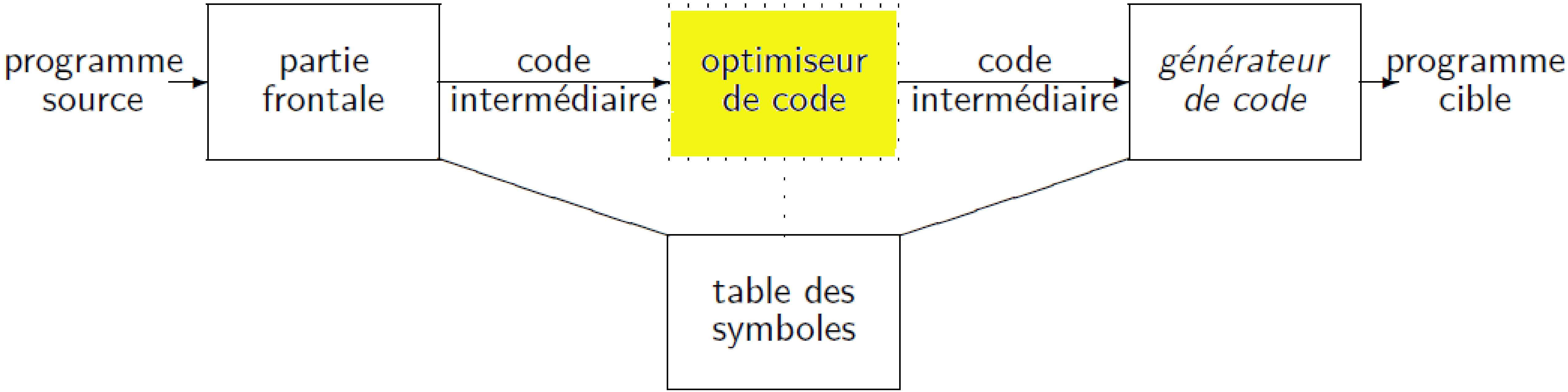
Plan

Rappel


programme source
partie frontale
code intermédiaire
optimiseur de code
code intermédiaire
générateur de code
programme cible
table des symboles

I- Optimisation

Pourquoi est-ce qu’on doit optimiser le code intermédiaire?

1. La génération du CI introduit des **redondances** : des sous-calculs qui peuvent être accélérés, partagés ou éliminés.

2. La paresse des programmeurs : un code qui peut être mis hors une boucle, etc.

I- Optimisation

1. redondances

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

I- Optimisation

1. redondances

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

I- Optimisation

1. redondances

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;



b2 = _t0 == _t1;



b3 = _t0 < _t1;
```

I- Optimisation

2. code hors boucle

```
while (x < y + z) {
    x = x - y;
}
```

```
_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

I- Optimisation

2. code hors boucle

```
while (x < y + z) {
    x = x - y;
}
```

```
_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

I- Optimisation

2. code hors boucle

```
while (x < y + z) {
    x = x - y;
}
```

```
    _t0 = y + z;
_L0:
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

I- Optimisation

- L’optimisation du CI n’implique pas L’obtention d’un code "**optimal**".

- L’objectif de cette étape est plutôt **l’amélioration** que **l’optimisation**.

- L’amélioration au niveau de :
  - **Temps d’exécution**
  - **Utilisation de la mémoire**
  - **Consommation d’énergie**: choisir des instructions simples
  - **Autres**: minimiser les appels de fonctions, réduire l’utilisation de matériel à virgule flottante, etc.

I- Optimisation

Un bon optimiseur:

- Ne devrait jamais changer le comportement observable d'un programme.
- Doit produire un CI aussi efficace que possible.
- Ne devrait pas prendre trop de temps pour traiter les entrées.

Par contre:

- Même les bons optimiseurs introduisent parfois des bogues dans le code.
- Les optimiseurs manquent souvent les optimisations "faciles" en raison des limitations de leurs algorithmes.
- Presque toutes les optimisations intéressantes sont NP-complet.

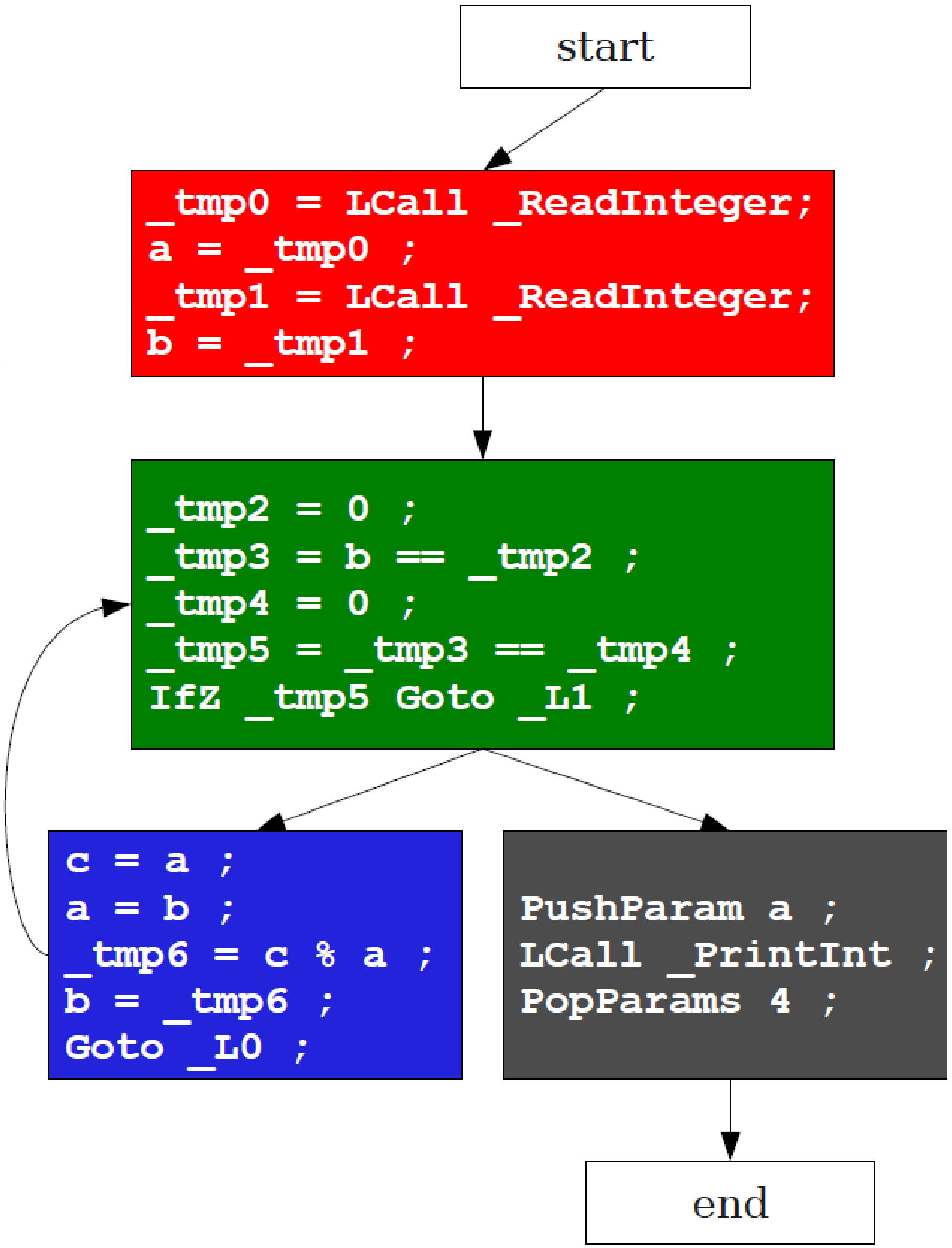
I- Optimisation

Types d'optimisations :

- Une optimisation est **locale** si elle fonctionne juste dans un seul bloc de base.
- Une optimisation est **globale** si elle fonctionne dans tous le **graph de flot de contrôle (CFG)**.
- Une optimisation est **interprocédurale** si elle fonctionne à travers les graphs de flot de contrôle de plusieurs fonctions.

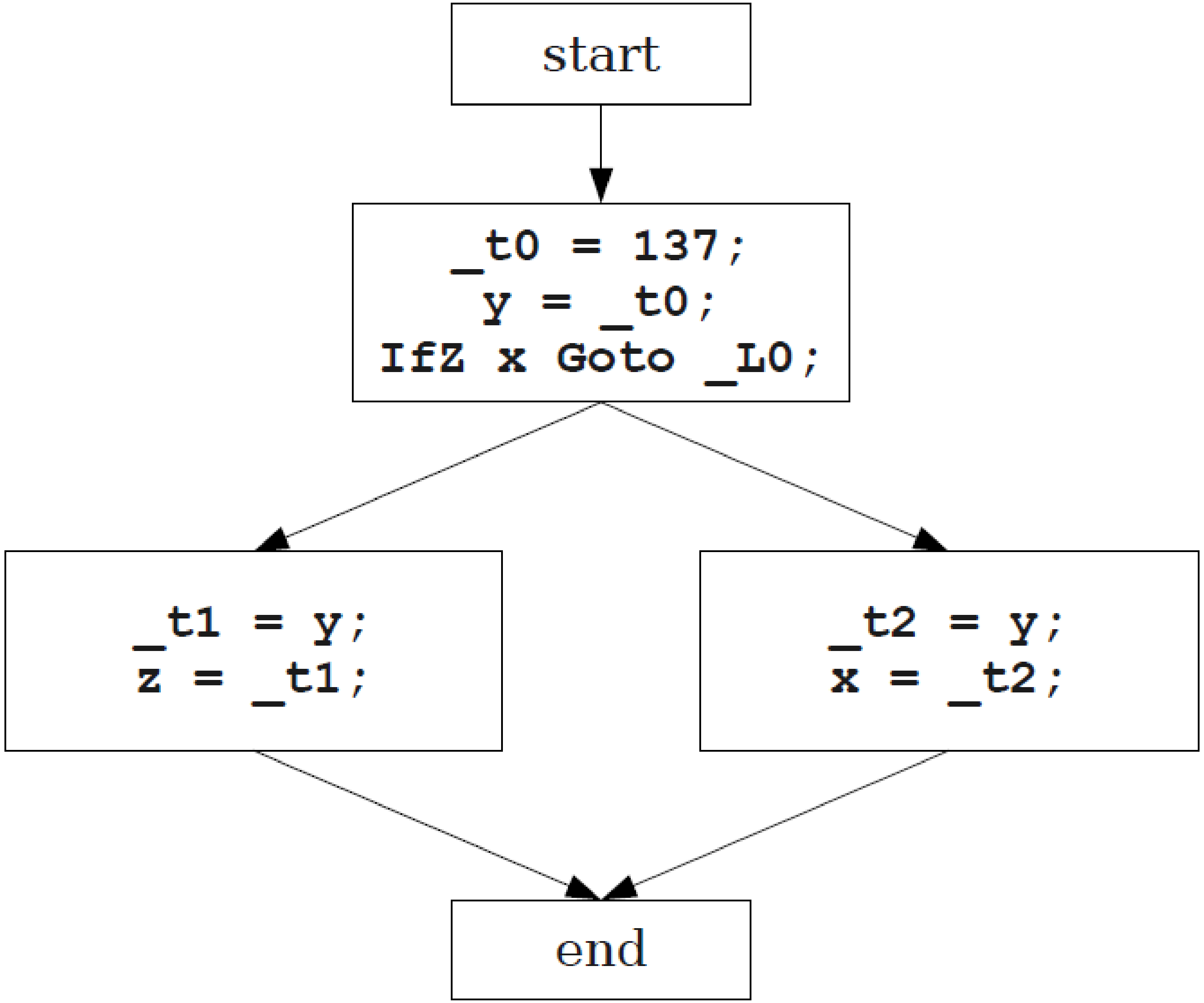
Exemple de graph de flot de contrôle

```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```



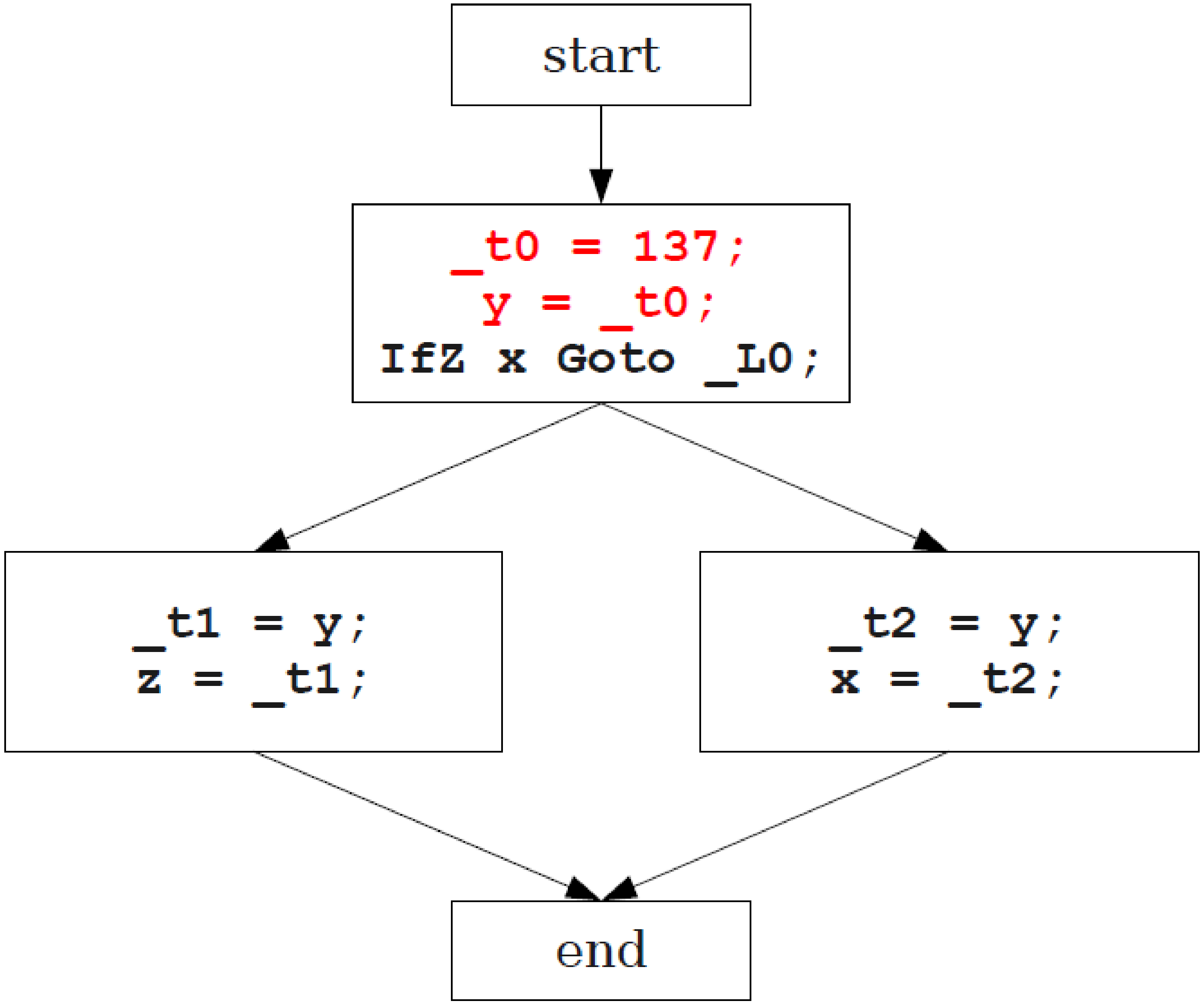
I-1. Exemple d'optimisations locales

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



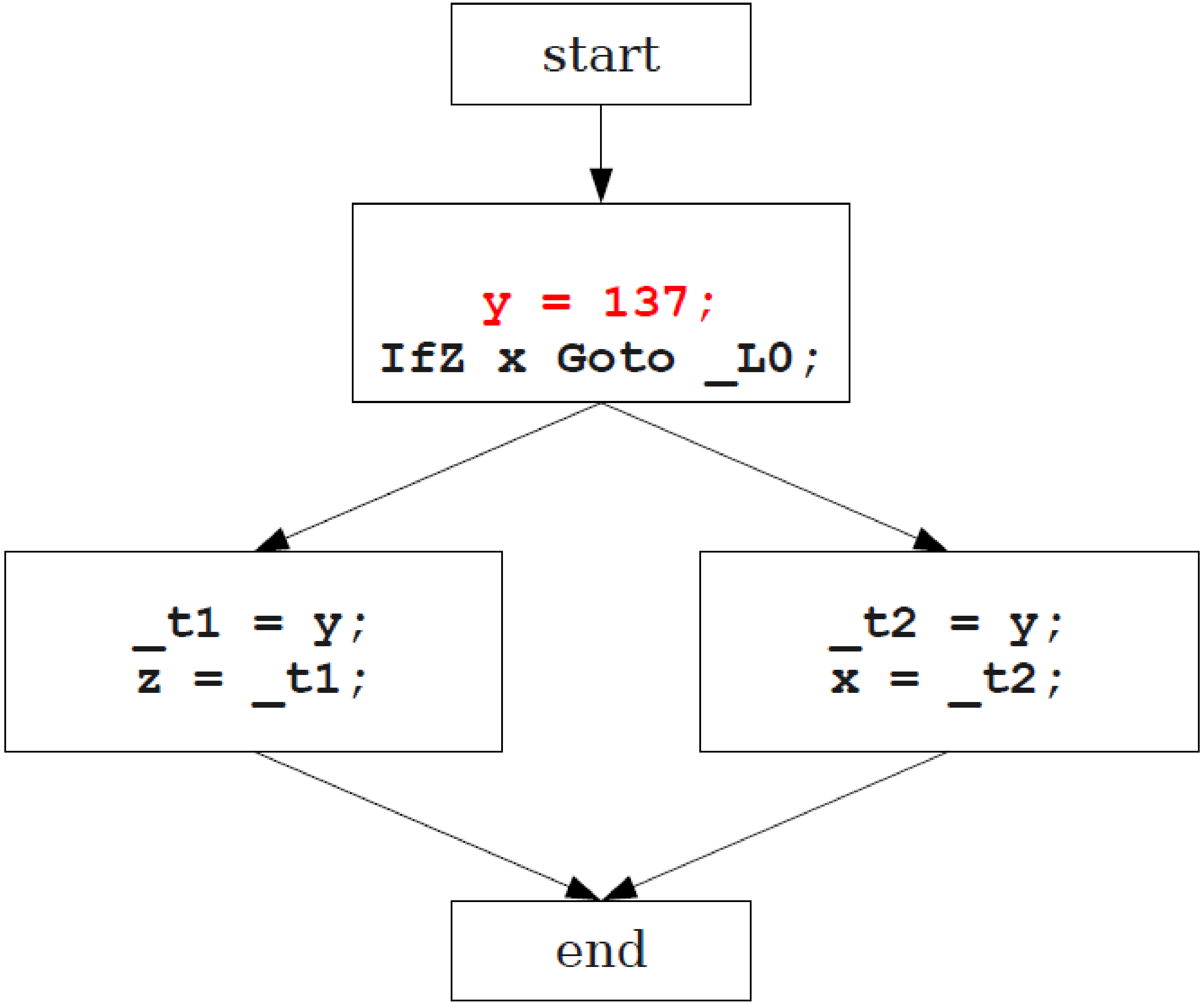
I-1. Exemple d'optimisations locales

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



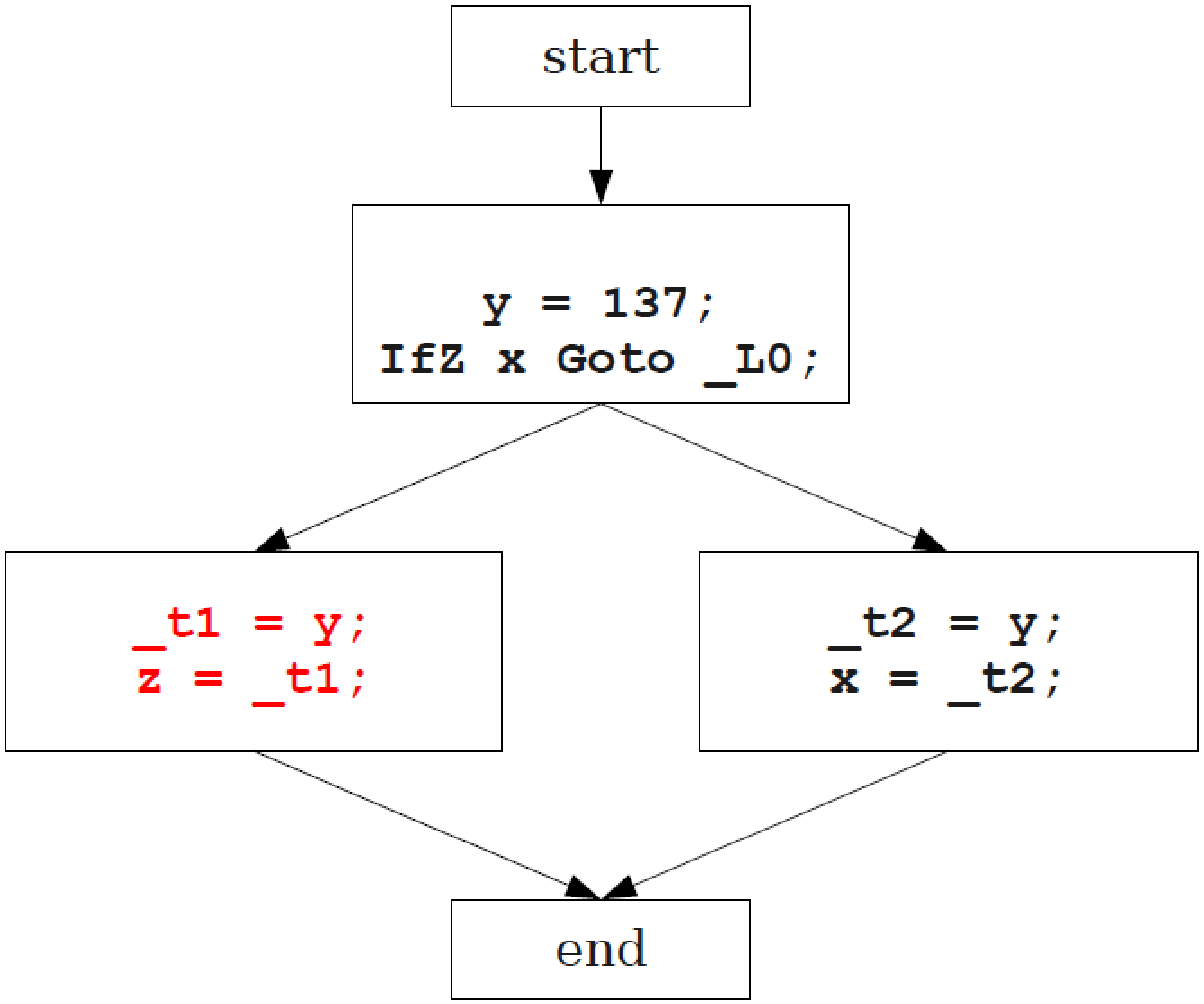
I-1. Exemple d'optimisations locales

