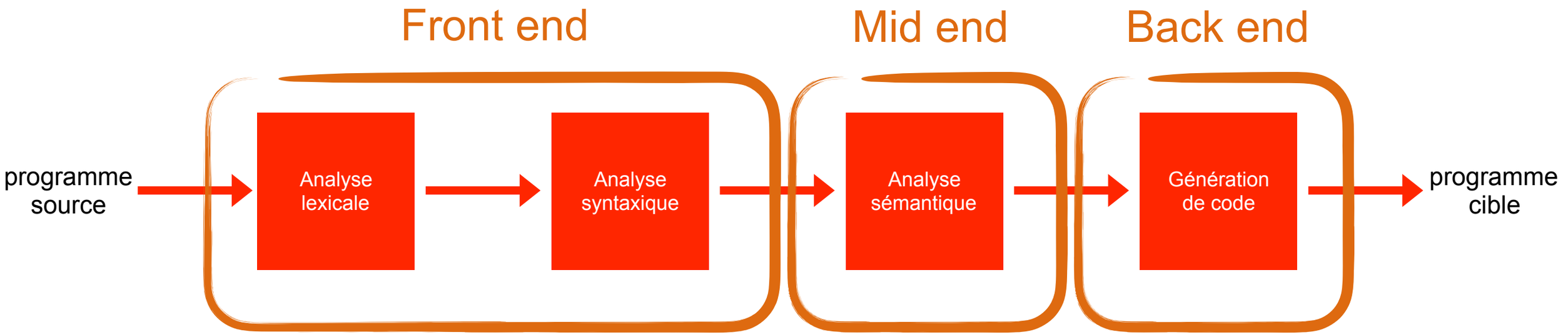
AST

Analyse statique pour l'optimisation de programme

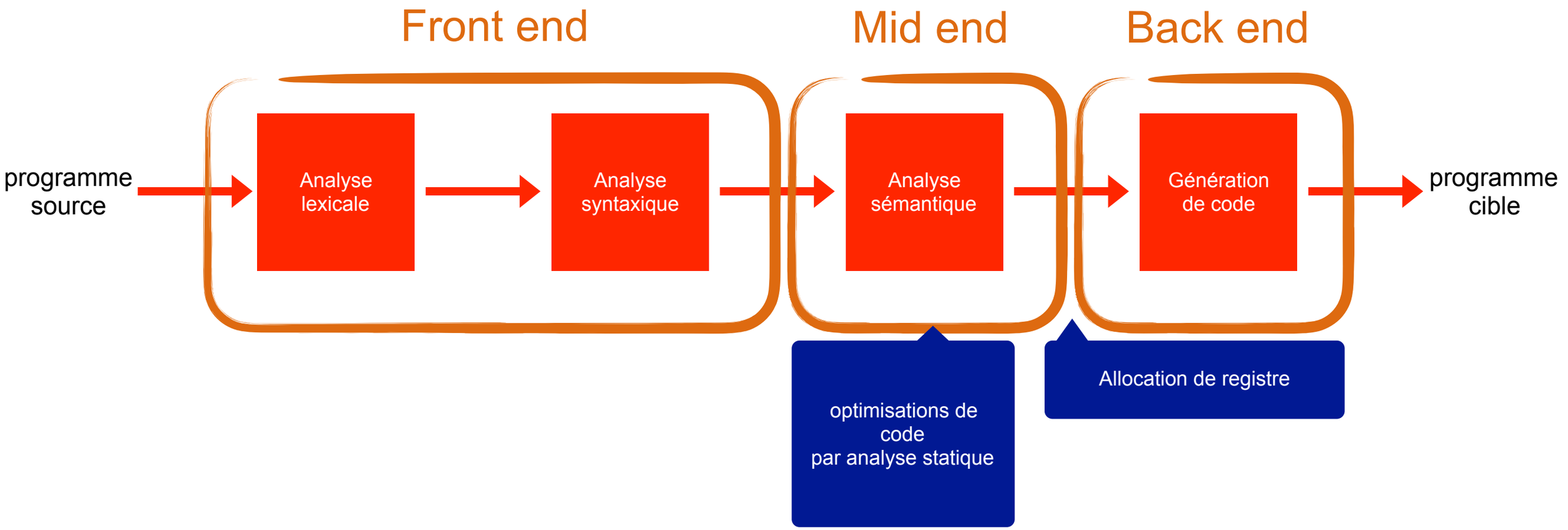
26 mars 2020

David Pichardie

Rappel : architecture globale

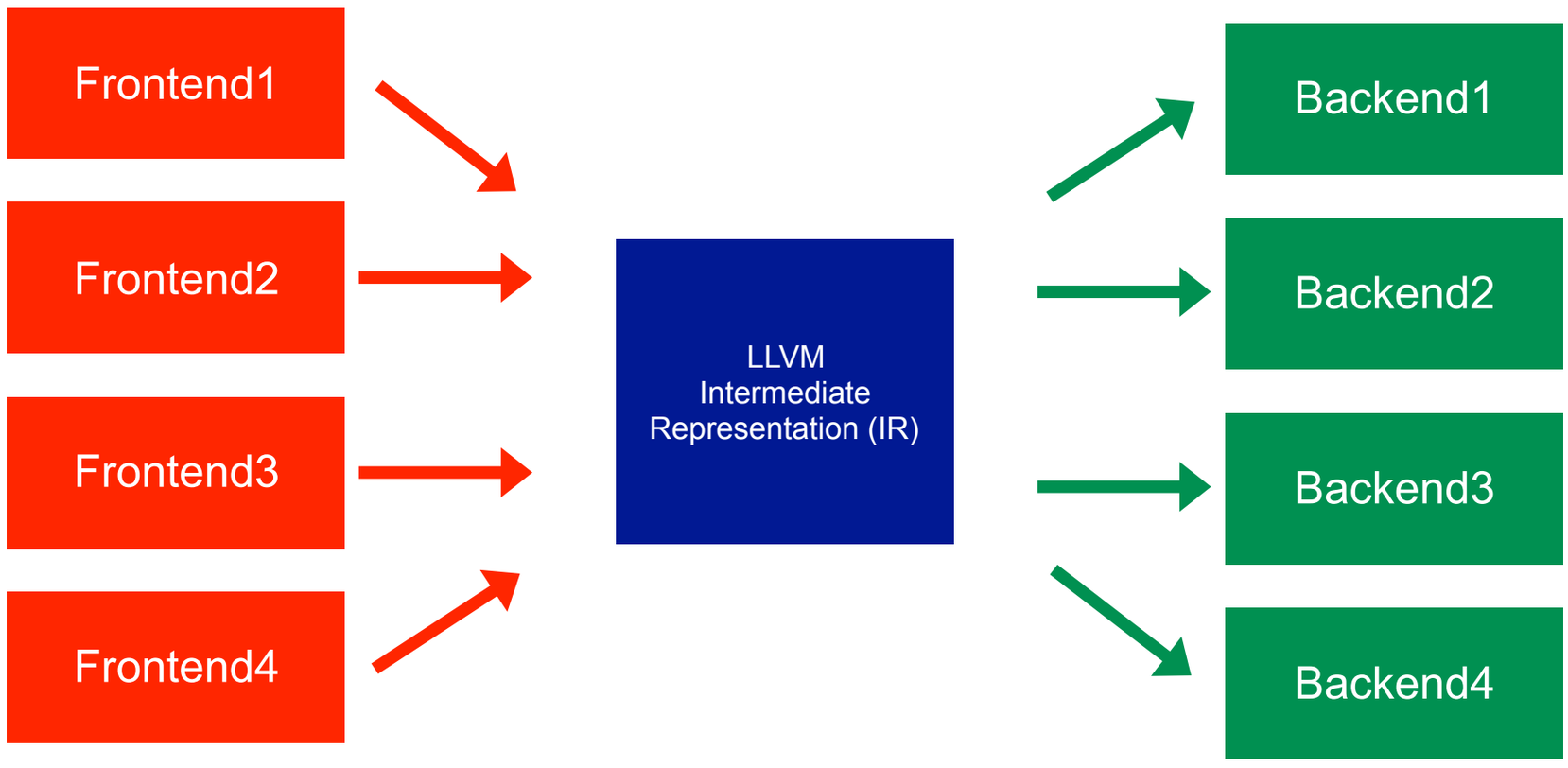


Rappel : architecture globale



LLVM mid-end

https://llvm.org



SSA : Static Single Assignment form

- Une représentation intermédiaire incontournable dans les compilateurs modernes (GCC, LLVM, VM Java HotSpot,…)

- Propriété clé :

> chaque variable est définie une seule fois
> (dans le texte du programme)

```
func F(a b)
  entry:
    [b] = a
    goto test
  test:
    y = [b]
    x = Lt(y 10)
    if x goto exit else loop
loop:
    t1 = [b]
    t2 = t1+1
    [b] = t2
    goto test
exit:
    ret a
```

SSA : Static Single Assignment form

non - SSA

```
func F(a b)
  entry:
    [b] = a
    goto test
  test:
    y = [b]
    x = Lt(y 10)
    if x goto exit else loop
loop:
    t = [b]
    t = t+1
    [b] = t
    goto test
exit:
    ret a
```

SSA

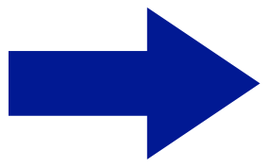
```
func F(a b)
  entry:
    [b] = a
    goto test
  test:
    y = [b]
    x = Lt(y 10)
    if x goto exit else loop
loop:
    t1 = [b]
    t2 = t1+1
    [b] = t2
    goto test
exit:
    ret a
```

pourtant exécuté plusieurs fois à cause de la boucle !

Passer en forme SSA

- Idée : renommer les variables
- Facile sur du code *linéaire*

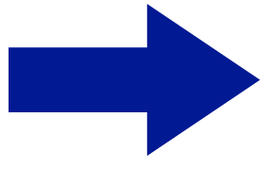
```
x = 1
y = x
x = 5
z = x
```



```
x1 = 1
y = x1
x2 = 5
z = x2
```

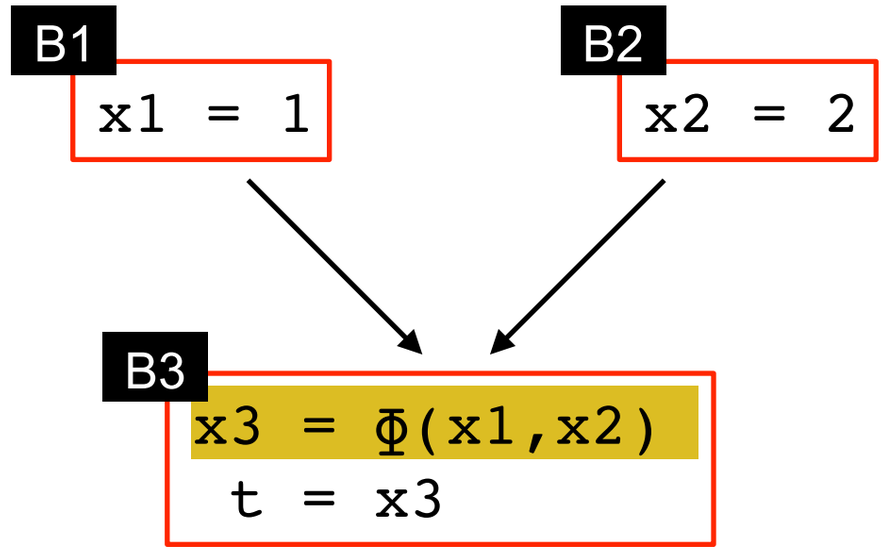
- Pas suffisant pour certains *points de jonction*

```
lab1:
  x = 1
  goto junc
lab2:
  x = 2
  goto junc
junc:
  t = x
```



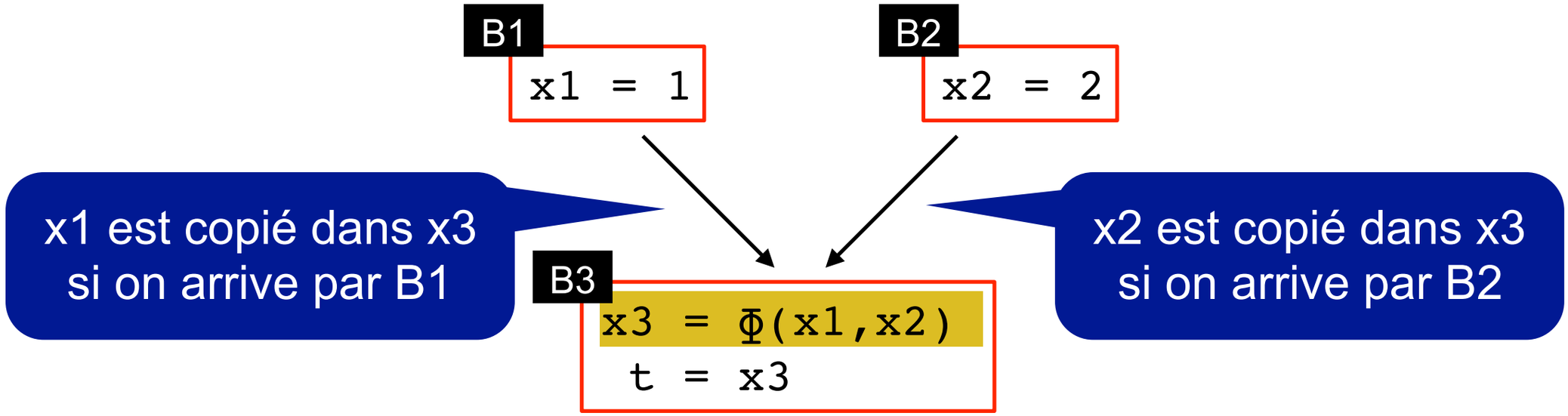
```
lab1:
  x1 = 1
  goto junc
lab2:
  x2 = 2
  goto junc
junc:
  t = x????
```

Une nouvelle instruction : la ϕ-fonction



- les ϕ-fonctions n’apparaissent qu’en début de bloc, et seulement ceux ayant plusieurs blocs prédécesseurs
- elles agissent comme des copies, mais en tenant compte du bloc de provenance

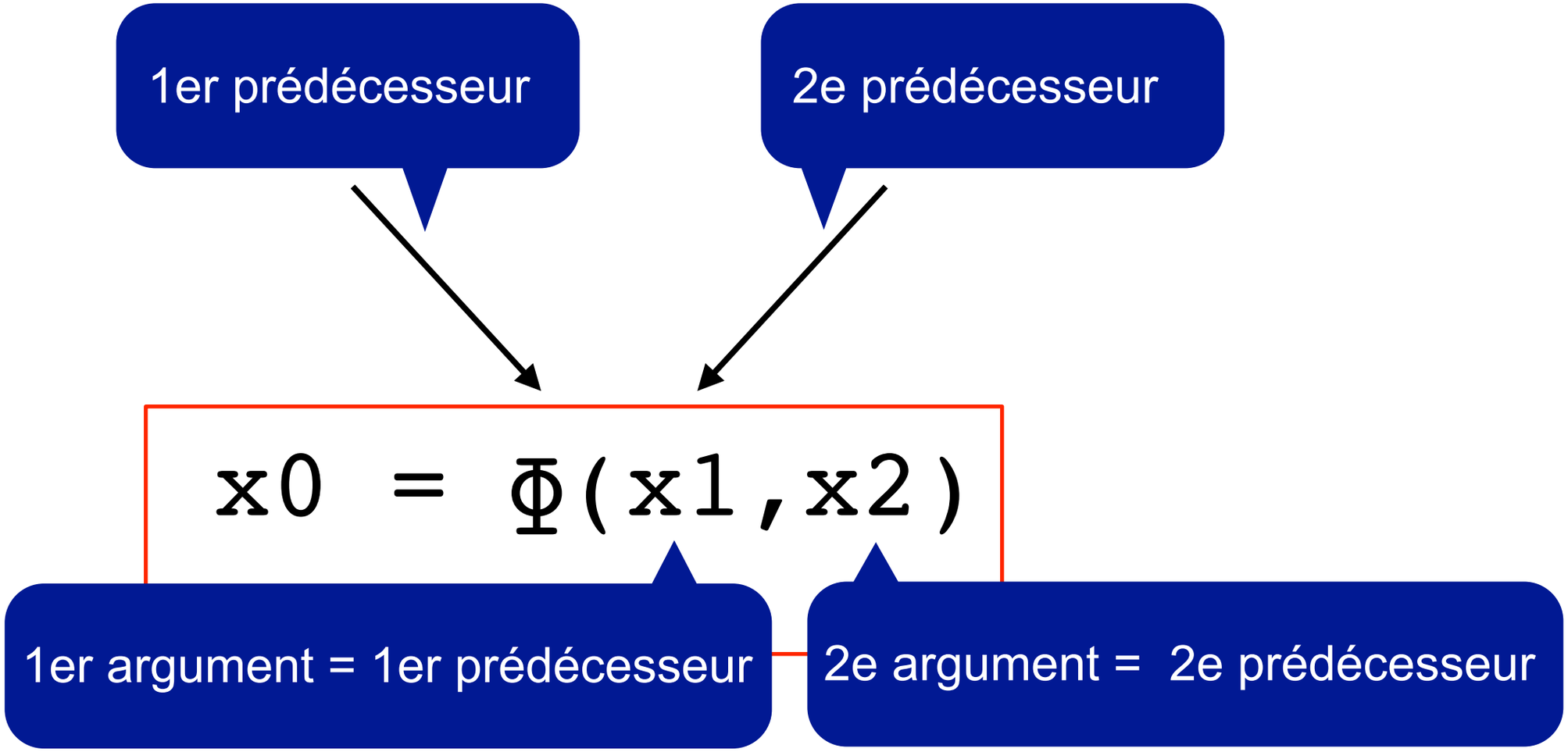
Une nouvelle instruction : la ϕ-fonction



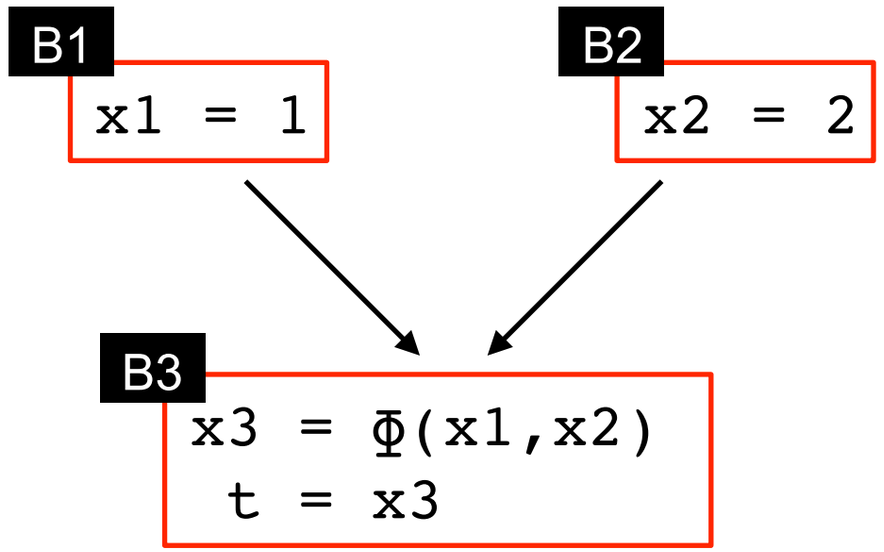
- les ϕ-fonctions n’apparaissent qu’en début de bloc, et seulement ceux ayant plusieurs blocs prédécesseurs

- elles agissent comme des copies, mais en tenant compte du bloc de provenance

Une nouvelle instruction : la ϕ-fonction



ϕ-fonction : syntaxe concrète



même syntaxe qu'en LLVM

- dans les programmes RTL, nous ajoutons une instruction `phi` dont les arguments font explicitement mention du bloc associé

```
B1:
  x1 = 1
  goto B3
B2:
  x2 = 2
  goto B3
B3:
  x3 = phi [x2,B2], [x1,B1]
  t = x3
```

ϕ-fonction : syntaxe abstraite en Java

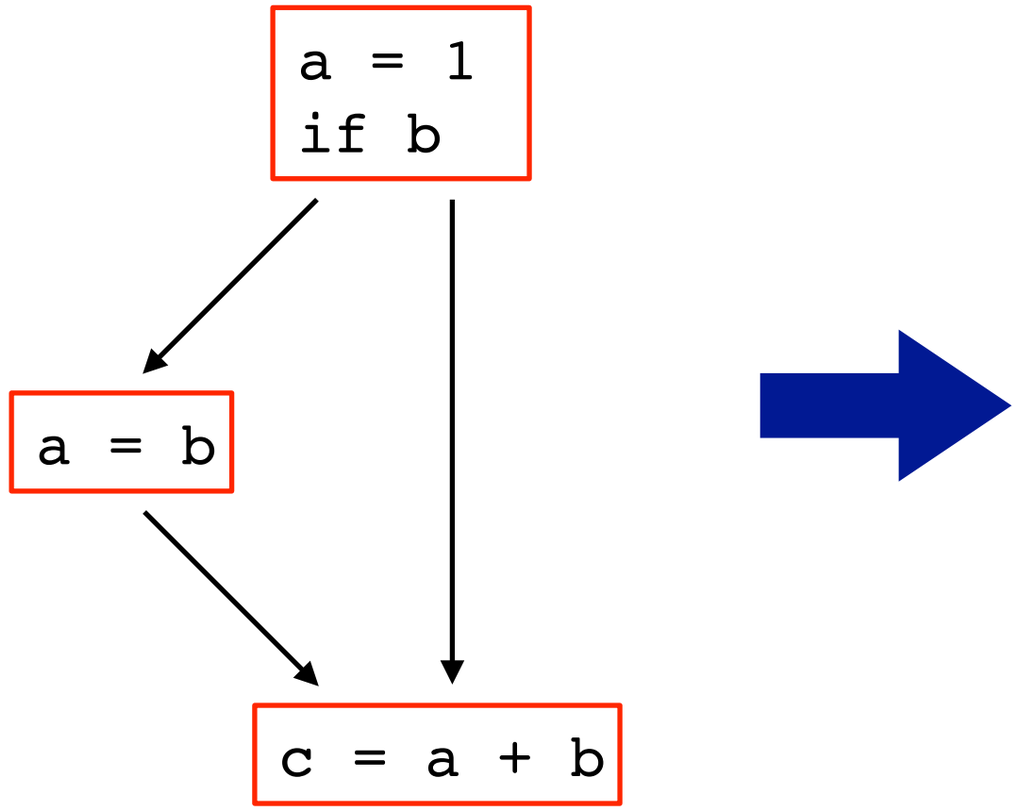
```
public class Phi {
    public final Ident target;
    public final List<PhiArg> args;
}

public class PhiArg {
    public final Operand operand;
    public final Block block;
}

public class Block {
    public final Ident label;
    public final List<Phi> phis;
    public final List<Instr> instrs;
    private EndInstr end;
}
```

