formulation en termes de but

Pour prouver

$$\{P\} \quad \begin{array}{l}\texttt{while } C : \\ \quad corps\end{array} \quad \{Q\},$$

on exhibe un invariant I :

$$\{I, C\} \quad corps \quad \{I\}$$

(21/1)

Formulation en termes de but

Pour prouver

$$\{P\} \quad \begin{array}{l}\texttt{while } C : \\ \quad corps\end{array} \quad \{Q\},$$

on exhibe un invariant I :

$$\{I, C\} \quad corps \quad \{I\}$$

qui vérifie

$$P \Rightarrow I \qquad \text{et} \qquad I \wedge \neg C \Rightarrow Q.$$

(21/1)

Formulation en termes de but

Pour prouver

$$\{P\} \quad \begin{array}{l}\texttt{while } C : \\ \quad corps\end{array} \quad \{Q\},$$

on exhibe un invariant I :

$$\{I, C\} \quad corps \quad \{I\}$$

qui vérifie

$$P \Rightarrow I \qquad \text{et} \qquad I \wedge \neg C \Rightarrow Q.$$

Il faut inventer l'invariant I.

(21/1)

Parfois c'est facile.

n^e terme d'une suite

```
def expo(a, n) :
  """P : n >= 0"""
  r = 1
  k = 0
  while k < n :
    r = r * a
    k = k + 1
  return r
```


(22/1)

Parfois c'est facile.

n^e terme d'une suite

```
def expo(a, n) :
  """P : n >= 0"""
  r = 1
  k = 0
  while k < n :
    r = r * a
    k = k + 1
  return r
```

$$r = a^k \wedge k \leqslant n$$

(22/1)

Parfois c'est facile.

Plus petit entier tel que

```
def pppa(n) :
  """P : n >= 0"""
  k = n + 1
  while not est_premier(k) :
    k = k + 1
  return k
```


(23/1)

Parfois c'est facile.

Plus petit entier tel que

```
def pppa(n) :
  """P : n >= 0"""
  k = n + 1
  while not est_premier(k) :
    k = k + 1
  return k
```

$$k > n \wedge \forall x \in]\!]n, k[\![, \; x \text{ n'est pas premier}$$

(23/1)

Parfois c'est facile.

Parcours de tableau

```
def appartient(x,t) :
  for e in t :
    if e == x :
      return True
  return False
```


(24/1)

Parfois c'est facile.

Parcours de tableau

```
def appartient(x,t) :
  for e in t :
    if e == x :
      return True
  return False
```

Pour tout a de t déjà vu,

$$a \neq x$$

(24/1)

Parfois c'est facile.

Parcours de tableau

```
def appartient(x,t) :
  for e in t :
    if e == x :
      return True
  return False
```

Pour tout a de t déjà vu,

$$a \neq x$$

Au départ, on n'a vu personne !

(24/1)

Parfois c'est facile.

Parcours de tableau

```
def appartient(x,t) :
  for e in t :
    if e == x :
      return True
  return False
```

Pour tout a de t déjà vu,

$$a \neq x$$

Au départ, on n'a vu personne !
Pour prouver l'invariance, on suppose un tour de boucle complet : la condition du `if` est fausse.

(24/1)

Parfois, c'est plus dur !

```
def exporap(a, n) :
  r = 1
  f = a
  p = n
  while p > 0 :
    if p % 2 == 1 :
      r = r * f
    f = f * f
    p = p // 2
  return r
```

Mettre en rapport :

Le but a^n

(25/1)

Parfois, c'est plus dur !

```
def exporap(a, n) :
  r = 1
  f = a
  p = n
  while p > 0 :
    if p % 2 == 1 :
      r = r * f
    f = f * f
    p = p // 2
  return r
```

Mettre en rapport :

Le but a^n

Ce qu'on a calculé `r`

(25/1)

Parfois, c'est plus dur !

```
def exporap(a, n) :
  r = 1
  f = a
  p = n
  while p > 0 :
    if p % 2 == 1 :
      r = r * f
    f = f * f
    p = p // 2
  return r
```

Mettre en rapport :

Le but a^n

Ce qu'on a calculé r

Ce qui reste à faire f^p

(25/1)

Parfois, c'est plus dur !

```
def exporap(a, n) :
  r = 1
  f = a
  p = n
  while p > 0 :
    if p % 2 == 1 :
      r = r * f
    f = f * f
    p = p // 2
  return r
```

Mettre en rapport :

Le but a^n

Ce qu'on a calculé r

Ce qui reste à faire f^p

$r \times f^p = a^n$

(25/1)

Parfois, c'est plus dur !

```
def exporap(a, n) :
  r = 1
  f = a
  p = n
  while p > 0 :
    if p % 2 == 1 :
      r = r * f
    f = f * f
    p = p // 2
  return r
```