```c
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



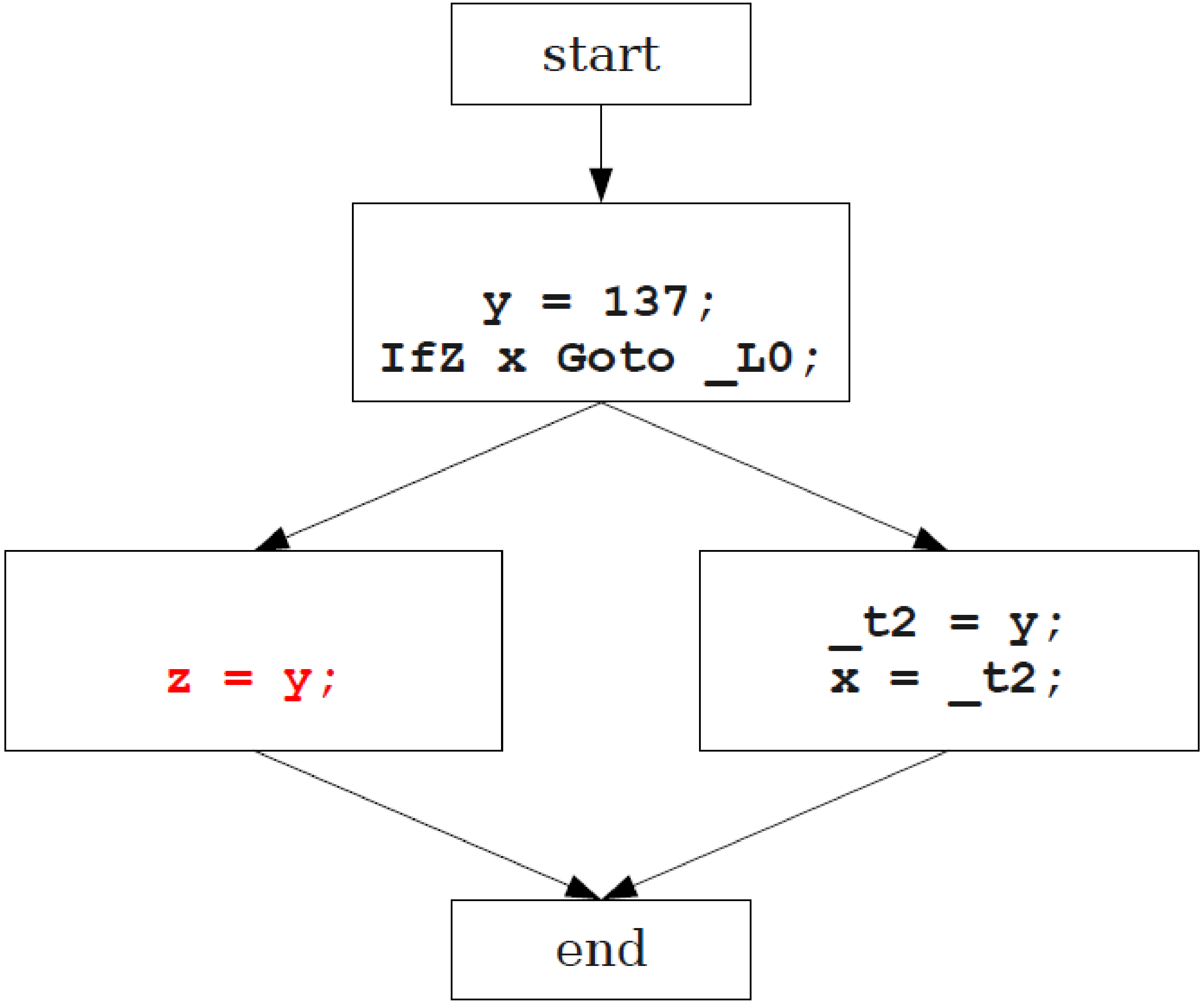
I-1. Exemple d'optimisations locales

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



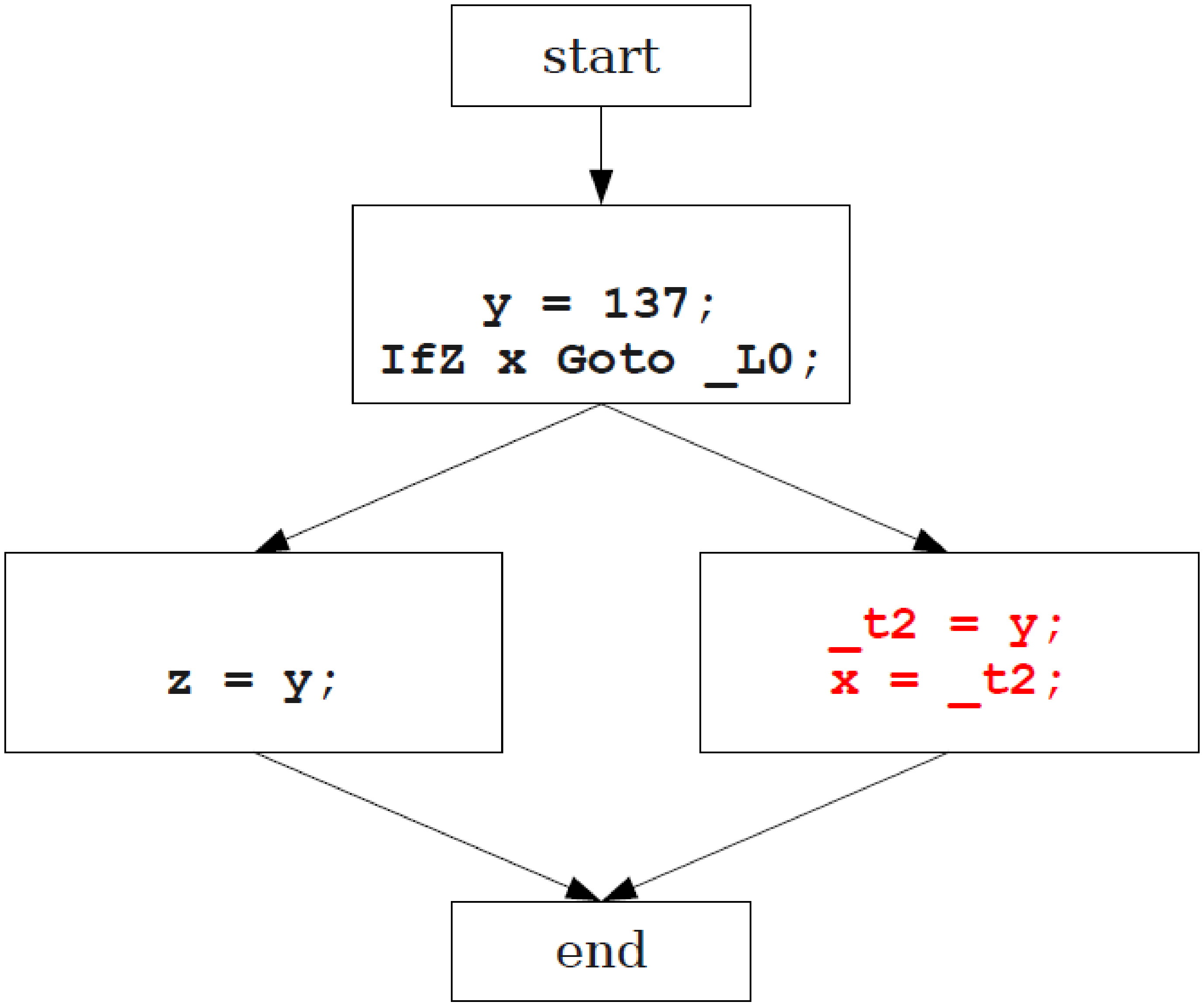
I-1. Exemple d'optimisations locales

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



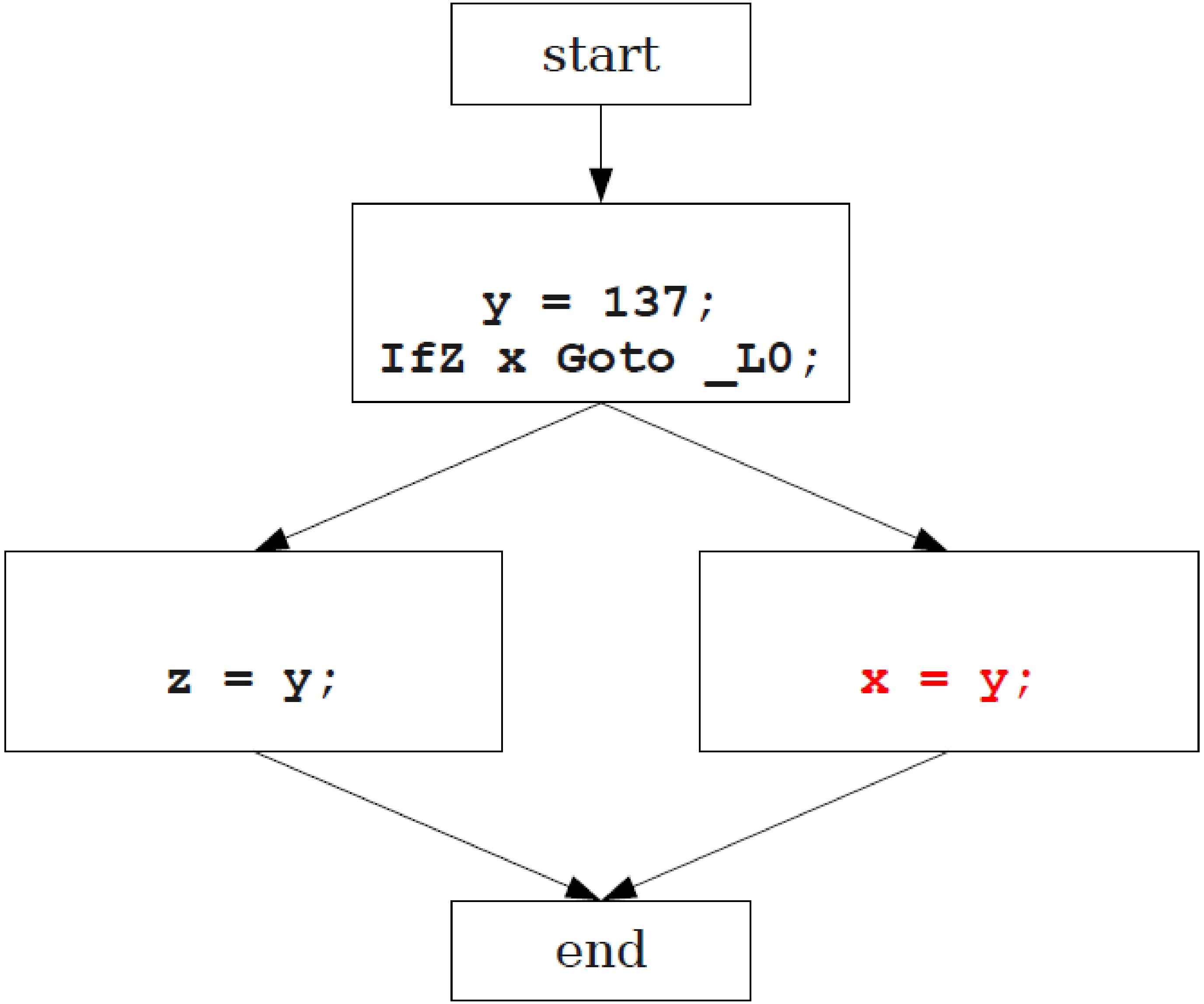
I-1. Exemple d'optimisations locales

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



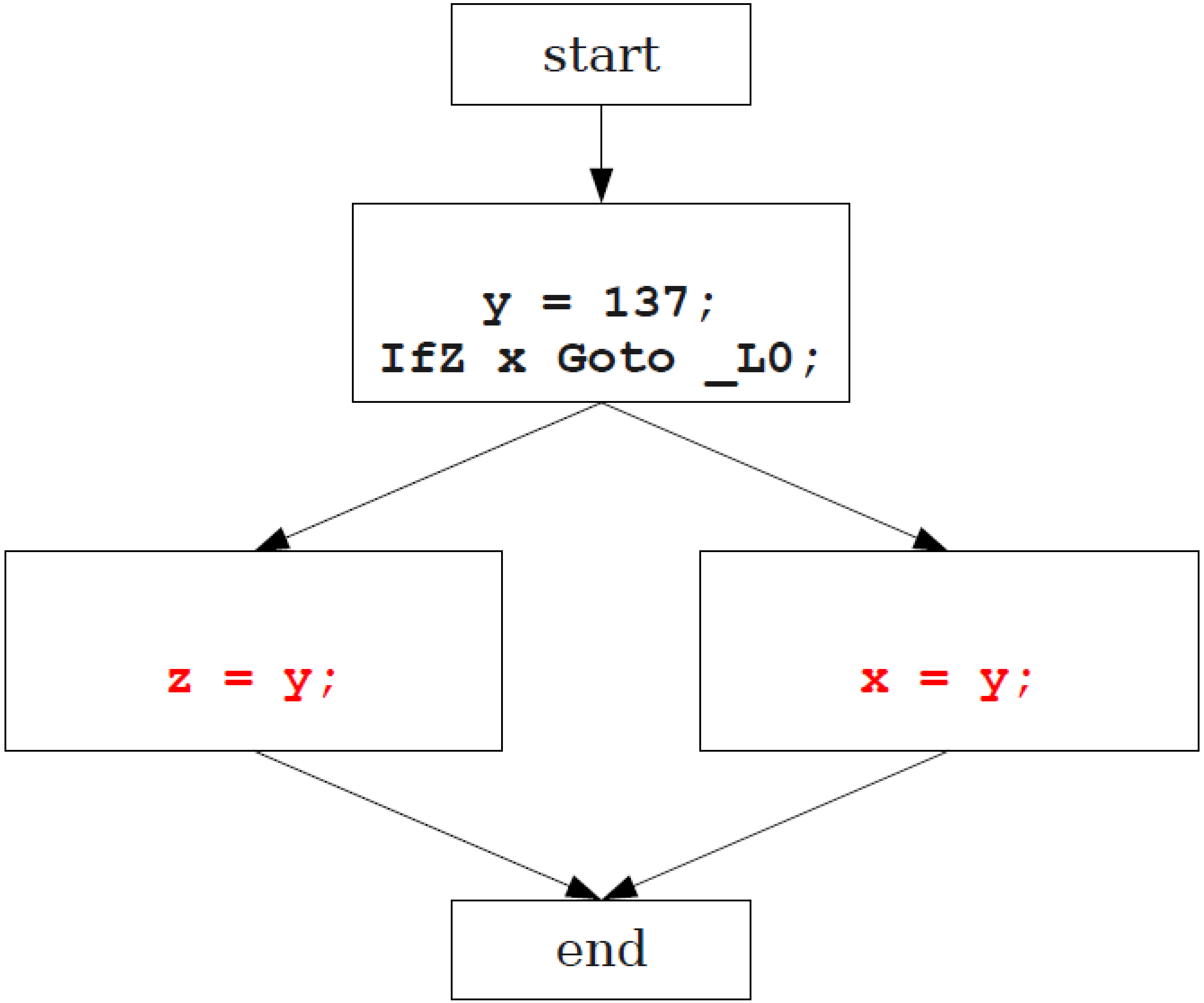
I-1. Exemple d'optimisations locales

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



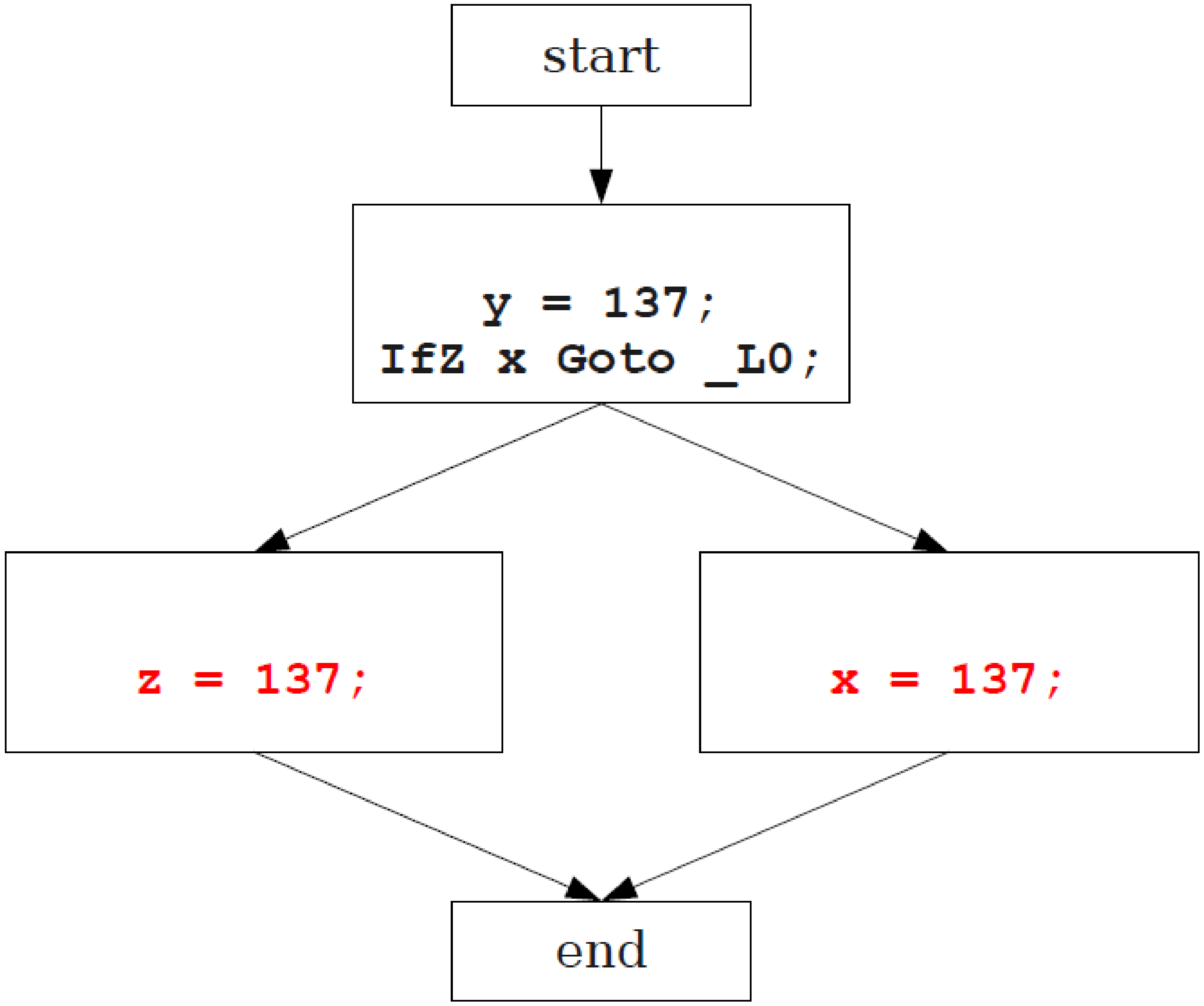
I-2. Exemple d'optimisations globales

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



I-2. Exemple d'optimisations globales

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



II- Optimisations locales

L’optimisation **locale** fonctionne juste dans un seul bloc de base.

II-1. Règles d'optimisations locales

a) **Élimination des sous-expressions communes**

b) **Propagation de copie**

c) **Élimination du code mort**

II- Règles d'optimisations locales

Exemple :

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.a) Élimination des sous-expressions communes

- Si nous avons deux affectations de variables:

$V_1 = a \; op \; b$

...

$V_2 = a \; op \; b$

et les valeurs de V_1 (*a* et *b)* n’ont pas changé entre les affectations, réécrire le code comme:

$V_1 = a \; op \; b$

...

$V_2 = V_1$

Cela permet d’éliminer le calcul inutile (*a op b*) et prépare des optimisations ultérieures.

II-1.a) Élimination des sous-expressions communes

Appliquer l'élimination des sous-expressions communes sur l'exemple précédent.

II-1.a) Élimination des sous-expressions communes

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.a) Élimination des sous-expressions communes

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.a) Élimination des sous-expressions communes

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.a) Élimination des sous-expressions communes

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.a) Élimination des sous-expressions communes

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.a) Élimination des sous-expressions communes

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

- Si nous avons une assignation de variable:

$$V_1 = V_2$$

Alors, tant que V_1 et V_2 ne sont pas réaffectés, nous pouvons réécrire les expressions sous la forme:

$$a = \dots V_1 \dots$$

comme

$$a = \dots V_2 \dots$$

à condition qu'une telle réécriture soit légale.

Cela aidera énormément pour les prochaines optimisations.

II-1.b) Propagation de copie

Appliquer la propagation de copie sur l’exemple précédent.

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp0 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp0 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.b) Propagation de copie

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.c) Élimination du code mort

- Une affectation à une variable *V* est dite **morte** si la valeur de cette affectation n'est jamais lue nulle part.