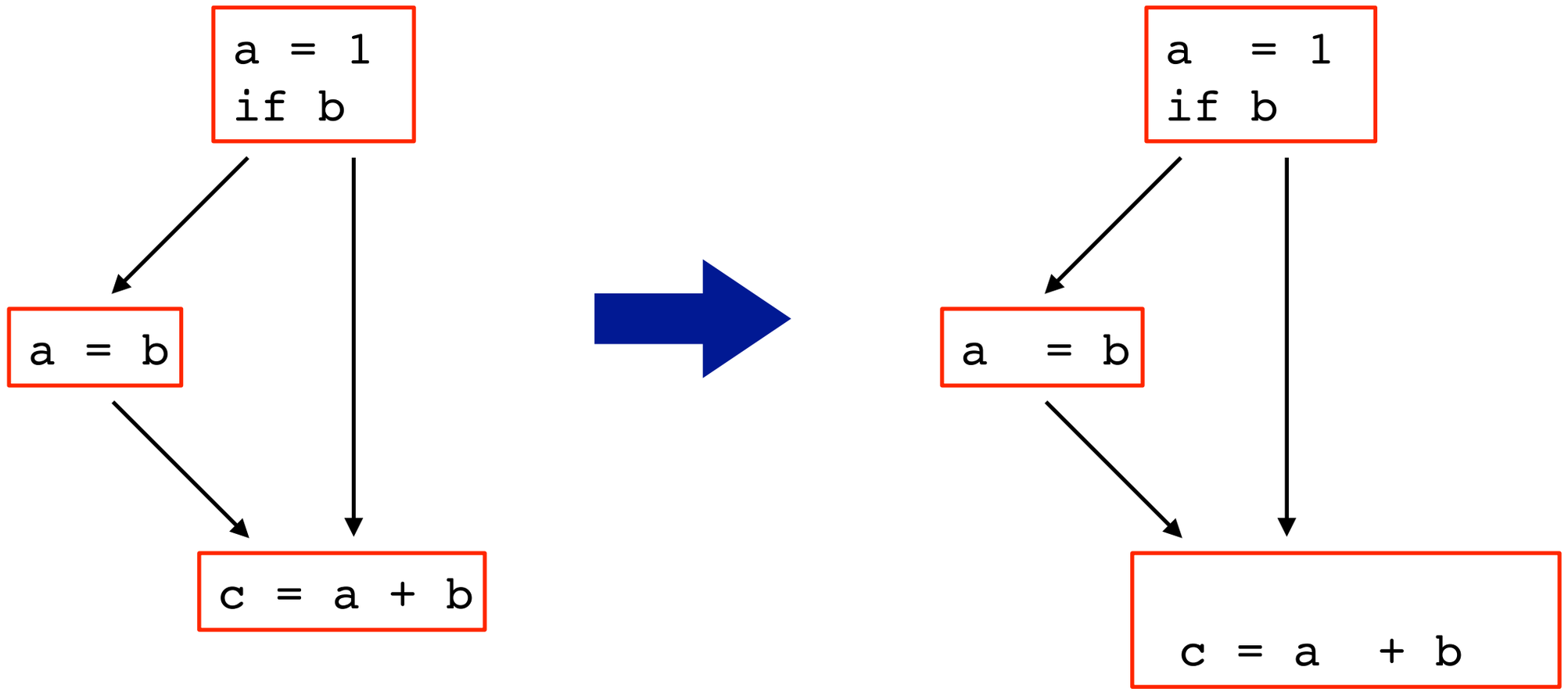
Exercice

- mettre ce programme sous une forme SSA



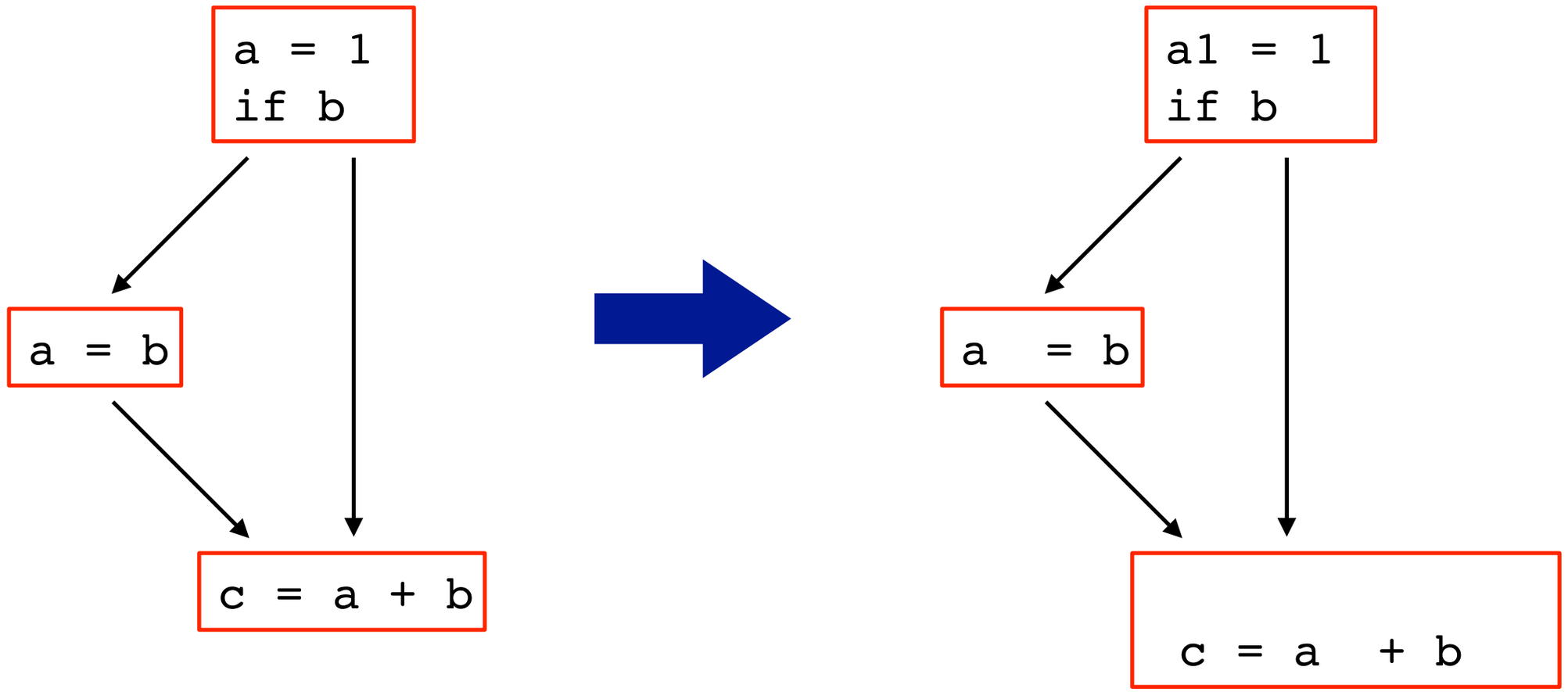
Exercice

- mettre ce programme sous une forme SSA



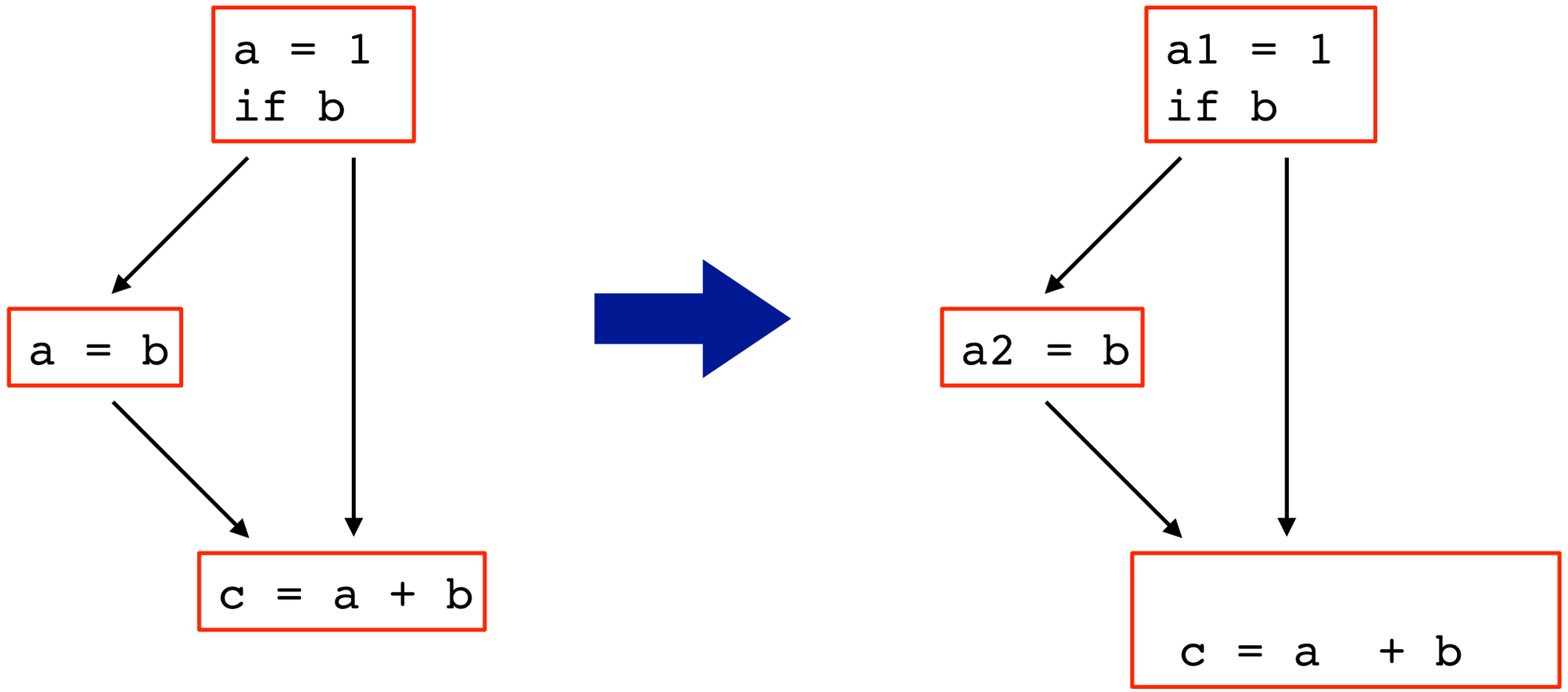
Exercice

- mettre ce programme sous une forme SSA



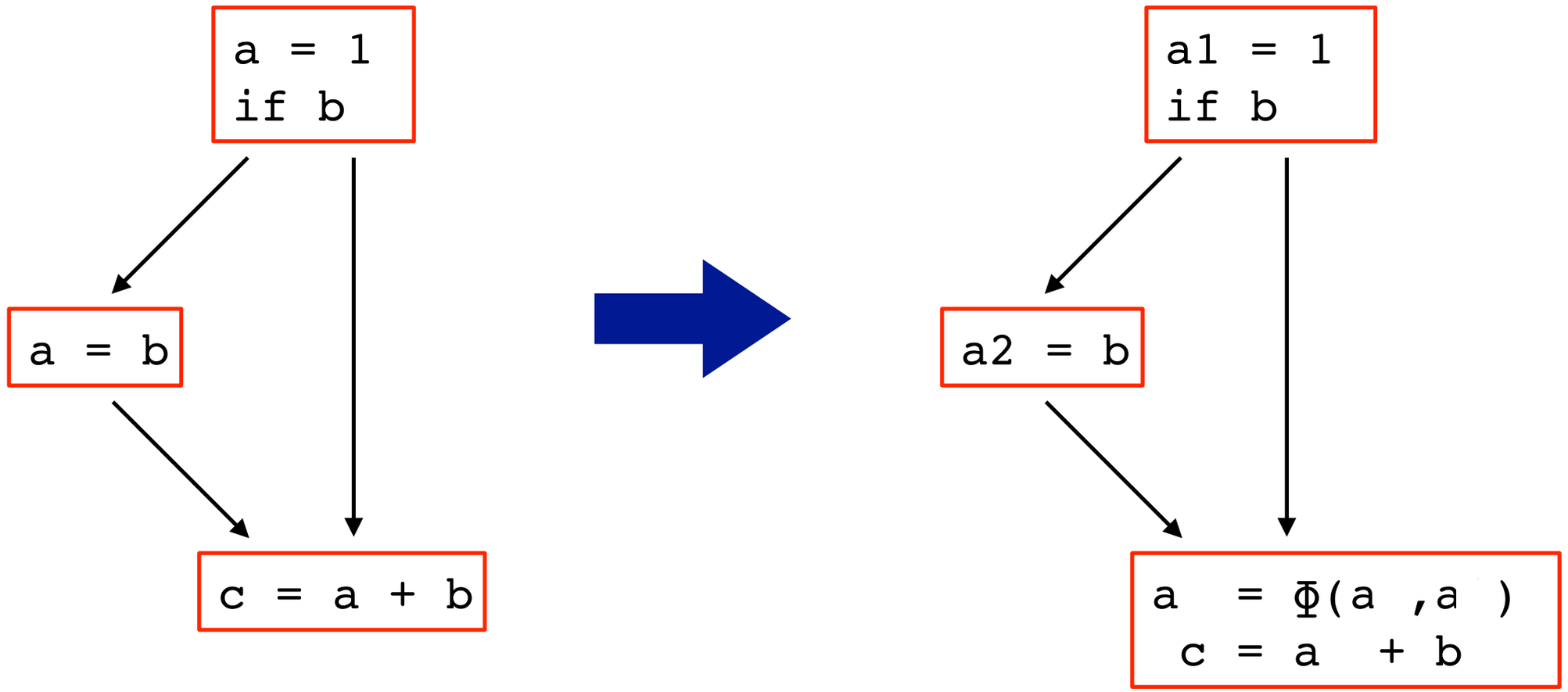
Exercice

- mettre ce programme sous une forme SSA



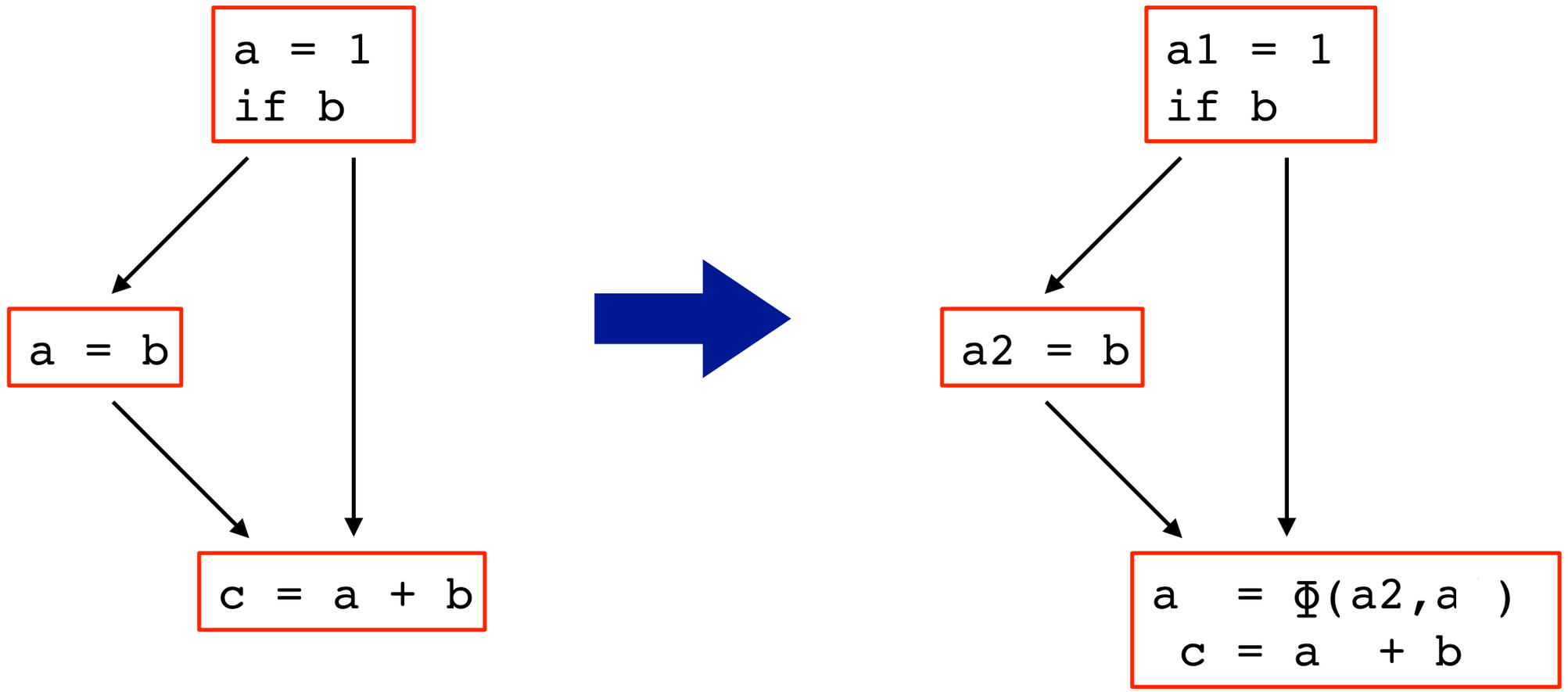
Exercice

- mettre ce programme sous une forme SSA



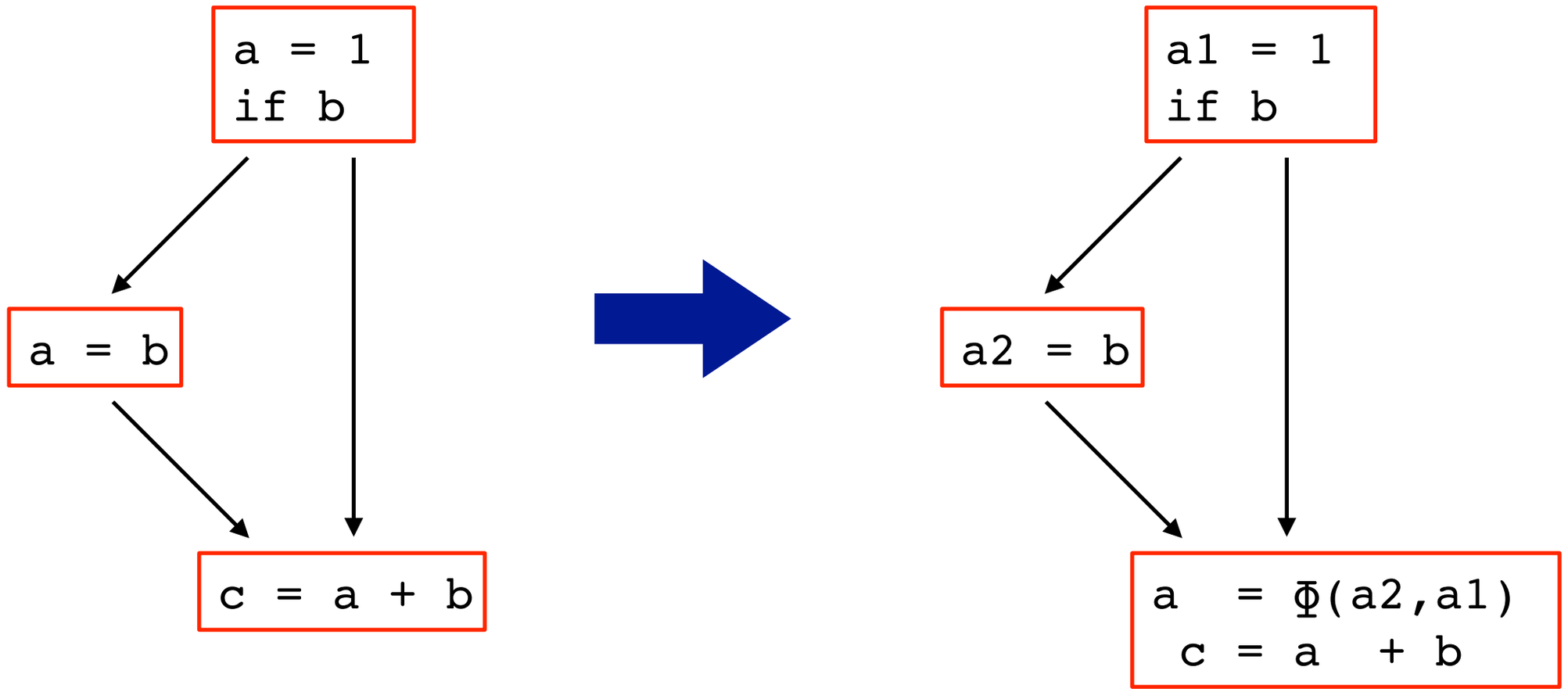
Exercice

- mettre ce programme sous une forme SSA



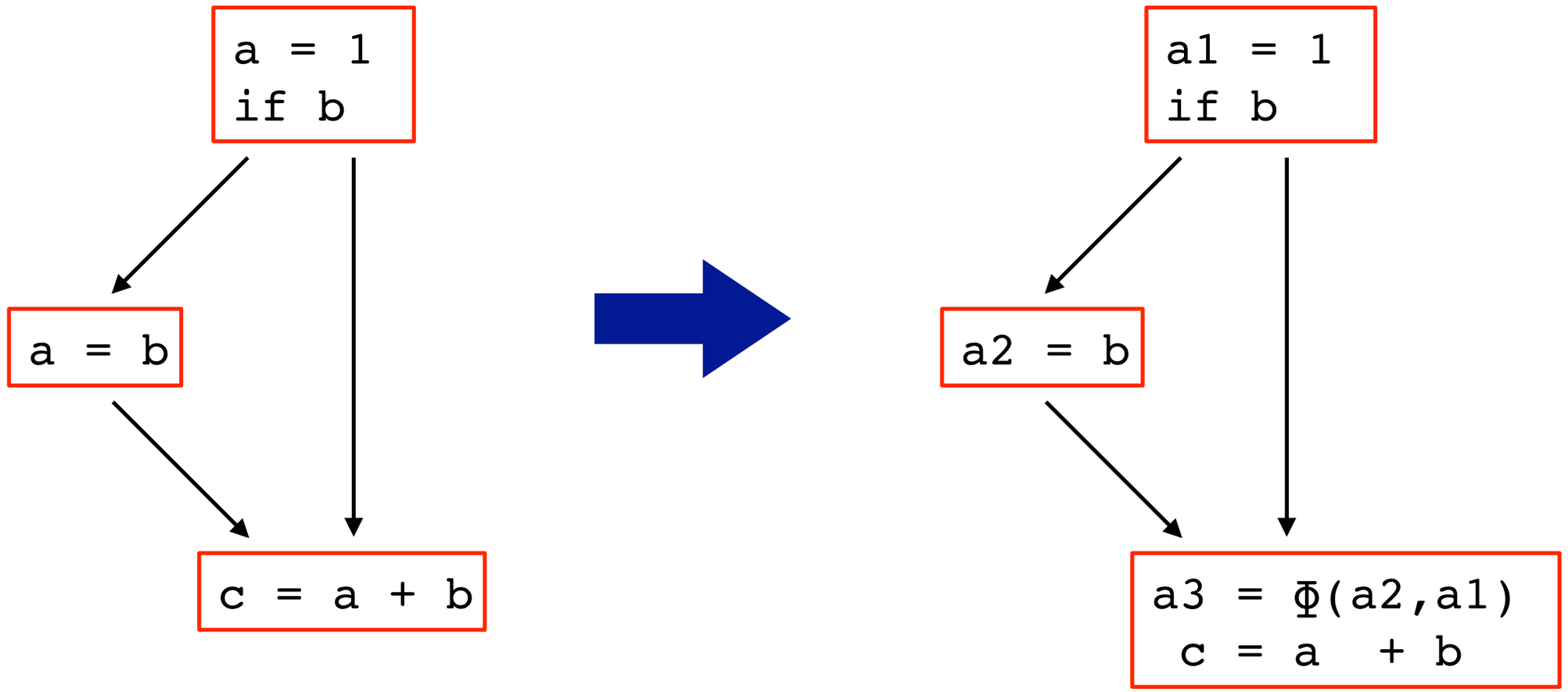
Exercice

- mettre ce programme sous une forme SSA



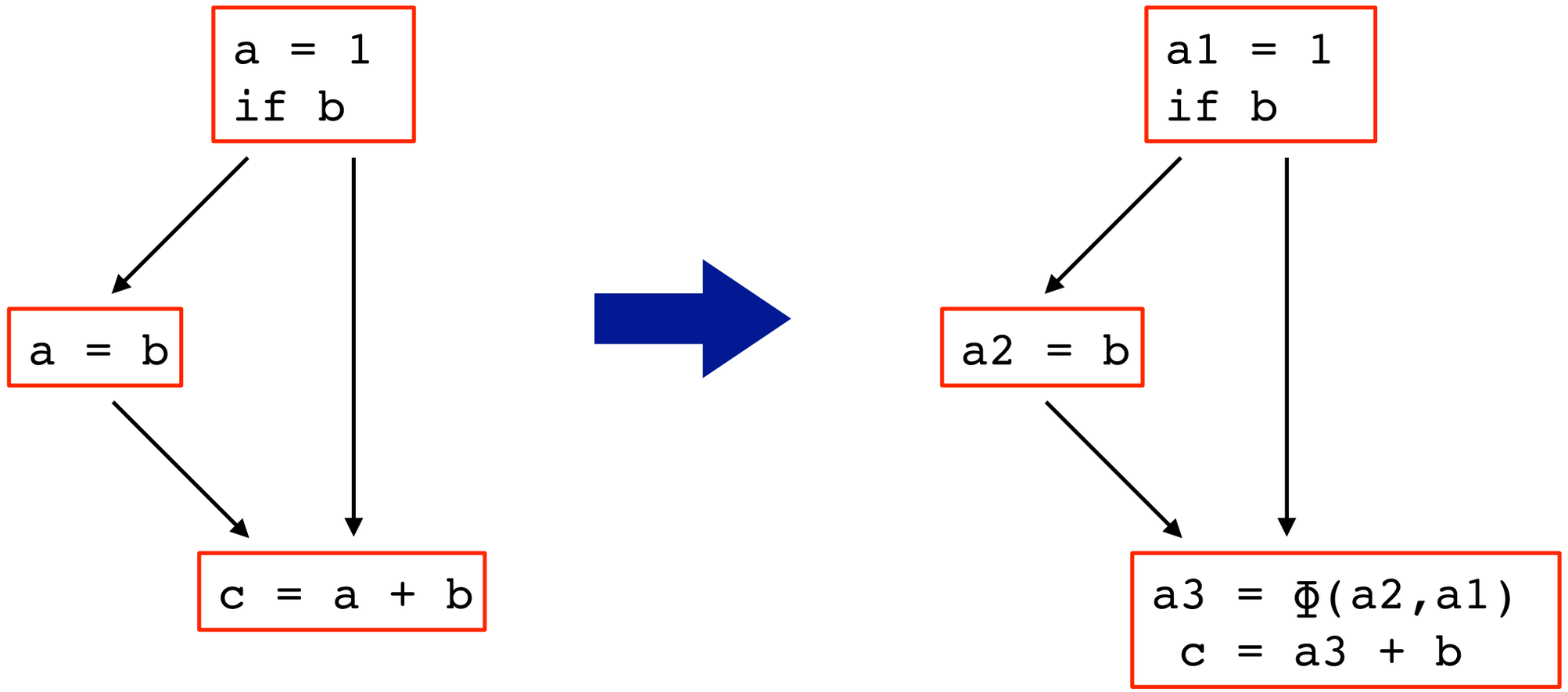
Exercice

- mettre ce programme sous une forme SSA



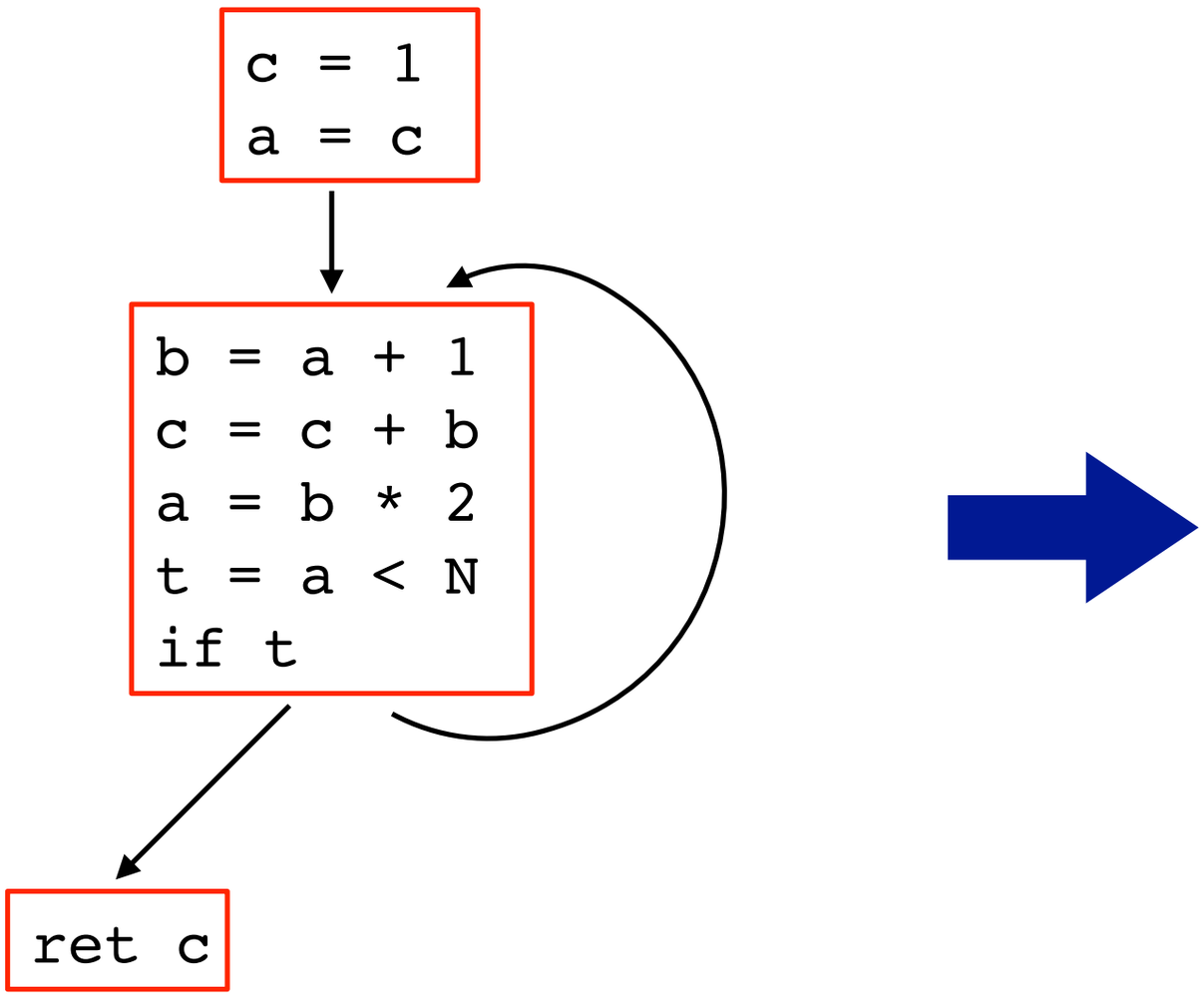
Exercice

- mettre ce programme sous une forme SSA



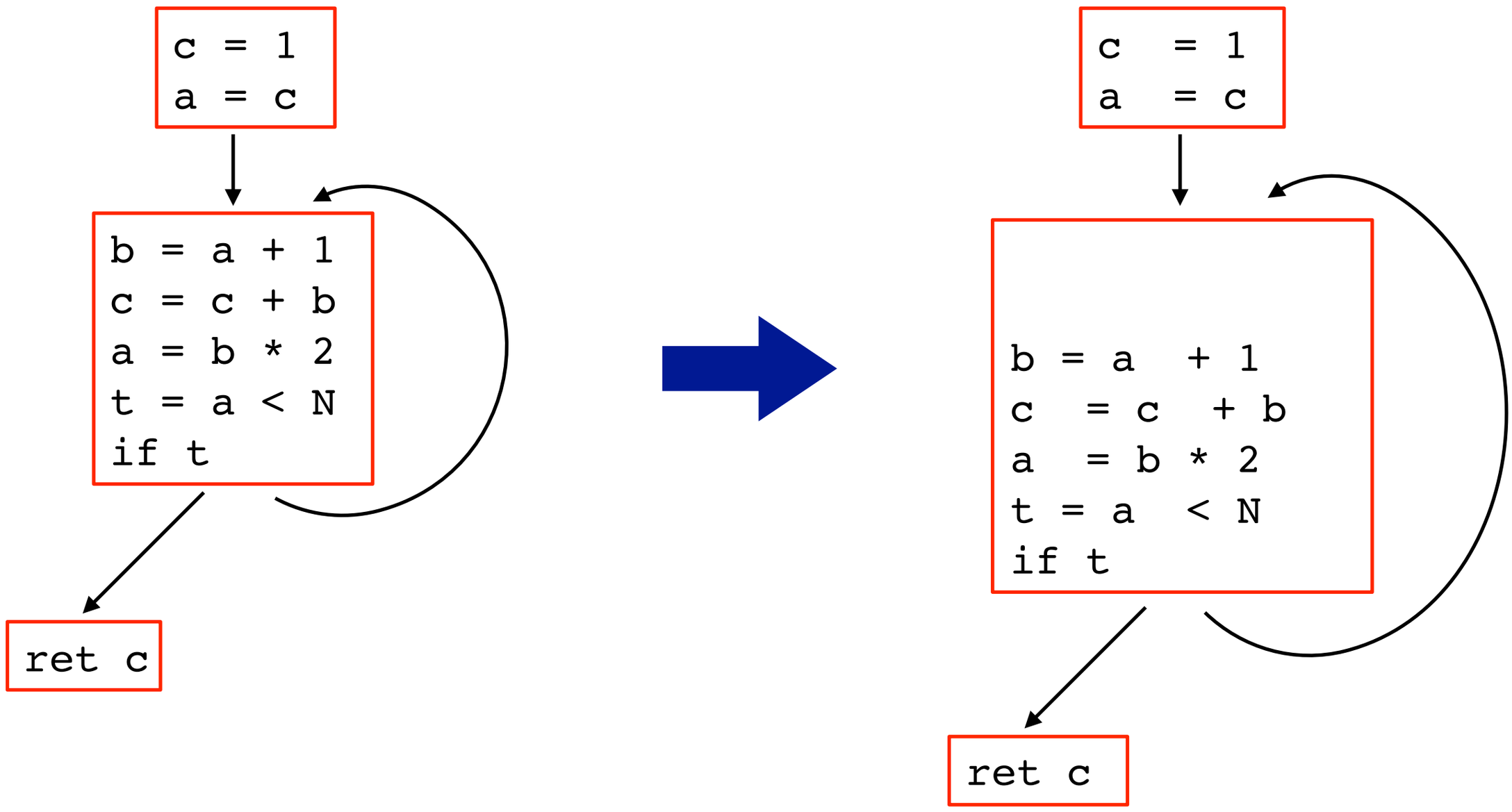
Exercice

- mettre ce programme sous une forme SSA