Mettre en rapport :

Le but a^n

Ce qu'on a calculé r

Ce qui reste à faire f^p

$r \times f^p = a^n \ (\wedge\ p \geqslant 0)$

(25/1)

Parfois, c'est plus dur !

```
def exporap(a, n) :
  r = 1
  f = a
  p = n
  while p > 0 :
    if p % 2 == 1 :
      r = r * f
    f = f * f
    p = p // 2
  return r
```

Mettre en rapport :

Le but a^n

Ce qu'on a calculé r

Ce qui reste à faire f^p

$r \times f^p = a^n \ (\wedge\ p \geqslant 0)$

Pas de méthode générale !
(théorème de Rice, 1951)

(25/1)

Problème de la terminaison.

Variant de boucle

Quantité entière positive qui décroit strictement à chaque tour de boucle.

Exponentiation rapide

p décroit strictement car dans la boucle $p > 0$ et on effectue p=p//2

(26/1)

Recherche dichotomique dans un tableau ordonné.

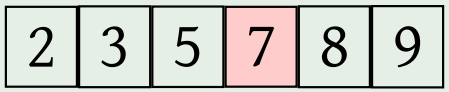
Recherche de 4 dans :

2	3	5	7	8	9

(27/1)

Recherche dichotomique dans un tableau ordonné.

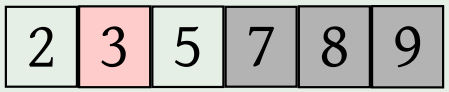
Recherche de 4 dans :



(27/1)

Recherche dichotomique dans un tableau ordonné.

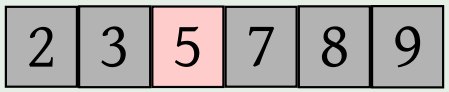
Recherche de 4 dans :



(27/1)

Recherche dichotomique dans un tableau ordonné.

Recherche de 4 dans :



(27/1)

Recherche dichotomique dans un tableau ordonné.

Recherche de 4 dans :

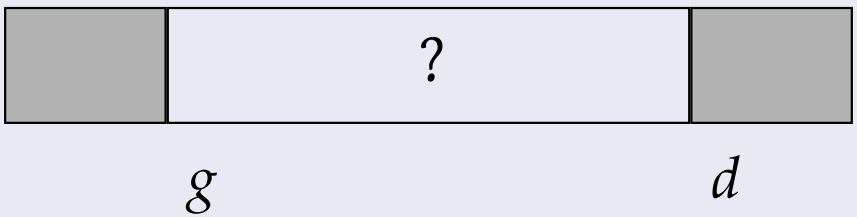
2	3	5	7	8	9

(27/1)

Recherche dichotomique dans un tableau ordonné.

Recherche de 4 dans :

Principe : recherche de x

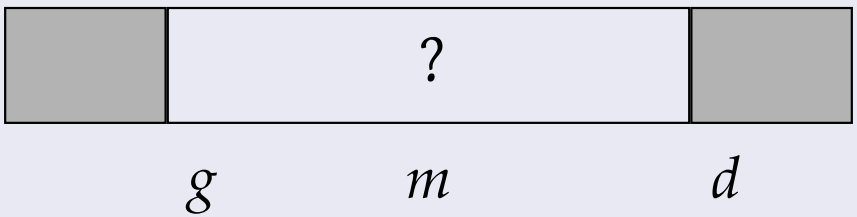


(27/1)

Recherche dichotomique dans un tableau ordonné.

Recherche de 4 dans :

Principe : recherche de x



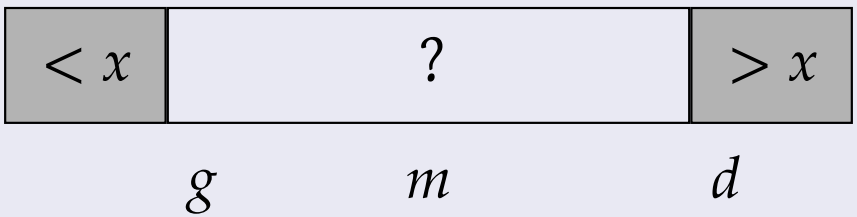
(27/1)

Recherche dichotomique dans un tableau ordonné.