- **Élimination du code mort** enlève les affectations mortes du CI.

- Déterminer si une affectation est morte dépend de quelle variable est affectée et du moment où elle est affectée.

II-1.c) Élimination du code mort

Appliquer l'élimination du code mort sur l'exemple précédent.

II-1.c) Élimination du code mort

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.c) Élimination du code mort

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;

_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.c) Élimination du code mort

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;

_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.c) Élimination du code mort

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;

a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.c) Élimination du code mort

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;

a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.c) Élimination du code mort

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;


_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.c) Élimination du code mort

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;


_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.c) Élimination du code mort

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;


_tmp4 = _tmp0 + b ;

_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.c) Élimination du code mort

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;


_tmp4 = _tmp0 + b ;

_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.c) Élimination du code mort

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;



_tmp4 = _tmp0 + b ;


_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.c) Élimination du code mort

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;


_tmp4 = _tmp0 + b ;


_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-1.c) Élimination du code mort

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;



_tmp4 = _tmp0 + b ;



_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Résultat de l'optimisation locale de l'exemple

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
_tmp4 = _tmp0 + b ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

II-2. Application de l’optimisation locale

Pour obtenir un effet maximum, il se peut de devoir **appliquer plusieurs fois** ces optimisations.

Exemple :

```
b = a * a;
c = a * a;
d = b + c;
e = b + b;
```

II-2. Application de l'optimisation locale

```
b = a * a;
c = a * a;
d = b + c;
e = b + b;
```

Élimination des sous-expressions communes

II-2. Application de l'optimisation locale

```
b = a * a;
c = b;
d = b + c;
e = b + b;
```

Élimination des sous-expressions communes

II-2. Application de l’optimisation locale

```
b = a * a;
c = b;
d = b + c;
e = b + b;
```

Propagation de copie

II-2. Application de l'optimisation locale

```
b = a * a;
c = b;
d = b + b;
e = b + b;
```

Propagation de copie

II-2. Application de l'optimisation locale

```
b = a * a;
c = b;
d = b + b;
e = b + b;
```

Élimination des sous-expressions communes

II-2. Application de l’optimisation locale

```
b = a * a;
c = b;
d = b + b;
e = d;
```

Élimination des sous-expressions communes

II-3. Autres types d’optimisation locale

- **Simplification arithmétique**:

Remplacer les opérations "difficiles" par des opérations plus faciles.

Exemple: réécrire x = 4 * a; comme x = a $\ll$ 2;

- **Évaluation des constantes**:

Évaluer les expressions au moment de la compilation si elles ont une valeur constante.

Exemple: réécrire x = 4 * 5; comme x = 20.

II-4. Implémentation de l’optimisation locale

L’optimisation locale se base sur les **expressions disponibles**.

- Une expression est "**disponible**" si une variable contient la valeur de cette expression.
- Dans l’élimination des sous-expressions communes, on remplace une expression disponible par la variable qui la contient.
- Dans la propagation de copie, on remplace l’utilisation d’une variable par l’expression disponible qu’elle contient.

II-4. Implémentation de l'optimisation locale

L'idée de l'implémentation:

Créer un ensemble d'expressions disponibles, qui doit être à jour, et l'utiliser pour les remplacements dans les optimisations locales.

II-4. Implémentation de l'optimisation locale

Exemple:

```
  a = b;
  c = b;
d = a + b;
e = a + b;
  d = b;
f = a + b;
```

II-4. Implémentation de l’optimisation locale

Ensemble d’expressions disponibles:

{ }

```
  a = b;
  c = b;
d = a + b;
e = a + b;
  d = b;
f = a + b;
```

II-4. Implémentation de l'optimisation locale

Ensemble d'expressions disponibles:

```
   {  }
  a = b;
{ a = b }
  c = b;

d = a + b;

e = a + b;

  d = b;

f = a + b;
```

II-4. Implémentation de l'optimisation locale

Ensemble d'expressions disponibles:

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;

e = a + b;

d = b;

f = a + b;
```

II-4. Implémentation de l’optimisation locale

Ensemble d’expressions disponibles:

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;

d = b;

f = a + b;
```

II-4. Implémentation de l'optimisation locale

Ensemble d'expressions disponibles:

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;

f = a + b;
```

II-4. Implémentation de l'optimisation locale

Ensemble d'expressions disponibles:

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
```

II-4. Implémentation de l'optimisation locale

Ensemble d'expressions disponibles:

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

II-4. Implémentation de l'optimisation locale

Élimination des sous-expressions communes: { }

```
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

II-4. Implémentation de l'optimisation locale

Élimination des sous-expressions communes: { }

```
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

II-4. Implémentation de l'optimisation locale

Élimination des sous-expressions communes:

```
{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

II-4. Implémentation de l'optimisation locale

Élimination des sous-expressions communes:

```
{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

II-4. Implémentation de l'optimisation locale

Élimination des sous-expressions communes: { }

```
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

II-4. Implémentation de l'optimisation locale

Élimination des sous-expressions communes: **{ }**

```
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = a;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

II-4. Implémentation de l'optimisation locale

Élimination des sous-expressions communes: { }

```
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = a;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

II-4. Implémentation de l'optimisation locale

Élimination des sous-expressions communes: { }

```
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = a;
{ a = b, c = b, d = b, e = a + b }
f = e;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

II-4. Implémentation de l'optimisation locale

Élimination des sous-expressions communes:

```
a = b;
c = a;
d = a + b;
e = d;
d = a;
f = e;
```

II-4.a) Analyse de vivacité

L’analyse qui correspond à l’élimination du code mort est appelée: **analyse de vivacité**.

- Une variable est "**vivante**" à un point d’un programme si, plus tard dans le programme, sa valeur sera lue avant d’être à nouveau écrite.

- L’élimination du code mort fonctionne en calculant la vivacité pour chaque variable, puis en éliminant les affectations aux variables mortes.

II-4.a) Analyse de vivacité

- Pour savoir si une variable sera utilisée à un moment donné, nous parcourons les instructions dans un bloc de base dans l’ordre inverse.

- Initialement, certains petits ensembles de valeurs sont connus pour être actifs (ceux qui dépendent du programme particulier).

- Lorsque nous voyons l’instruction a = b + c :
  - Juste avant l’instruction, a n’est pas vivante, puisque sa valeur est sur le point d’être écrasée.
  - Juste avant l’instruction, b et c sont vivantes, puisque nous sommes sur le point de lire leurs valeurs.
  - (Et si on avait a = a + b :?)

II-4. Implémentation de l'optimisation locale

Analyse de vivacité :

```
a = b;
c = b;
d = a + b;
e = a + b;
d = b;
f = a + b;
```

II-4. Implémentation de l'optimisation locale

Analyse de vivacité :

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
```

variables vivantes à la sortie du bloc → **{b, d}**

II-4. Implémentation de l'optimisation locale

Analyse de vivacité :

```
a = b;

c = a;

d = a + b;

e = d;

d = a;
{ b, d, e }
f = e;
{ b, d }
```

II-4. Implémentation de l'optimisation locale

Analyse de vivacité :

```
a = b;

c = a;

d = a + b;

e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

II-4. Implémentation de l'optimisation locale

Analyse de vivacité :

```
a = b;

c = a;

d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

II-4. Implémentation de l'optimisation locale

Analyse de vivacité :

```
a = b;

c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

II-4. Implémentation de l'optimisation locale

Analyse de vivacité :

```
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

II-4. Implémentation de l'optimisation locale

Analyse de vivacité :

```
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

II-4. Implémentation de l'optimisation locale

Élimination du code mort:

```
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

II-4. Implémentation de l'optimisation locale

Élimination du code mort:

```
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }

{ b, d }
```

II-4. Implémentation de l'optimisation locale

Élimination du code mort:

```
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }

{ b, d }
```

II-4. Implémentation de l’optimisation locale

Élimination du code mort:

```
{ b }
a = b;
{ a, b }

{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }

{ b, d }
```

II-4. Implémentation de l'optimisation locale

Élimination du code mort:

```
a = b;



d = a + b;

e = d;

d = a;
```

II-4. Implémentation de l’optimisation locale

Analyse de vivacité II:

```
a = b;



d = a + b;

e = d;

d = a;
{ b, d }
```

II-4. Implémentation de l'optimisation locale

Analyse de vivacité II:

```
a = b;



d = a + b;

e = d;
{ a, b }
d = a;
{ b, d }
```

II-4. Implémentation de l'optimisation locale

Analyse de vivacité II:

```
a = b;



d = a + b;
{ a, b, d }
e = d;
{ a, b }
d = a;
{ b, d }
```

II-4. Implémentation de l’optimisation locale

Analyse de vivacité II:

```
a = b;

{ a, b }

d = a + b;
{ a, b, d }
e = d;
{ a, b }
d = a;
{ b, d }
```

II-4. Implémentation de l'optimisation locale

Analyse de vivacité II:

```
{ b }
a = b;

{ a, b }

d = a + b;
{ a, b, d }
e = d;
{ a, b }
d = a;
{ b, d }
```

II-4. Implémentation de l'optimisation locale

Élimination du code mort:

```
{ b }
a = b;

{ a, b }

d = a + b;
{ a, b, d }
e = d;
{ a, b }
d = a;
{ b, d }
```

II-4. Implémentation de l’optimisation locale

Élimination du code mort:

```
{ b }
a = b;

{ a, b }

d = a + b;
{ a, b, d }

{ a, b }
d = a;
{ b, d }
```

II-4. Implémentation de l'optimisation locale

Élimination du code mort:

```
a = b;

d = a + b;

d = a;
```

II-4. Implémentation de l'optimisation locale

Analyse de vivacité III:

```
a = b;


d = a + b;


d = a;
{b, d}
```

II-4. Implémentation de l'optimisation locale

Analyse de vivacité III:

```
a = b;




d = a + b;

{a, b}

d = a;
{b, d}
```

II-4. Implémentation de l'optimisation locale

Analyse de vivacité III:

```
a = b;

{a, b}

d = a + b;

{a, b}

d = a;
{b, d}
```

II-4. Implémentation de l’optimisation locale

Analyse de vivacité III:

```
{b}
a = b;

{a, b}

d = a + b;

{a, b}

d = a;
{b, d}
```

II-4. Implémentation de l'optimisation locale

Élimination du code mort:

```
{b}
a = b;

{a, b}

d = a + b;

{a, b}

d = a;
{b, d}
```

II-4. Implémentation de l'optimisation locale

Élimination du code mort:

```
{b}
a = b;

{a, b}



{a, b}

d = a;
{b, d}
```

II-4. Implémentation de l'optimisation locale

Élimination du code mort:

```
a = b;




d = a;
```

II-4.b) Algorithme combiné

```
a = b;
c = a;
d = a + b;
e = d;
d = a;
f = e;
```

II-4.b) Algorithme combiné

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
{b, d}
```

II-4.b) Algorithme combiné

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
{b, d}
```

II-4.b) Algorithme combiné

```
a = b;
c = a;
d = a + b;
e = d;
d = a;
```

{b, d}

II-4.b) Algorithme combiné

```
a = b;

c = a;

d = a + b;

e = d;
{a, b}
d = a;


{b, d}
```

II-4.b) Algorithme combiné

```
a = b;

c = a;

d = a + b;

e = d;
{a, b}
d = a;


{b, d}
```

II-4.b) Algorithme combiné

```
a = b;

c = a;

d = a + b;
```

{a, b}

```
d = a;
```

{b, d}

II-4.b) Algorithme combiné

```
a = b;

c = a;

d = a + b;


{a, b}
d = a;


{b, d}
```

II-4.b) Algorithme combiné

```
a = b;

c = a;




{a, b}
d = a;


{b, d}
```

II-4.b) Algorithme combiné

```
a = b;

c = a;

{a, b}
d = a;

{b, d}
```

II-4.b) Algorithme combiné

```
a = b;
```

```
{a, b}
d = a;
```

```
{b, d}
```

II-4.b) Algorithme combiné

```
{b}
a = b;

{a, b}
d = a;