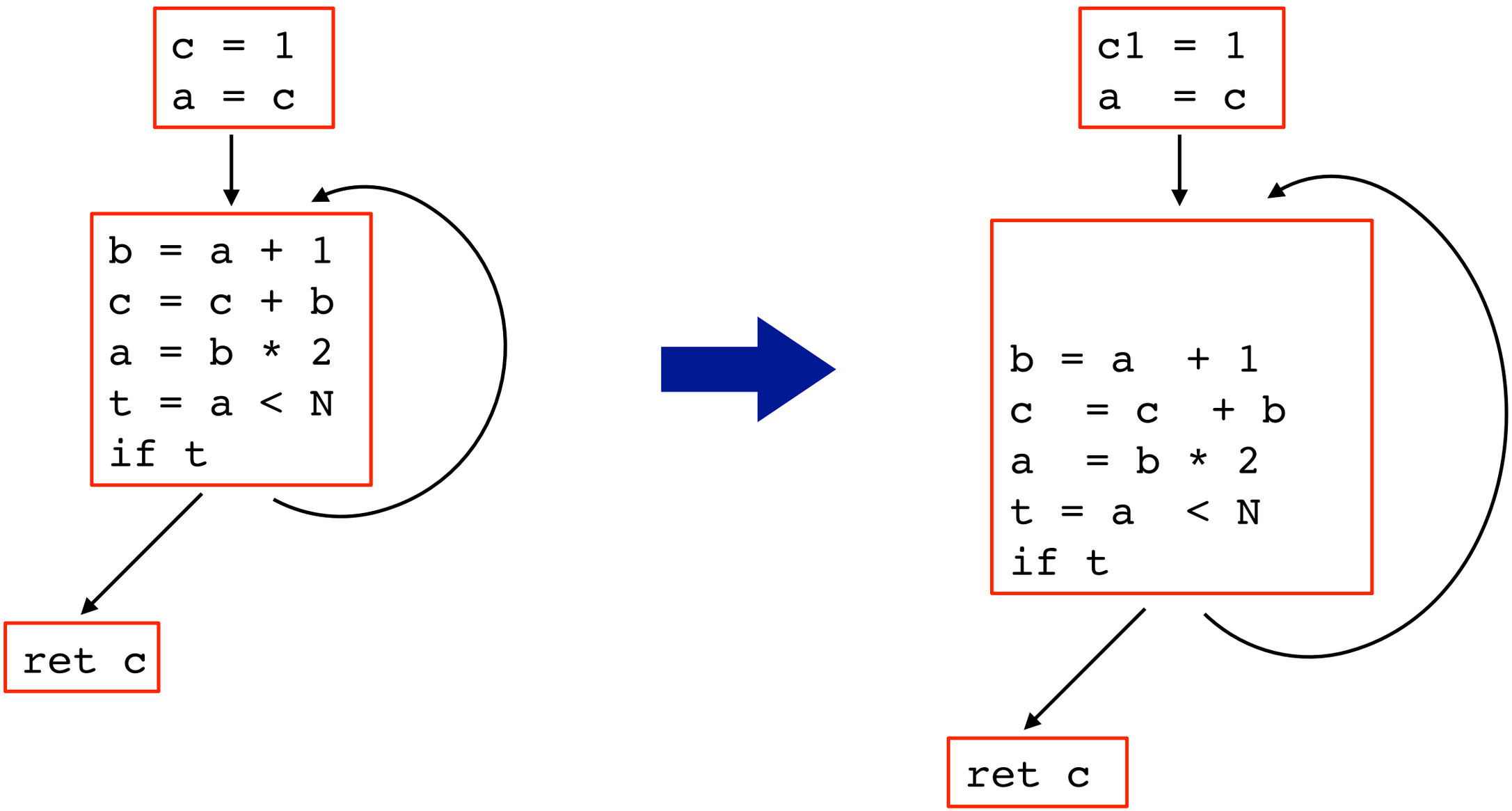
Exercice

- mettre ce programme sous une forme SSA



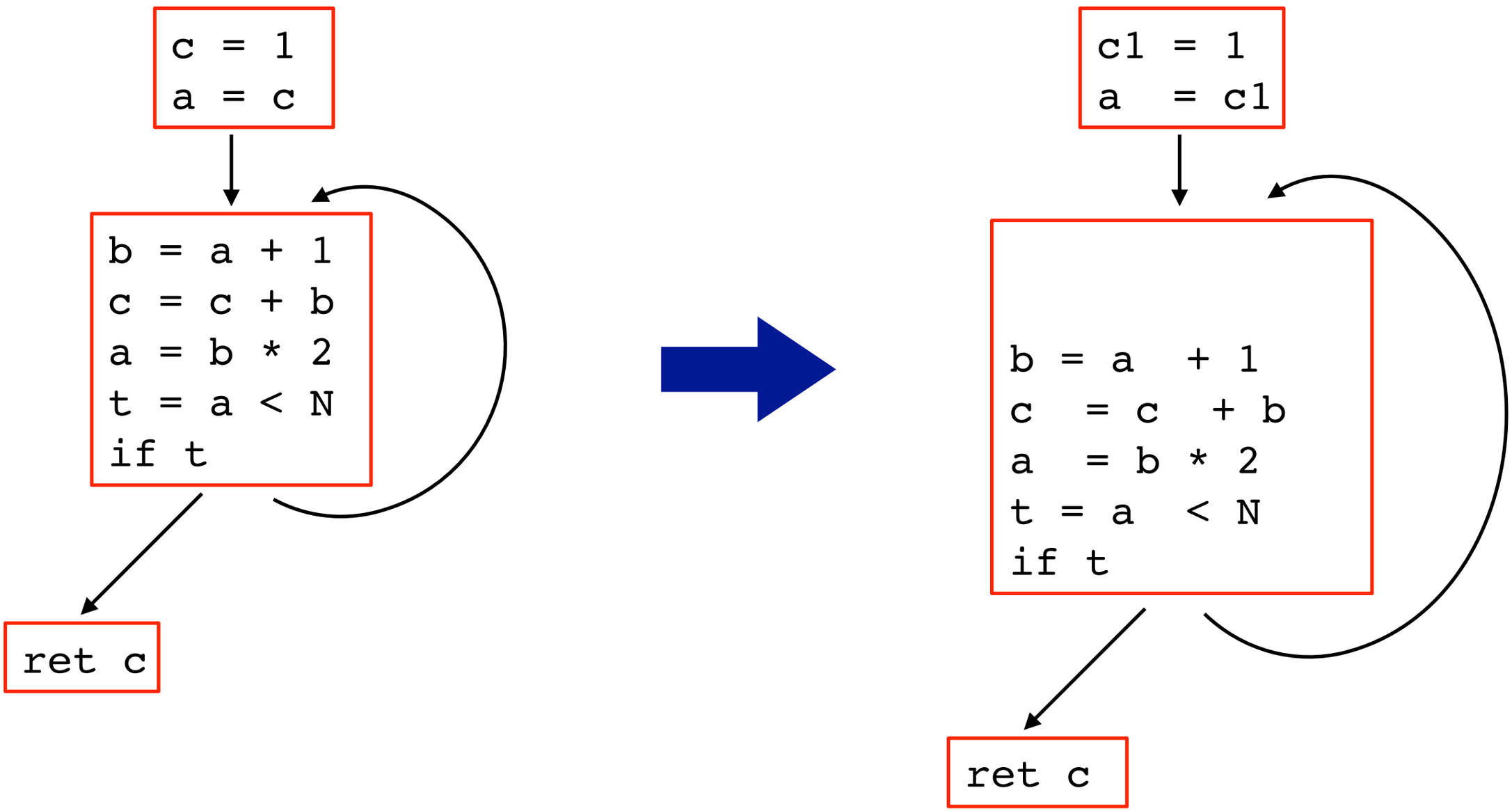
Exercice

- mettre ce programme sous une forme SSA



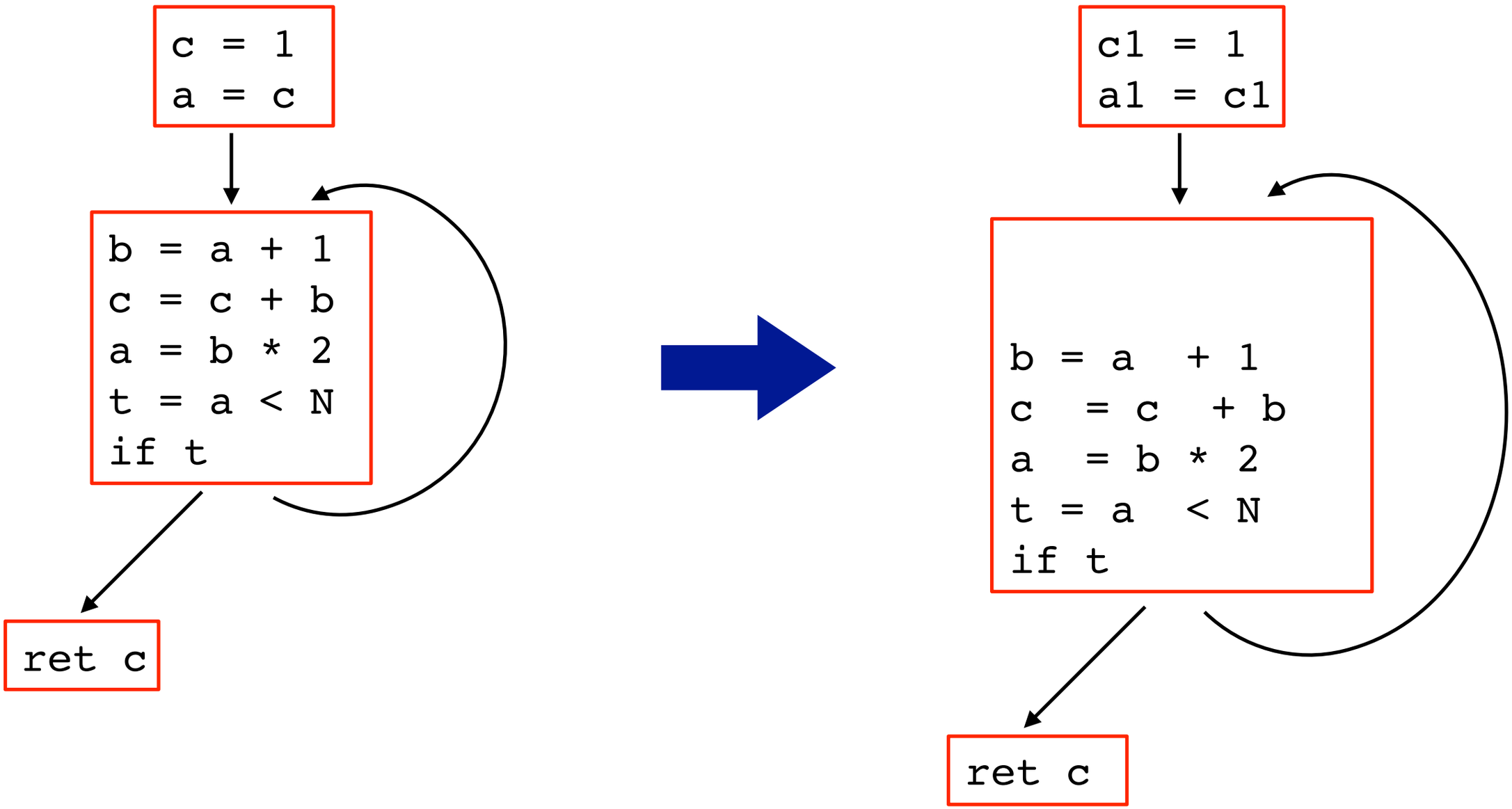
Exercice

- mettre ce programme sous une forme SSA



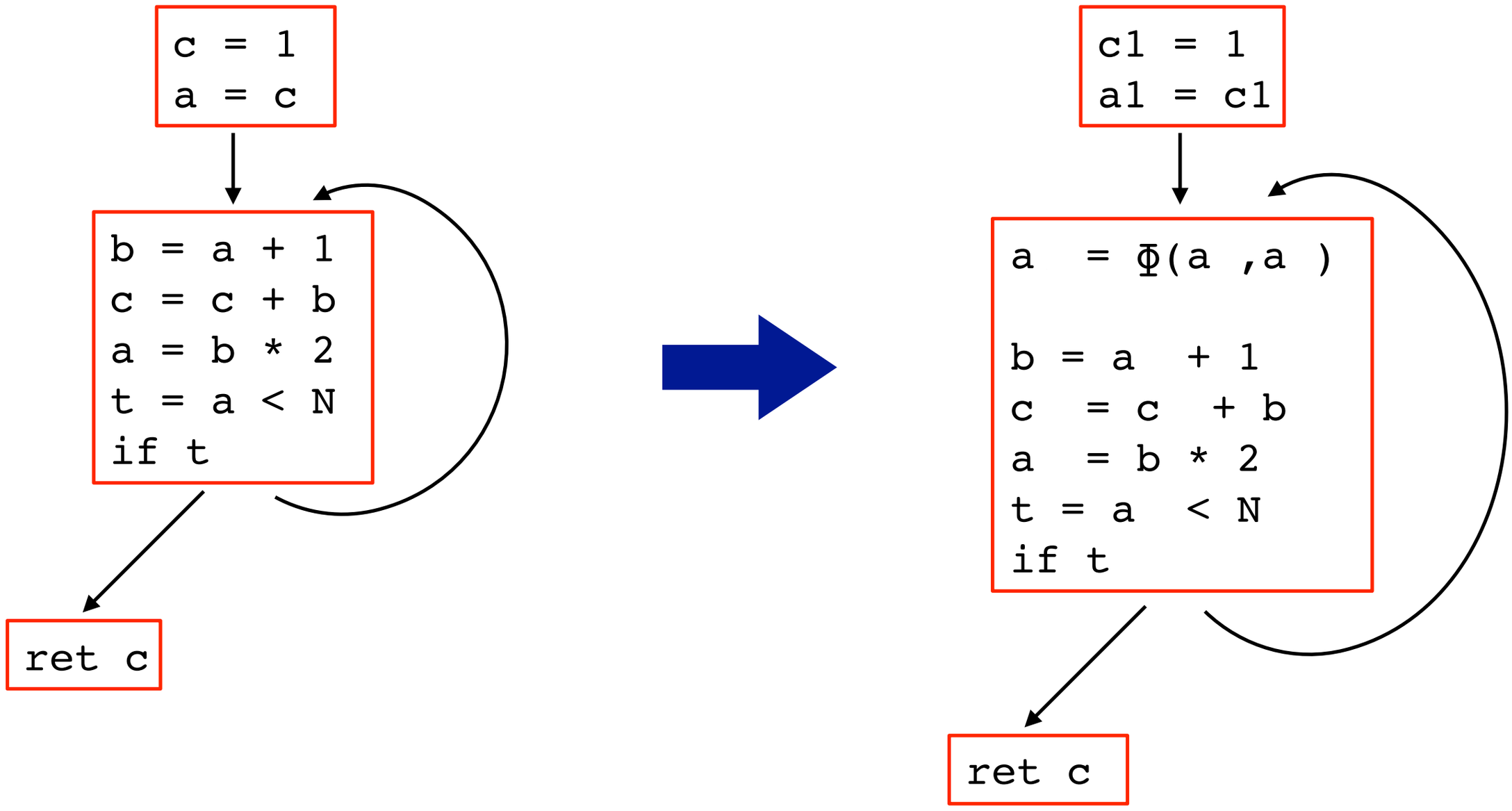
Exercice

- mettre ce programme sous une forme SSA



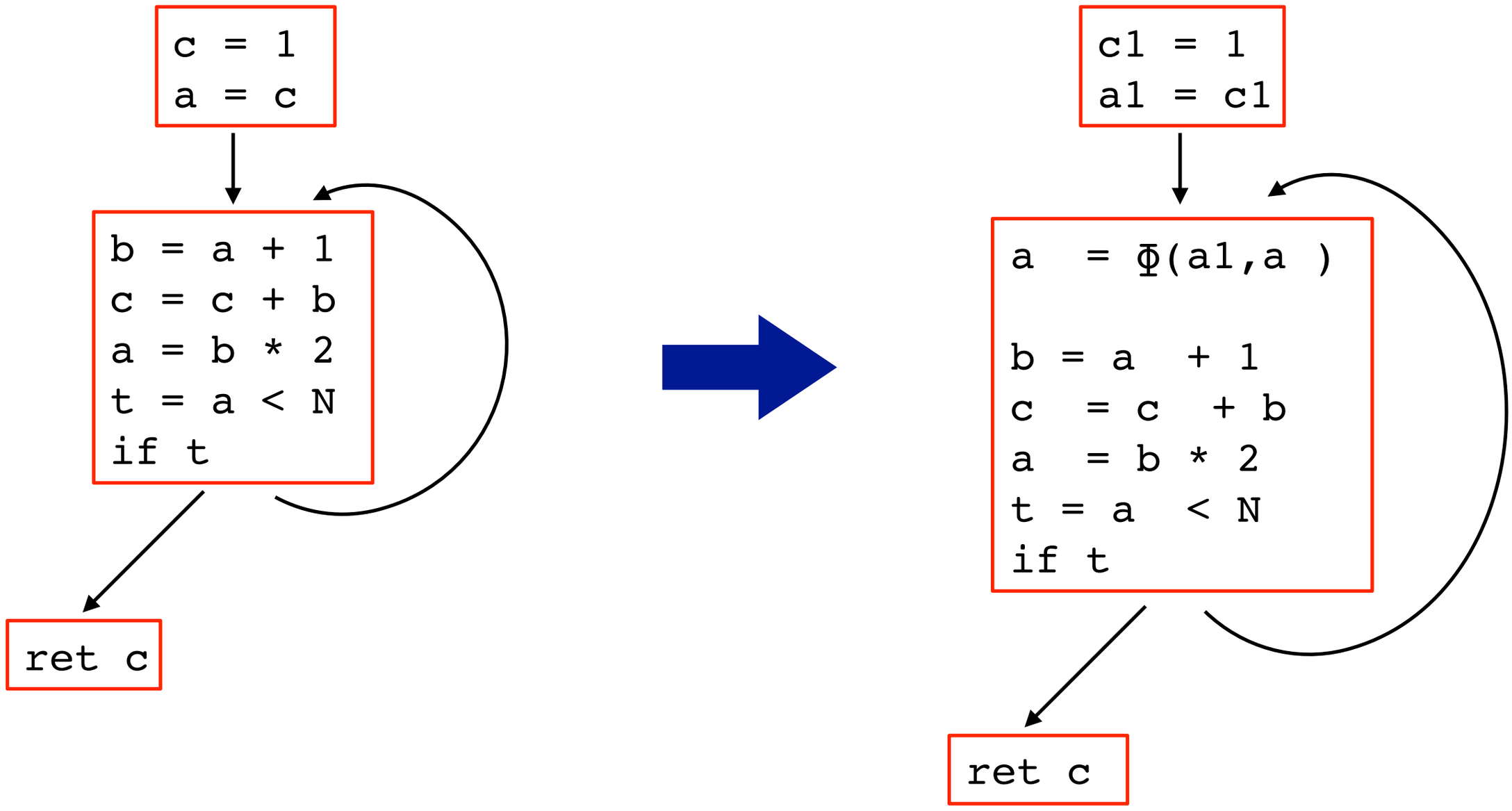
Exercice

- mettre ce programme sous une forme SSA



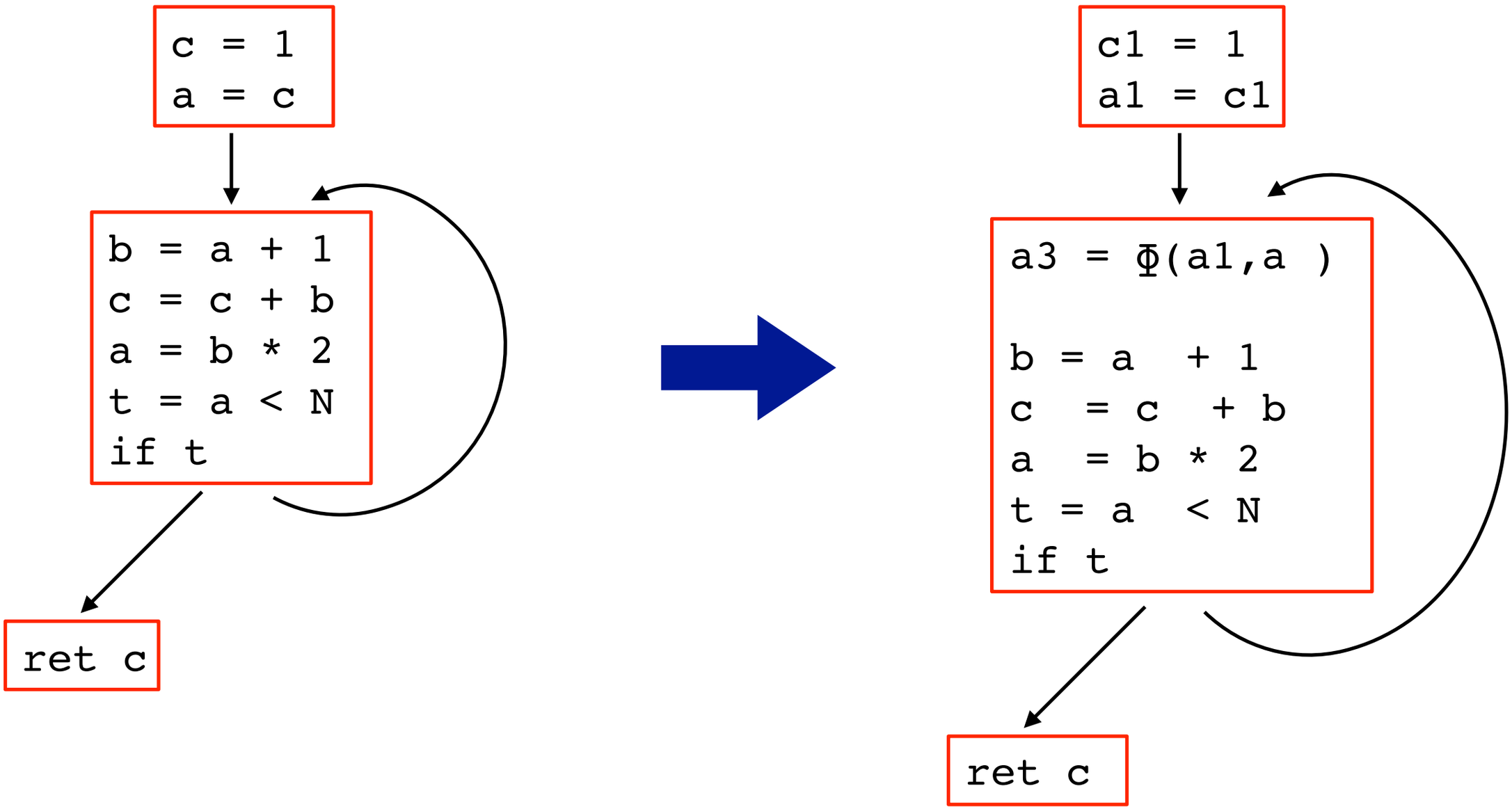
Exercice

- mettre ce programme sous une forme SSA



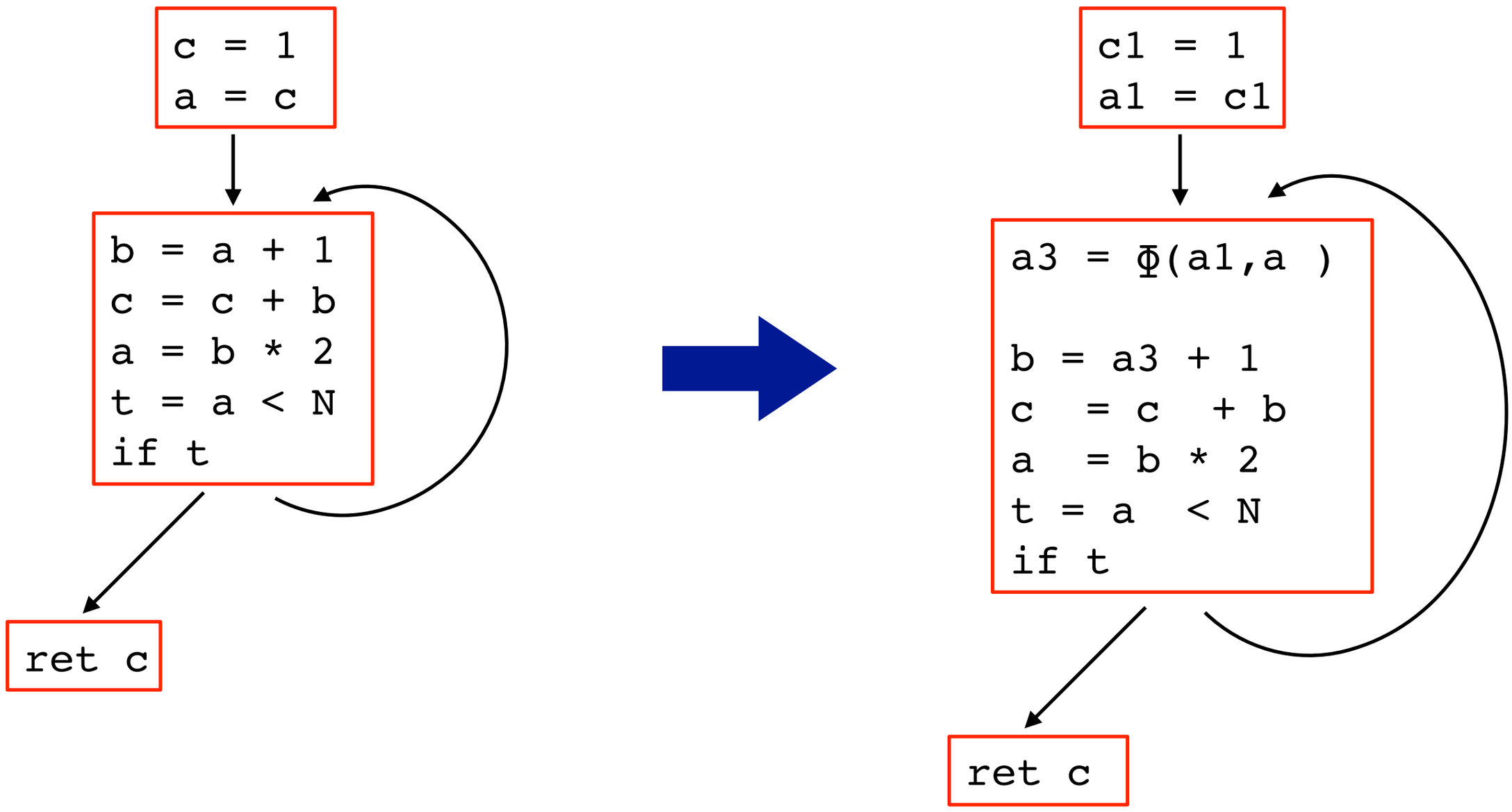
Exercice

- mettre ce programme sous une forme SSA



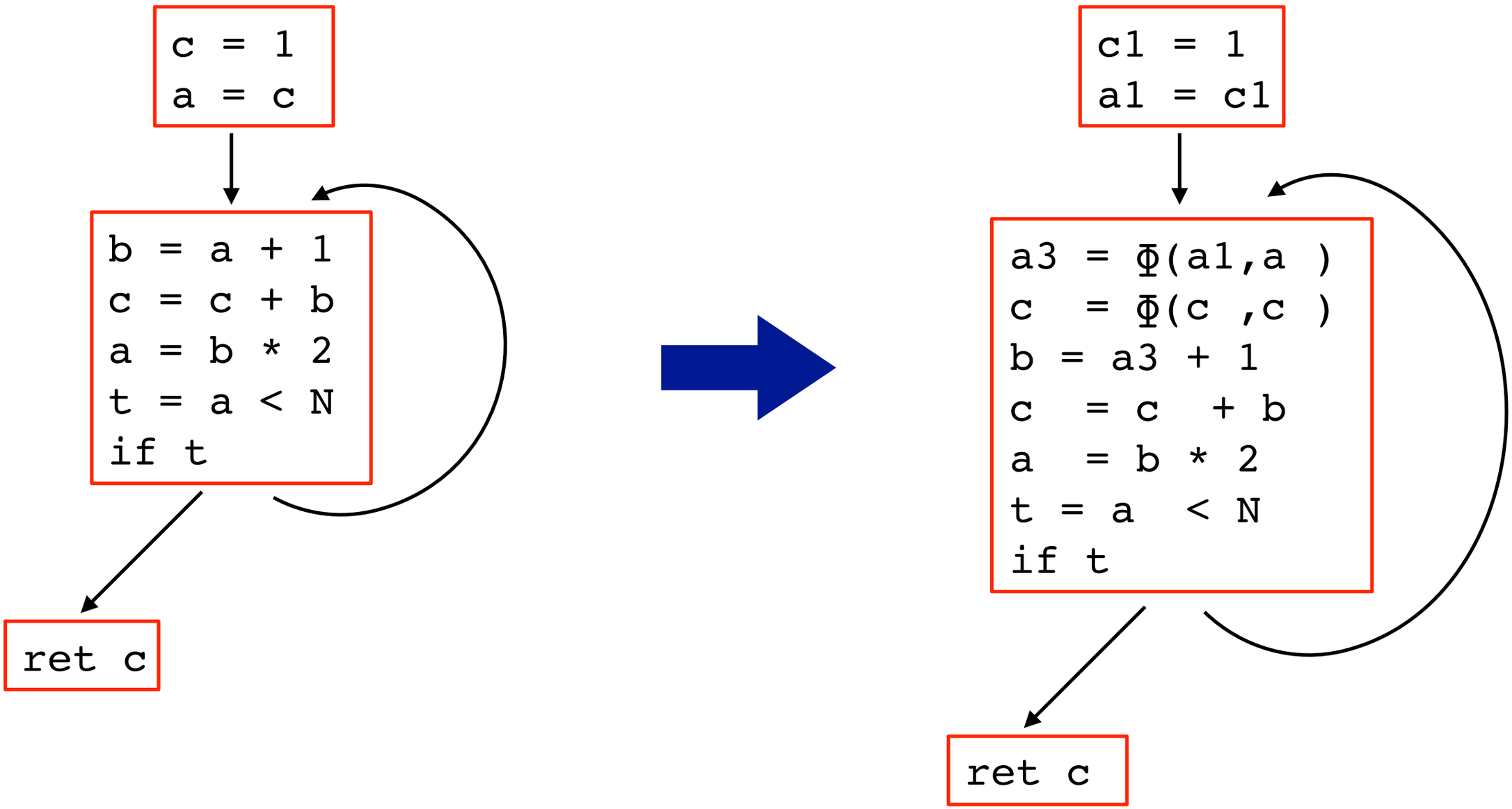
Exercice

- mettre ce programme sous une forme SSA



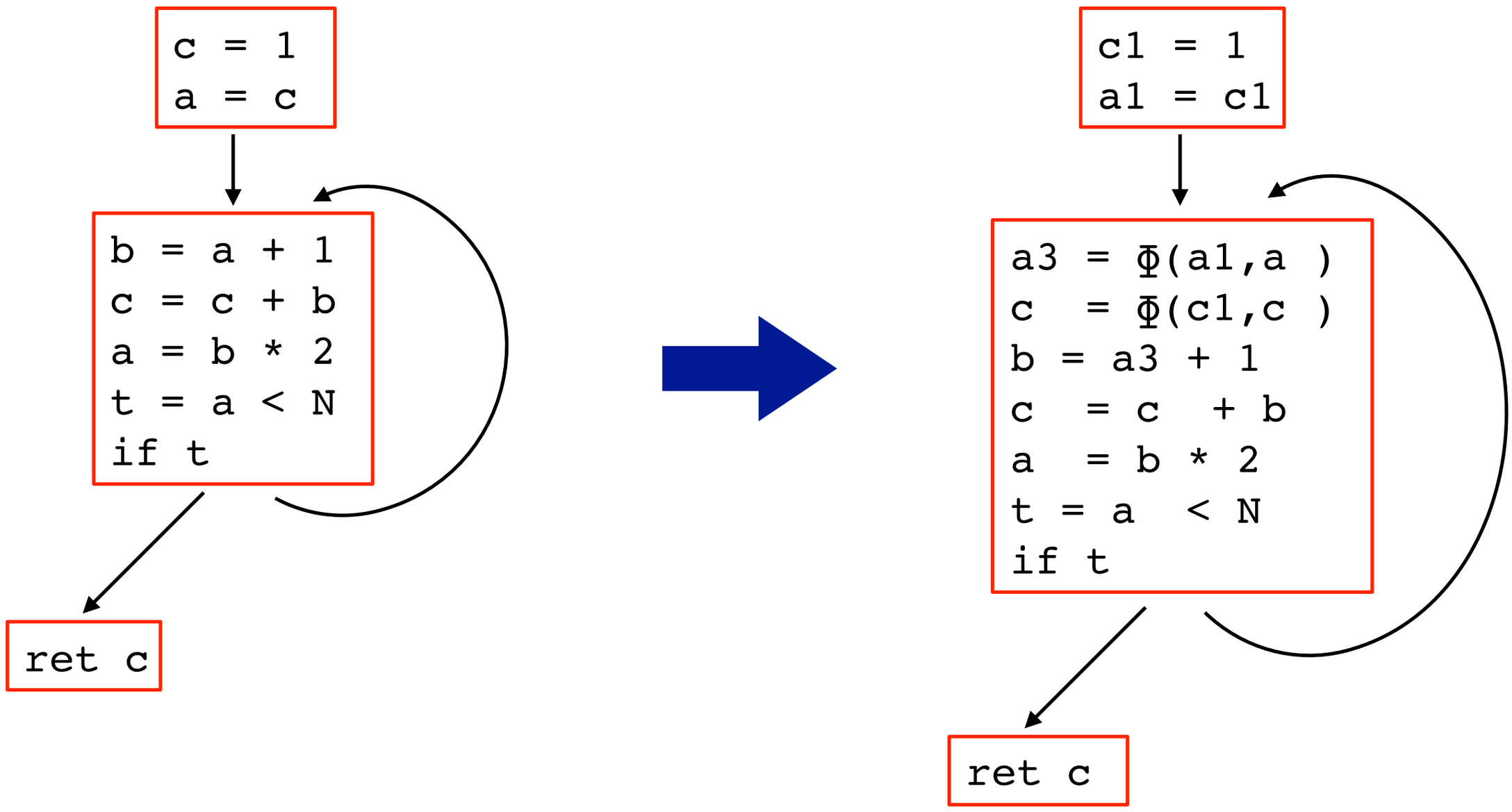
Exercice

- mettre ce programme sous une forme SSA



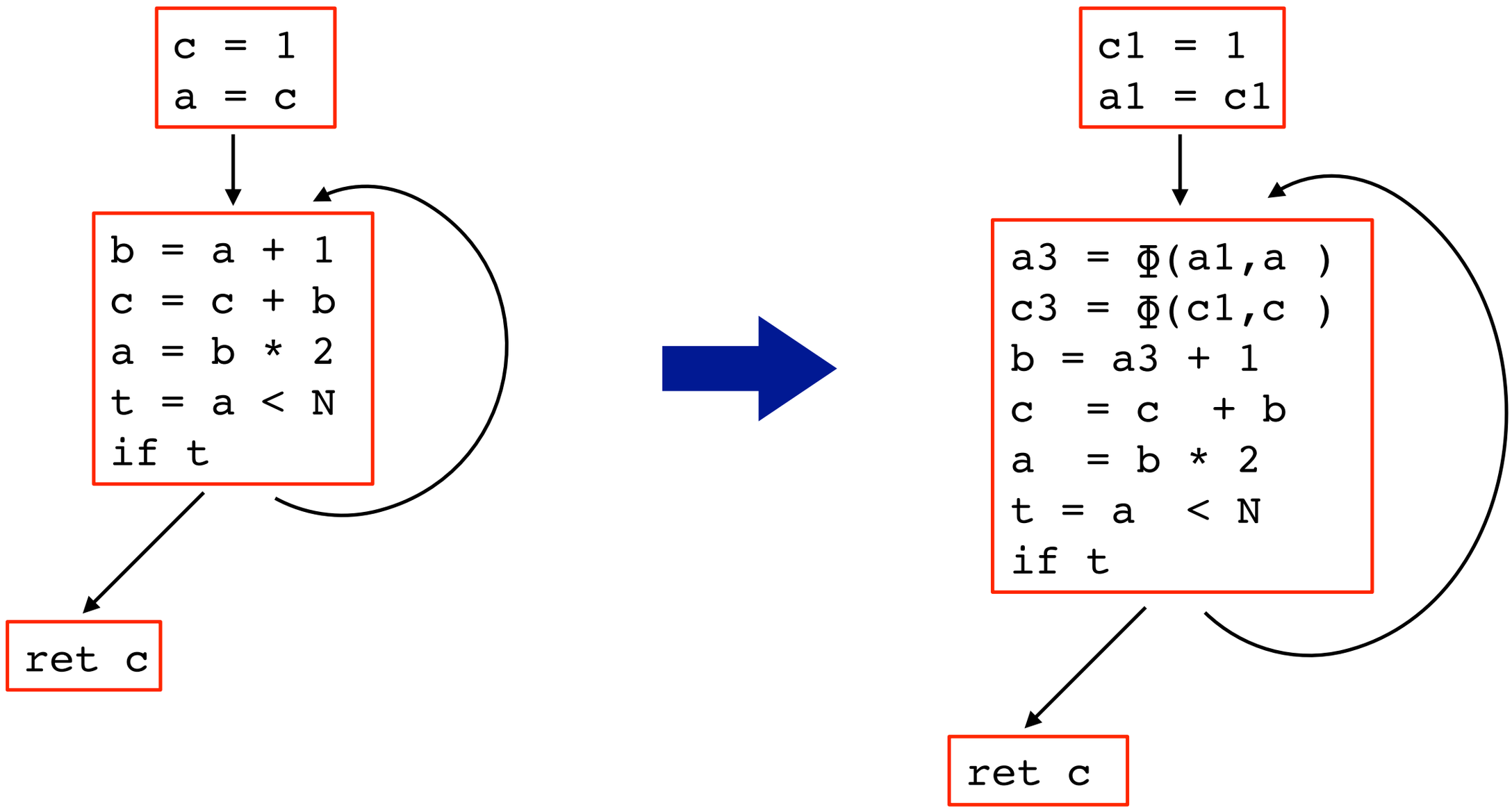
Exercice

- mettre ce programme sous une forme SSA