Recherche de 4 dans :

2	3	5	7	8	9

Principe : recherche de x



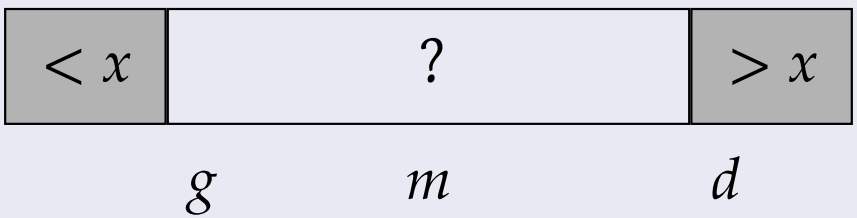
(27/1)

Recherche dichotomique dans un tableau ordonné.

Recherche de 4 dans :

2	3	5	7	8	9

Principe : recherche de x



$$I = \begin{cases} g \leqslant d, \\ \forall i \in [\![0\,, g[\![, & t[i] < x, \\ \forall i \in [\![d\,, \texttt{len}(t)[\![, & t[i] > x \end{cases}$$

(27/1)

On déduit le code de l'invariant qu'on veut.

(28/1)

Terminaison : variant naturel

$$d - g.$$

```
def dicho(x, t) :
  g = 0
  d = len(t)
  while d-g > 0 :
    m = (g+d)//2
    if t[m] == x :
      return True
    elif t[m] > x :
      d = m
    else :
      g = m
  return False
```


(29/1)

Terminaison : variant naturel

$$d - g.$$

```
def dicho(x, t) :
  g = 0
  d = len(t)
  while d-g > 0 :
    m = (g+d)//2
    if t[m] == x :
      return True
    elif t[m] > x :
      d = m
    else :
      g = m
  return False
```

À montrer :

$$\left\lfloor \frac{g+d}{2} \right\rfloor < d$$

$$\left\lfloor \frac{g+d}{2} \right\rfloor > g$$

(29/1)

Terminaison : variant naturel

$$d - g.$$

```
def dicho(x, t) :
  g = 0
  d = len(t)
  while d-g > 0 :
    m = (g+d)//2
    if t[m] == x :
      return True
    elif t[m] > x :
      d = m
    else :
      g = m + 1
  return False
```

À montrer :

$$\left\lfloor \frac{g+d}{2} \right\rfloor < d$$

$$\left\lfloor \frac{g+d}{2} \right\rfloor + 1 > g$$

(29/1)

Question intéressante

Soit P un programme et x un paramètre.
Est-ce que $P(x)$ s'arrête ?

(30/1)

Question intéressante

Soit P un programme et x un paramètre.
Est-ce que $P(x)$ s'arrête ?

Question fondamentale

Existe-t-il un algorithme H qui réponde à cette question ?

(30/1)

Par l'absurde : soit H un tel programme.

(31/1)

Par l'absurde : soit H un tel programme.
Soit D le programme de paramètre x :

Si $H(x, x)$ **alors**
  Boucler indéfiniment
Sinon
  Renvoyer 1917
FinSi

(31/1)

Par l'absurde : soit H un tel programme.
Soit D le programme de paramètre x :

Si $H(x, x)$ **alors**
  Boucler indéfiniment
Sinon
  Renvoyer 1917
FinSi

Est-ce que $D(D)$ s'arrête ?

(31/1)

Toute propriété non triviale des programmes est indécidable.

Est-ce que $P(x)$ renvoie 0 ?

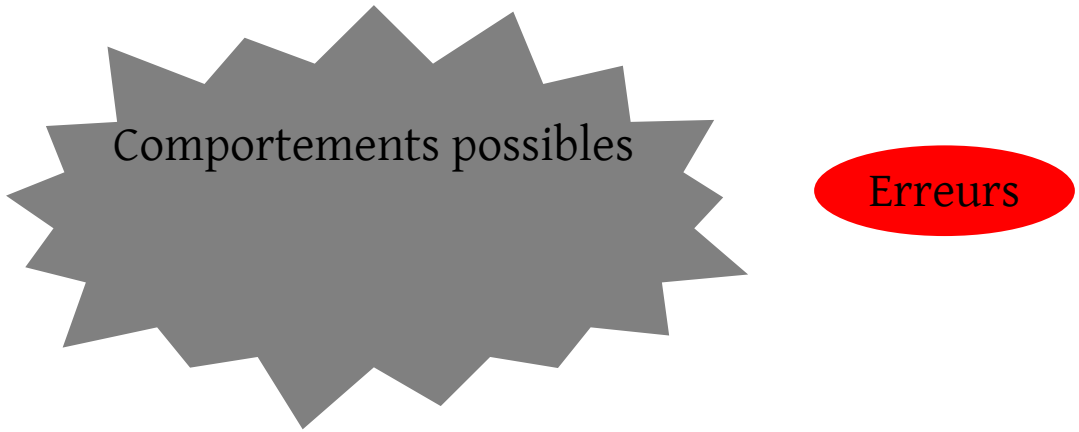
Par l'absurde : soit Z un programme le décidant.
Soit $T_{P,x}$ le programme sans paramètre :

$P(x)$
Renvoyer 0

Le programme H qui pour les paramètres P et x construit $T_{P,x}$ puis exécute $Z(T_{P,x})$ décide le problème de l'arrêt.