{b, d}
```

II-4.b) Algorithme combiné

```
a = b;
```

```
d = a;
```

III- Optimisation globale

- Une **analyse globale** est une analyse sur le graphe de flot de contrôle en entier.
- Elle est plus puissante et plus compliquée que l’analyse locale.
- La plupart des optimisations locales peuvent être appliquées au niveau globale.
- Exemple d’optimisations globales:
  - Élimination globale du code mort
  - Propagation constante globale
  - Élimination partielle des redondances.

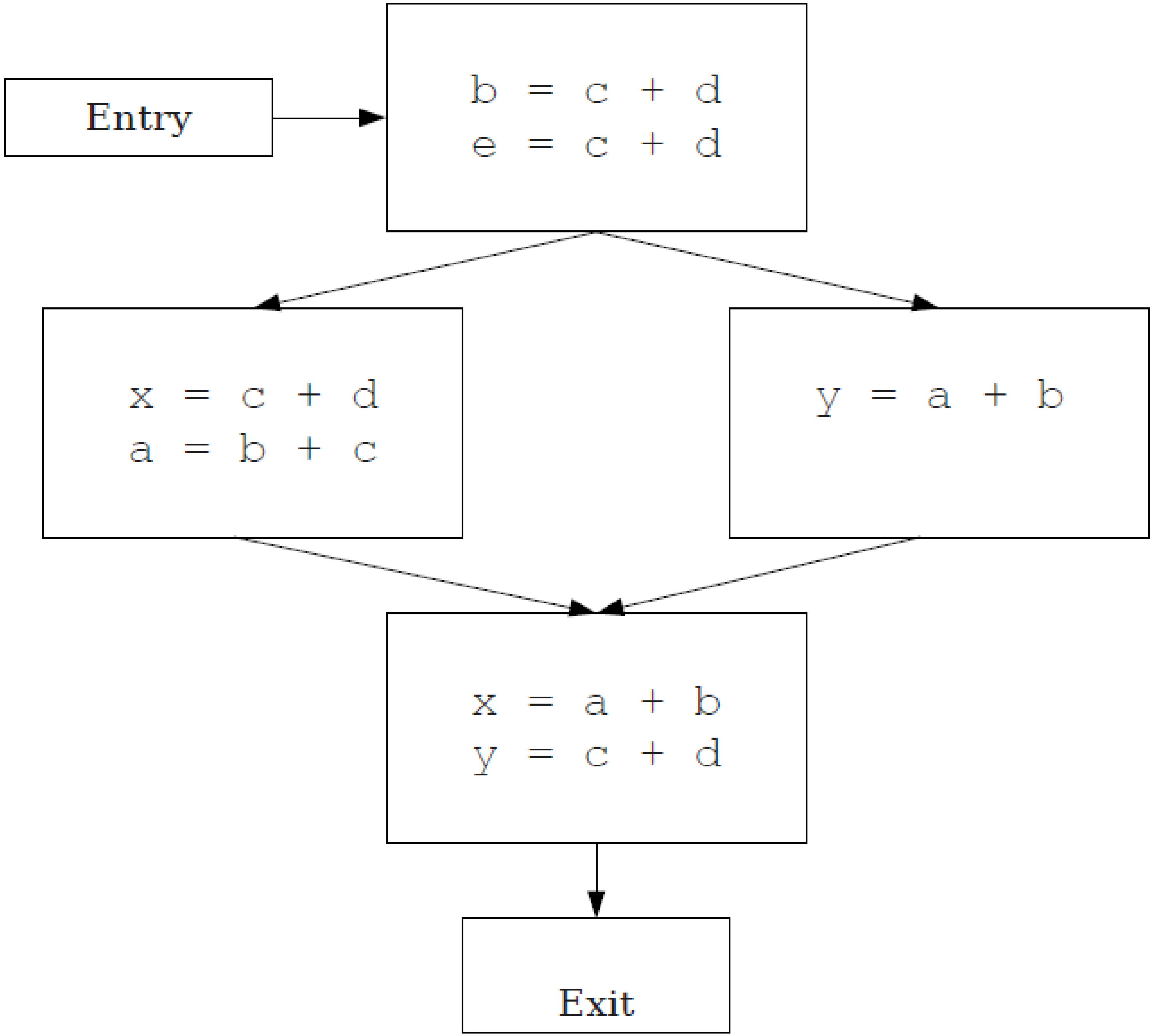
III-1. Élimination globale du code mort

- L’élimination locale du code mort avait besoin de connaître les variables vivantes à la sortie d’un bloc basique.

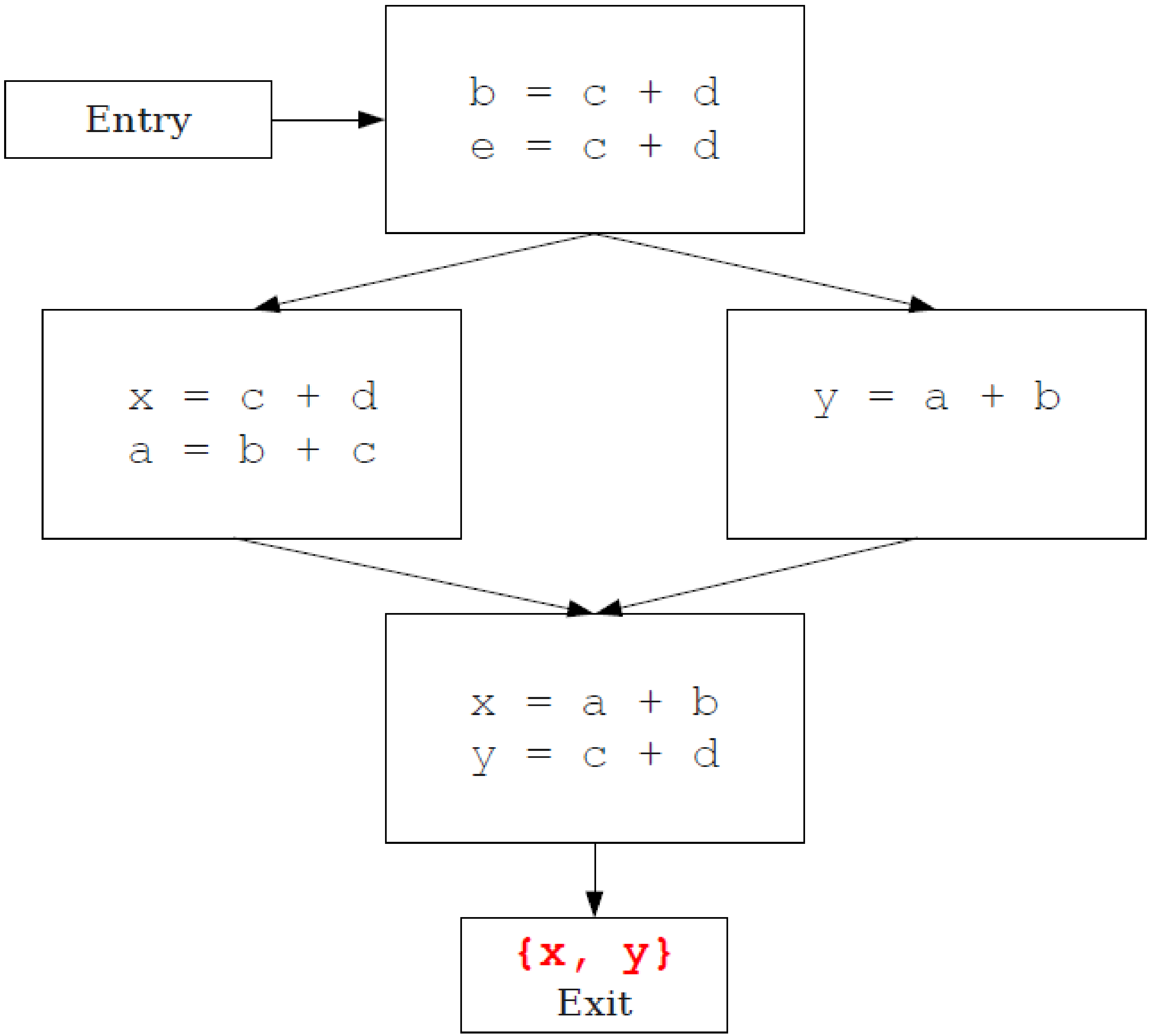
- Cette information ne peut être calculée que dans le cadre d’une analyse globale.

- Comment modifier l’analyse de vivacité pour gérer un CFG?

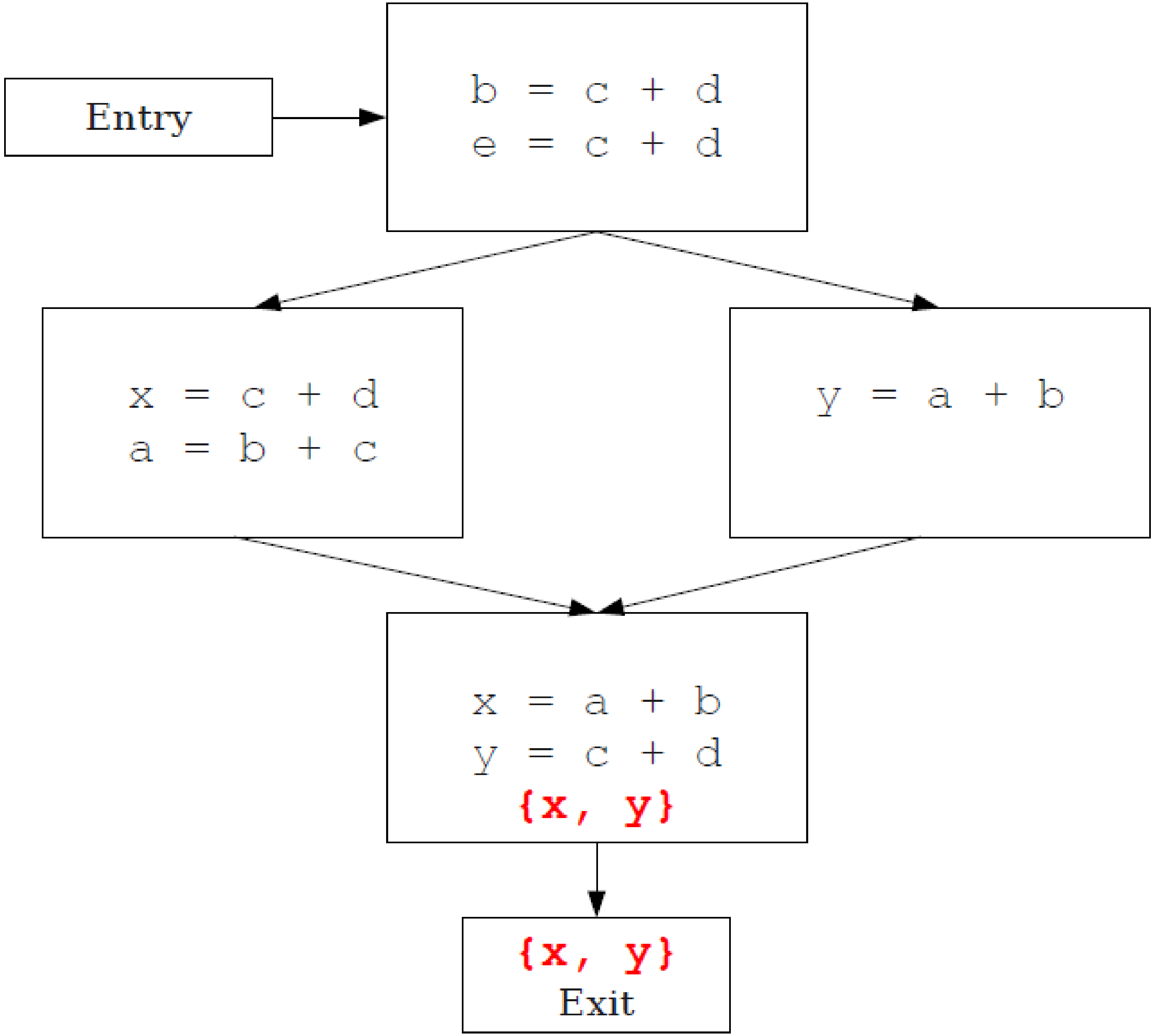