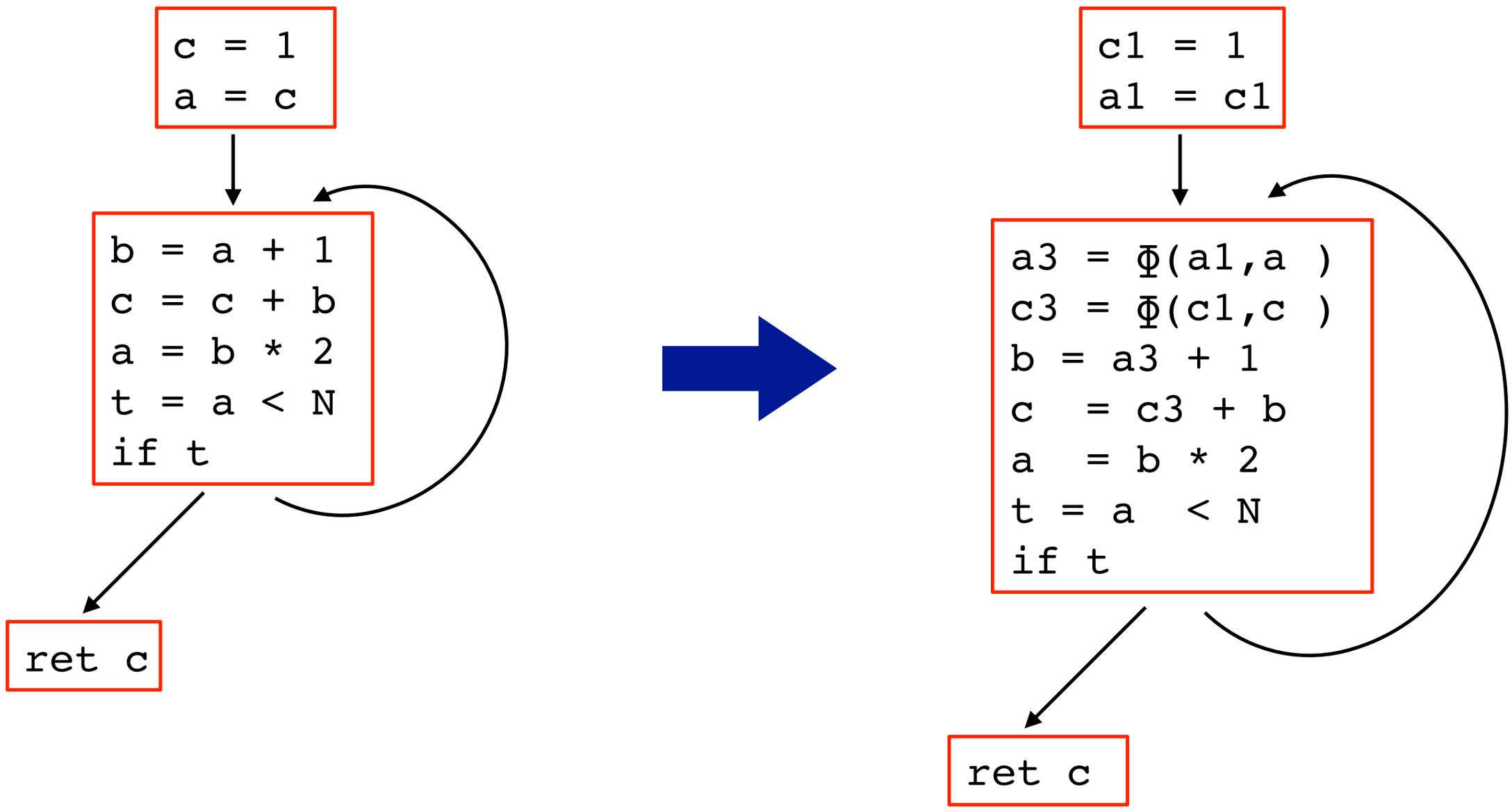
Exercice

- mettre ce programme sous une forme SSA



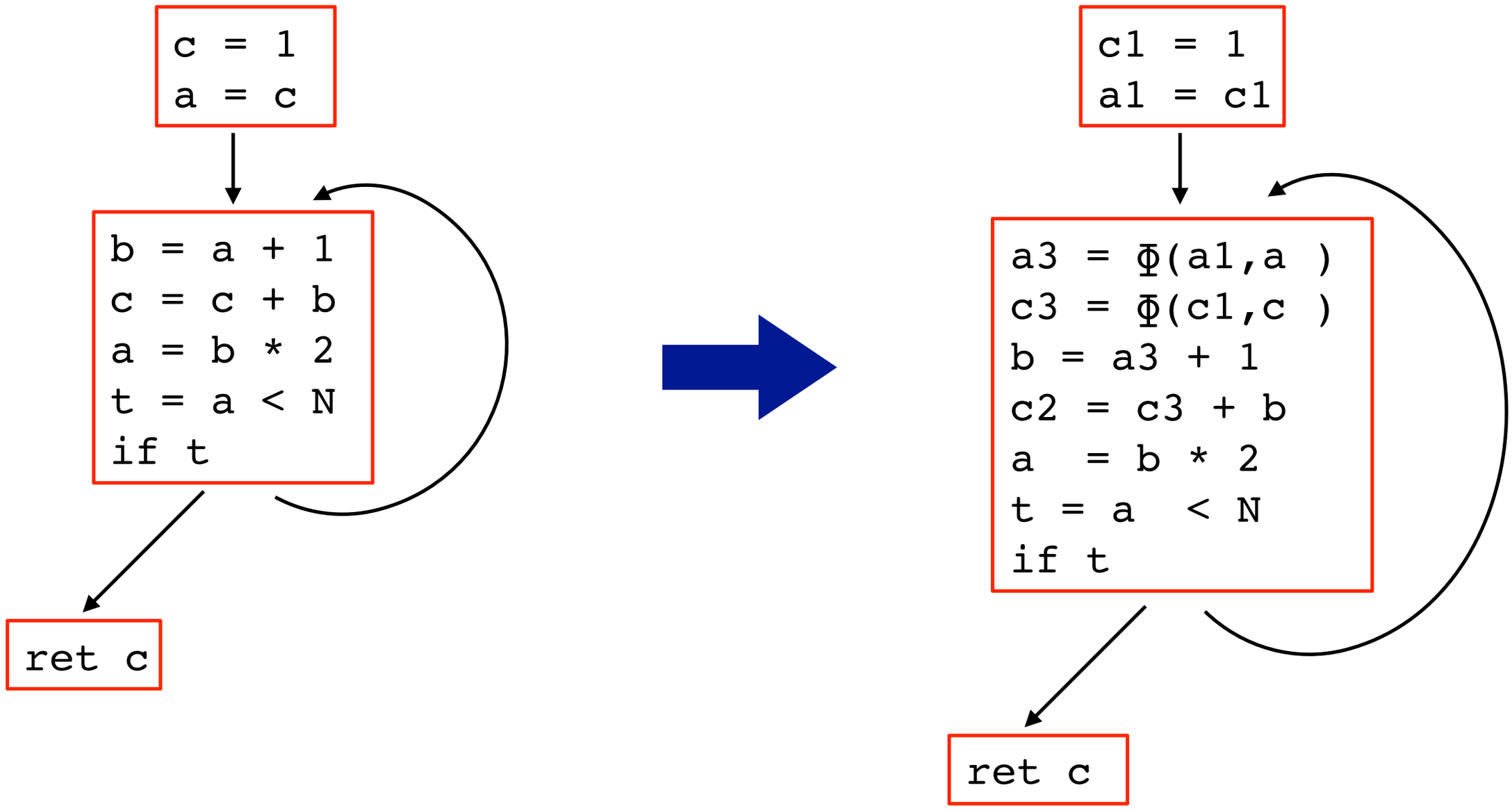
Exercice

- mettre ce programme sous une forme SSA



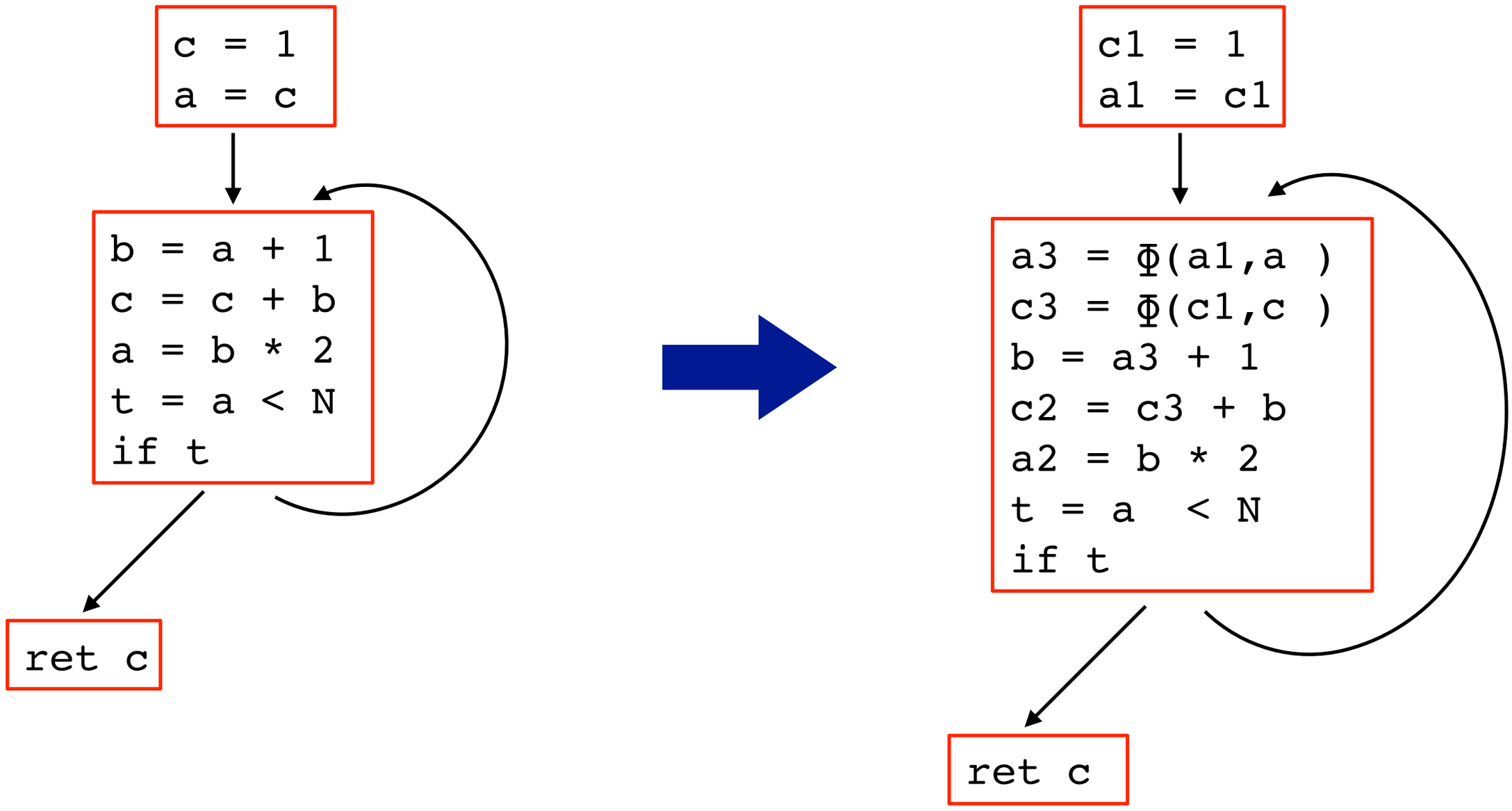
Exercice

- mettre ce programme sous une forme SSA



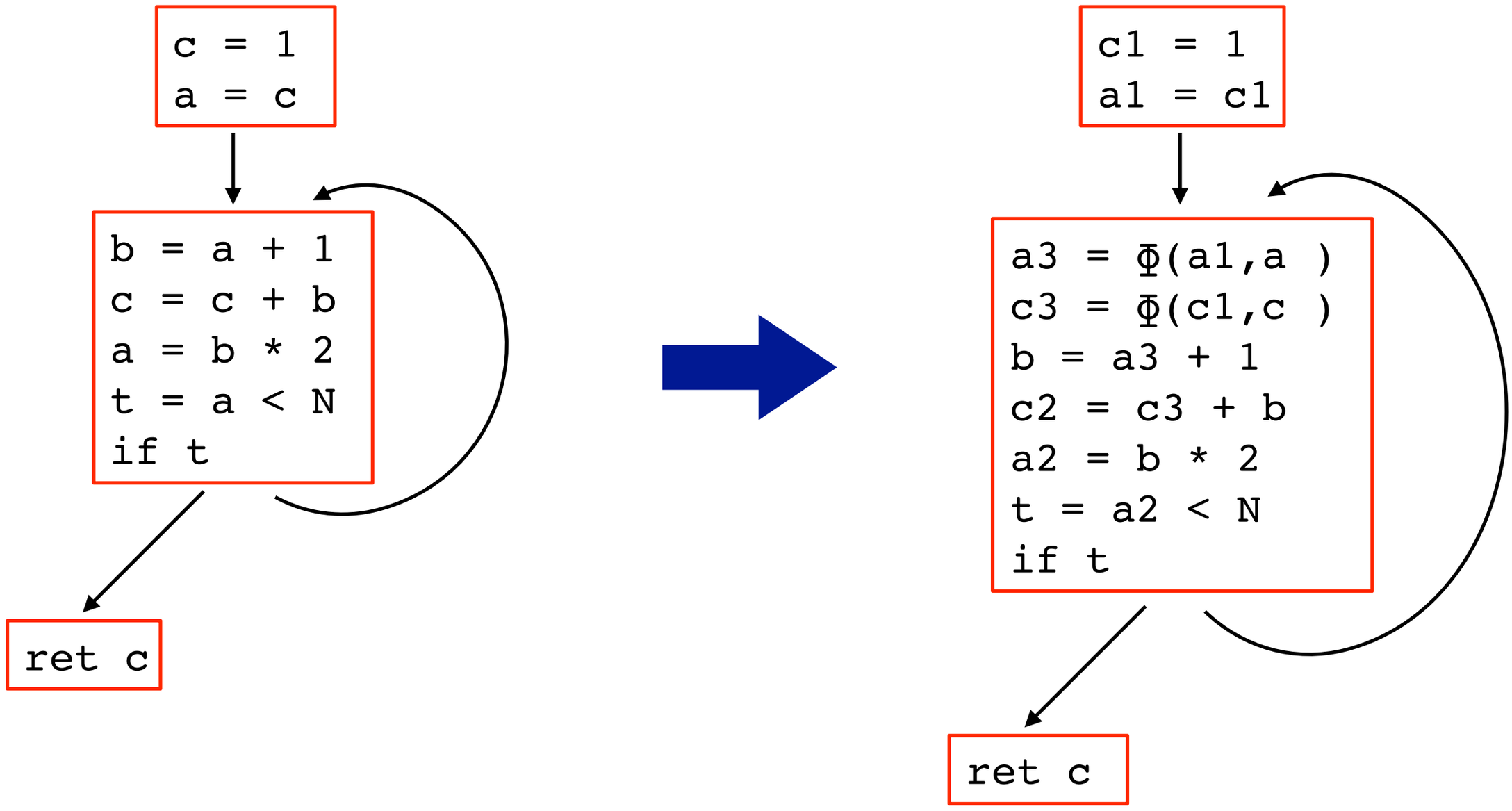
Exercice

- mettre ce programme sous une forme SSA



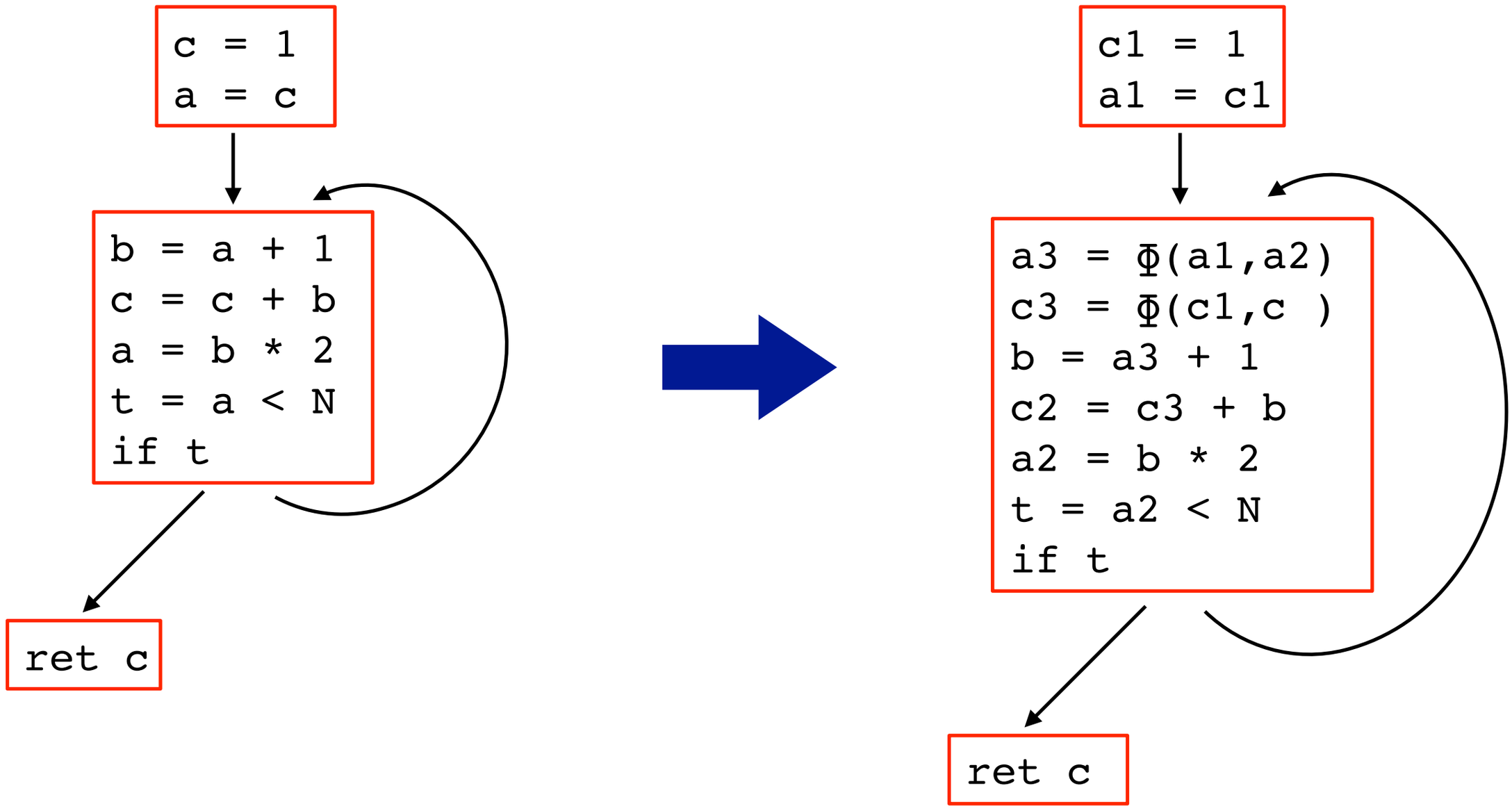
Exercice

- mettre ce programme sous une forme SSA



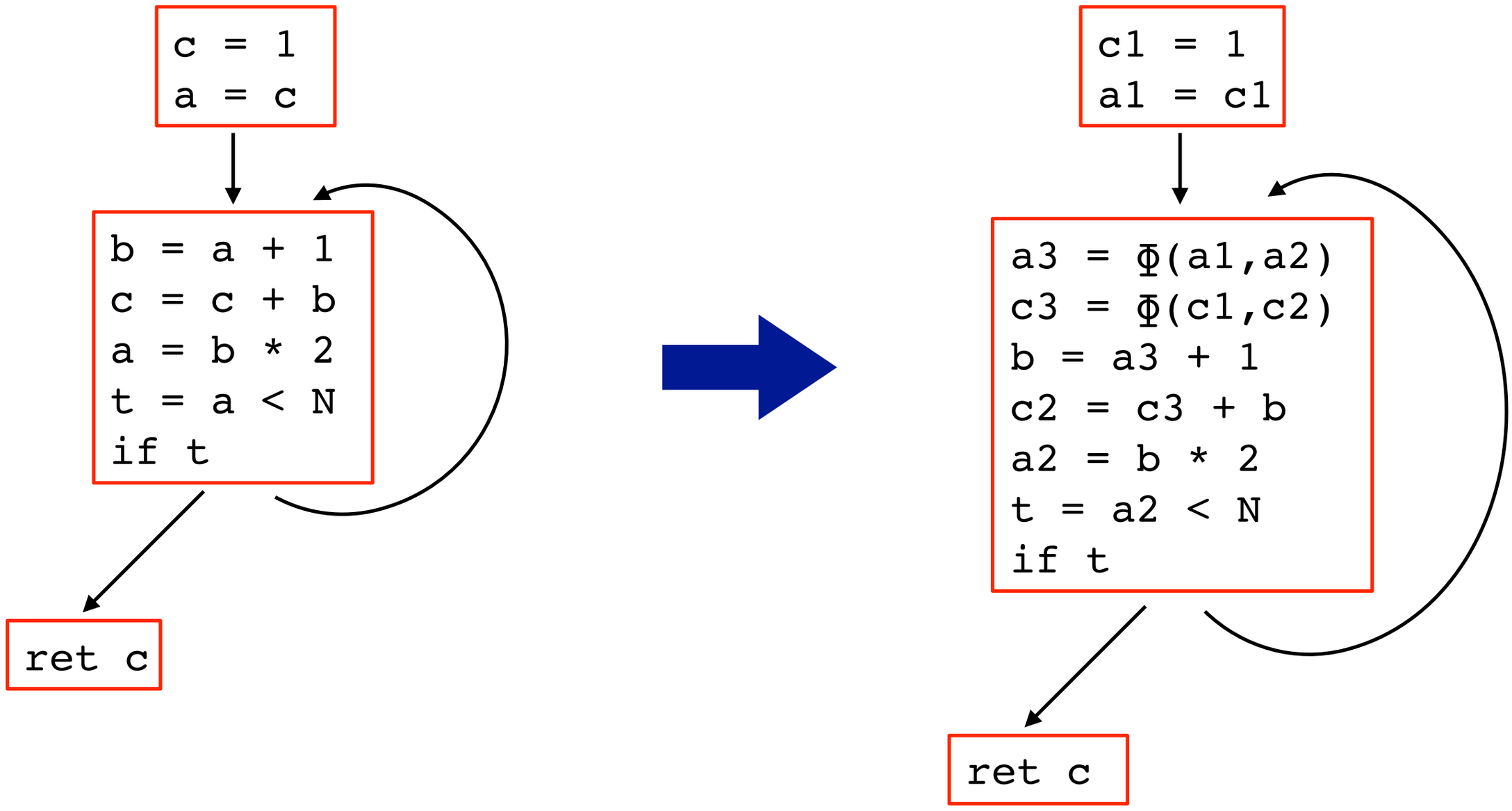
Exercice

- mettre ce programme sous une forme SSA



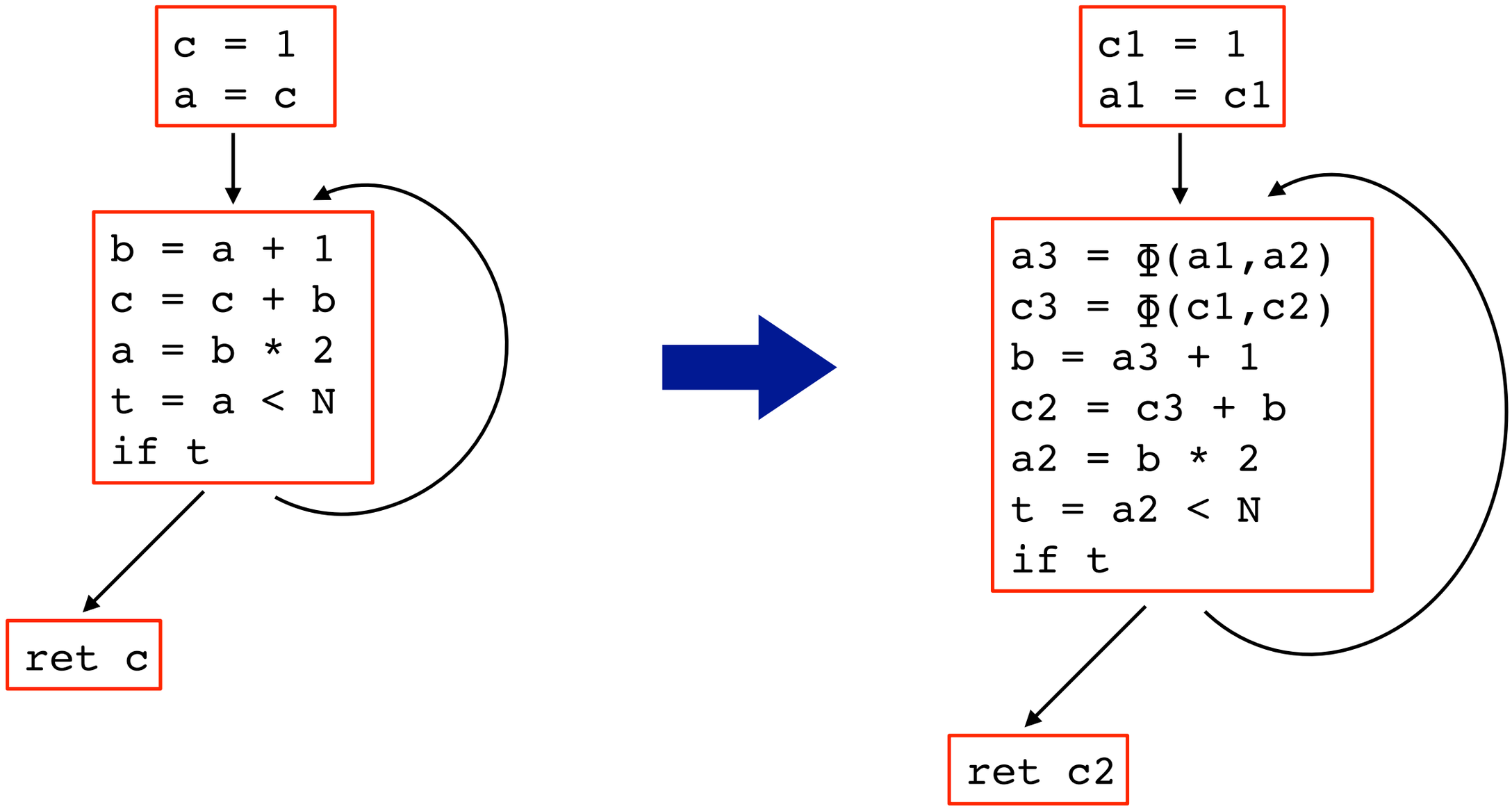
Exercice

- mettre ce programme sous une forme SSA



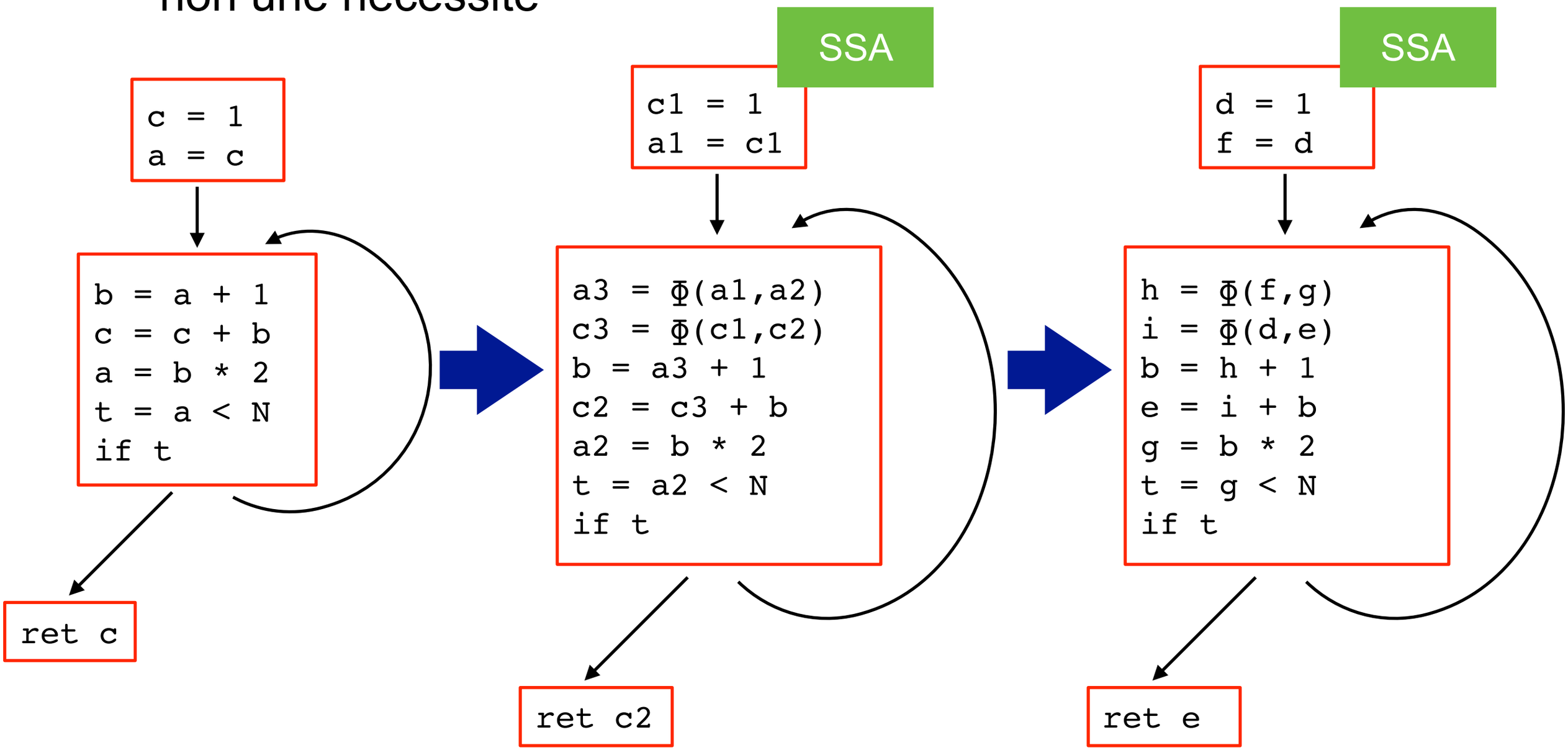
Exercice

- mettre ce programme sous une forme SSA



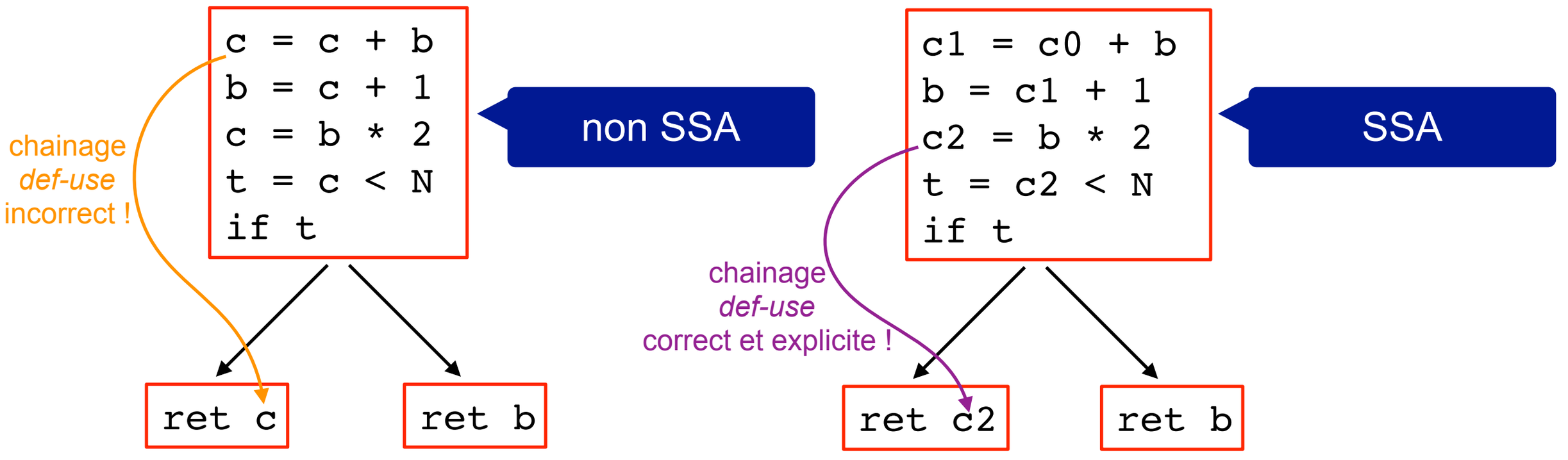
Remarque

- la *numérotation* est juste une technique de mise en forme SSA, et non une nécessité



SSA : quel interêt ?