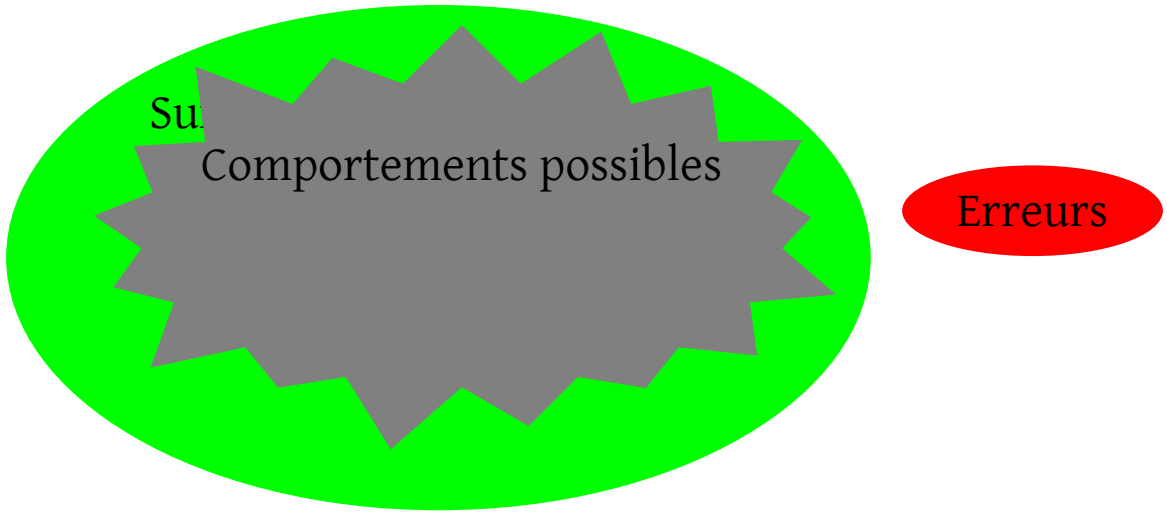
(32/1)

On procède par surapproximation.



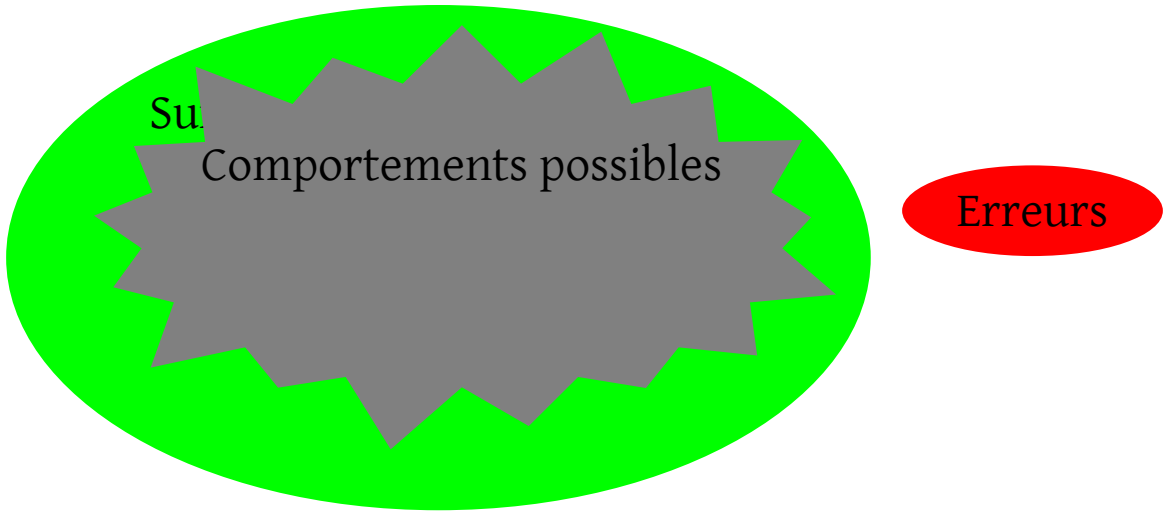
(33/1)

On procède par surapproximation.



(33/1)

On procède par surapproximation.



Analyse statique

Vérifier de manière automatique, sans exécuter le programme.
Nécessite un modèle :

- **Correct** pour dire des choses vraies,
- **Grossier** pour éviter le problème de l'arrêt,
- **Précis** pour dire des choses intéressantes.

(33/1)