III-1. Élimination globale du code mort



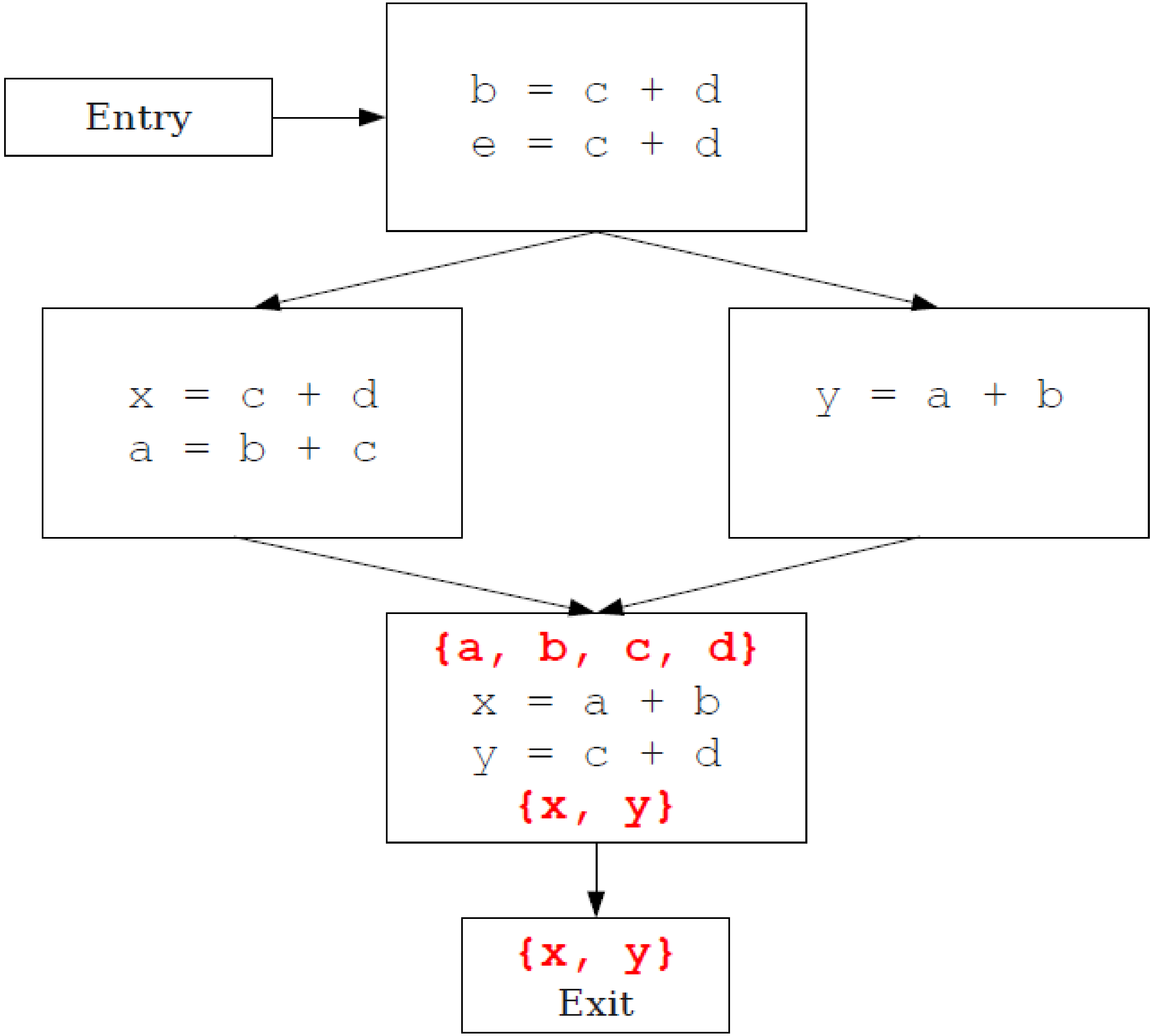
III-1. Élimination globale du code mort



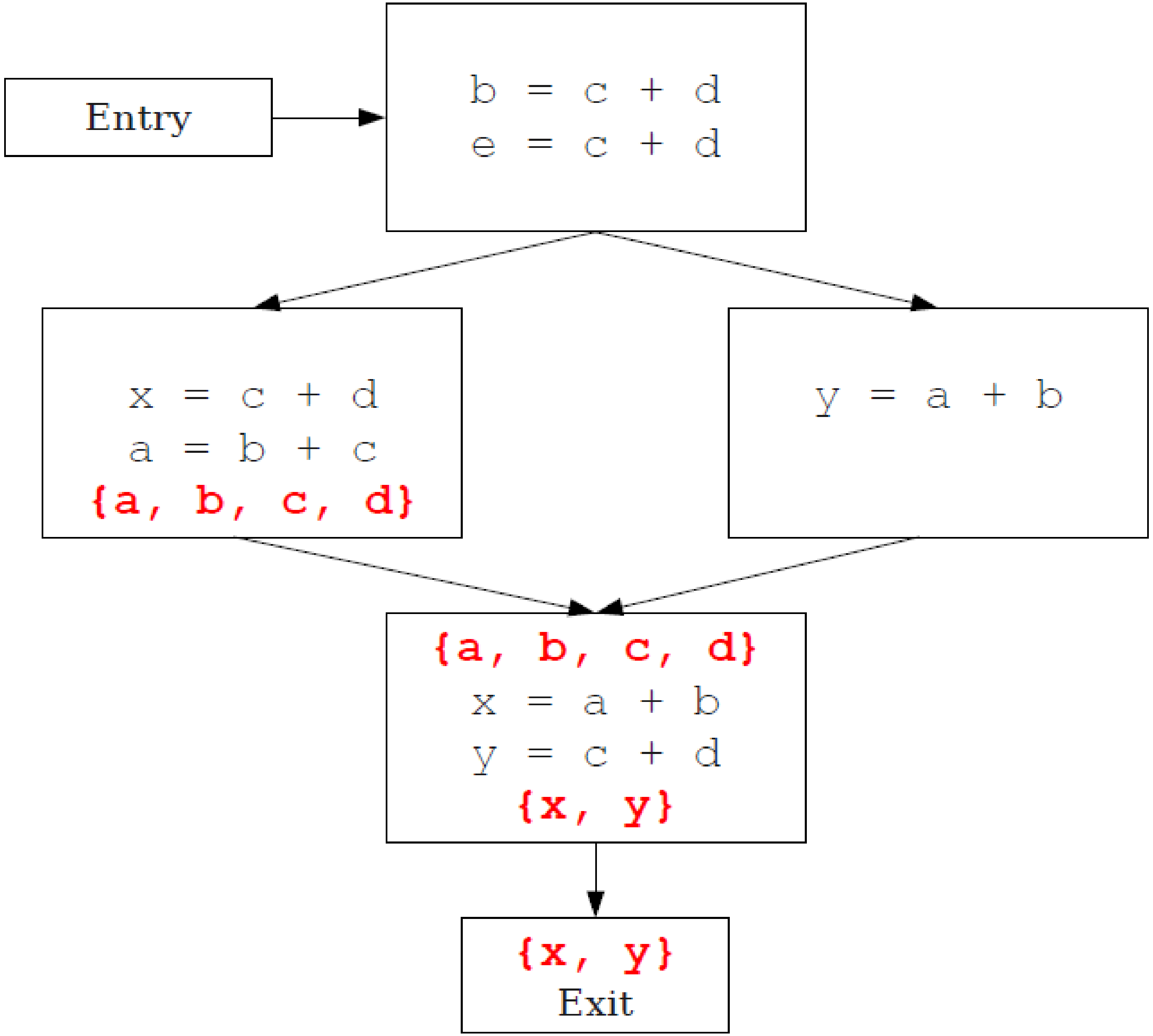
III-1. Élimination globale du code mort



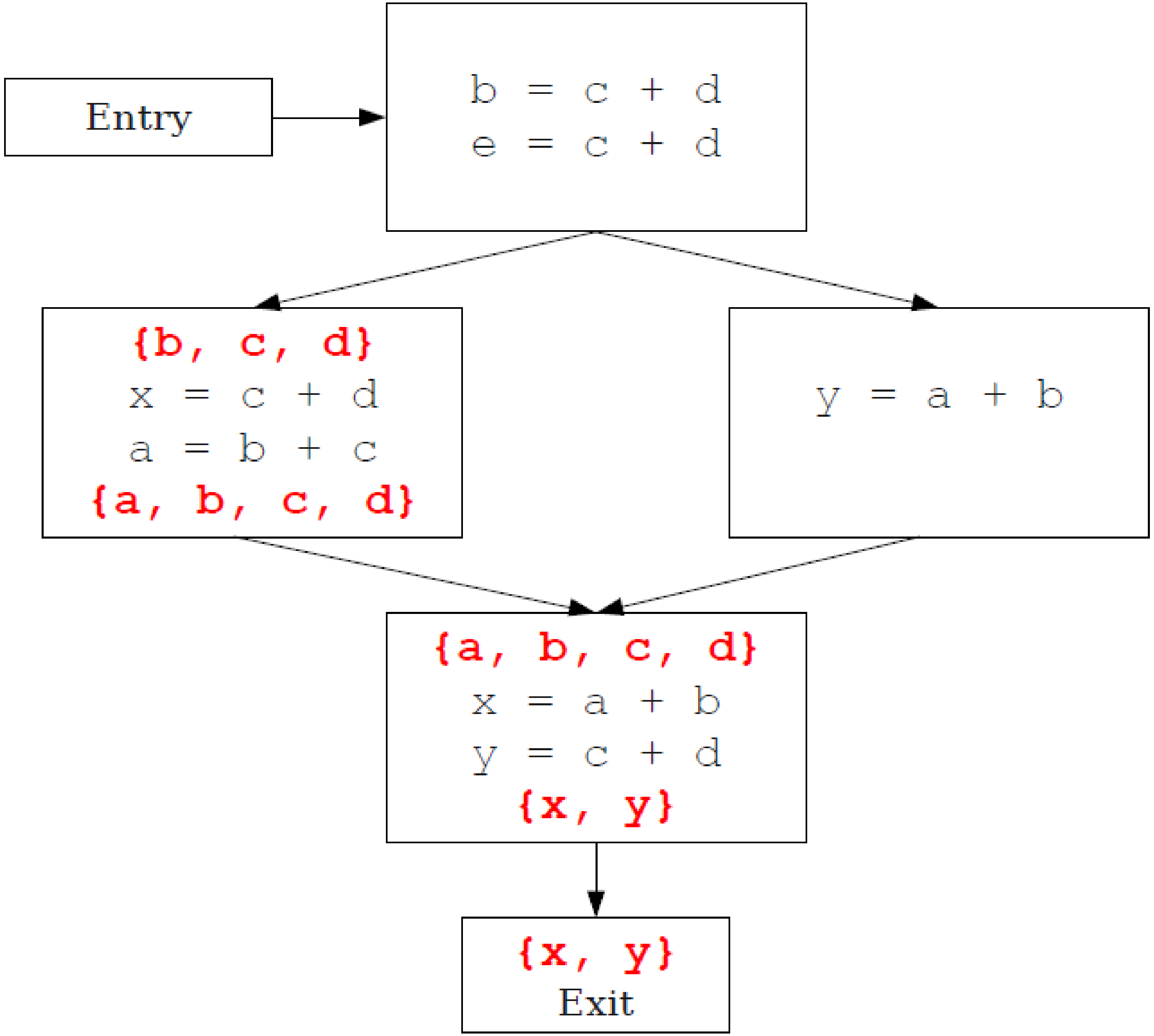
III-1. Élimination globale du code mort



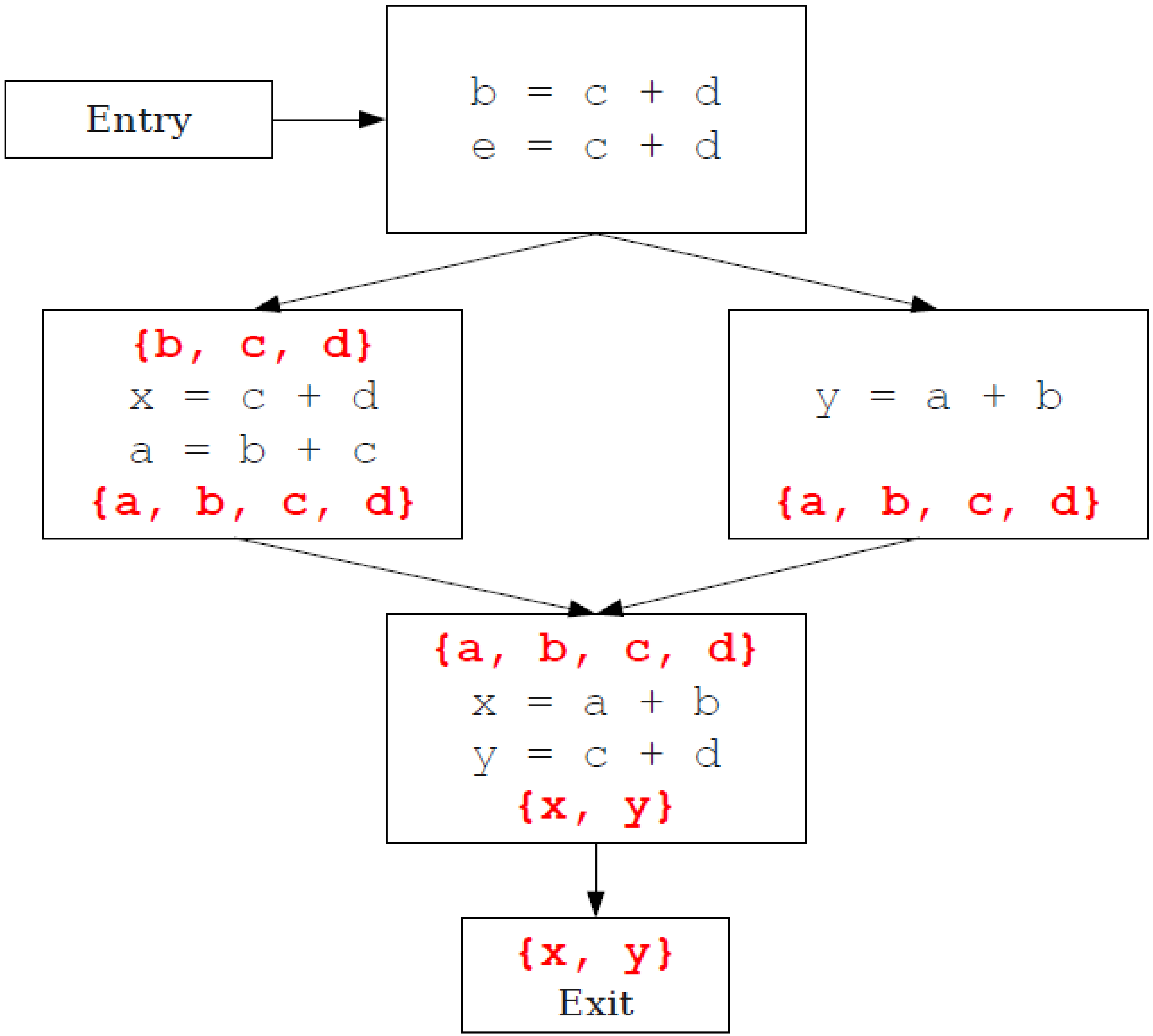
III-1. Élimination globale du code mort



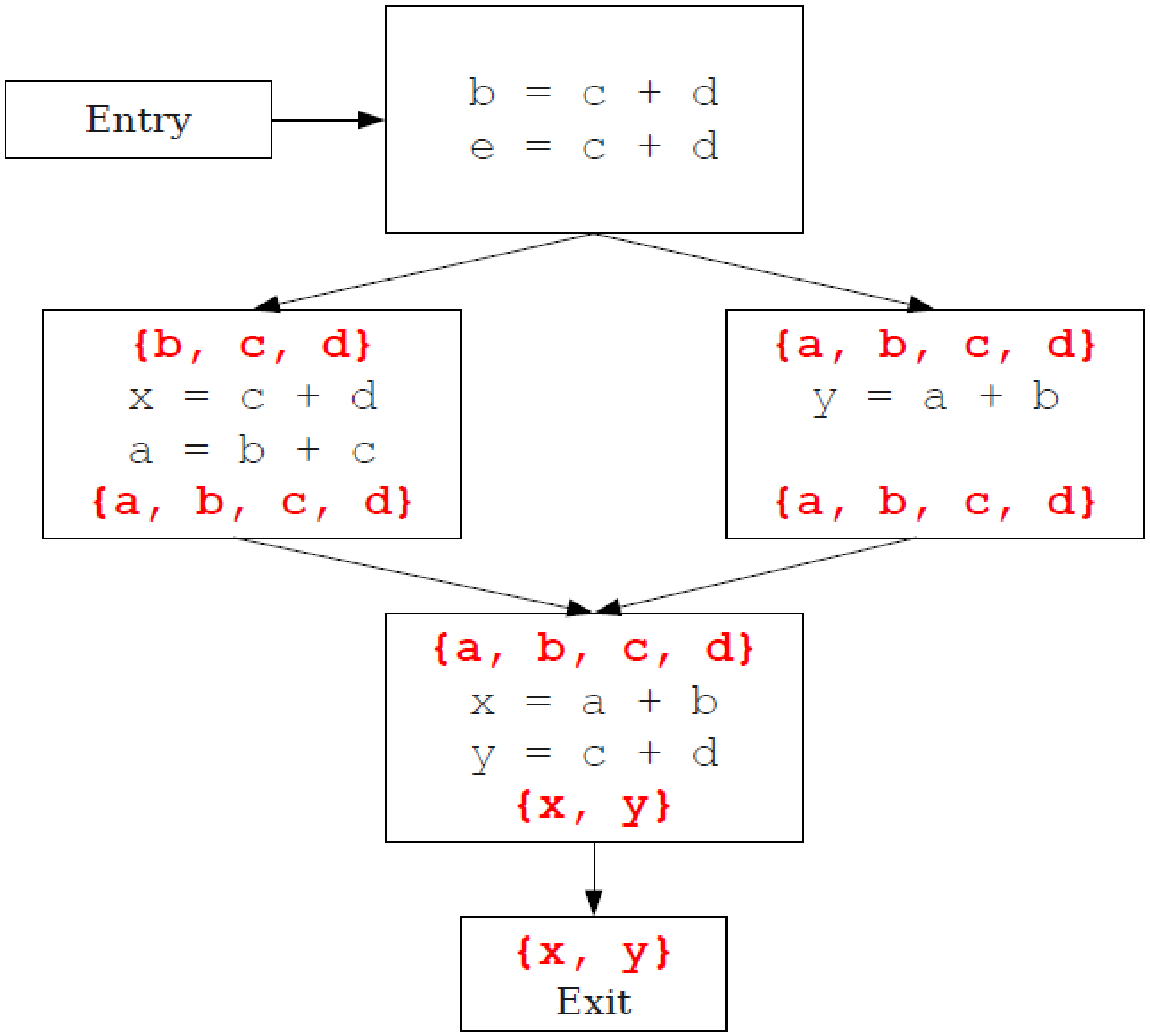
III-1. Élimination globale du code mort



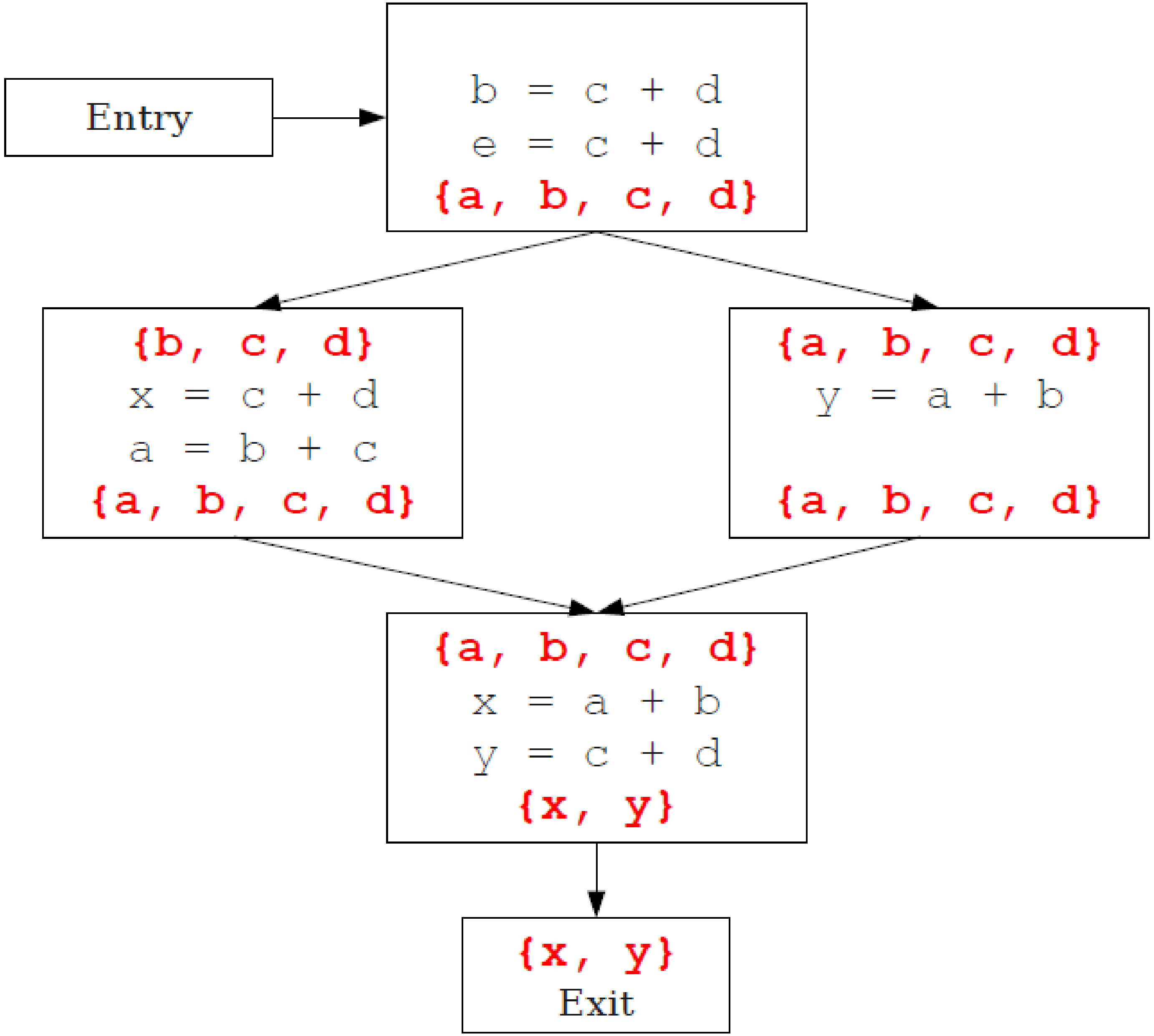
III-1. Élimination globale du code mort



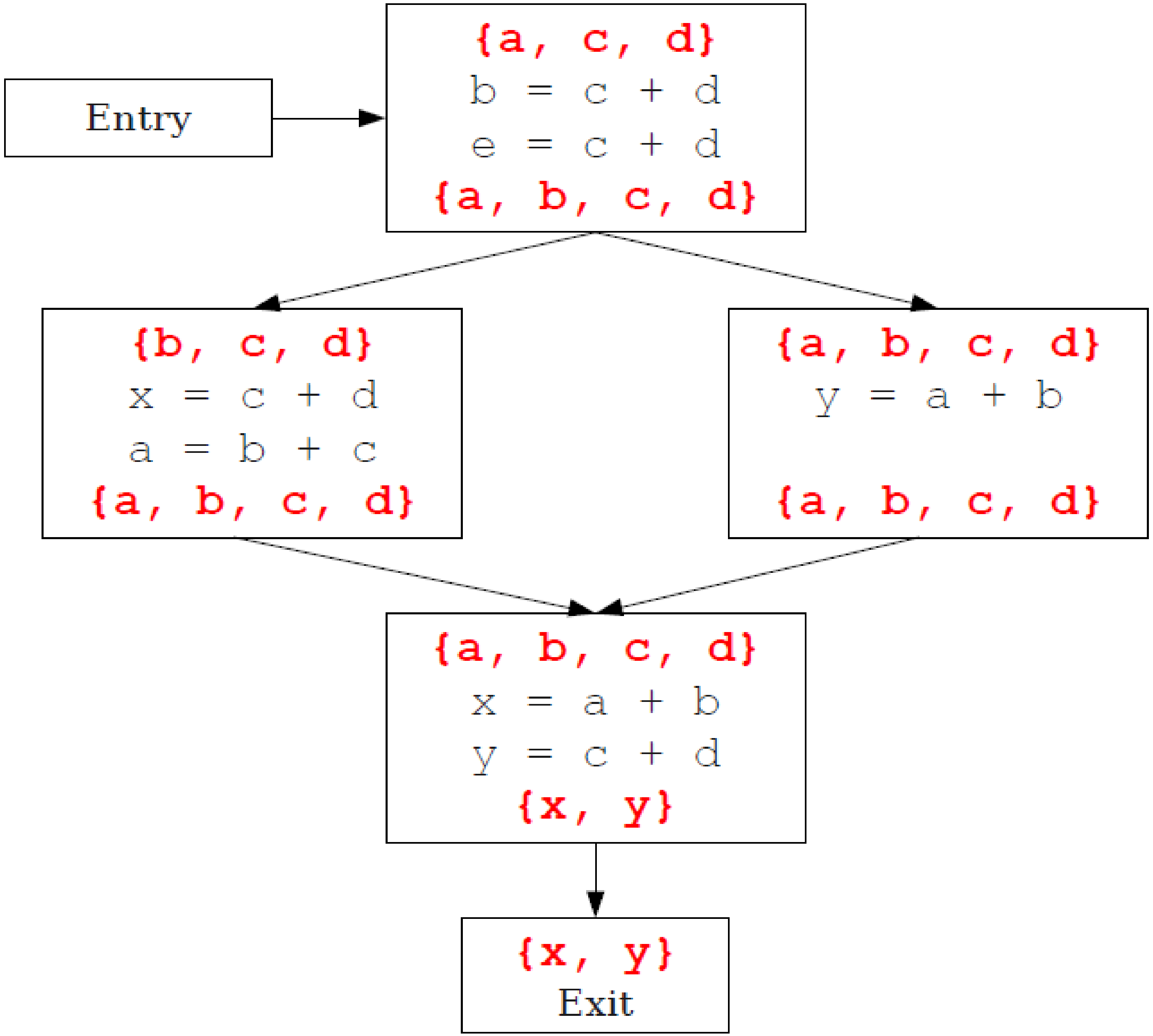
III-1. Élimination globale du code mort



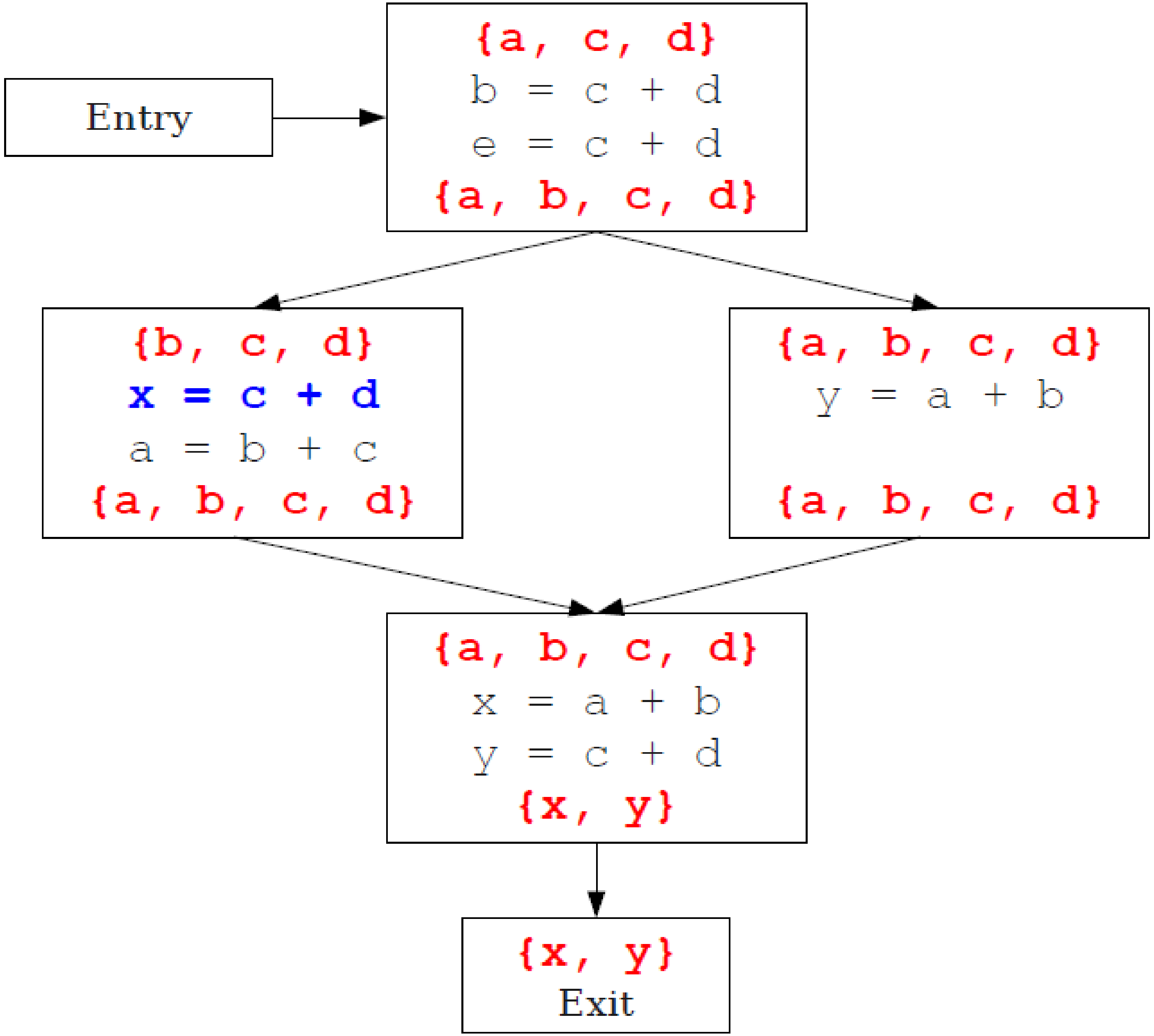
III-1. Élimination globale du code mort



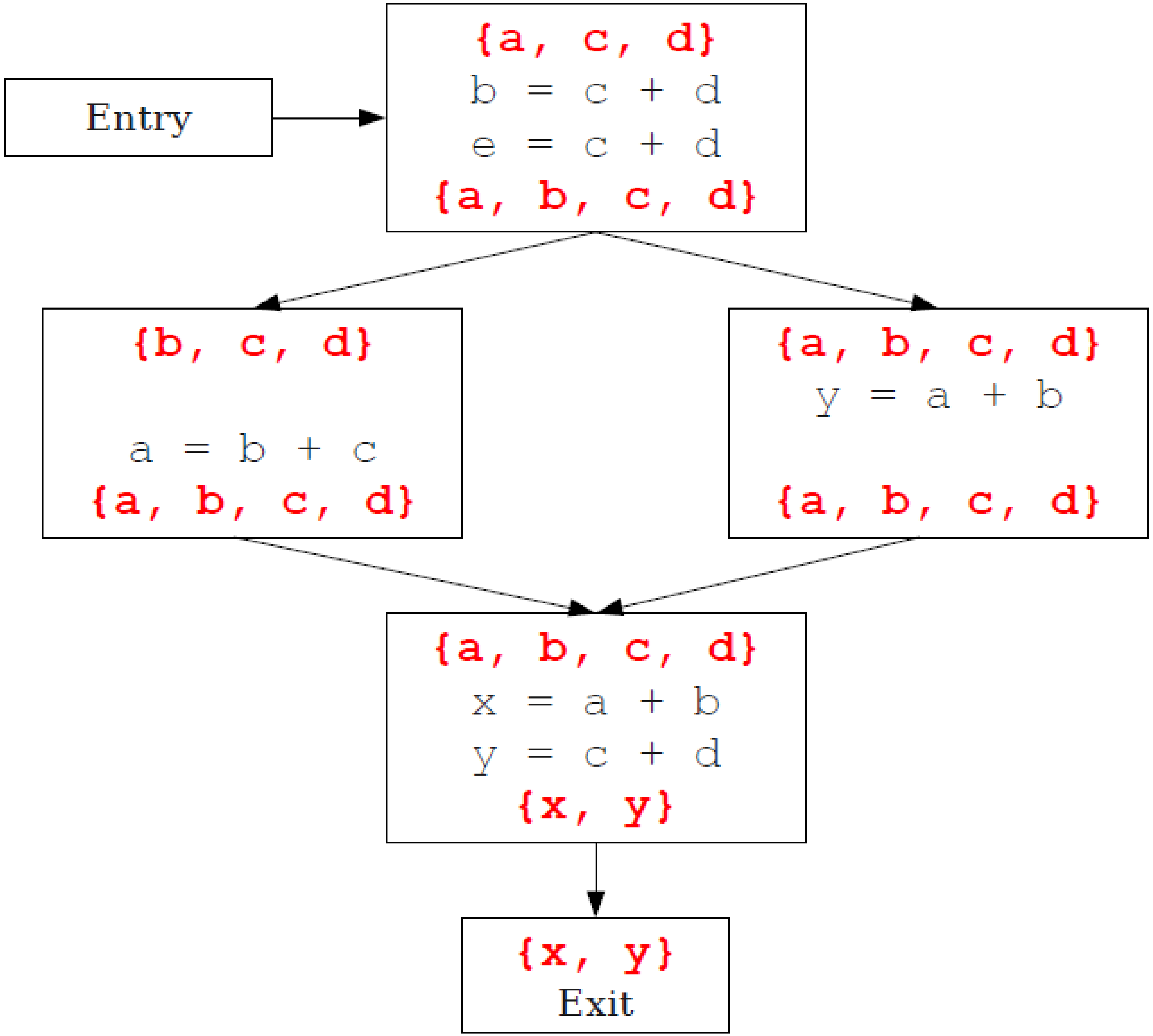
III-1. Élimination globale du code mort



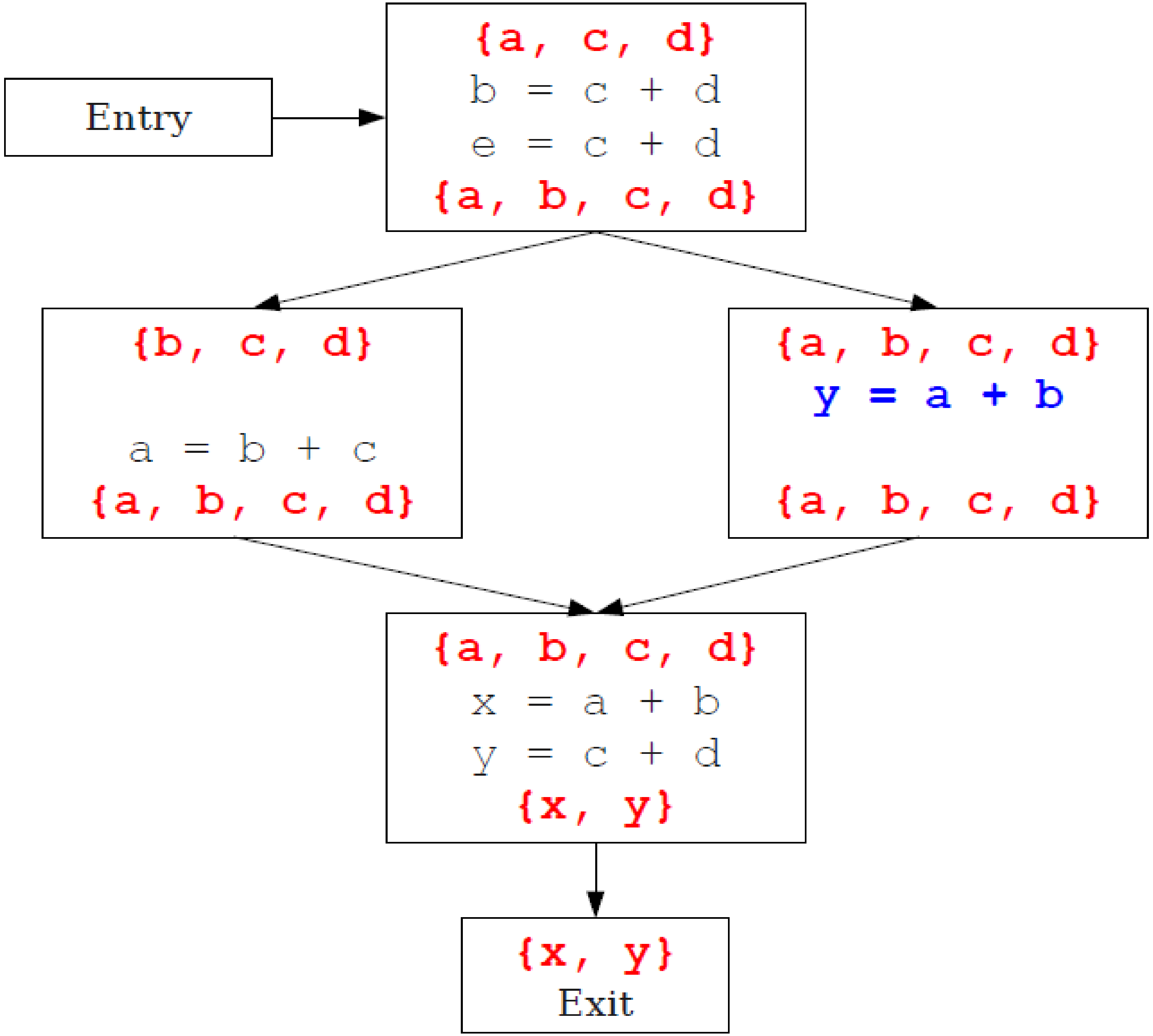
III-1. Élimination globale du code mort



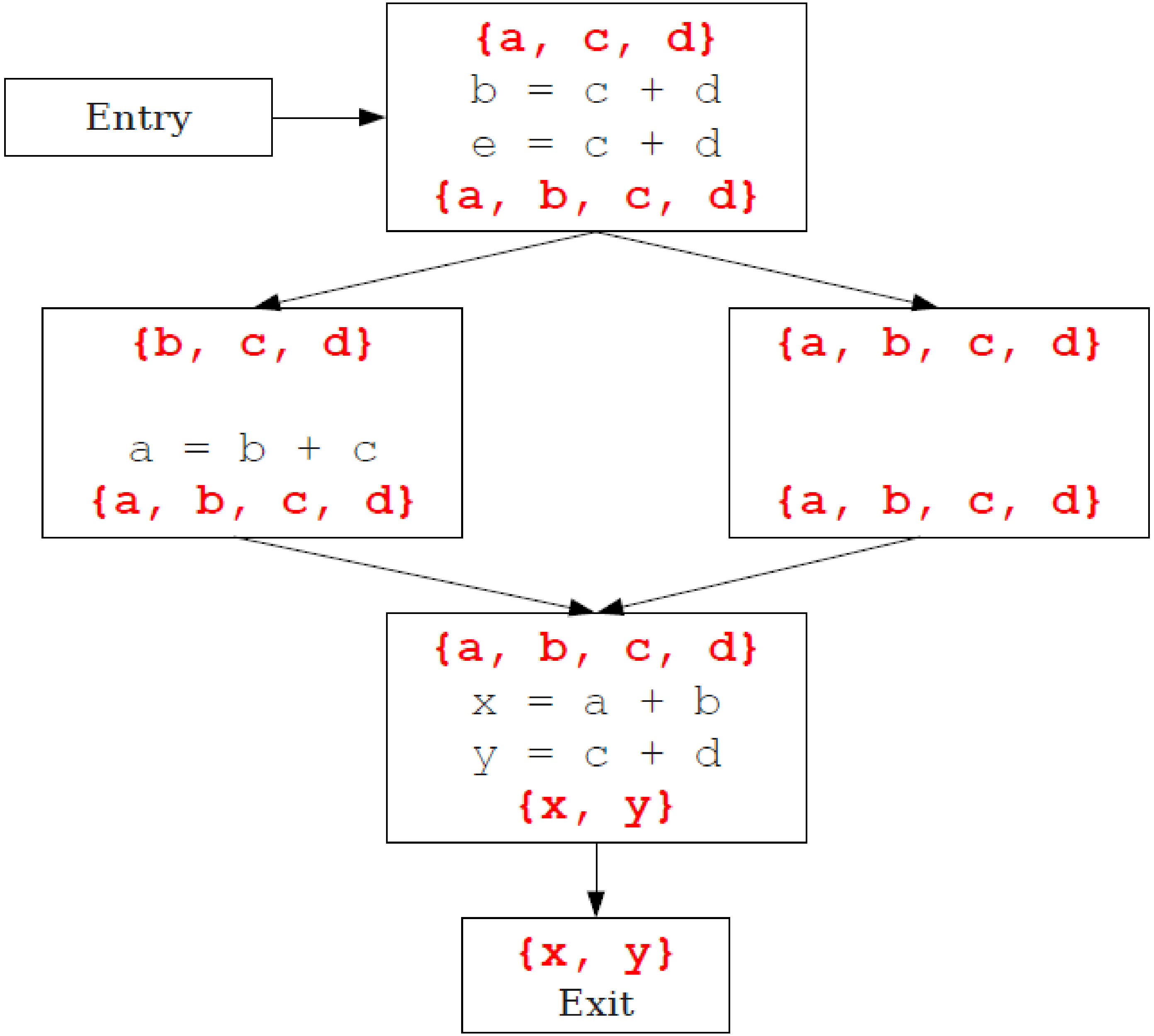