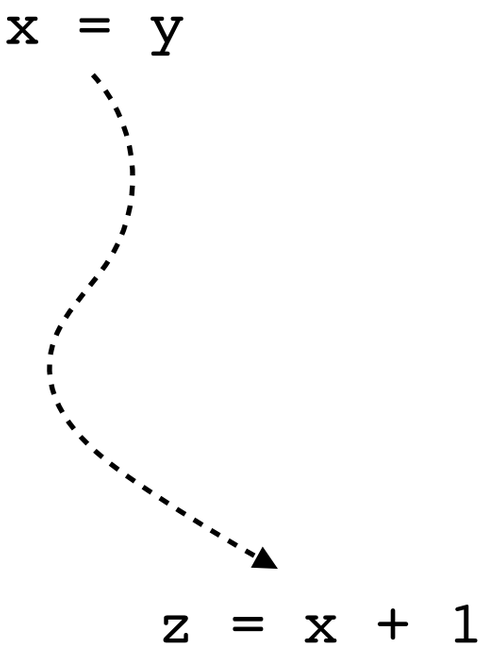
- chaque variable a une unique **définition accessible**
- les chaînes *use-def* deviennent triviales : on peut associer à chaque véritable son unique point de définition
- les chaînes *def-use* deviennent explicites : les occurrences syntaxiques donnent des chaînages corrects



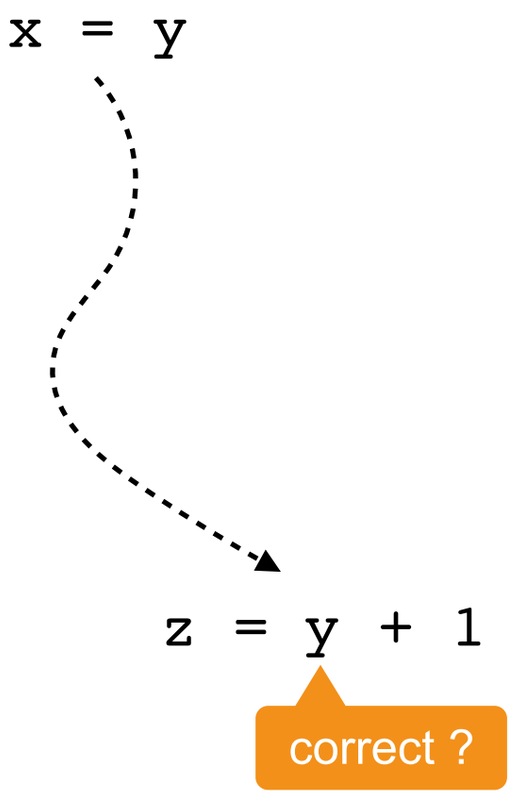
SSA : propagation des copies

- sans forme SSA, une copie `x = y` ne peut être propagée sur un usage de `x` (en remplaçant `x` par `y`) que si, sur tout les chemins menant de la copie à l'usage (sans repasser par la copie), ni `x` ni `y` ne sont redéfinis

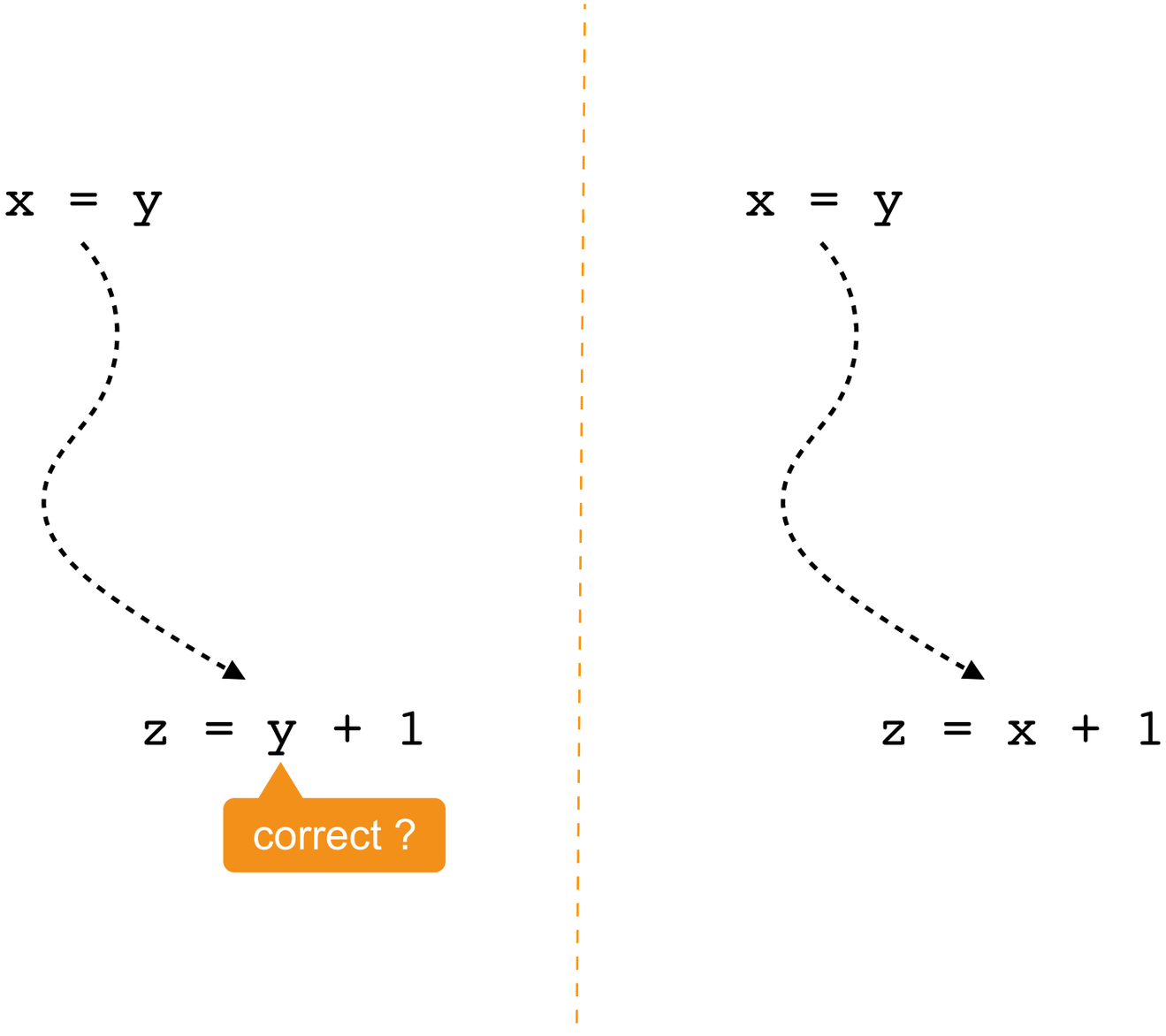
SSA : propagation des copies



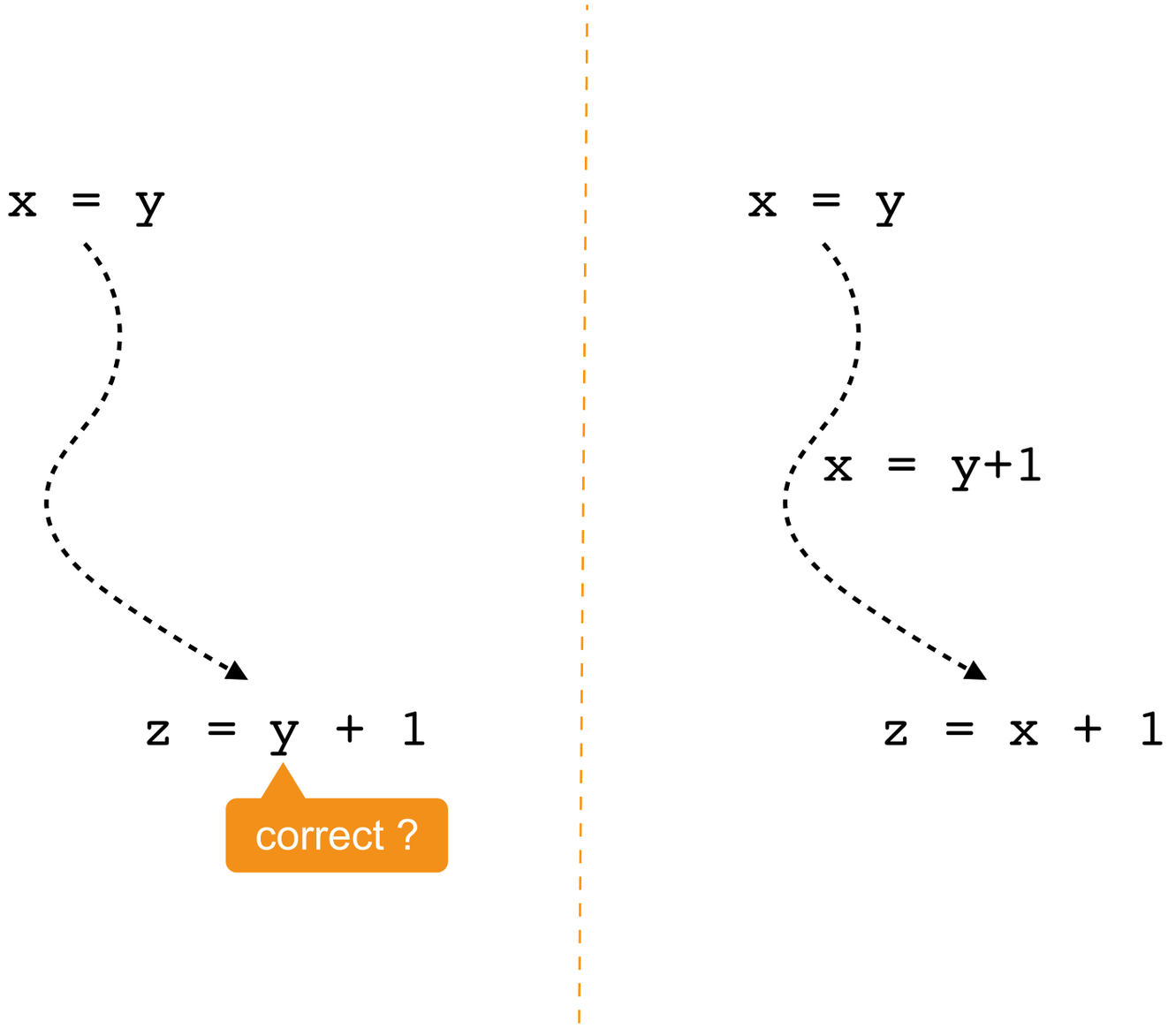
SSA : propagation des copies



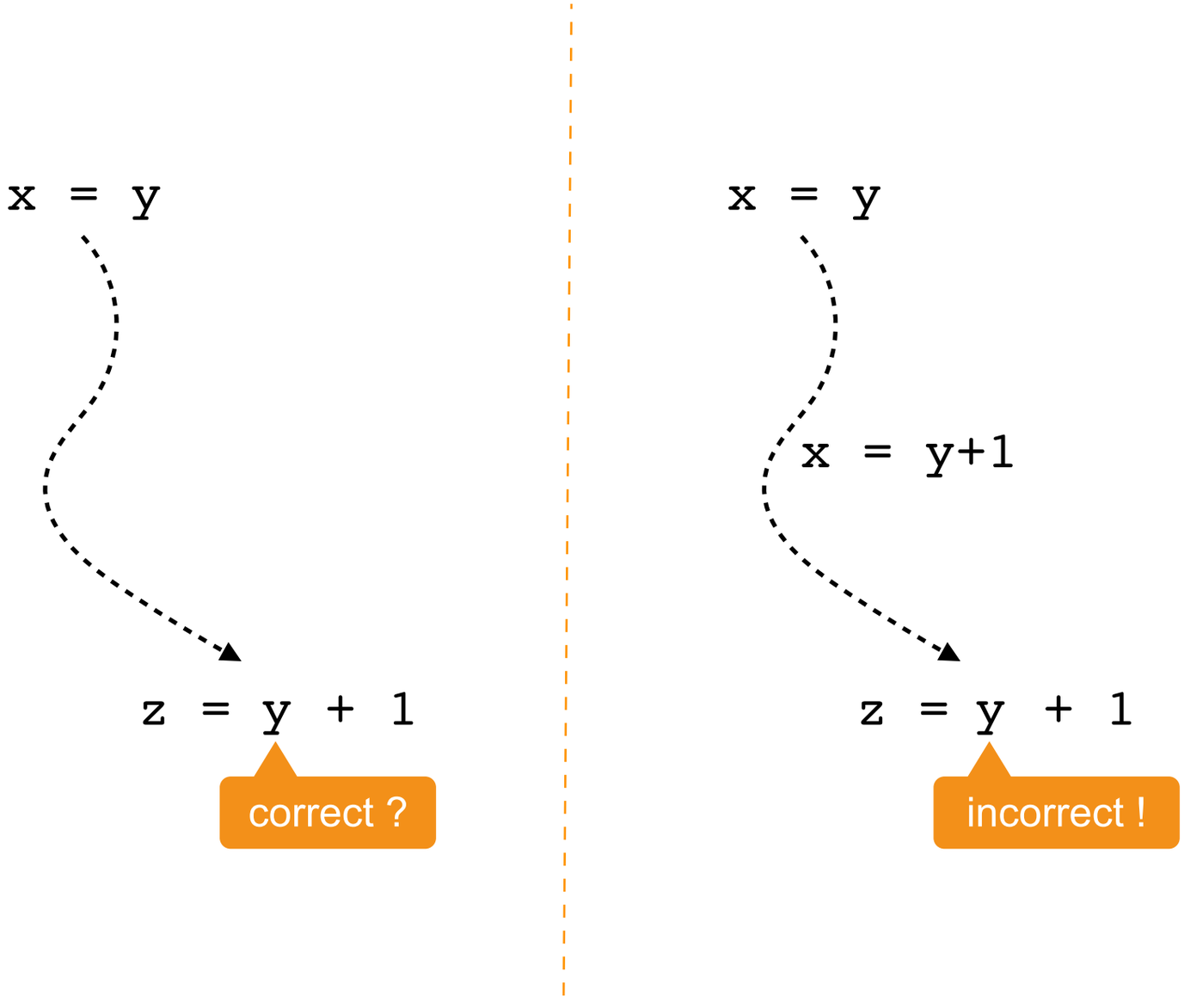
SSA : propagation des copies



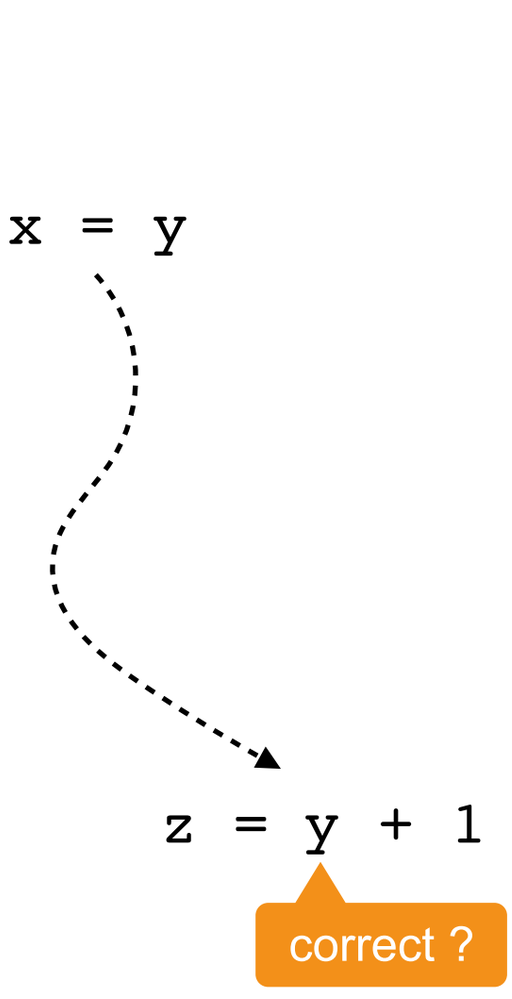
SSA : propagation des copies

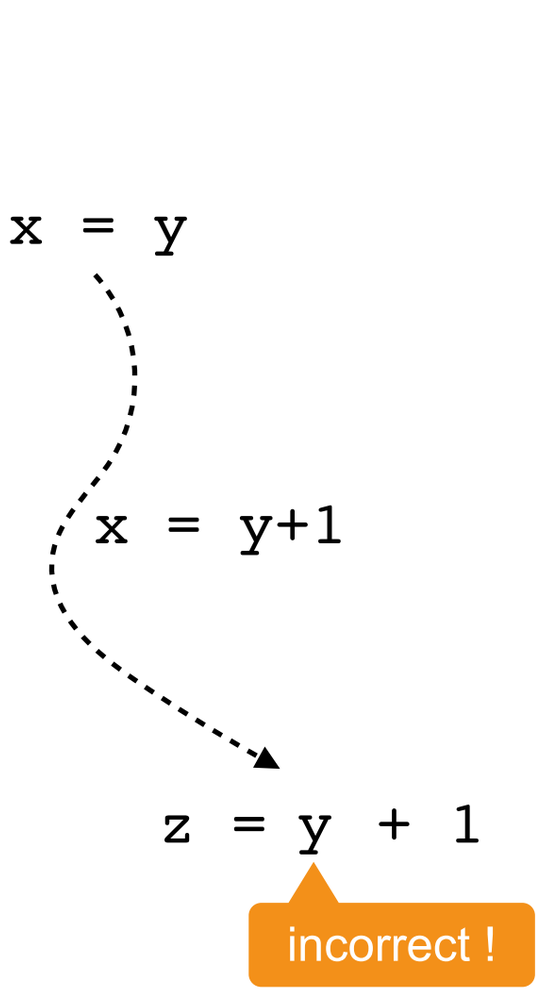


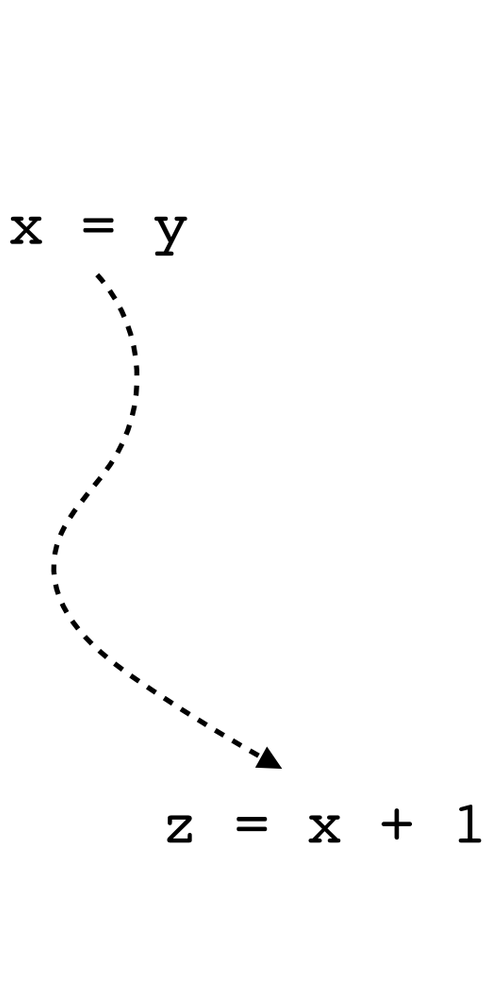
SSA : propagation des copies



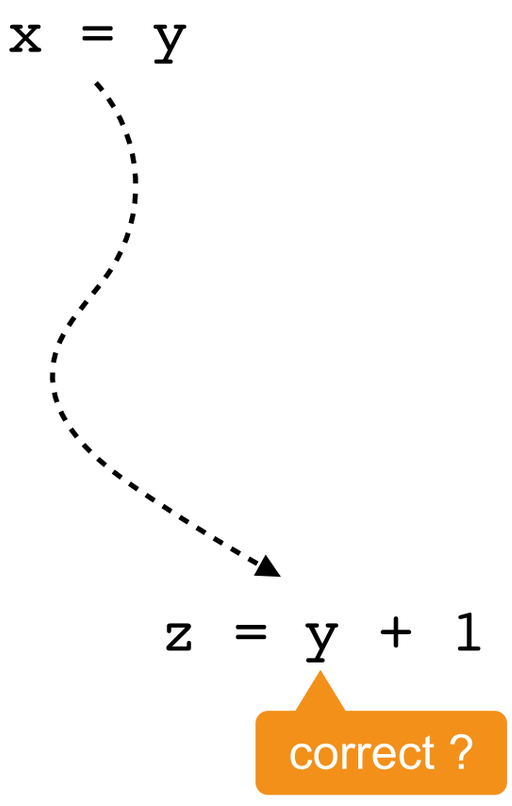
SSA : propagation des copies

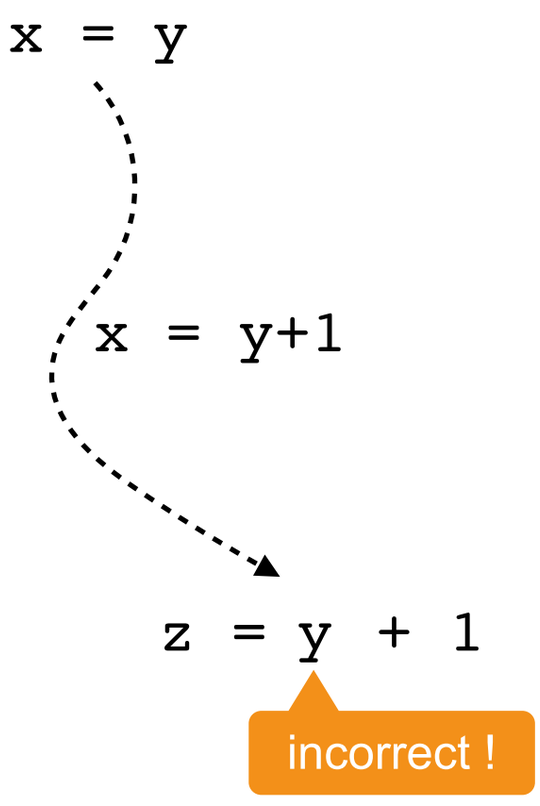


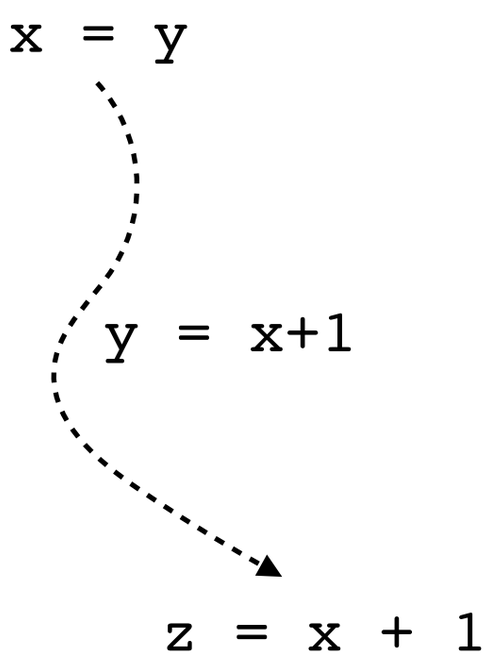




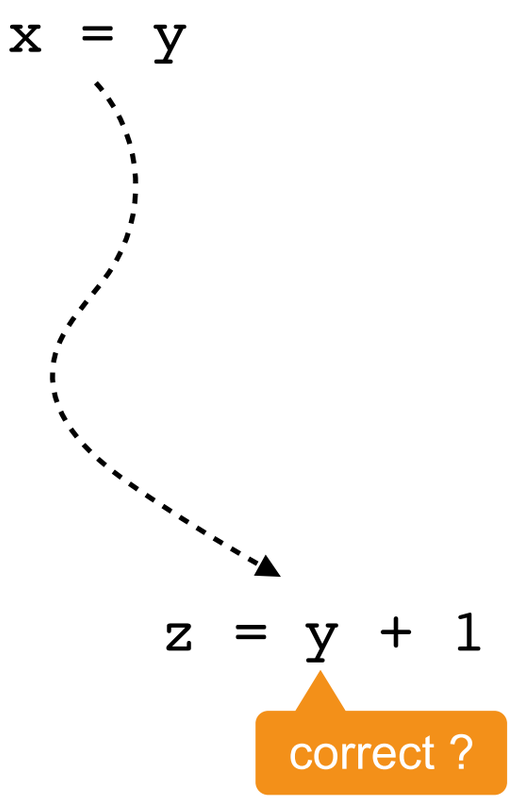
SSA : propagation des copies

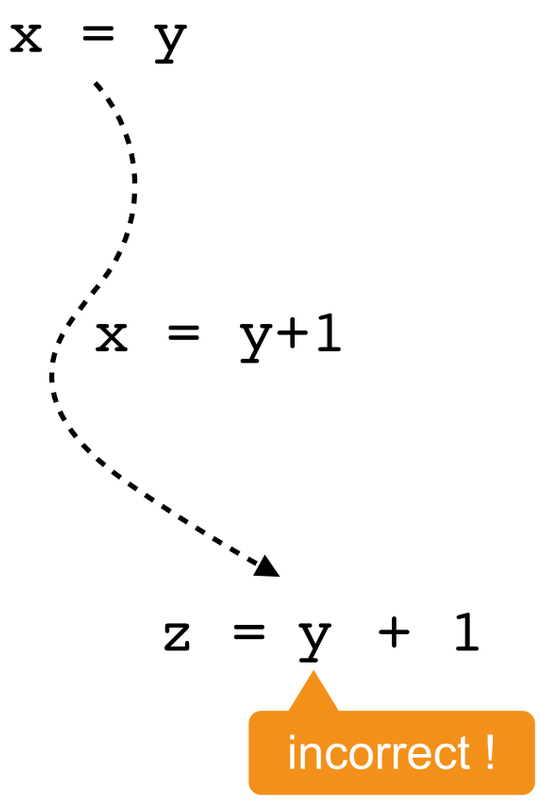


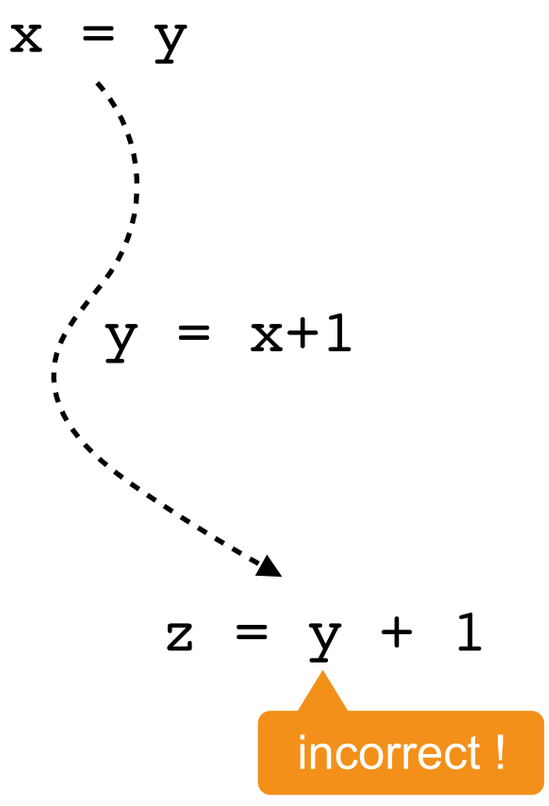




SSA : propagation des copies







SSA : propagation des copies

- sans forme SSA, une copie `x = y` ne peut être propagée sur un usage de `x` (en remplaçant `x` par `y`) que si, sur tout les chemins menant de la copie à l'usage (sans repasser par la copie), ni `x` ni `y` ne sont redéfinis

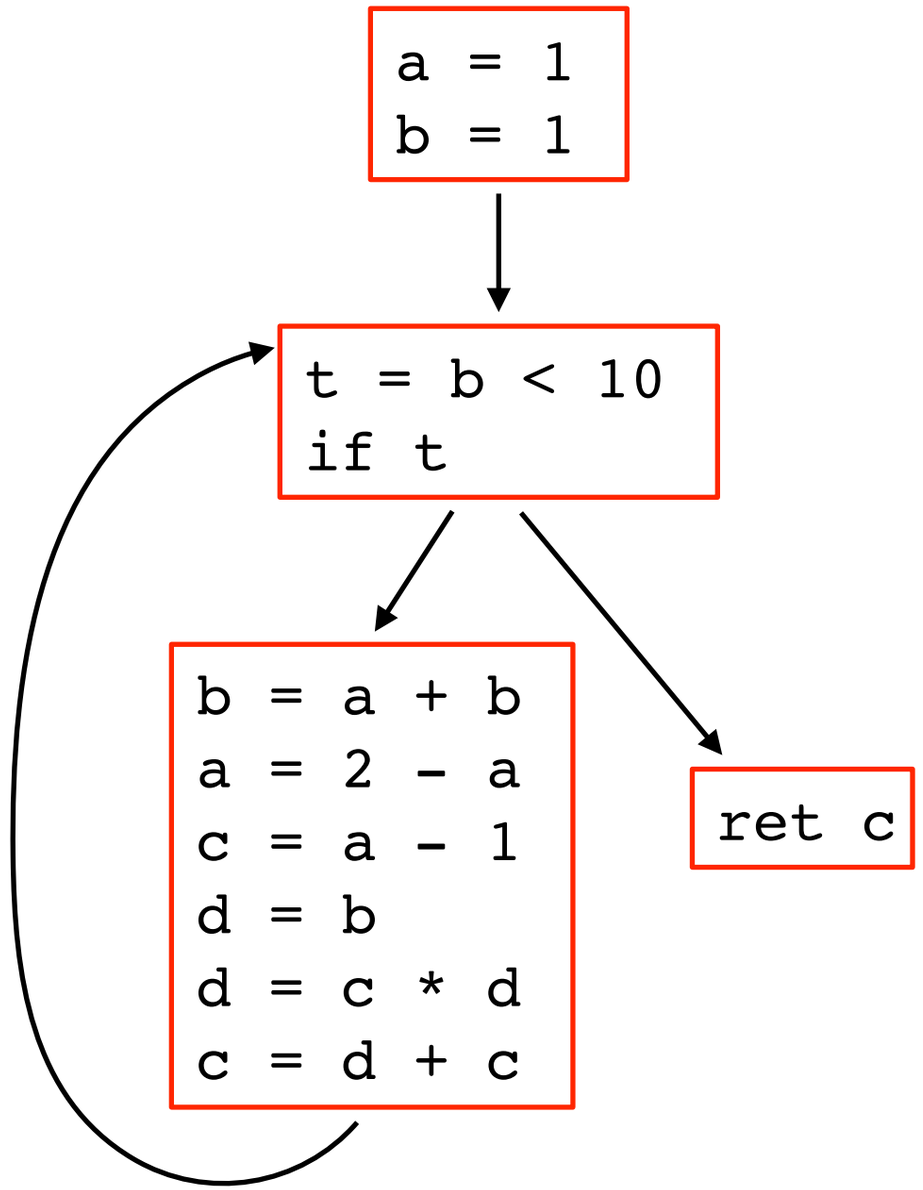
- en forme SSA,

- une copie `x = y` peut être propagée (en remplaçant `x` par `y`) vers toutes les apparitions syntaxiques de `x`

- ces propagations successives peuvent donner lieu à des instructions `z = Φ(y,y)` que l'on peut simplifier en une copie `z = y`, puis on peut itérer le procédé

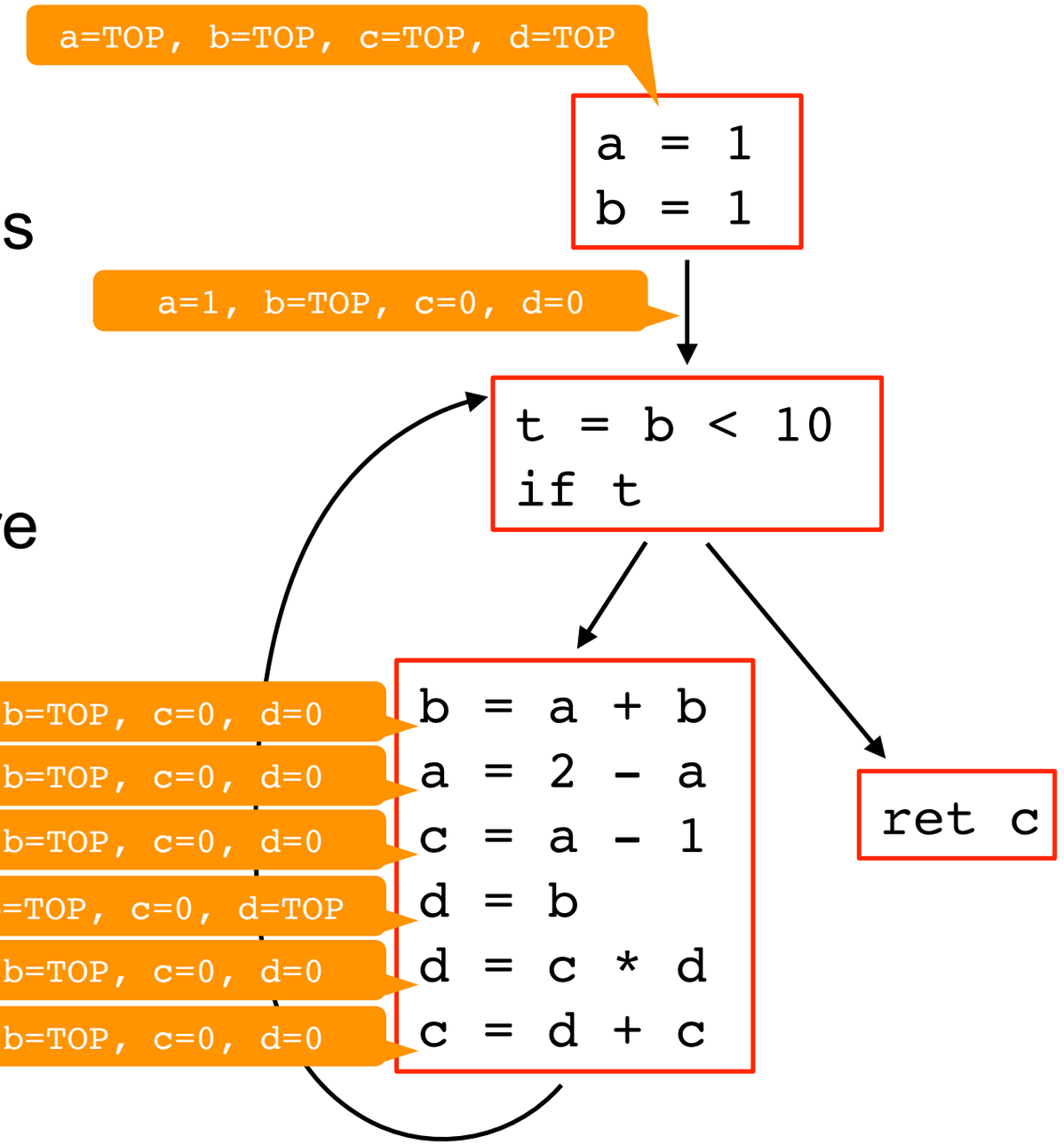
SSA : propagation des constantes

- l'analyse vue au dernier cours était *flow-sensitive* : nous avons calculé une information en chaque noeud du programme
- la forme SSA va nous permettre de calculer moins de choses, sans perdre de précision



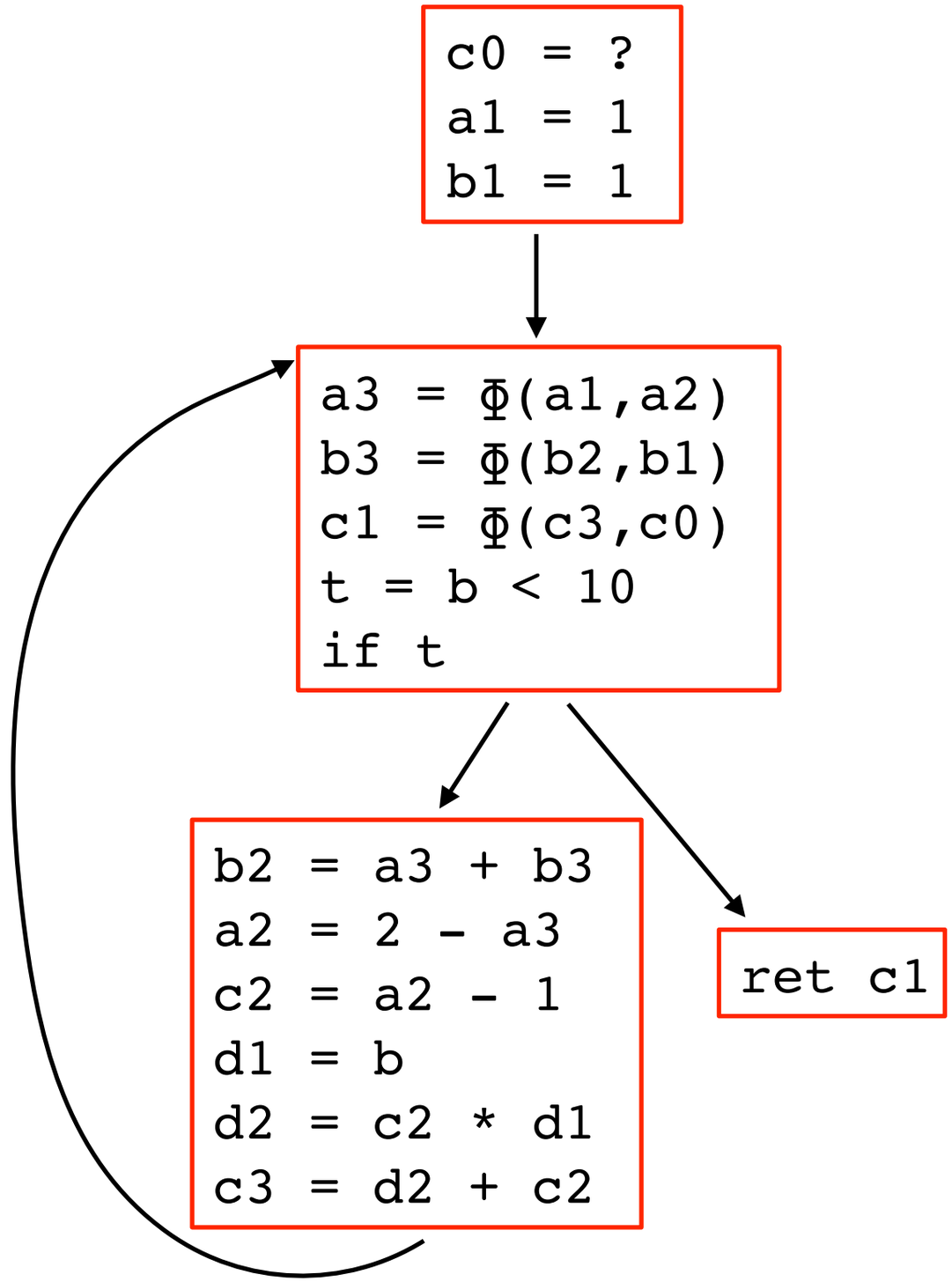
SSA : propagation des constantes

- l'analyse vue au dernier cours était *flow-sensitive* : nous avons calculé une information en chaque noeud du programme
- la forme SSA va nous permettre de calculer moins de choses, sans perdre de précision



SSA : propagation des constantes

- l'analyse vue au dernier cours était *flow-sensitive* : nous avons calculé une information en chaque noeud du programme
- la forme SSA va nous permettre de calculer moins de choses, sans perdre de précision

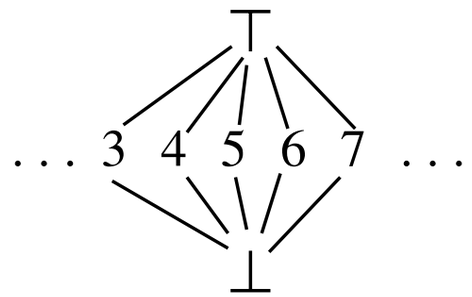



```
a1=1, a2=1, a3=1,
b1=1, b2=TOP, b3=TOP,
c0=TOP, c1=TOP, c2=0, c3=0
d1=TOP, d2=0,
t=TOP
```

SSA : propagation des constantes

- au lieu de calculer une `Map<Ident,IntOrTop>` en chaque noeud du programme, nous allons calculer une seule `Map<Ident,BotOrIntOrTop>` pour tout le programme

- le type `BotOrIntOrTop` représente le treillis



- le point fixe `Const : Map<Ident,BotOrIntOrTop>` recherché doit satisfaire les équations suivantes

```
pour tout ident x,
  si x est un paramètre, Const[x] = ⊤
  si x = cst, Const[x] = cst
  si x = y, Const[x] = Const[y]
  si x = Add(y z), Const[x] = addConst(Const[y],Const[z])
  si x = func …, Const[x] = ⊤
  si x = Φ(y,z), Const[x] = Const[y] ⊔ Const[z]
  …
```

chaque ident se voit ainsi attaché une *unique équation*

un seul cas possible

même opérateur que dans l'analyse du cours précédent

borne sup du treillis

SSA : propagation des constantes

Recherche du (plus petit) point fixe

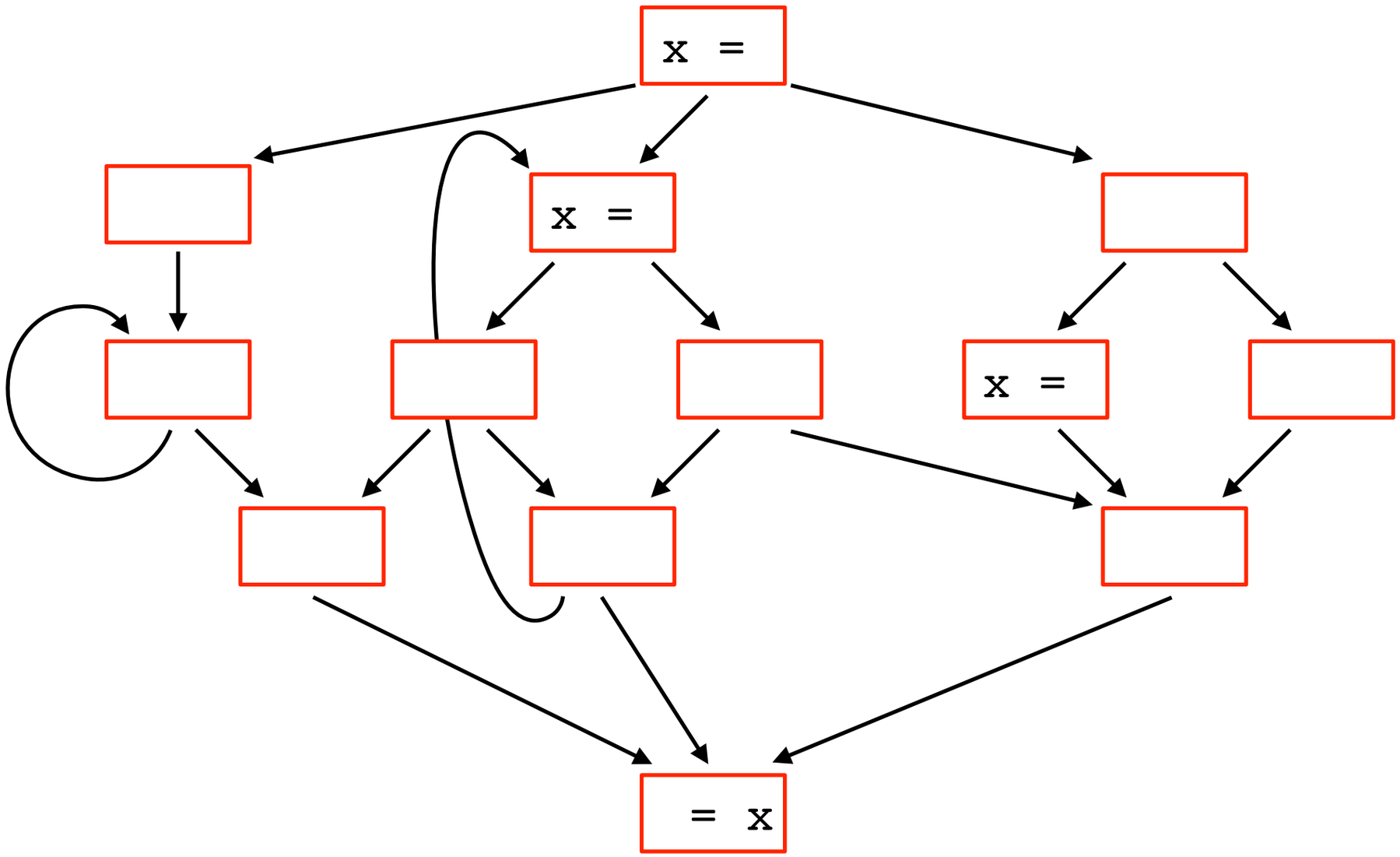
- algorithme d'itération :

```
pour tout ident x, Const[x] := ⊥
tant qu'il existe un ident x qui ne satisfait pas son équation
  Const[x] := terme de droite de l'unique équation de x
```

- on peut accélérer l'itération en tenant à jour une *worklist* `W` des identifiants à surveiller
  - au départ `W` contient uniquement les identifiants dont l'équation à un terme droit indépendant de `Const`
  - quand un identifiant `x` est mis à jour, on ajoute à `W` tous les identifiants *id* tels que *id* utilise `x` et son équation n'est pas encore assurée (on *réveille id*)
  - quand `W` est vide, le point fixe est atteint, sinon on pioche (en l'enlevant de `W`) un identifiant `x` dedans et on le met à jour

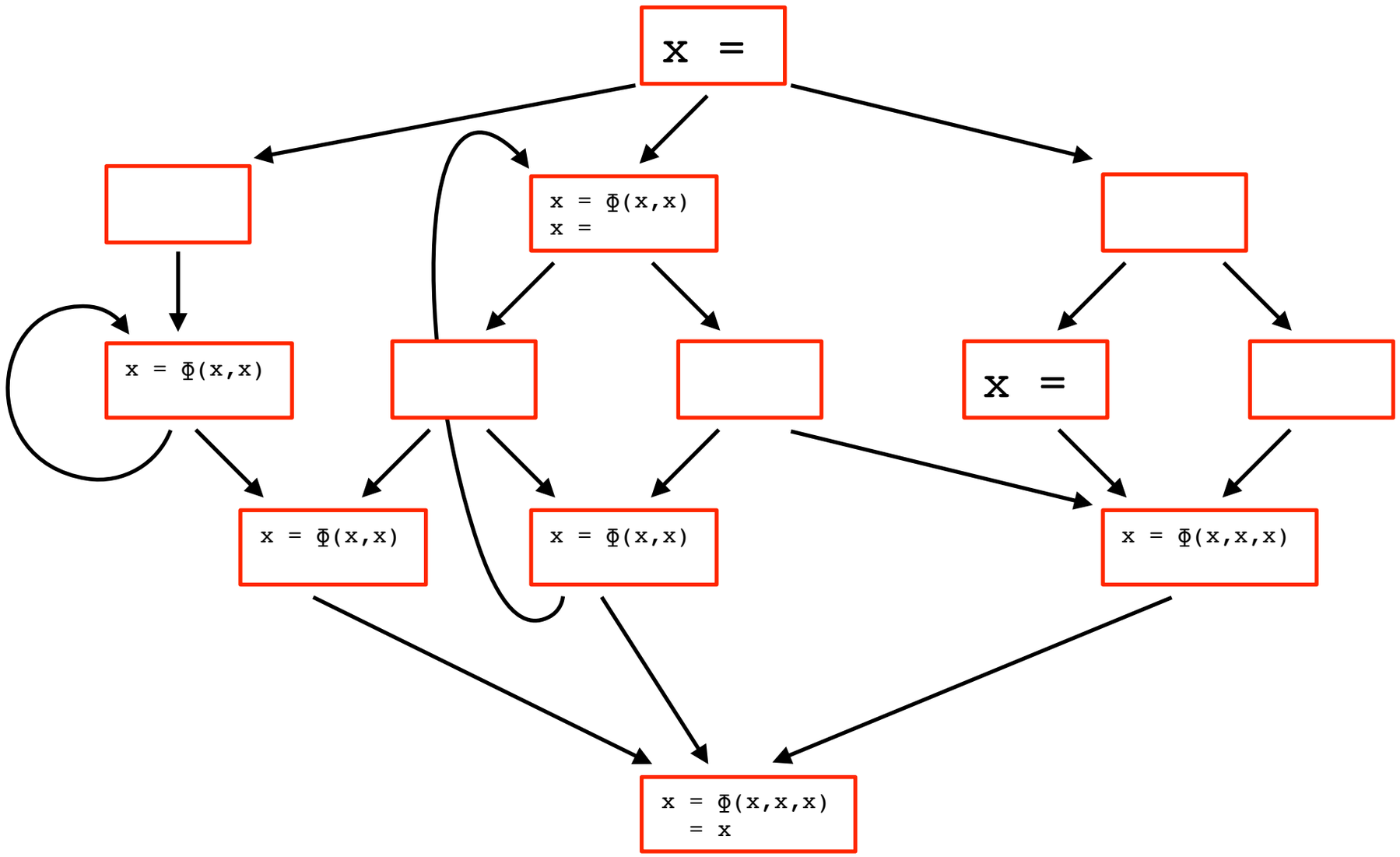
Algorithmes de mise en forme SSA

- Algorithme naif : pour chaque variable, en chaque point de jonction, on ajoute une ϕ redéfinition



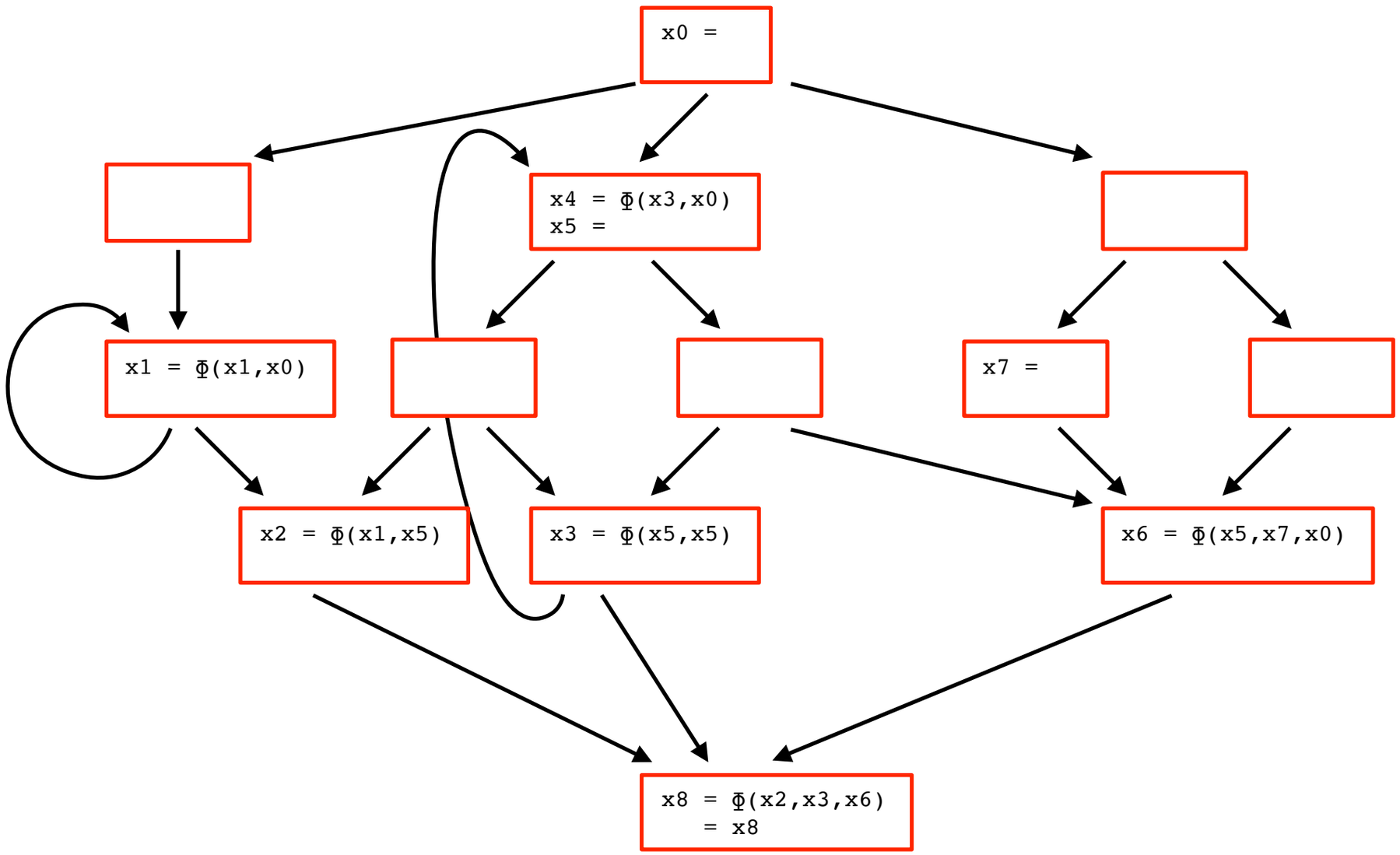
Algorithmes de mise en forme SSA

- Algorithme naif : pour chaque variable, en chaque point de jonction, on ajoute une ϕ redéfinition



Algorithmes de mise en forme SSA

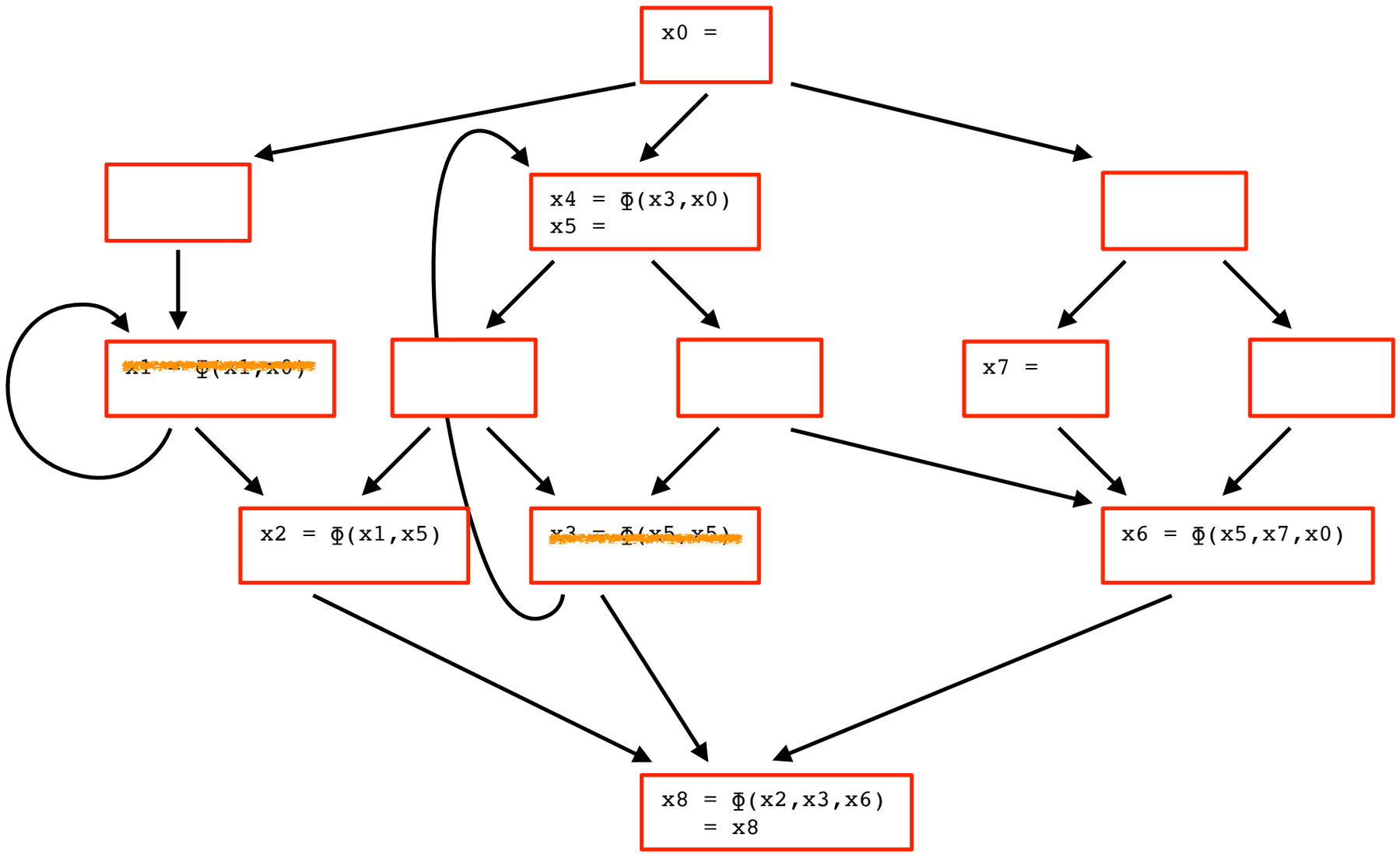
- Algorithme naif : pour chaque variable, en chaque point de jonction, on ajoute une ϕ redéfinition



Algorithmes de mise en forme SSA

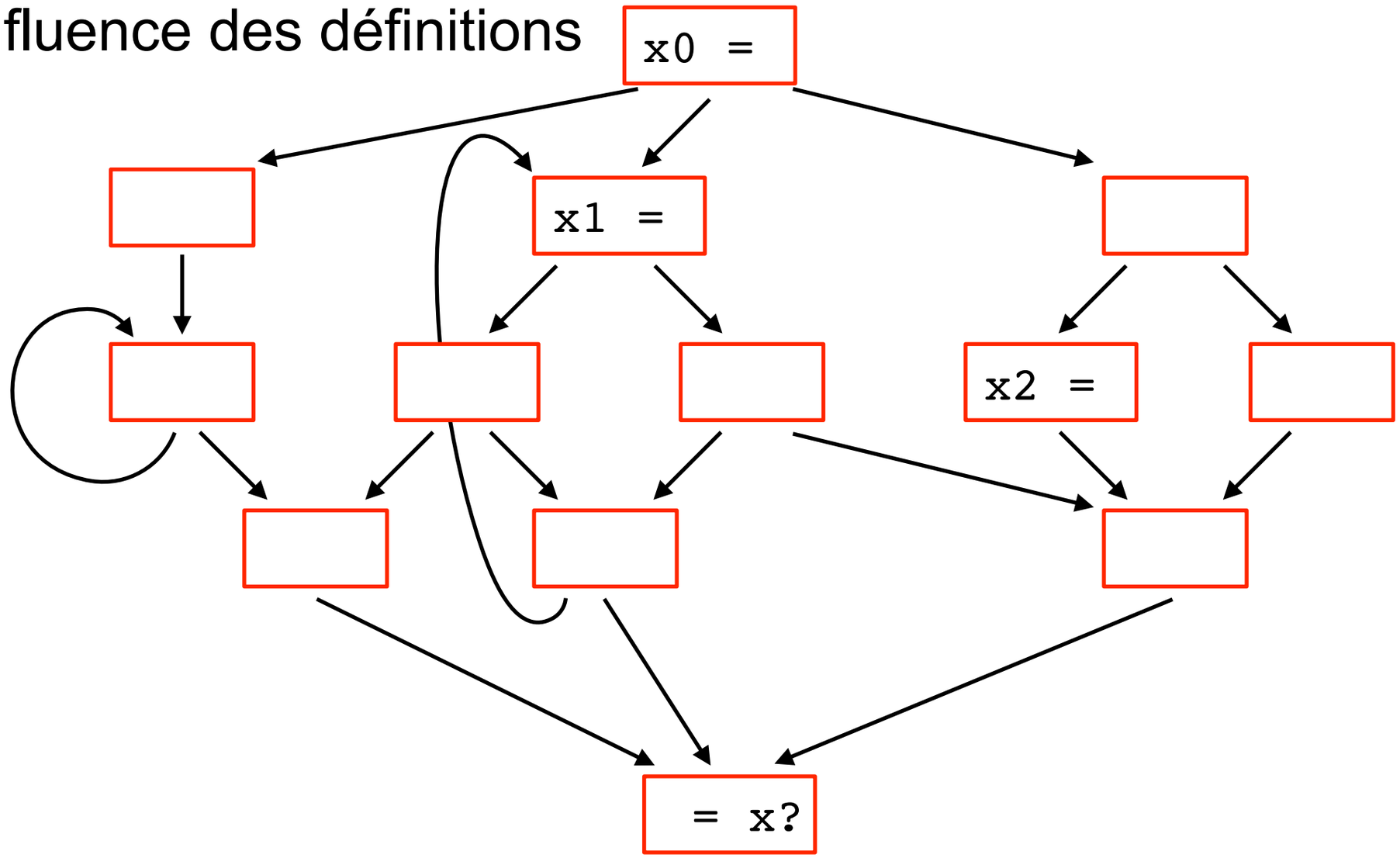
- Algorithme naif : beaucoup de ϕ sont insérés inutilement !

➡ impact négatif direct sur les performances en temps et mémoire des optimisations



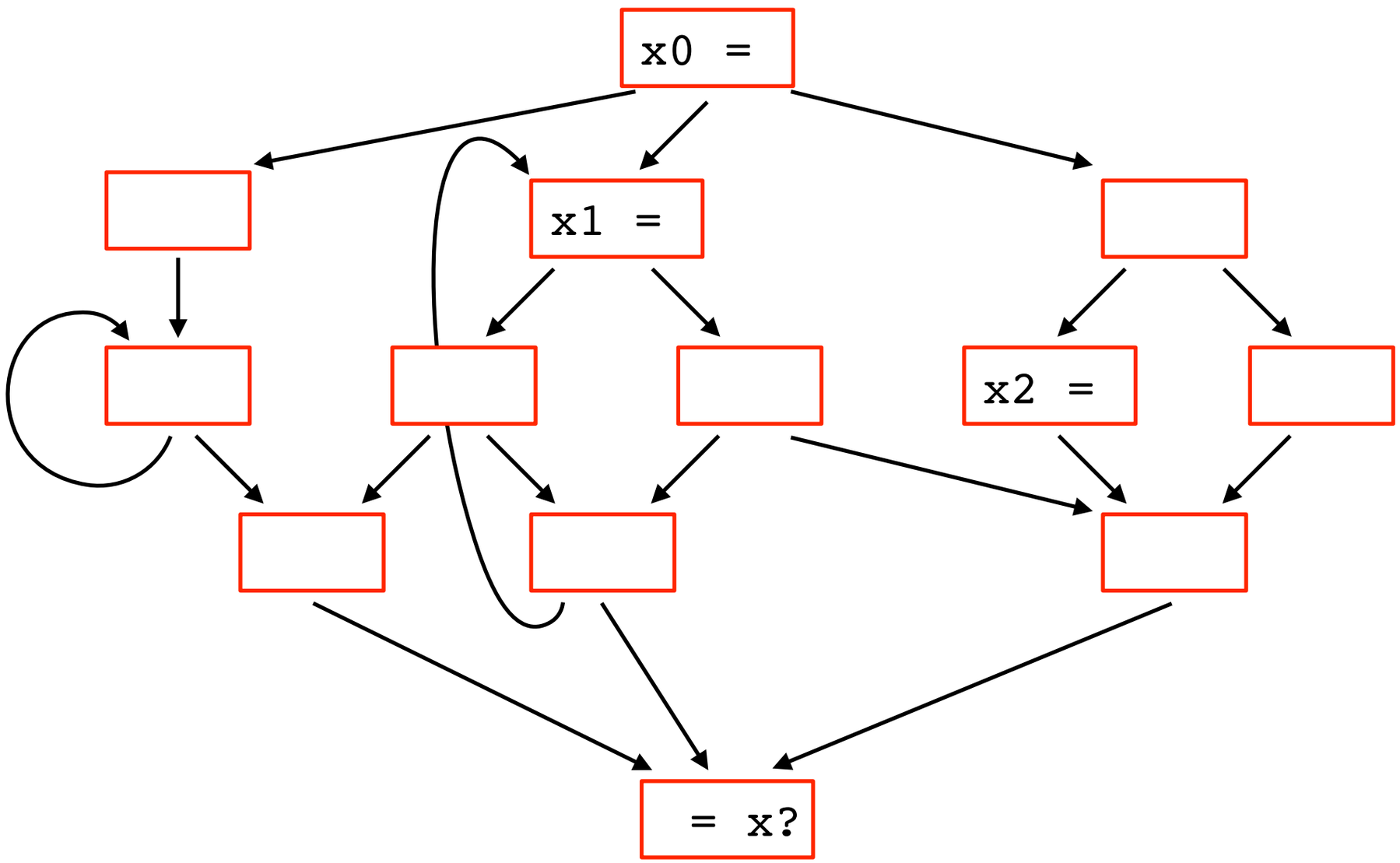
Algorithmes de mise en forme SSA

- Optimisation 1 : inutile de redéfinir une variable morte
- Optimisation 2 : placer les ϕ à la *frontière* des zones d'influence des définitions



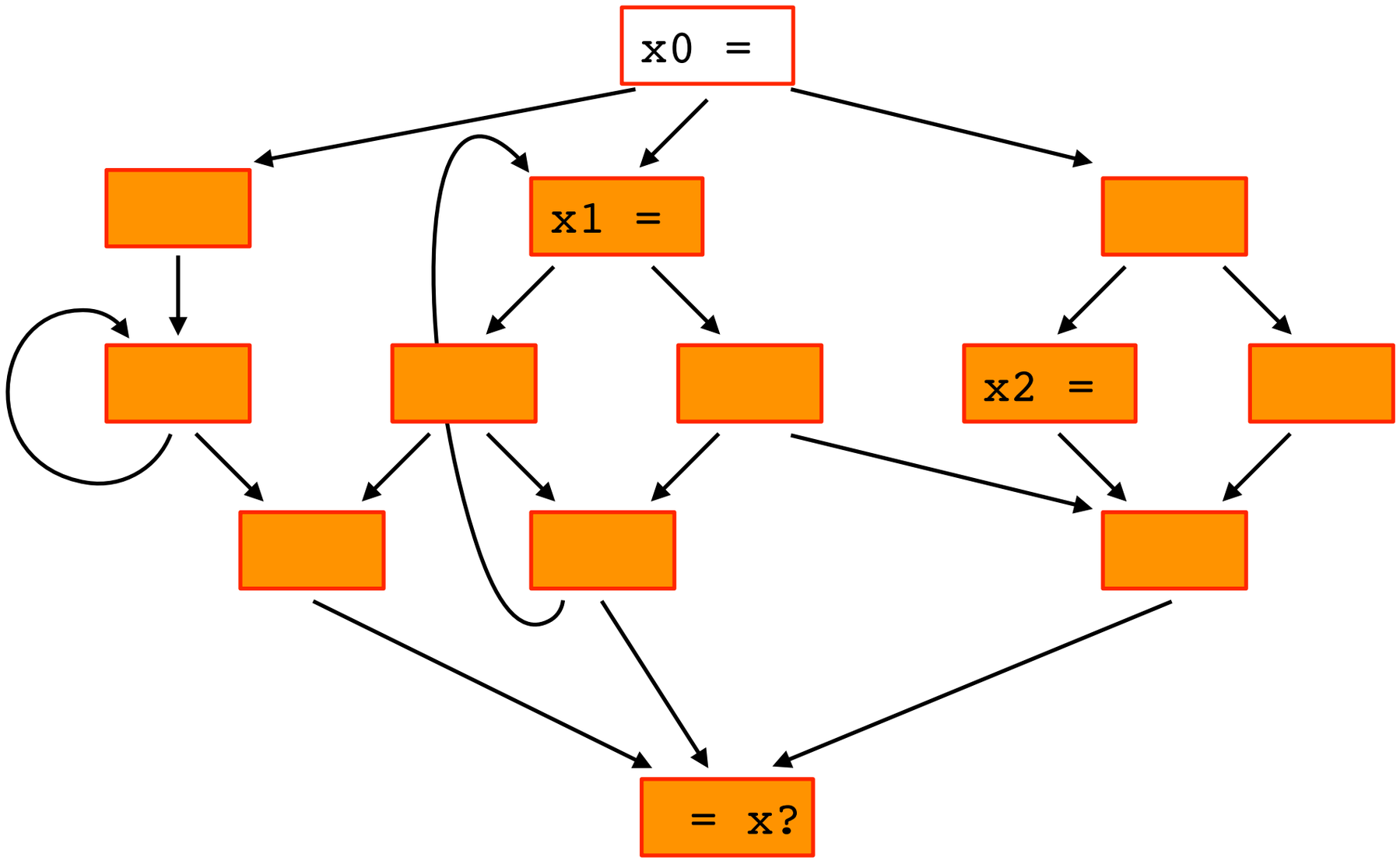
Algorithmes de mise en forme SSA

- Comment déterminer ces *zones d'influence* ?