III-1. Élimination globale du code mort



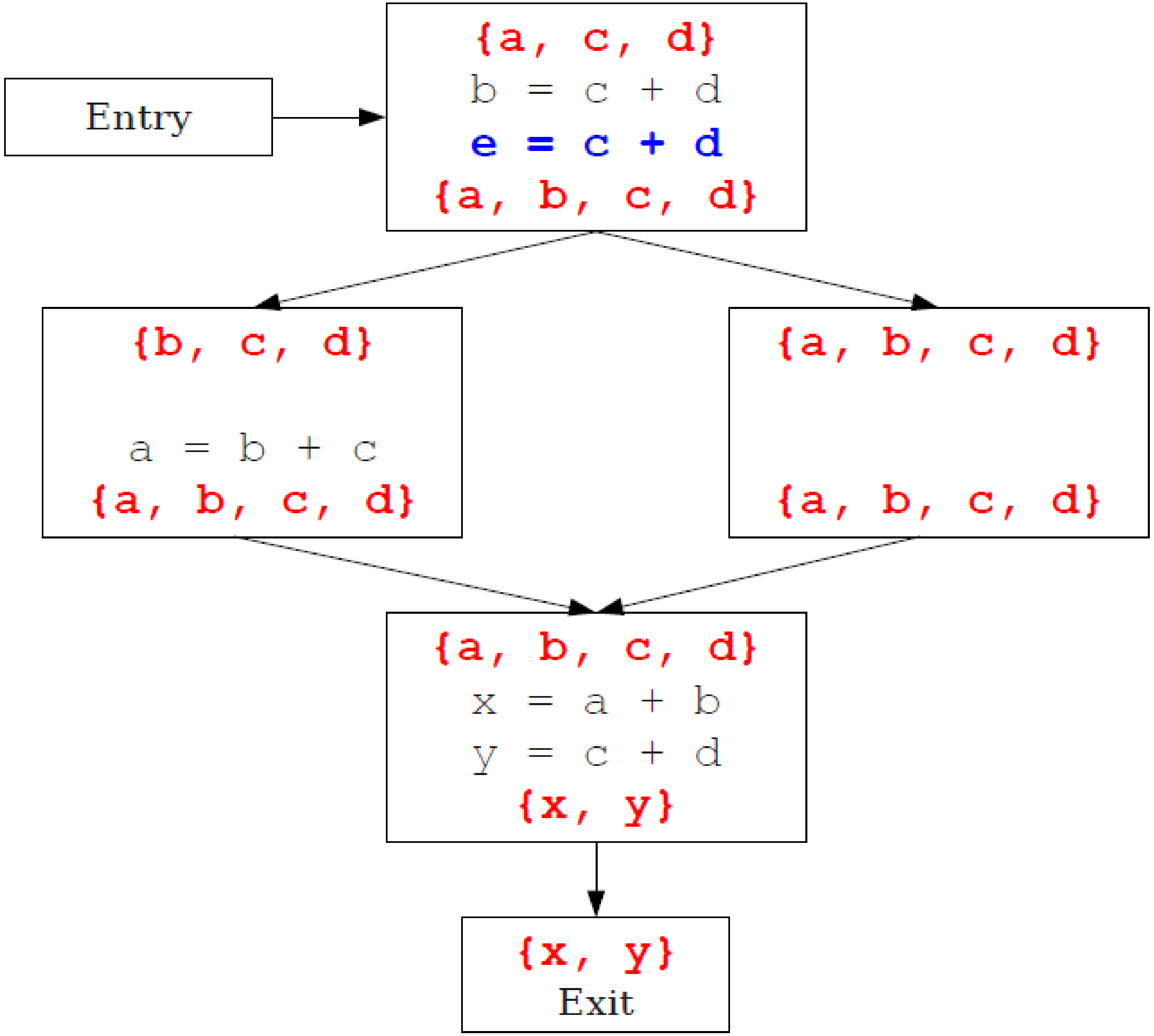
III-1. Élimination globale du code mort



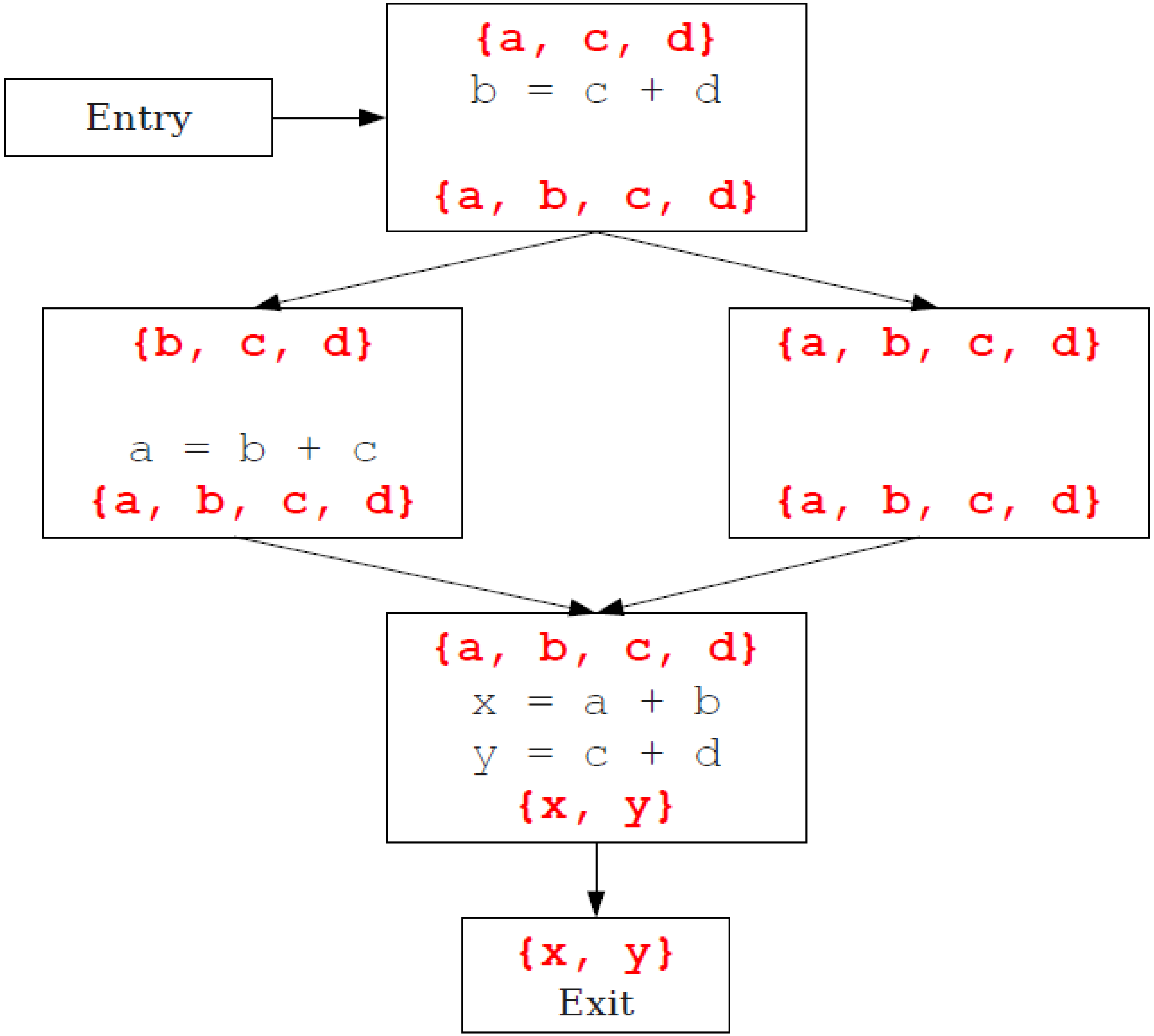
III-1. Élimination globale du code mort



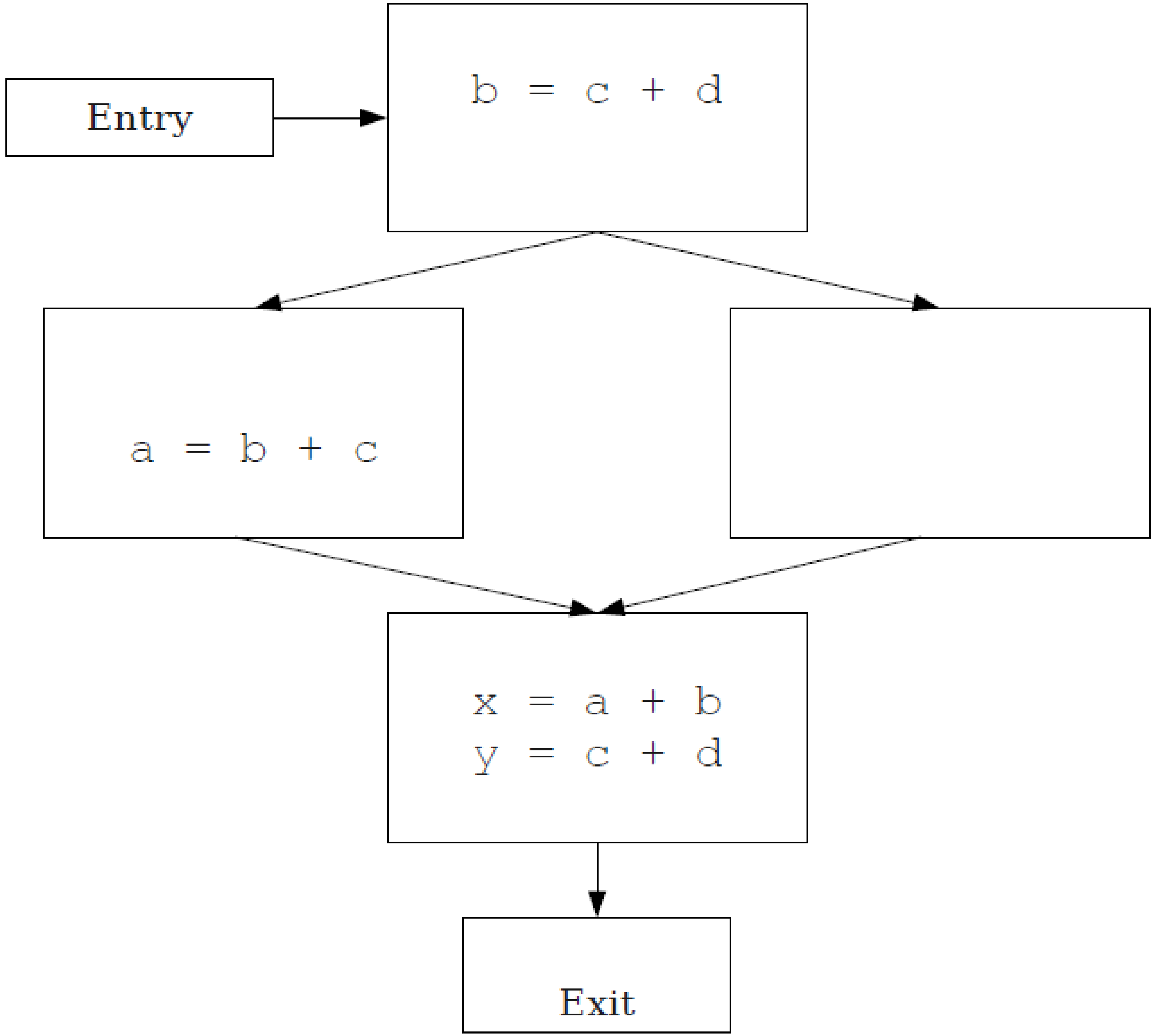
III-1. Élimination globale du code mort



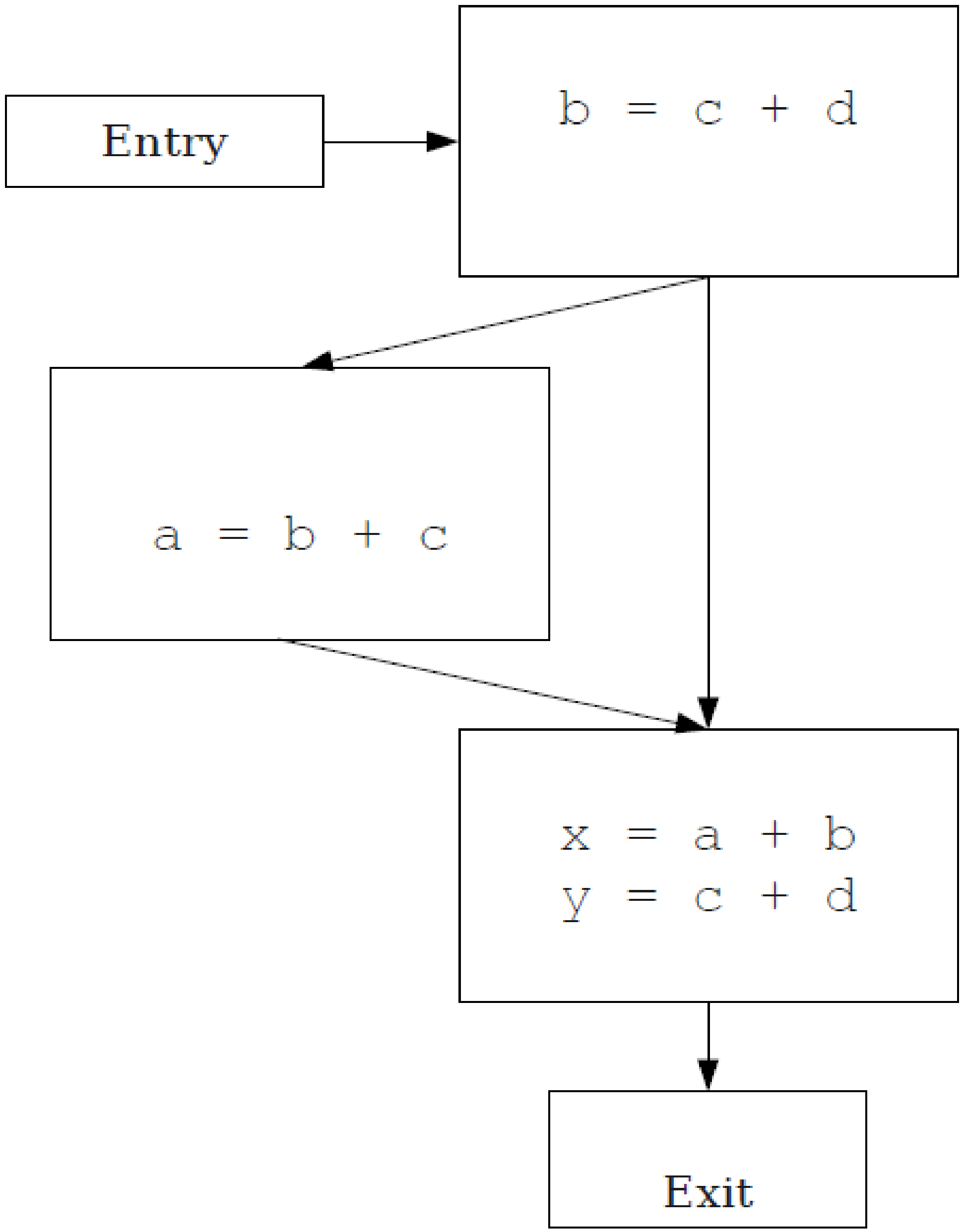
III-1. Élimination globale du code mort



III-1. Élimination globale du code mort



III-1. Élimination globale du code mort



III- Optimisations globales

Les différences majeures avec l’optimisation locale:

- Dans une analyse locale, chaque instruction a exactement un prédécesseur.
- Dans une analyse globale, chaque instruction peut avoir plusieurs prédécesseurs.
- Une analyse globale doit pouvoir combiner des informations provenant de tous les prédécesseurs d’un bloc de base.

III- Optimisations globales

Les différences majeures avec l’optimisation locale:

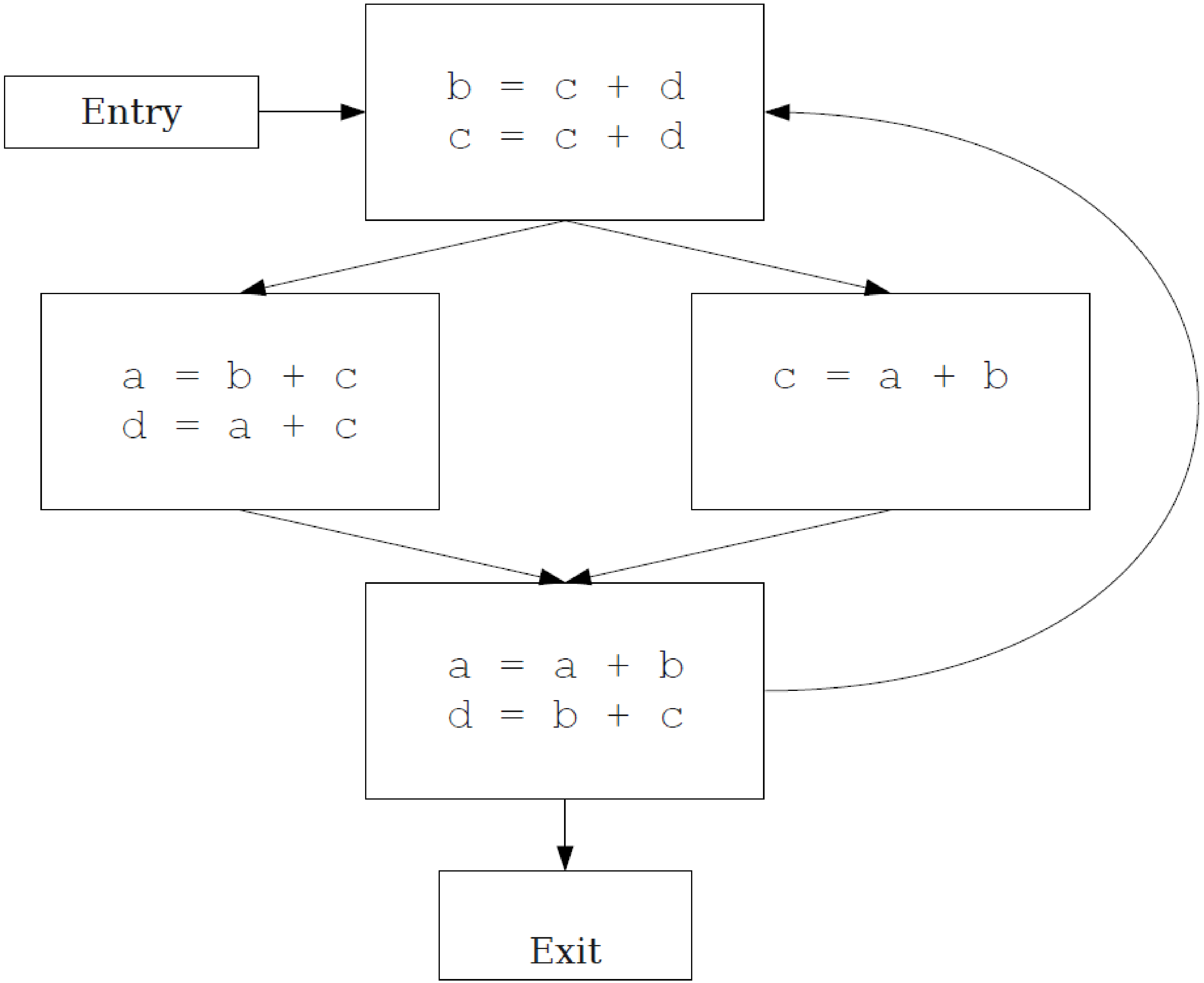
- Dans une analyse locale, il n’y a qu’un seul chemin possible à travers un bloc de base.
- Dans une analyse globale, il peut y avoir plusieurs chemins à travers un CFG.
- Il peut être nécessaire de recalculer les valeurs plusieurs fois à mesure que d’autres informations deviennent disponibles (sans boucle infinie).

III-2. CFG avec boucles

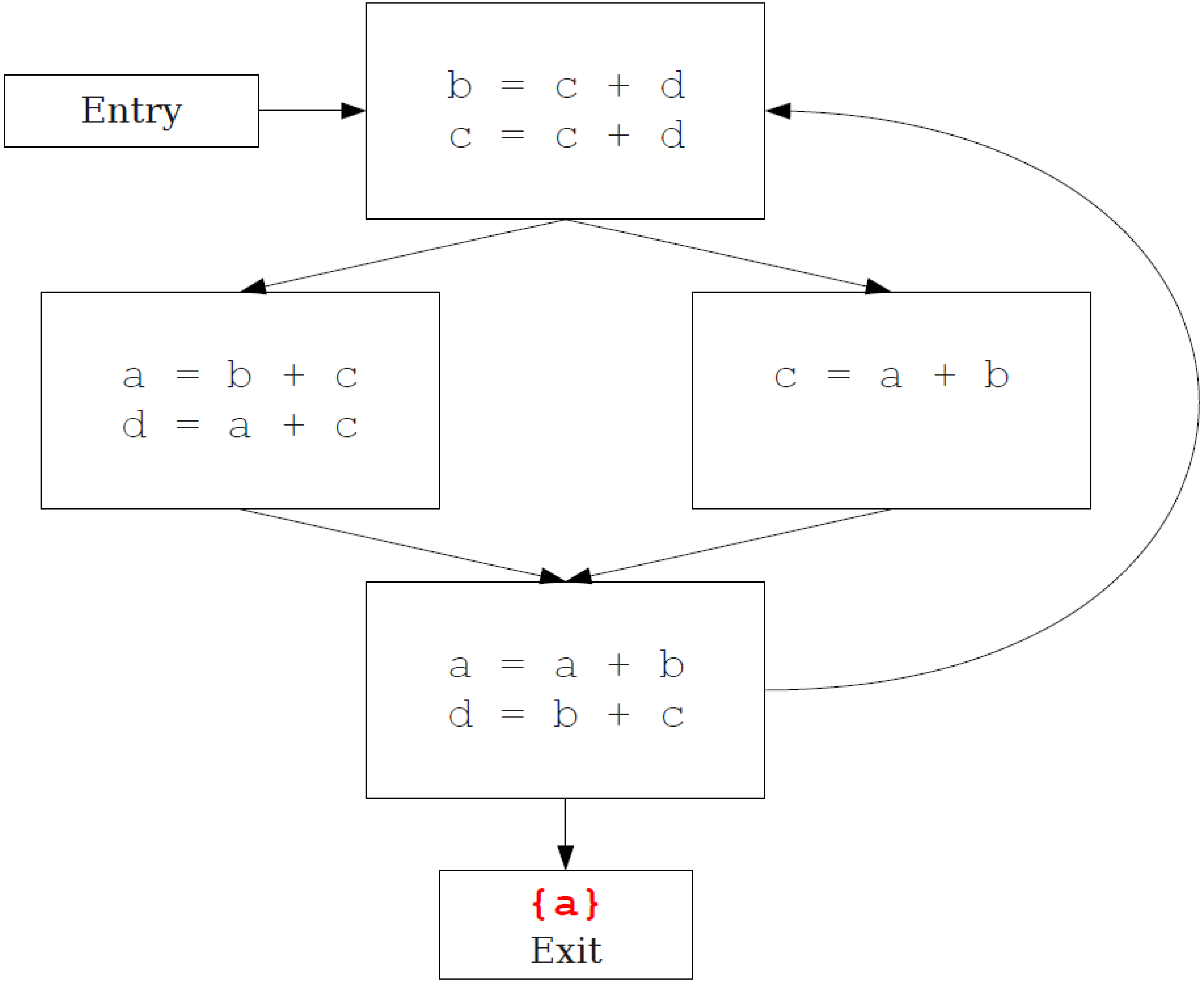
Approximation: Supposons que chaque chemin possible à travers le CFG correspond à une exécution valide.

- Inclut tous les chemins réalisables, mais aussi d'autres chemins supplémentaires.
- Rend l'analyse faisable.
- Peut rendre l'analyse moins précise (mais toujours correcte).

III-2. CFG avec boucles



III-2. CFG avec boucles



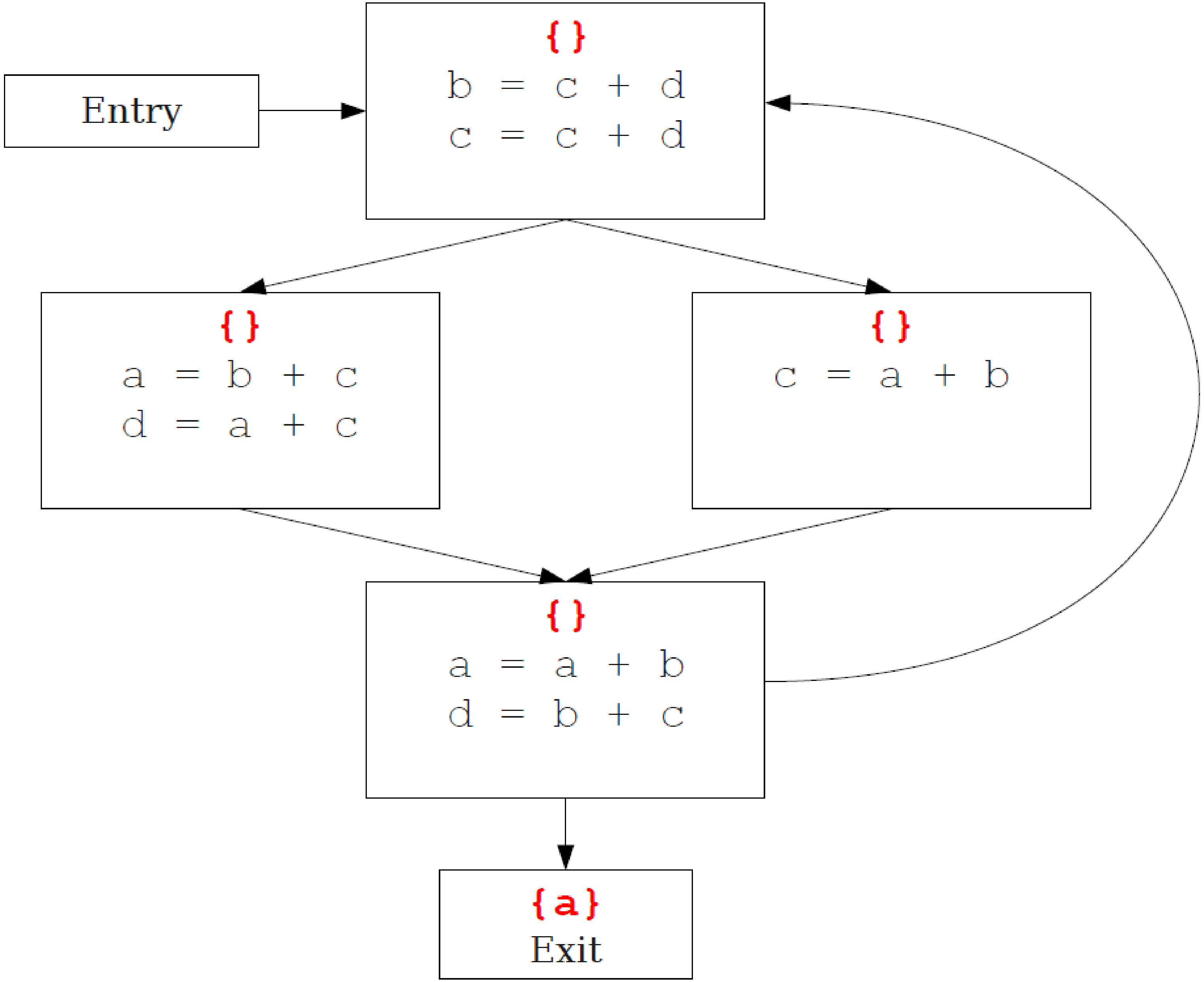
III-2. CFG avec boucles

- Dans une analyse locale, il y a toujours une "première" déclaration bien définie pour commencer le traitement.