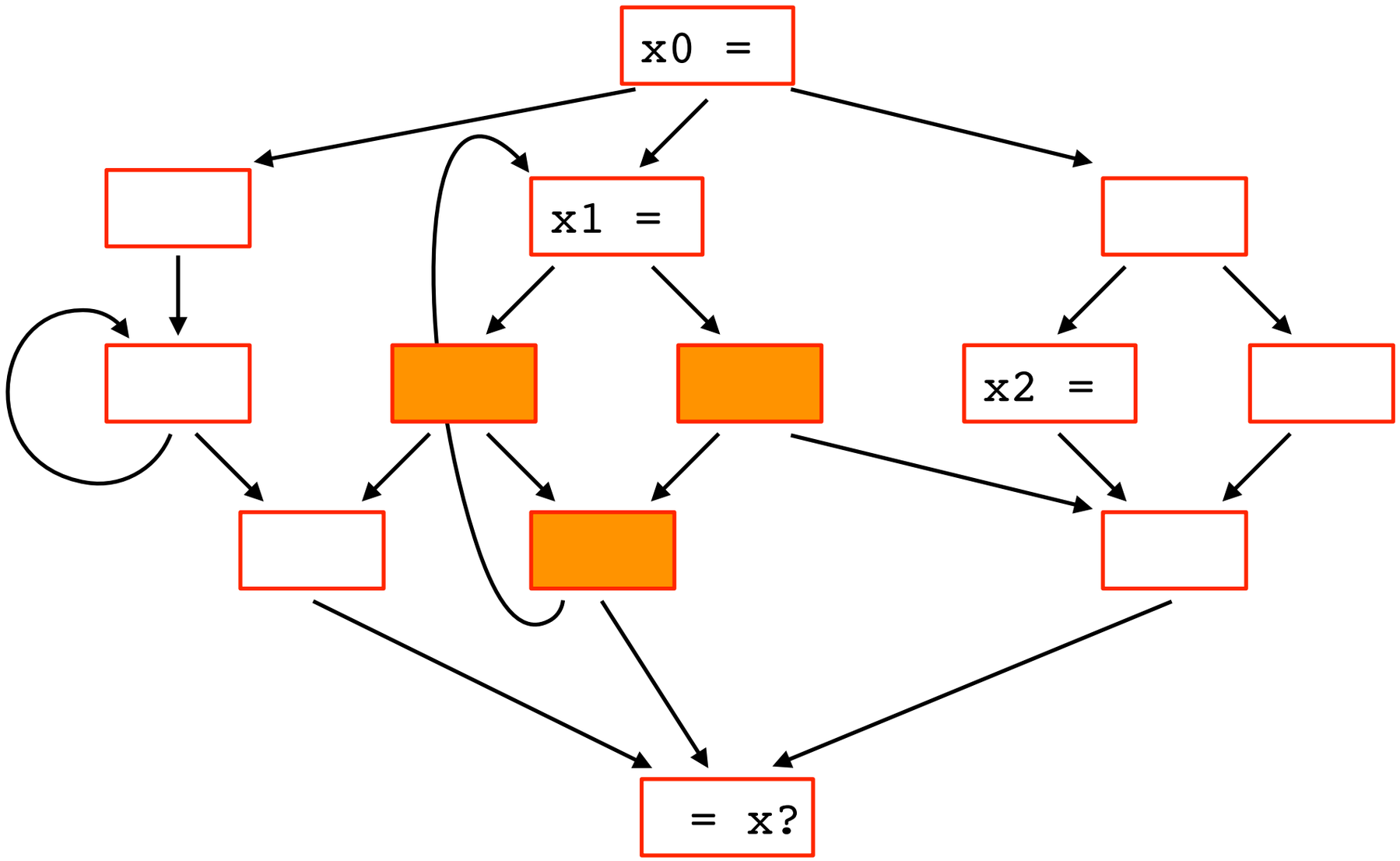
Algorithmes de mise en forme SSA

Les noeuds où `x0` à une valeur définie



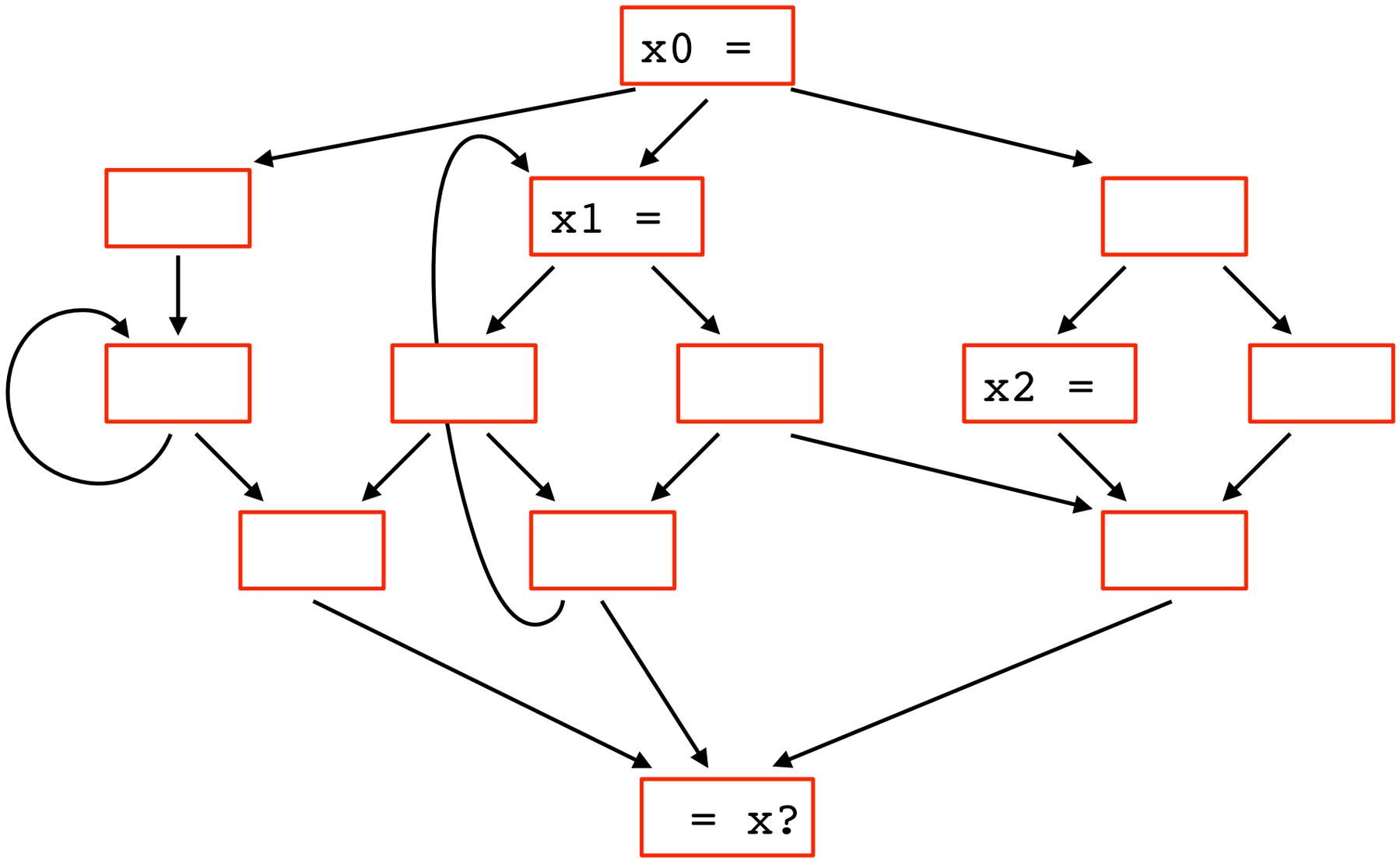
Algorithmes de mise en forme SSA

Les noeuds où `x1` à une valeur définie



Algorithmes de mise en forme SSA

Les noeuds où `x2` à une valeur définie

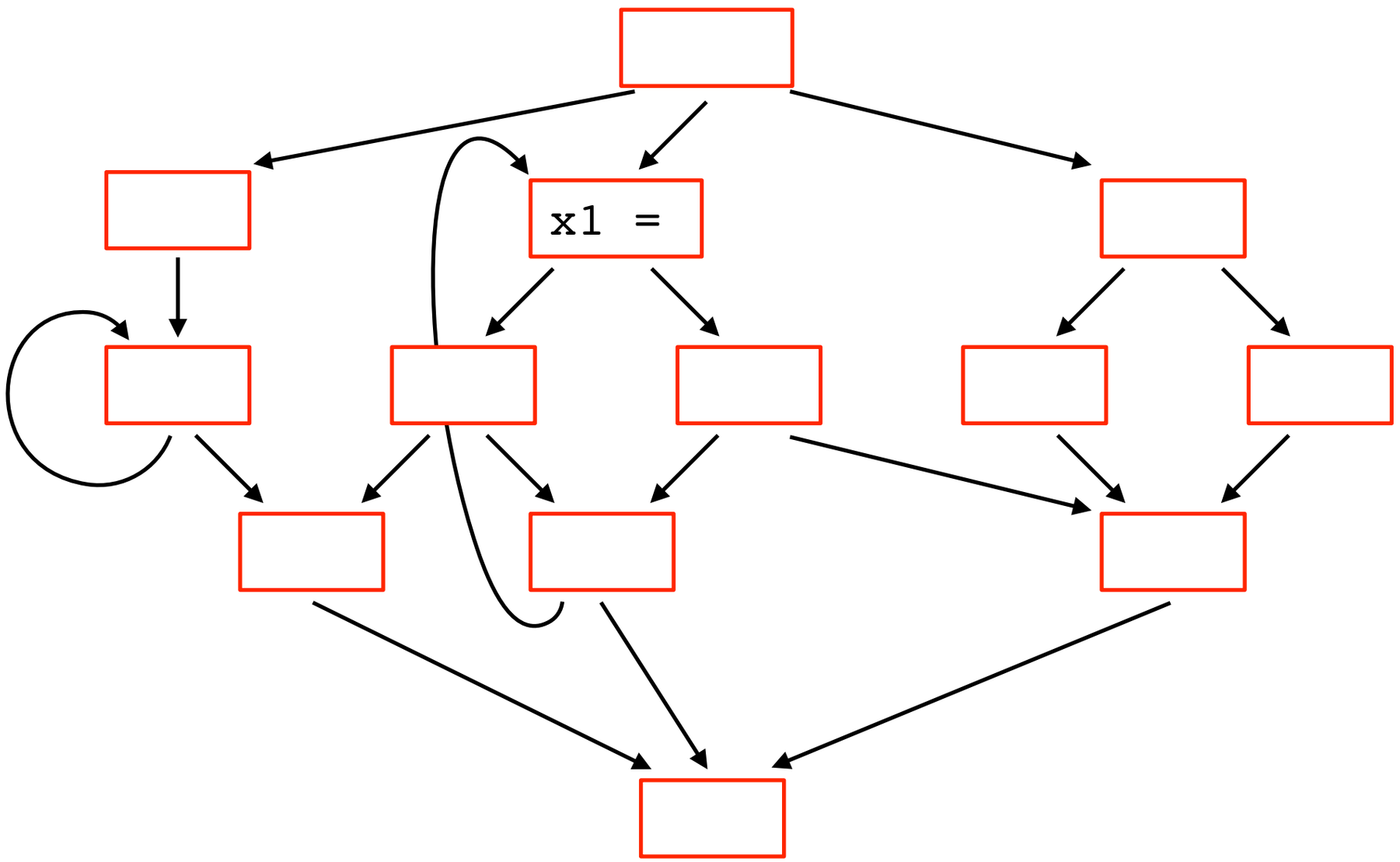


Dominance dans un graphe orienté

- On considère un graphe orienté qui possède un unique noeud *d'entrée e* tel que
  - *e* n'a pas de prédécesseurs
  - tous les noeuds du graphe sont accessibles depuis *e*
- On dit qu'un **noeud *d domine* un noeud *n*** du graphe si *d* apparaît sur tout les chemins menant de *e* à *n*

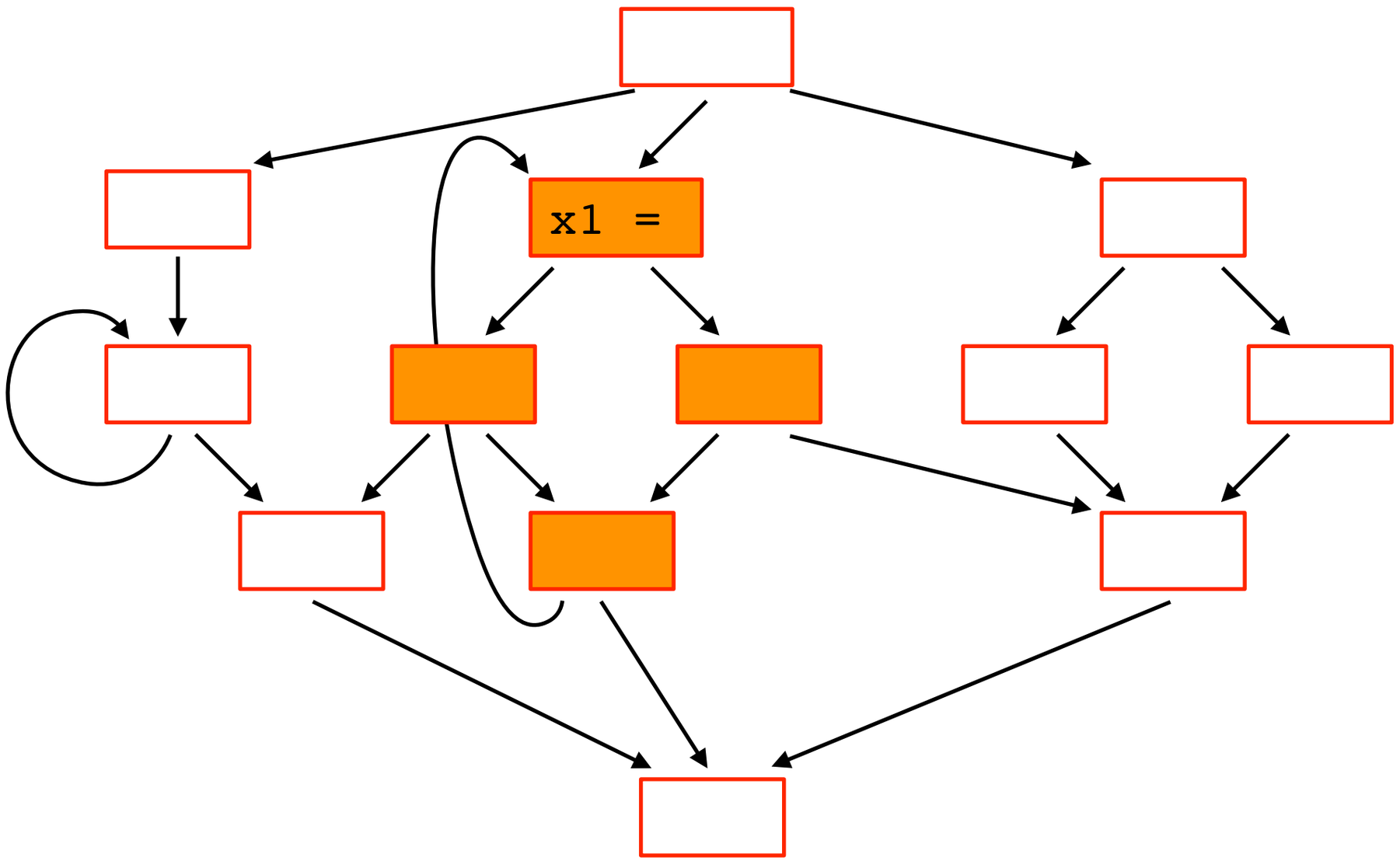
Dominance

Quels sont les noeuds dominés par la définition de x1 ?



Dominance

Quels sont les noeuds dominés par la définition de x1 ?



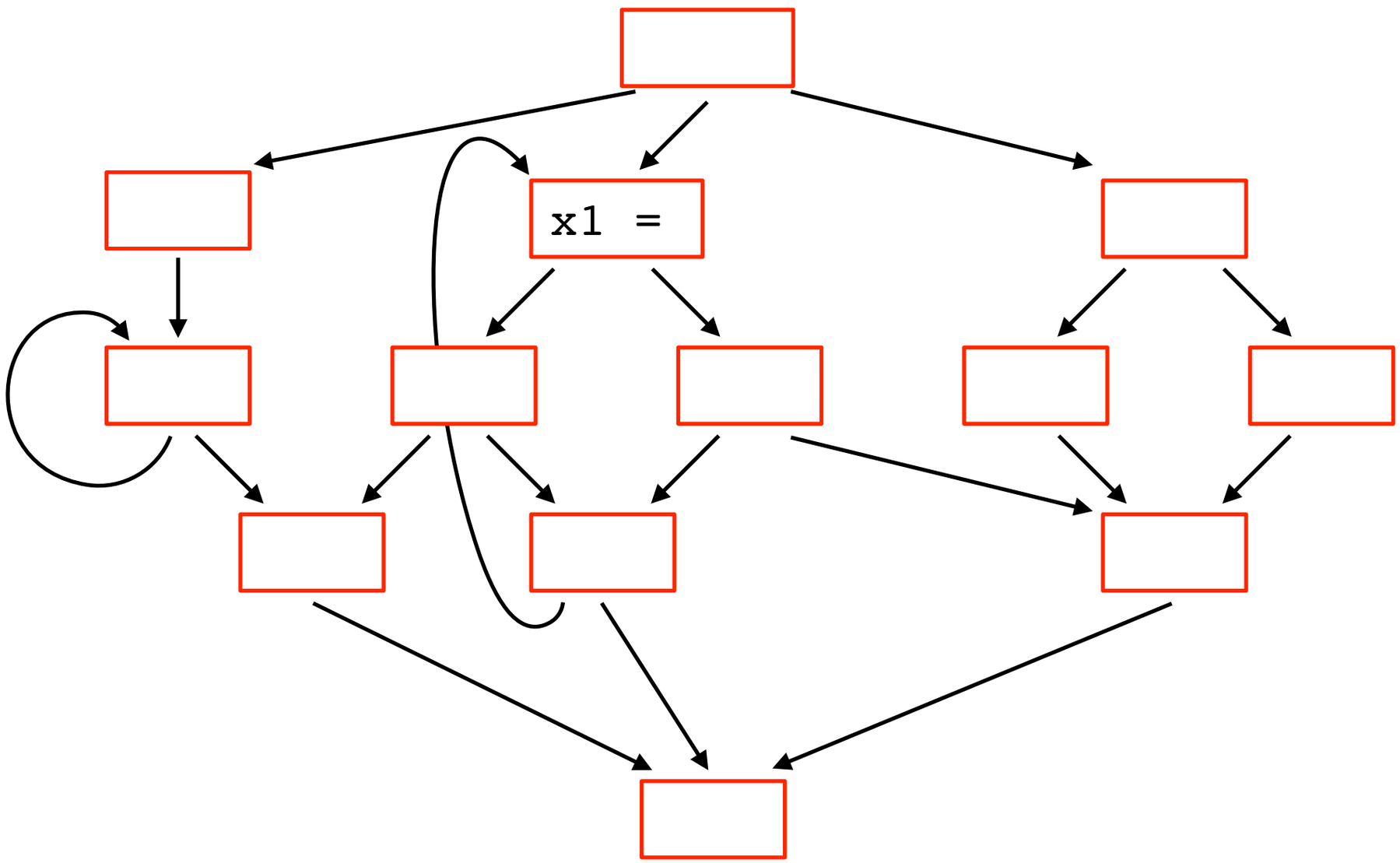
Algorithmes de mise en forme SSA

- Il suffit de placer les ϕ aux *frontières de dominances*
- On dit qu'un noeud *d* ***domine strictement*** un noeud *n* si *d* domine *n* et d ≠ n
- La *frontière de dominance* d'un noeud *d* est l'ensemble des noeuds *n* tels que
  - *d* domine un prédécesseur de *n*
  - mais *d* ne domine pas strictement *n*

n est donc le ***premier*** noeud que *d* ne domine pas strictement

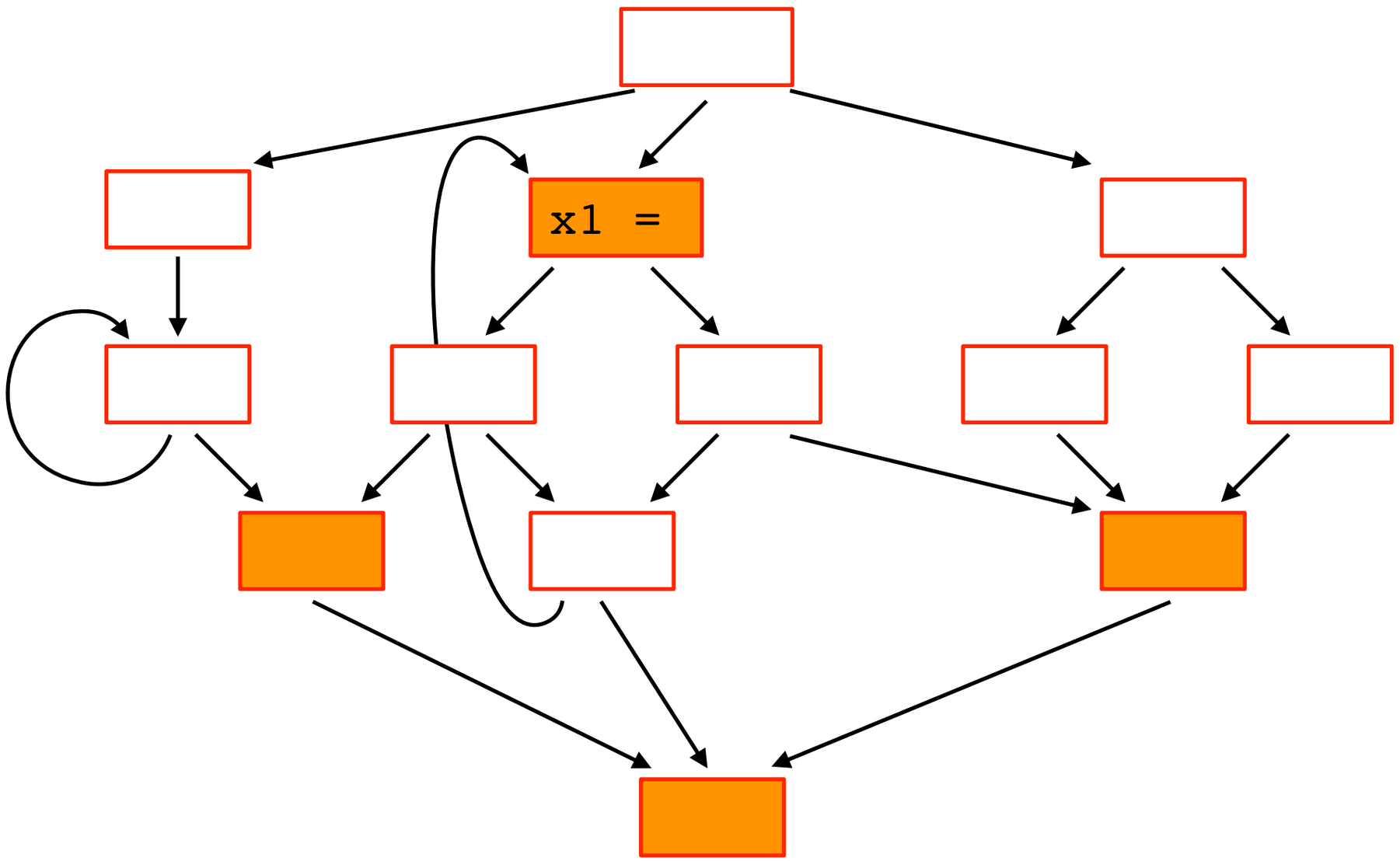
Dominance

Quelle est la frontière de dominance de la définition de x1 ?



Dominance

Quelle est la frontière de dominance de la définition de x1 ?



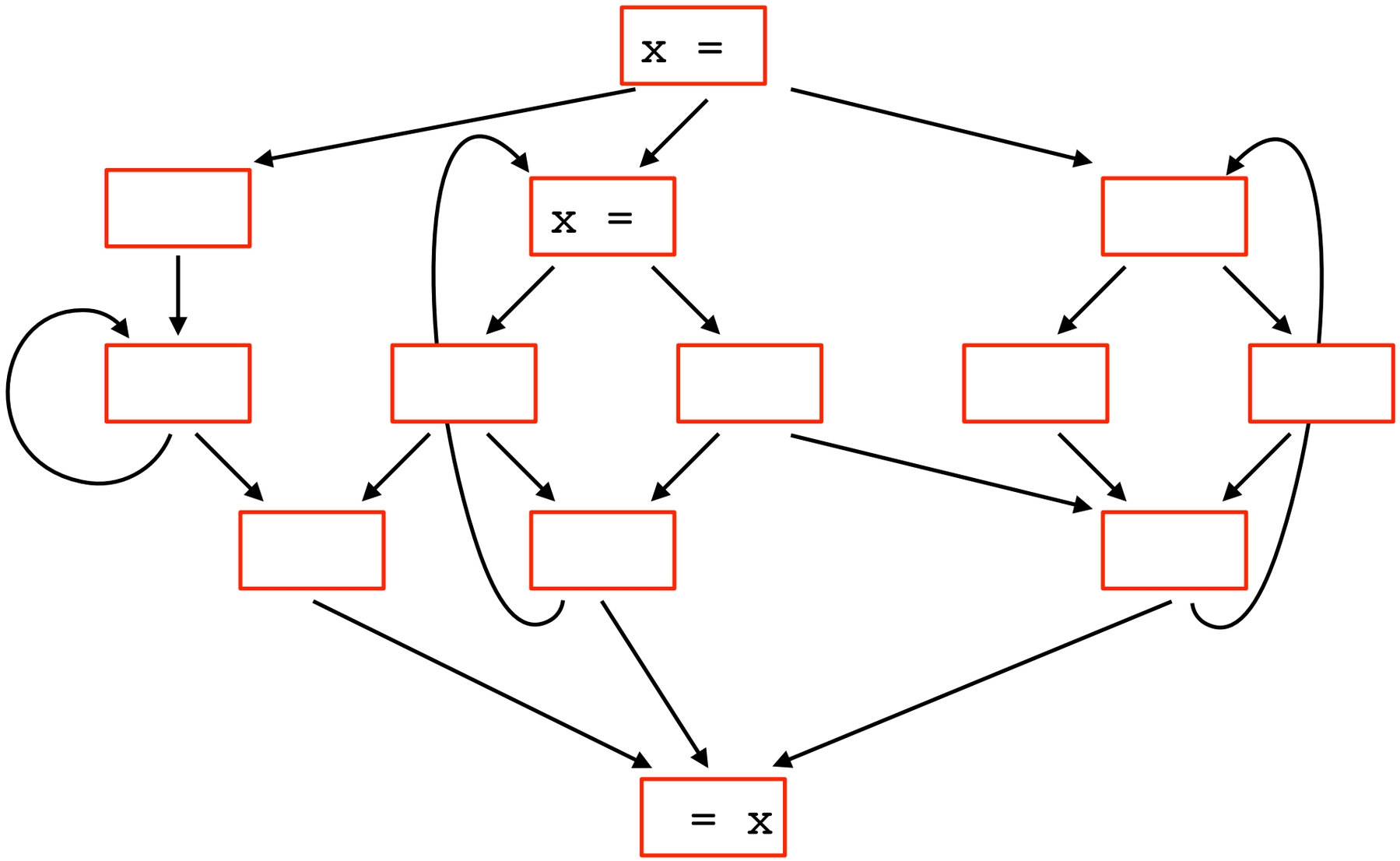
Algorithmes de mise en forme SSA

- **Critère de placement** : si un noeud *n* contient la définition d'une variable *x*, alors tout les noeud de la frontière de dominance de *n* ont besoin d'une ϕ fonction pour *x*

- Mais ajouter une ϕ instruction ajoute une définition donc il faut itérer le procédé jusqu'à ce que plus aucune ϕ fonction ne soit nécessaire

Placement par frontières de dominance itérées

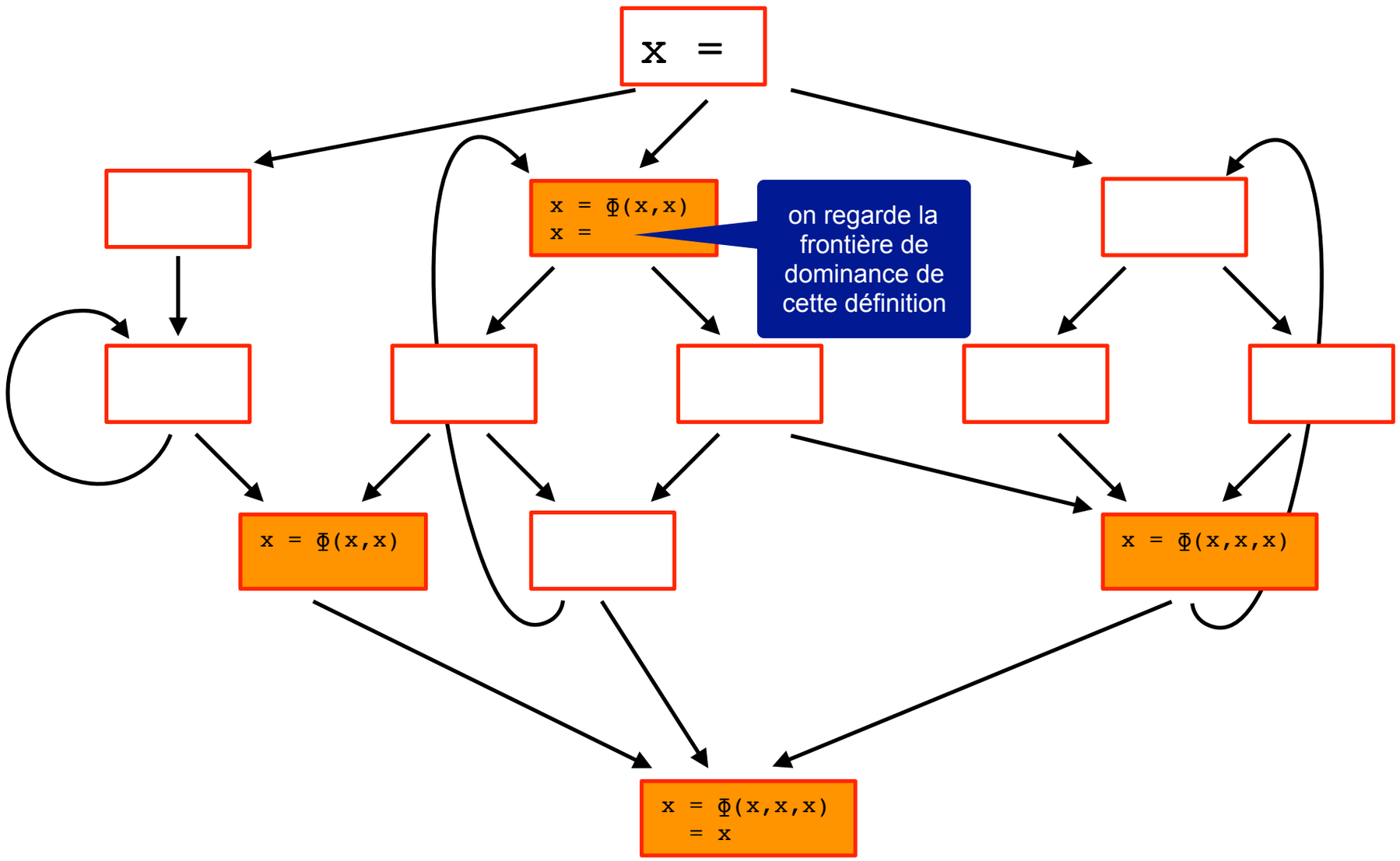
Placer les ϕ fonctions avec le critère des frontières de dominance itérées



Placement par frontières de dominance itérées

Placer les ϕ fonctions avec le critère des frontières de dominance itérées

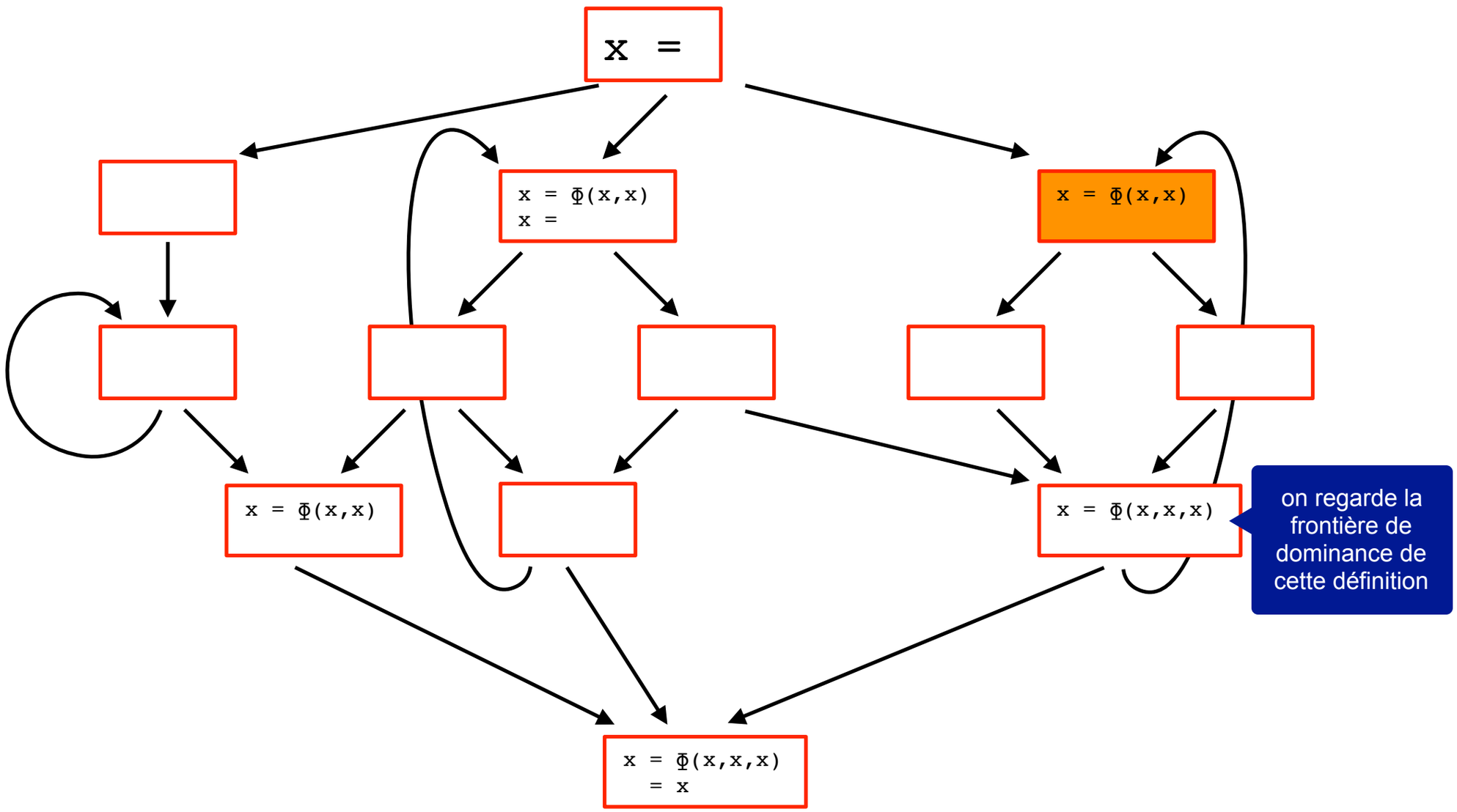
Etape 1



Placement par frontières de dominance itérées

Placer les ϕ fonctions avec le critère des frontières de dominance itérées

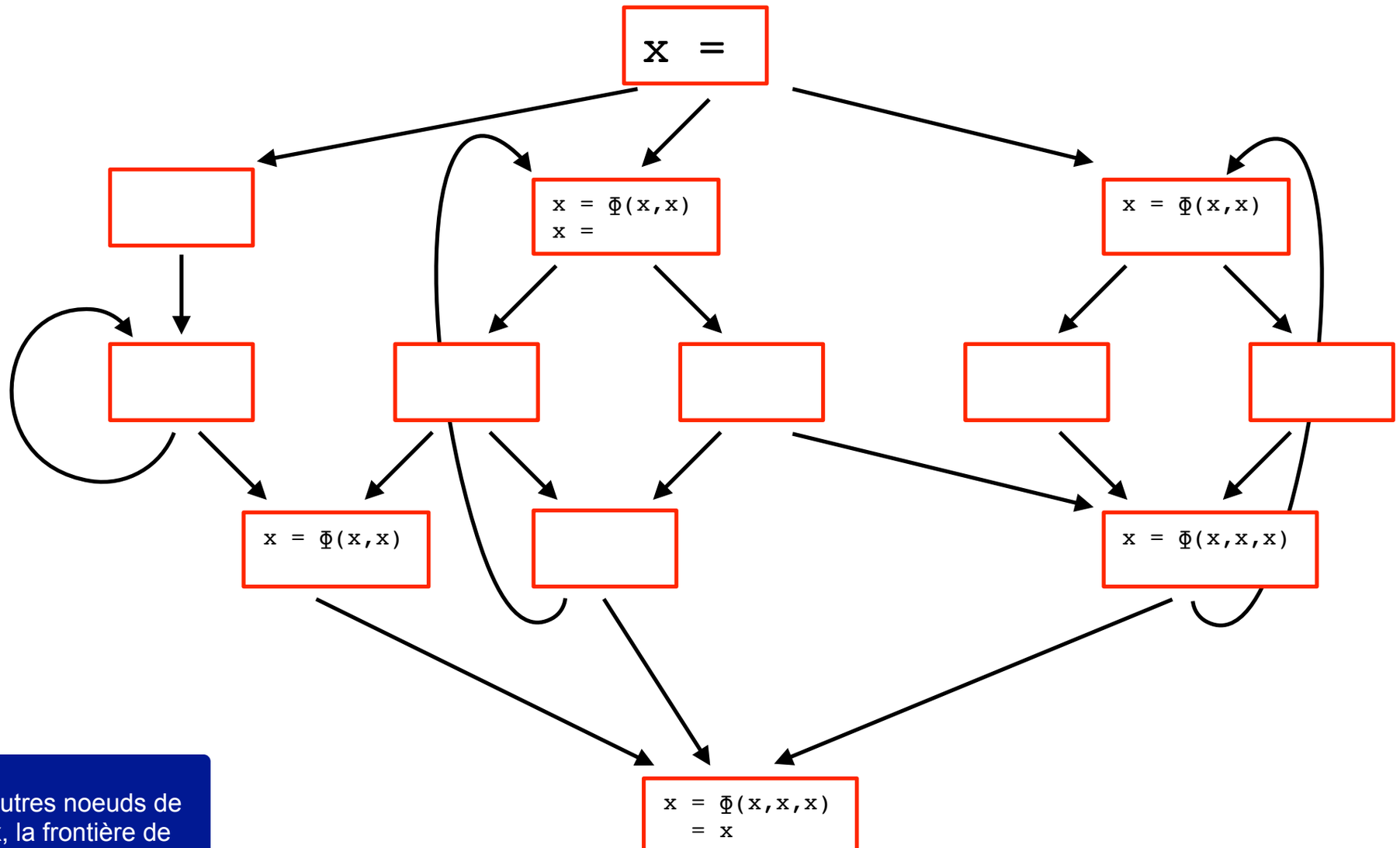
Etape 2



Placement par frontières de dominance itérées

Placer les ϕ fonctions avec le critère des frontières de dominance itérées

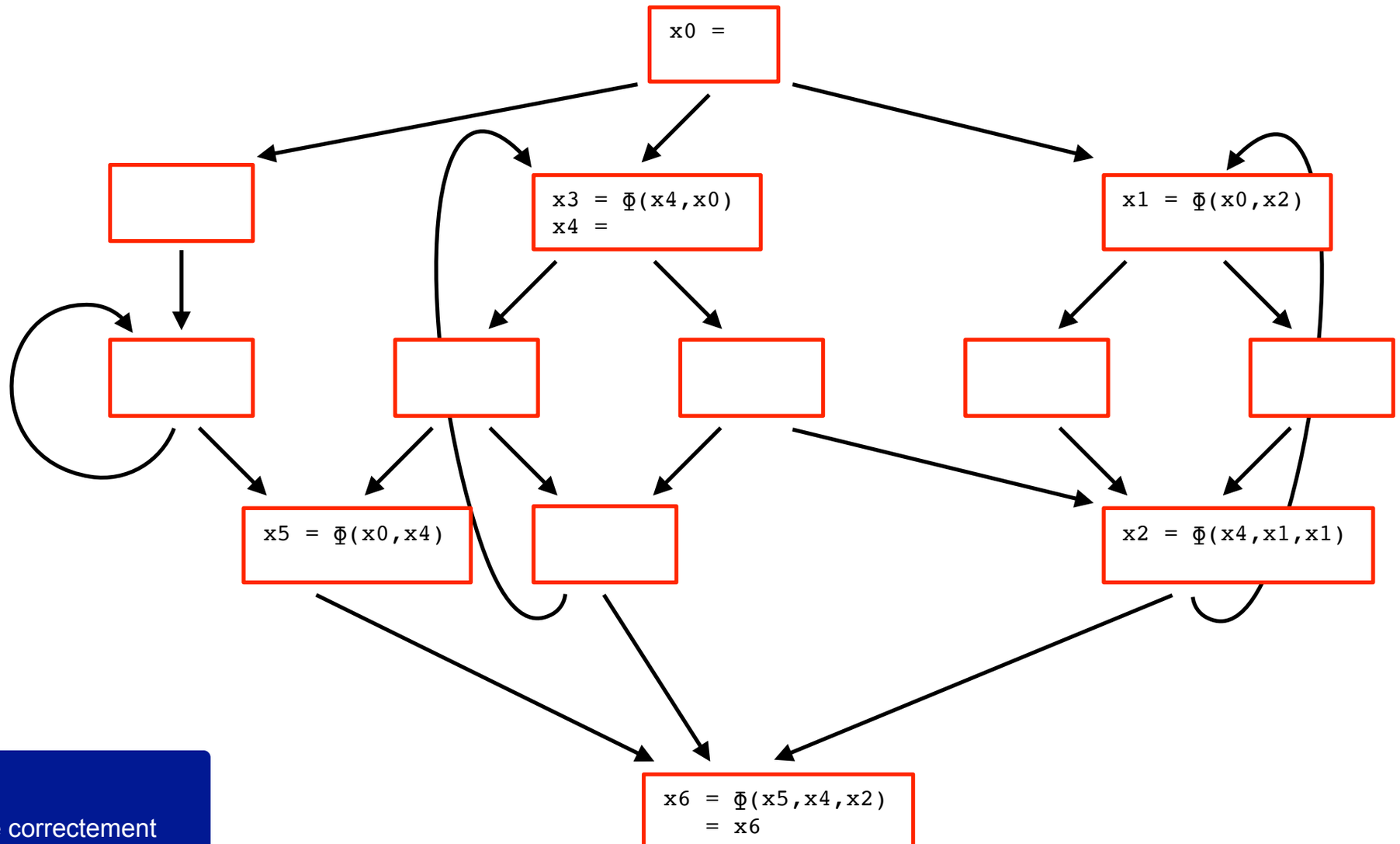
Etape 3



pour tous les autres noeuds de définition de x, la frontière de dominance contient déjà un ϕ

Placement par frontières de dominance itérées

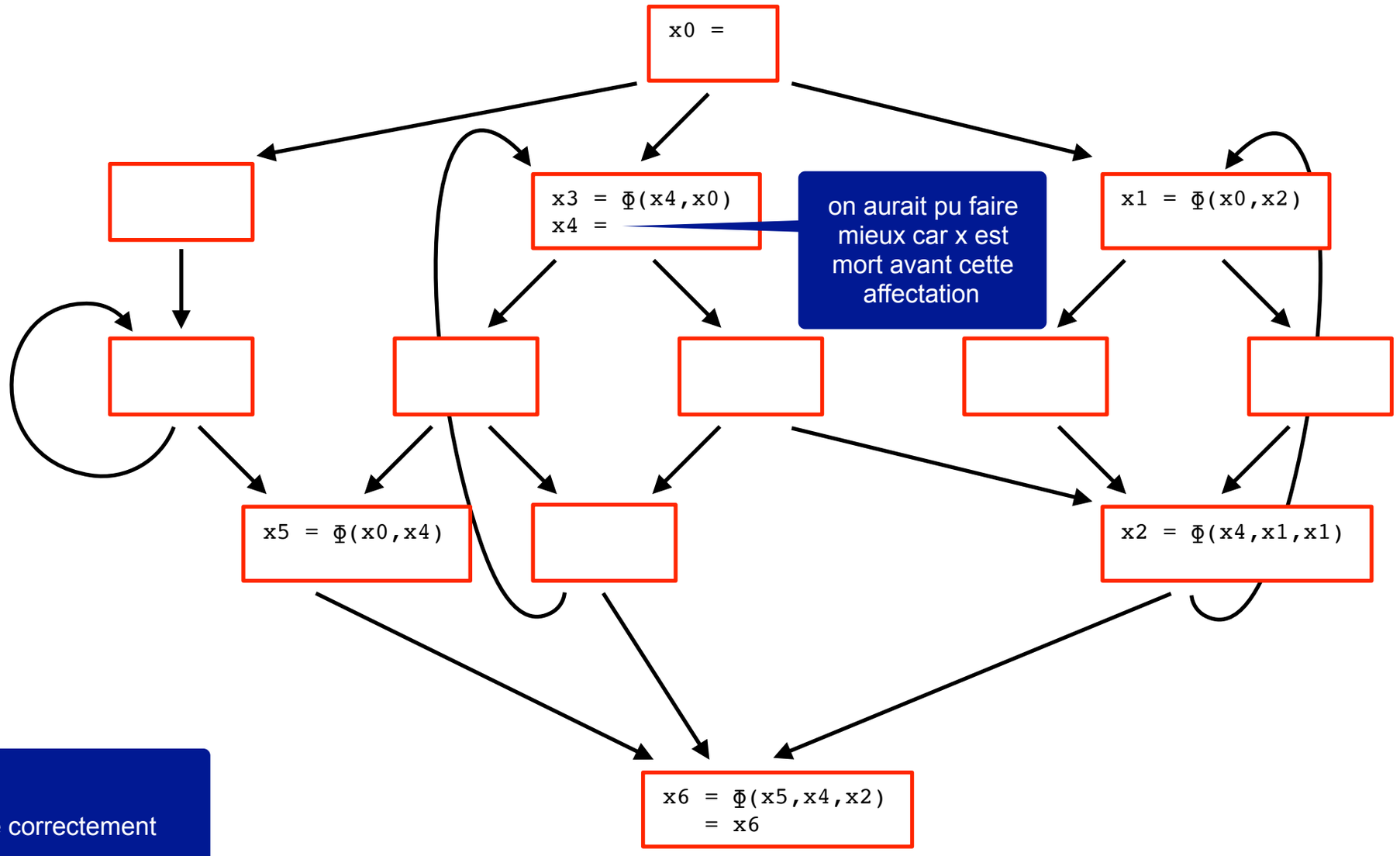
Placer les ϕ fonctions avec le critère des frontières de dominance itérées



on numérote correctement

Placement par frontières de dominance itérées

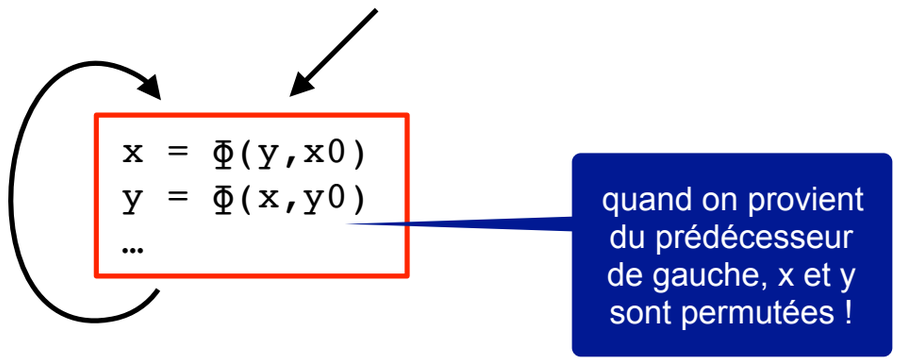
Placer les ϕ fonctions avec le critère des frontières de dominance itérées



on numérote correctement

SSA : quelques subtilités

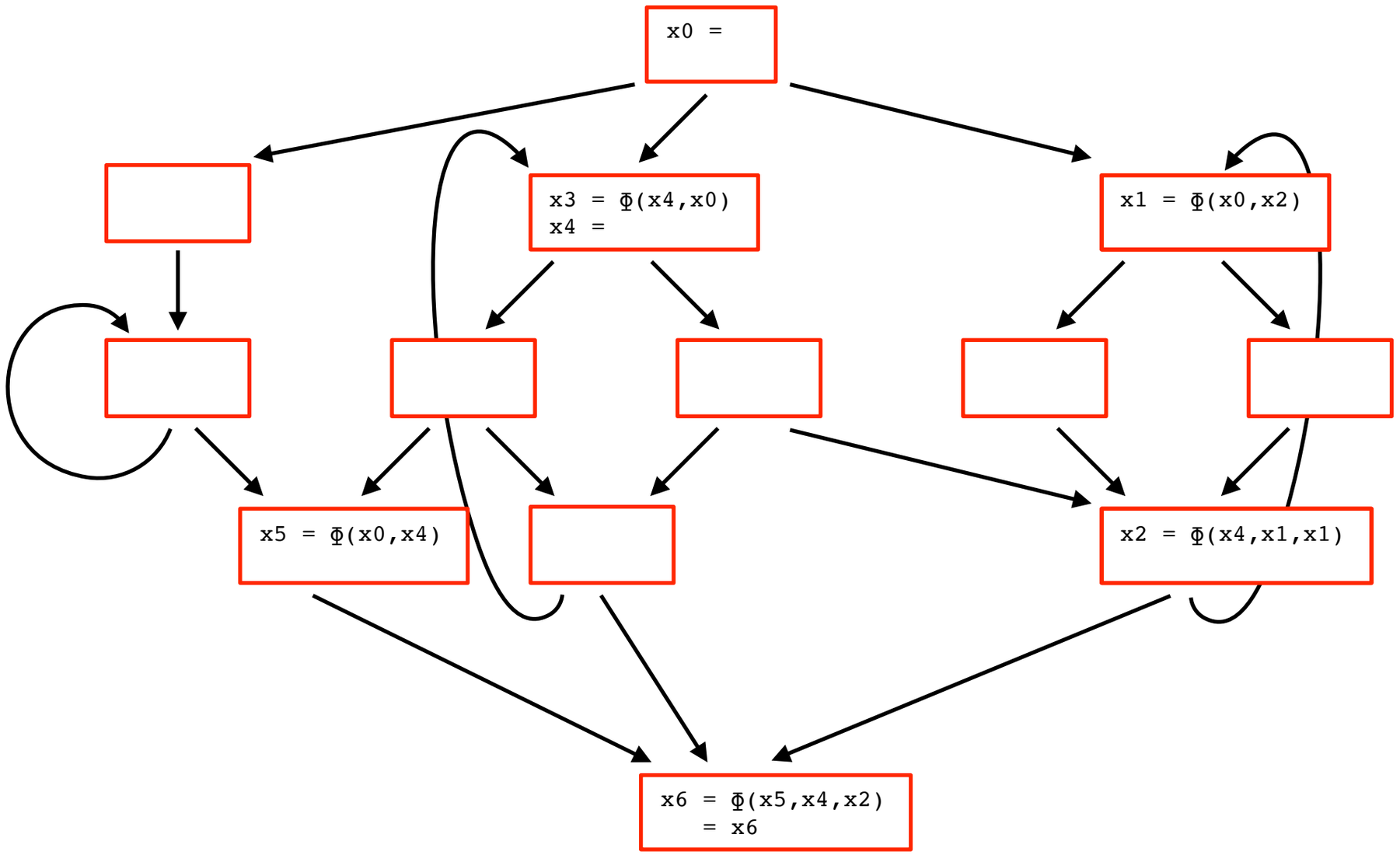
- Un bloc de plusieurs ϕ fonctions a une sémantique **parallèle**



- **Propriété fondamentale :** *toute définition domine ses usages*
  - si *x* est utilisée dans une instruction (non ϕ) d'un noeud *n*, alors la définition de *x* se trouve soit avant son usage dans le noeud, soit dans un noeud qui domine strictement *n*
  - si *x* est utilisée dans une ϕ instruction d'un noeud *n*, comme i-*ème* argument, alors le noeud de la définition de *x* domine le i-*ème* prédécesseur de *n*

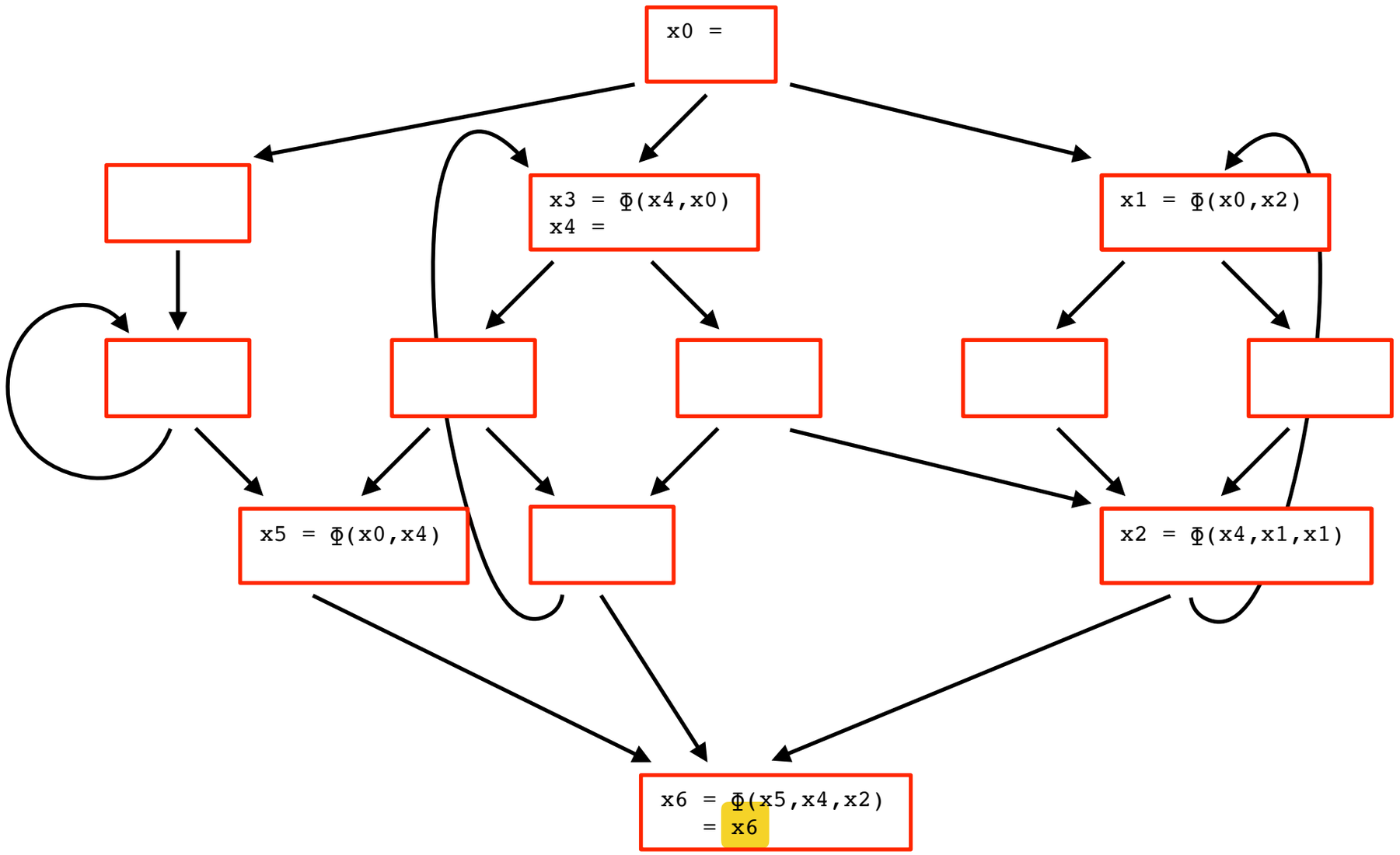
Exercice

Expliquer la propriété « *toute définition domine ses usages* » pour les variables `x6`, puis `x4`



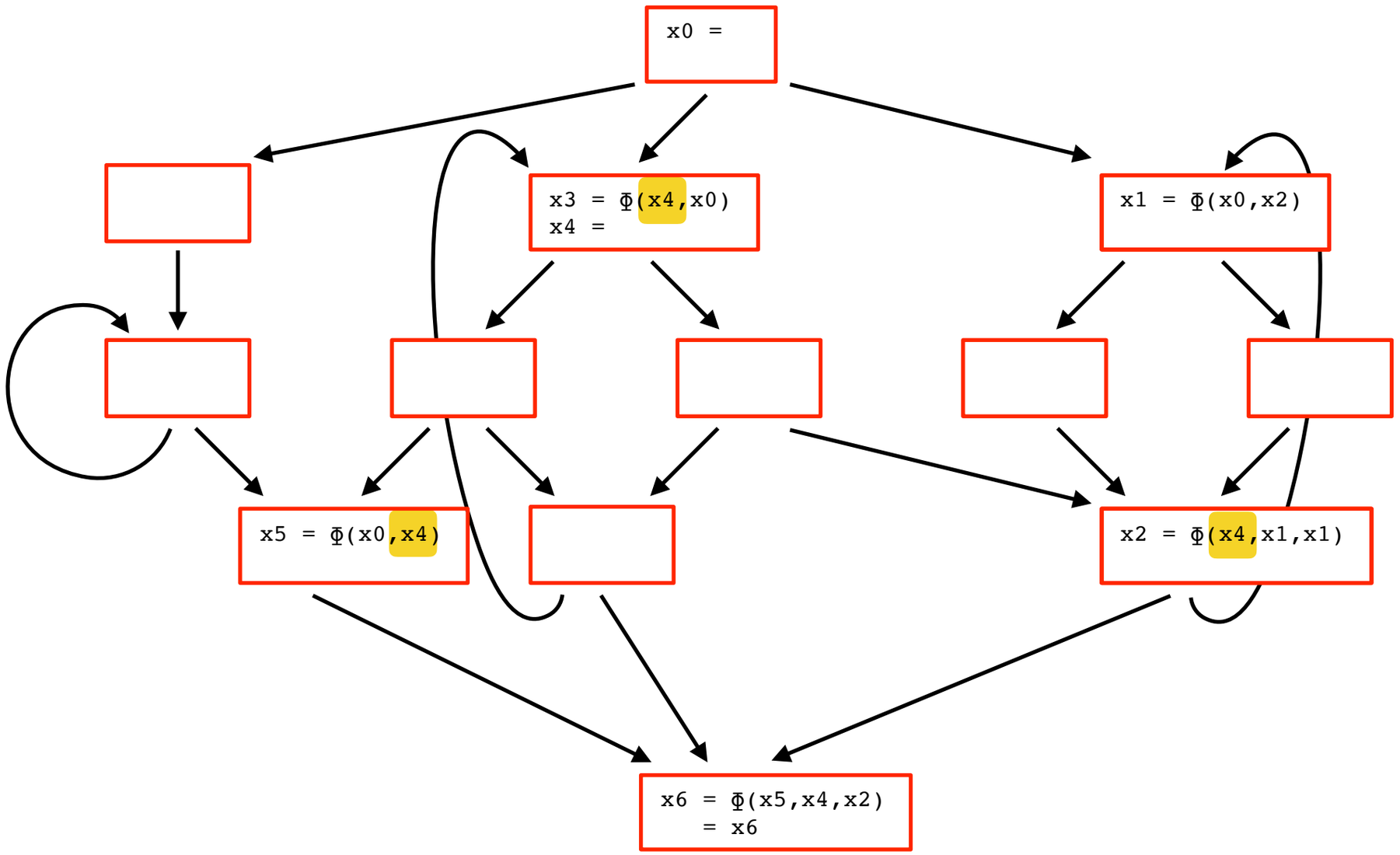
Exercice

Expliquer la propriété « *toute définition domine ses usages* » pour les variables `x6`, puis `x4`



Exercice

Expliquer la propriété « *toute définition domine ses usages* » pour les variables `x6`, puis `x4`



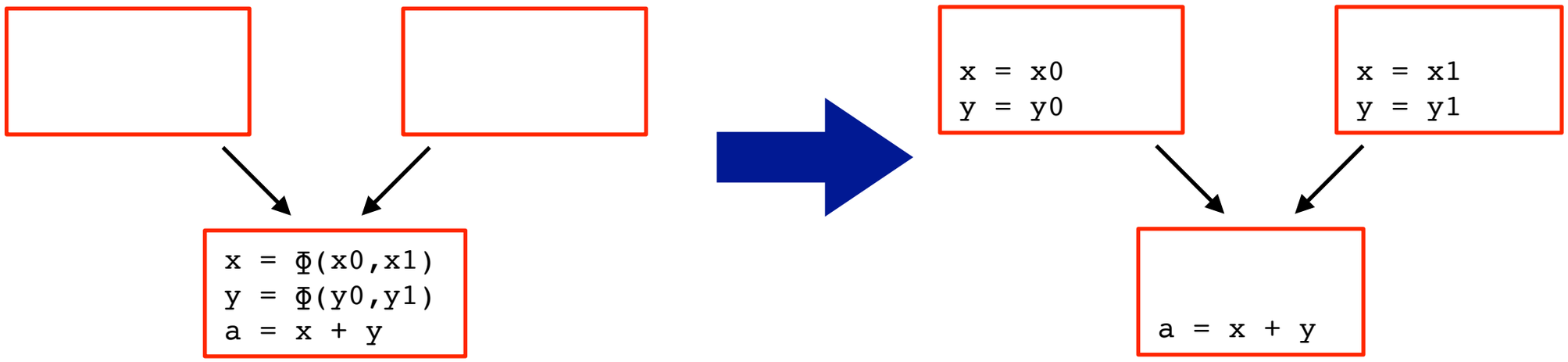
Algorithmes de mise en forme SSA

- **Efficacité** : il existe des algorithmes quasi-linéaire en pratique pour calculer la relation de dominance et déterminer les frontières de dominance itérées

- **Préservation** : lors d'une transformation en forme SSA, il faut veiller à conserver la propriété « *toute définition domine ses usages* »

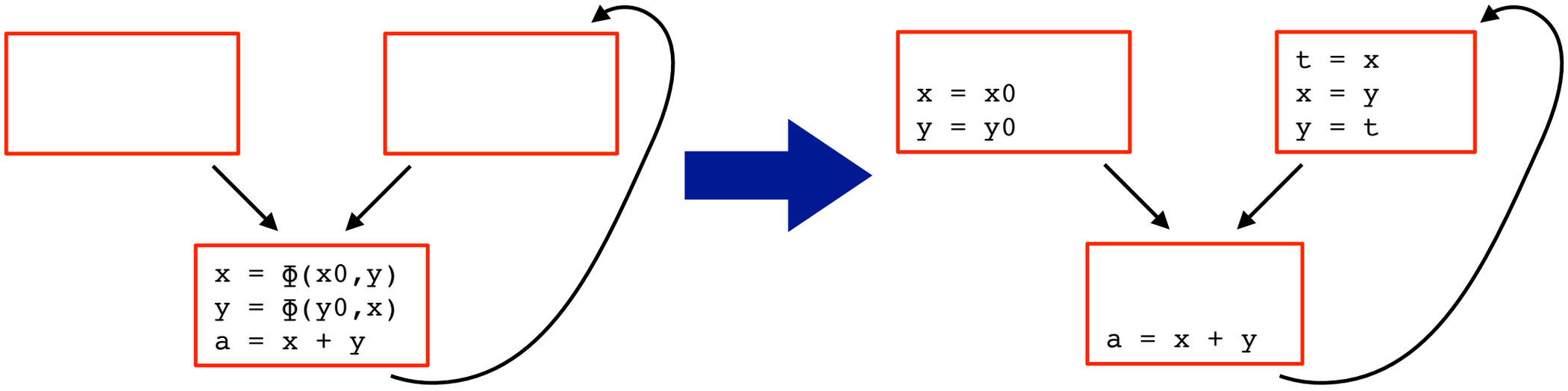
Sortir de la forme SSA

- **Algorithme naïf** : on supprime les *ϕ blocs* en plaçant des blocs de copies appropriés dans les prédécesseurs



Sortir de la forme SSA

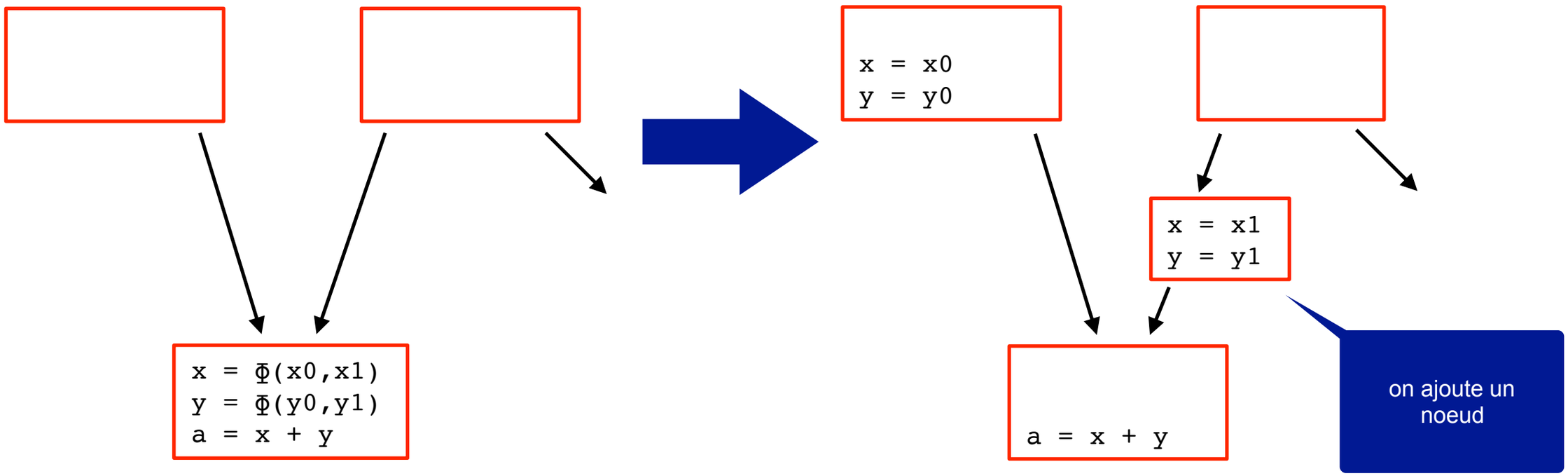
- **subtilité #1 [*swap problem*]** : comme les ϕ blocs ont une sémantique parallèle, il faut dé-parallèliser le bloc de copie créé



Sortir de la forme SSA

- *arc critique* : arc qui part d'un noeud avec plusieurs successeurs et arrive en un noeud avec plusieurs prédécesseurs

- **subtilité #2 [*critical edges*]** : si il existe des *arcs critiques*, on va faire remonter des copies qui peuvent interferer inutilement avec d'autres chemins d'executions. La solution : ajouter des noeuds



Conclusion

- une représentation astucieuse avec des propriétés fortes
- on dispose d'algorithmes très efficaces (et subtiles) pour passer en forme SSA et en sortir
- un sujet de recherche encore actif
  - études de variantes de SSA
  - vérification formelle sur SSA

Classe SSAConstMap

```
// construction d'une fonction de domaine dom,
// où chaque ident est associé à BOT
static public SSAConstMap buildBot(Set<Ident> dom)

// applique la fonction this sur id
public BotOrIntOrTop get(Ident id)

// modifie la map courante en associant l'ident id à la valeur v
public void set(Ident id, BotOrIntOrTop v)

public boolean equals(Object o)

public String toString()
```

Classe BotOrIntOrTop

```
// construction de BOT
static public BotOrIntOrTop buildBot()

// construction de TOP
static public BotOrIntOrTop buildTop()

// construction d'une constante entière
static public BotOrIntOrTop buildInt(int i)

// teste si this est BOT
public boolean isBot()

// teste si this est TOP
public boolean isTop()

// renvoie i si this est un entier i,
// échoue avec une exception si this est TOP
public int getInt()

// renvoie une nouvelle valeur égal au join de this et v
public BotOrIntOrTop join(BotOrIntOrTop v)

public String toString()

public boolean equals(Object o)
```