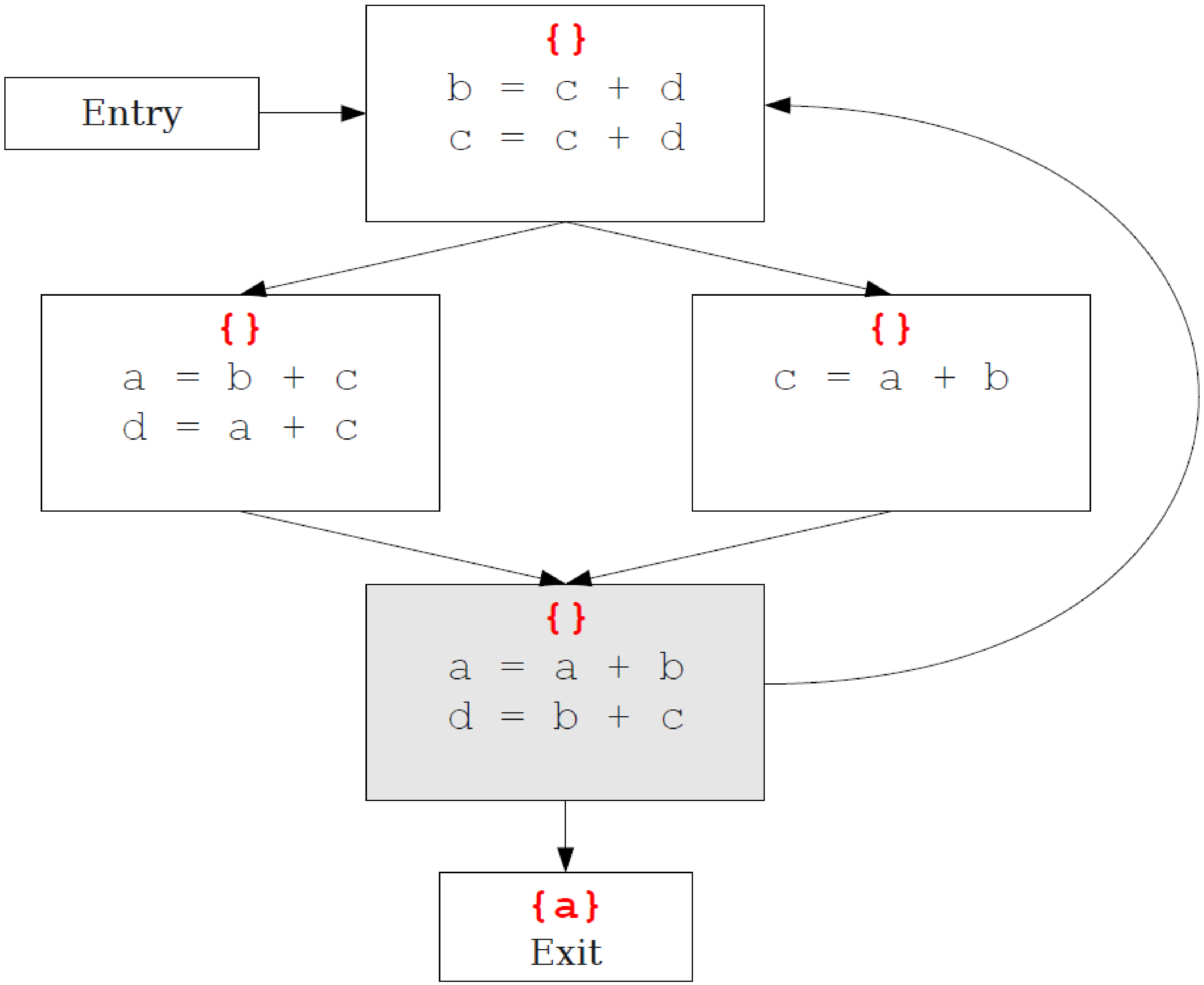
- Dans une analyse globale avec des boucles, chaque bloc de base peut dépendre de tous les autres blocs de base.

- Pour résoudre ce problème, nous devons affecter des valeurs initiales à tous les blocs du CFG.

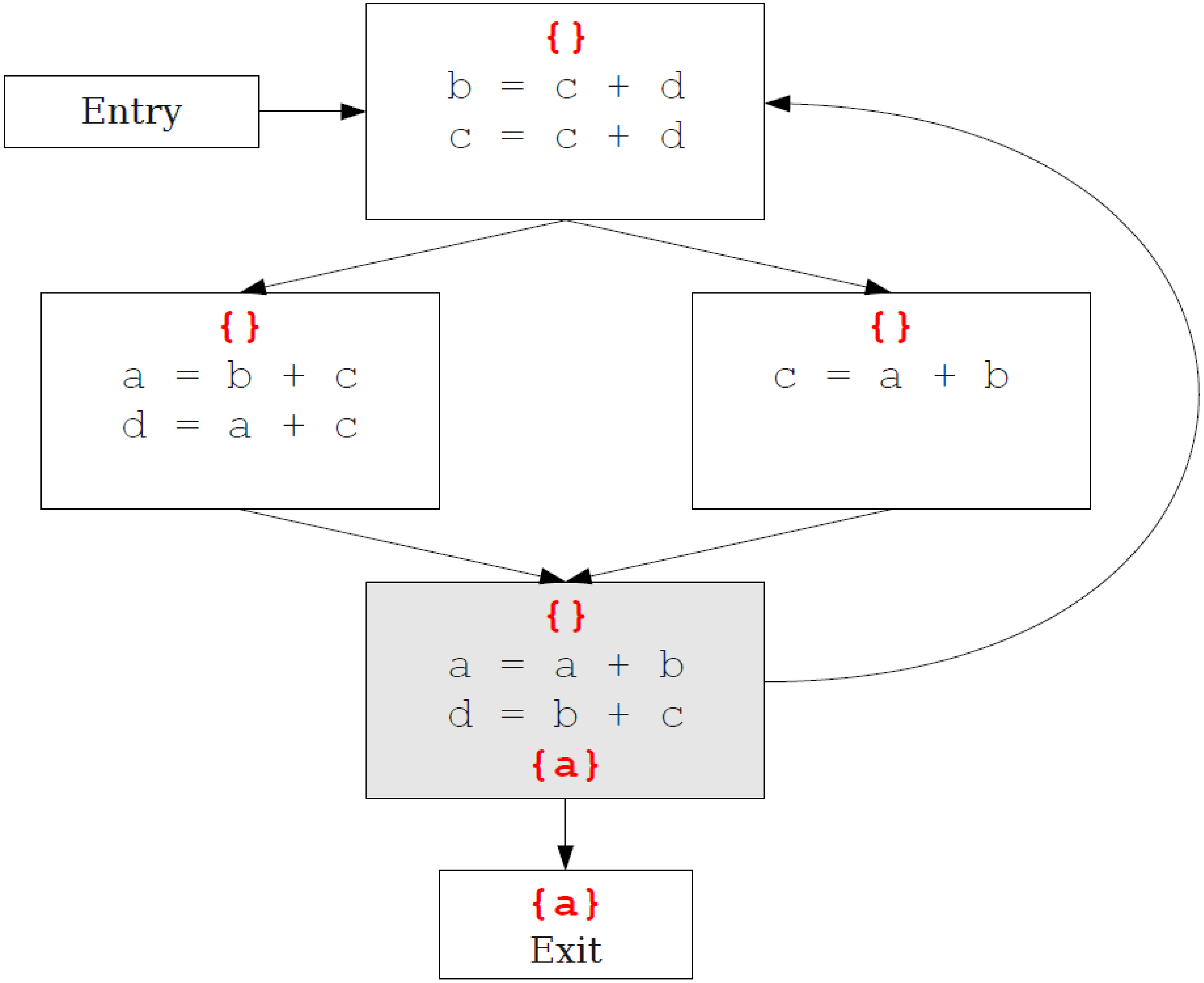
III-2. CFG avec boucles



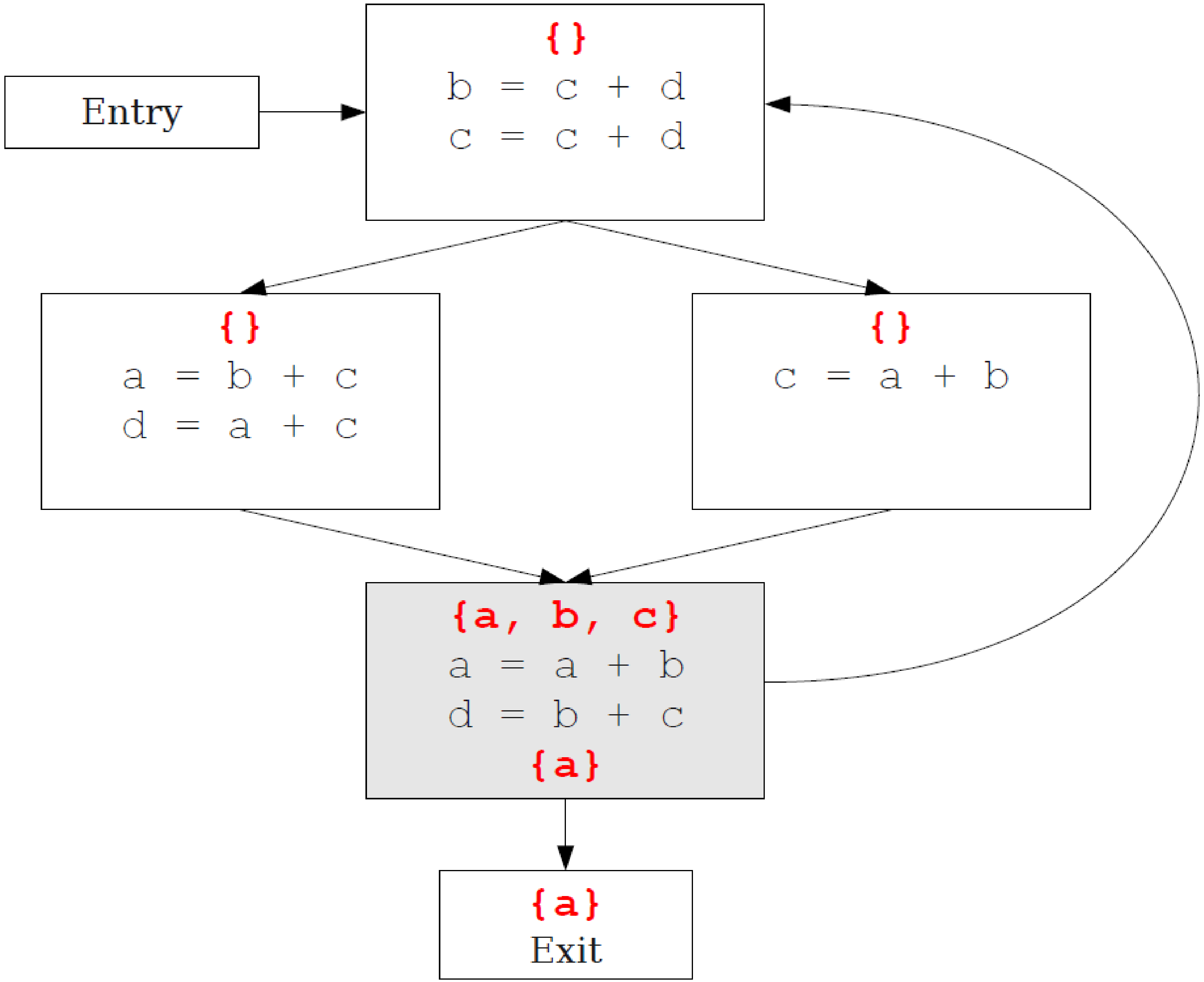
III-2. CFG avec boucles



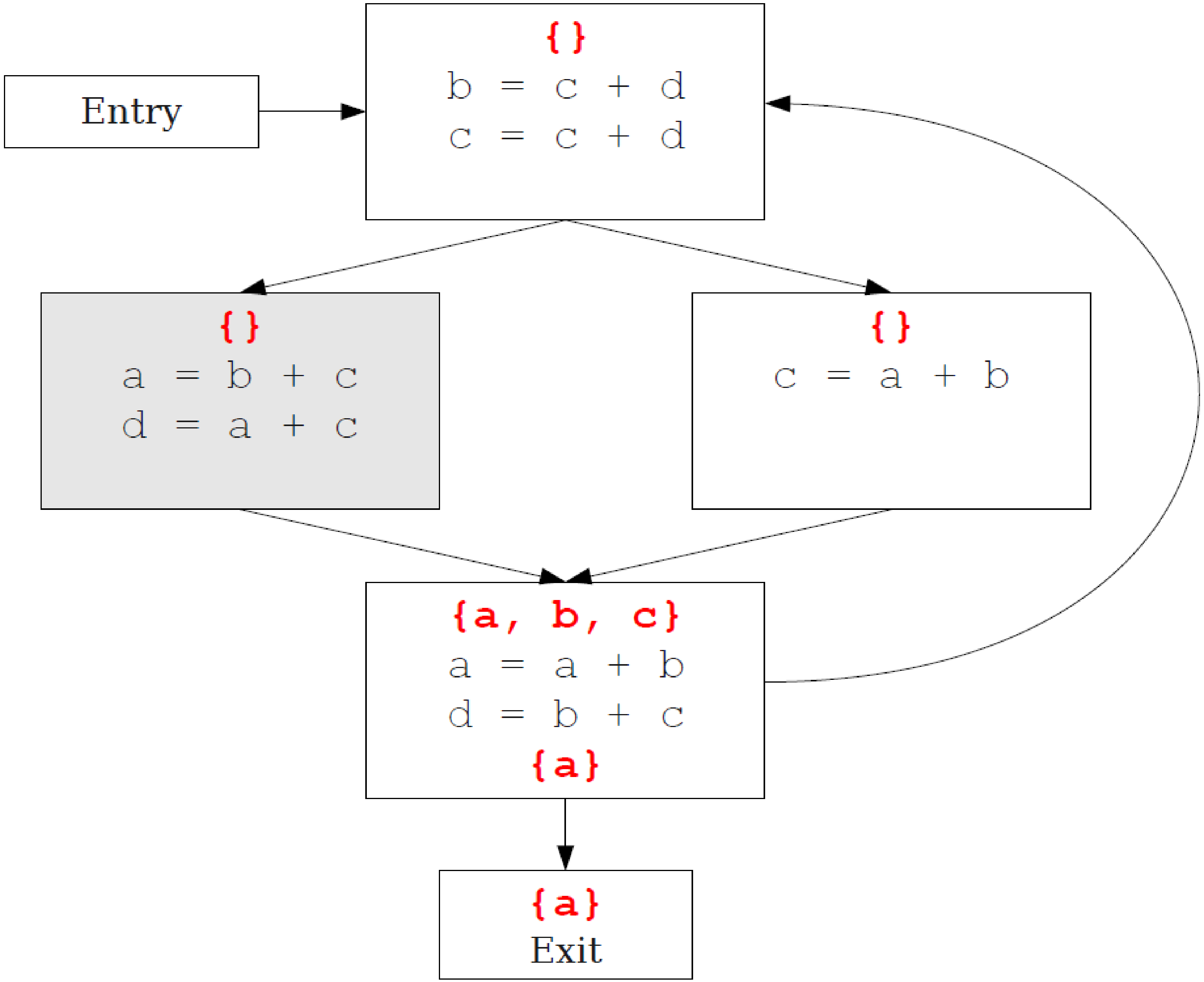
III-2. CFG avec boucles



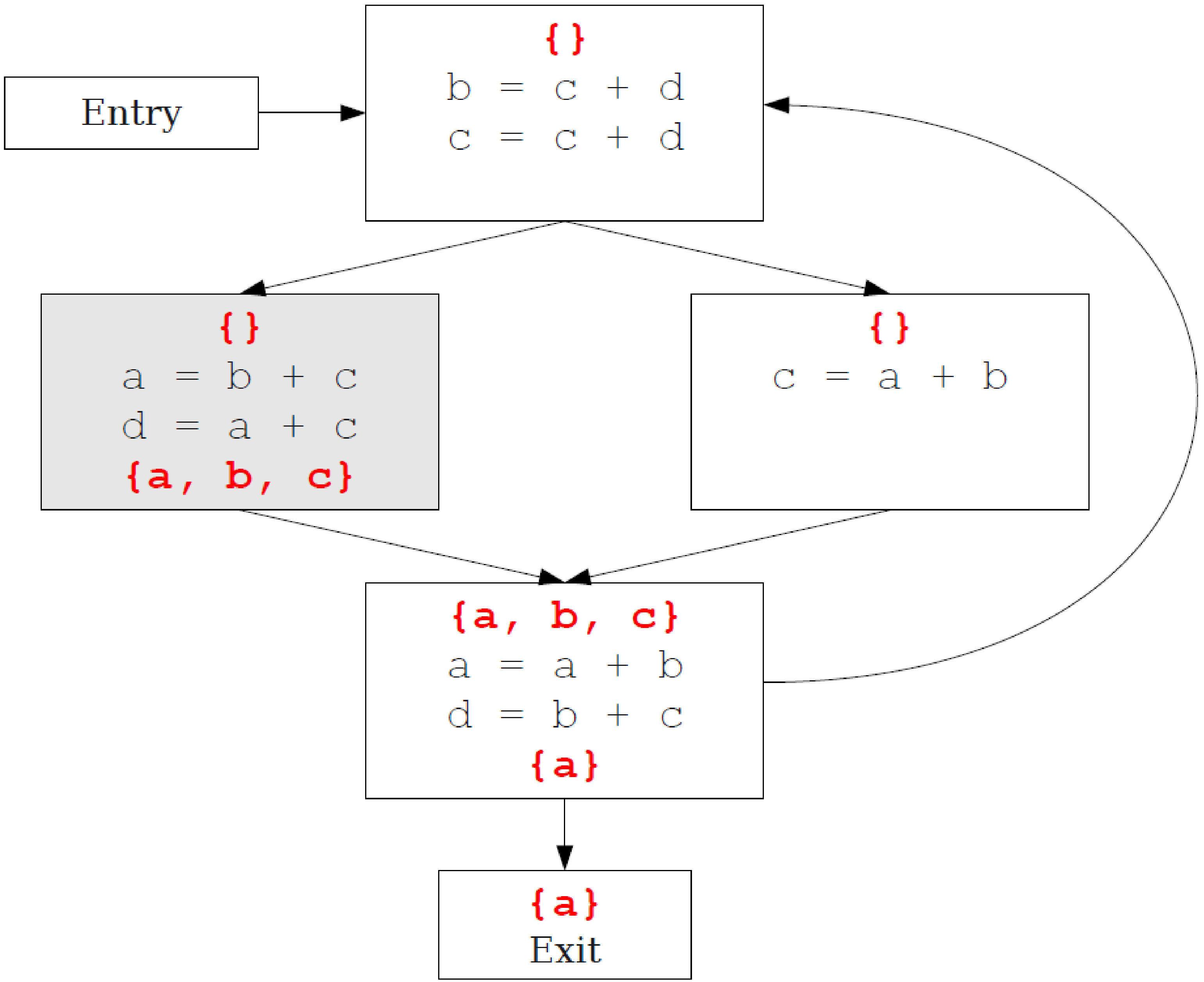
III-2. CFG avec boucles



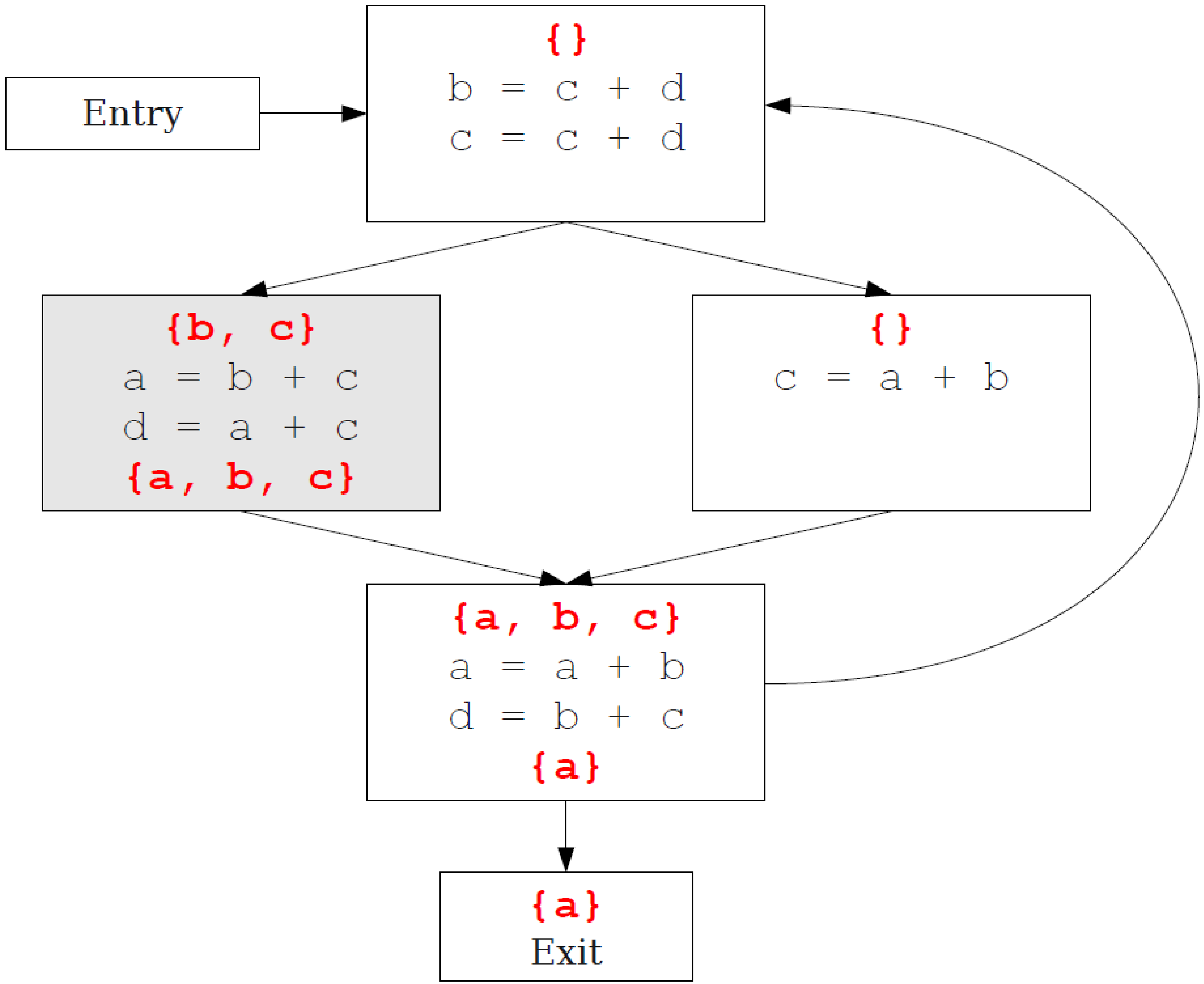
III-2. CFG avec boucles



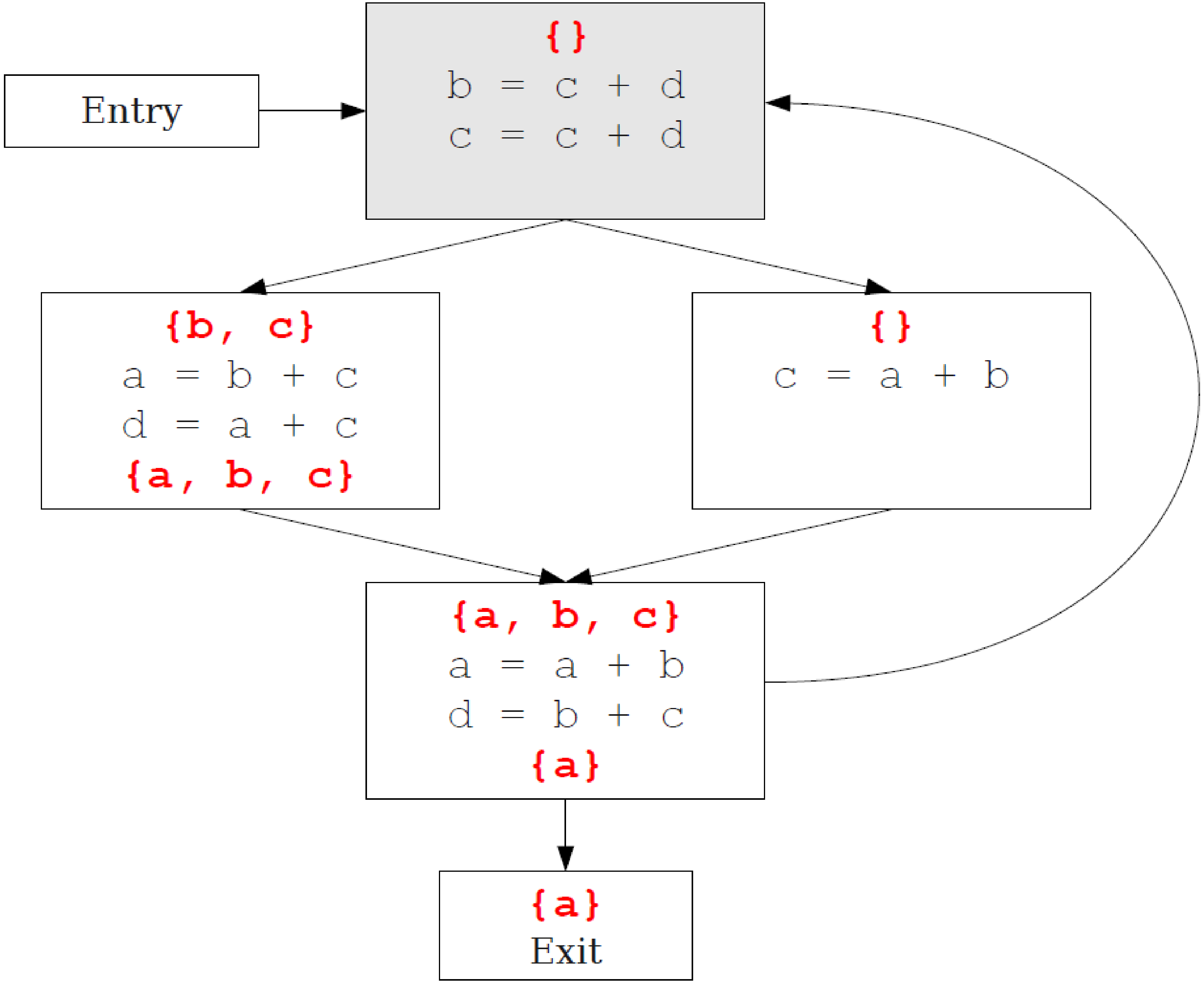
III-2. CFG avec boucles



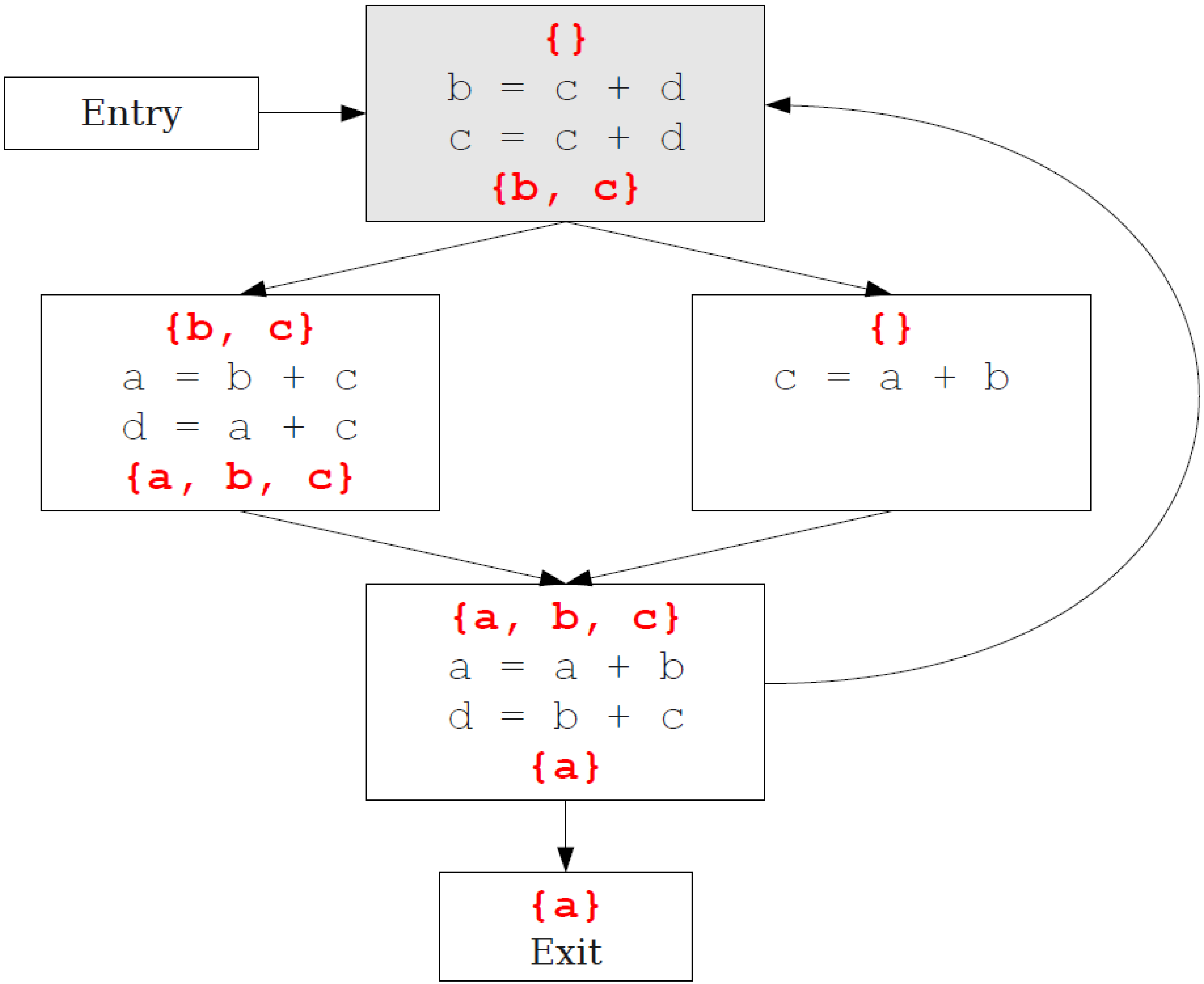
III-2. CFG avec boucles



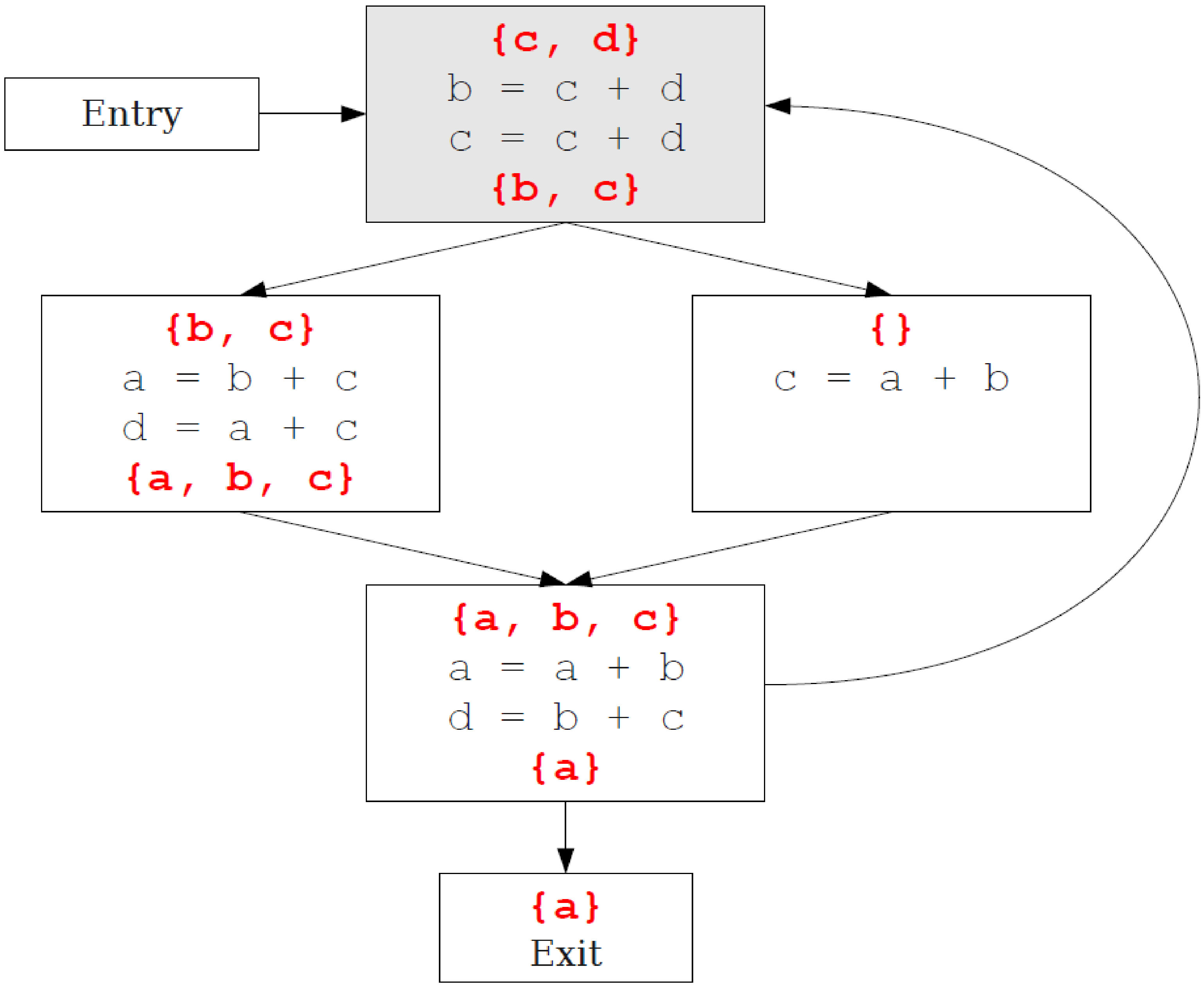
III-2. CFG avec boucles



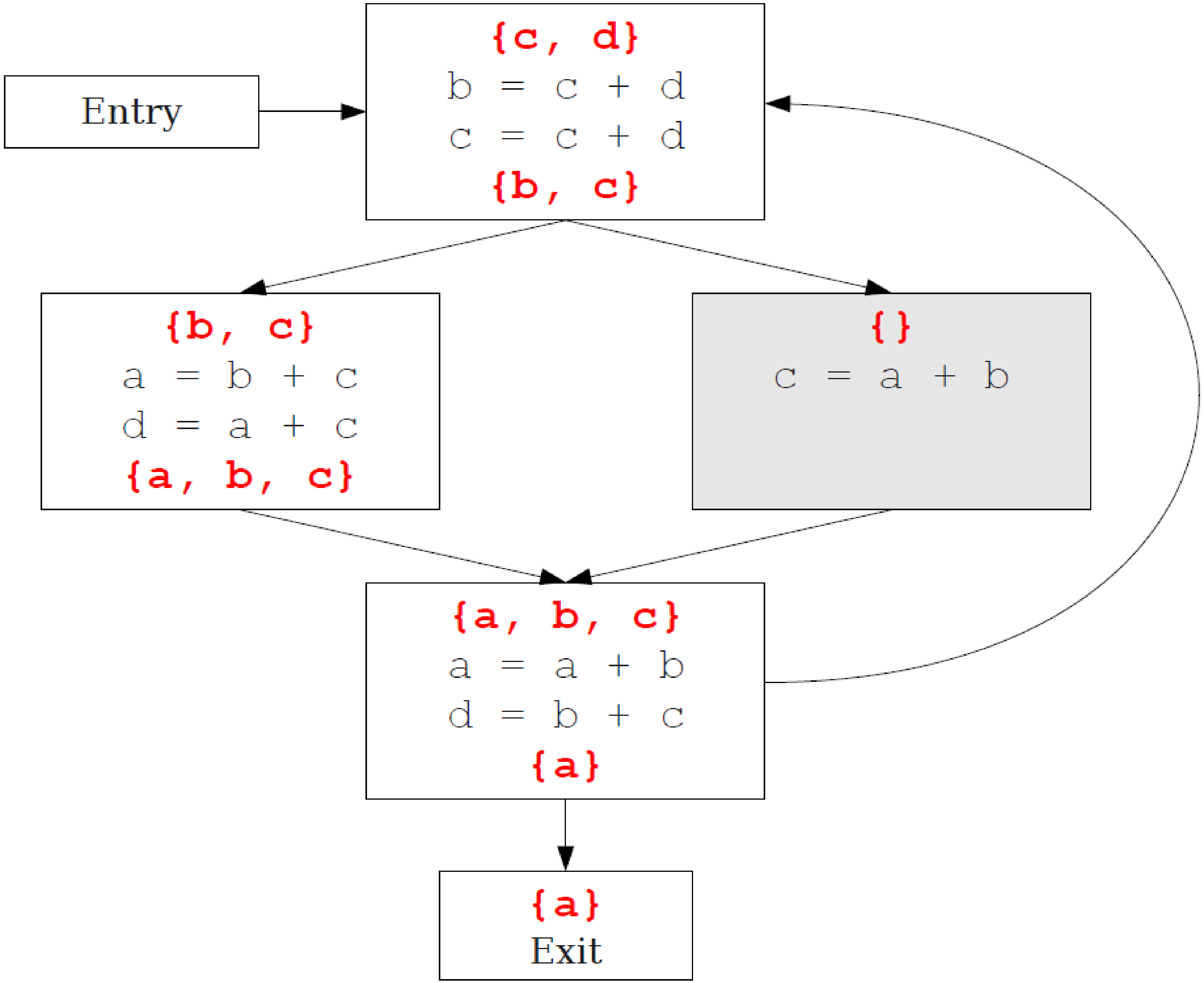
III-2. CFG avec boucles



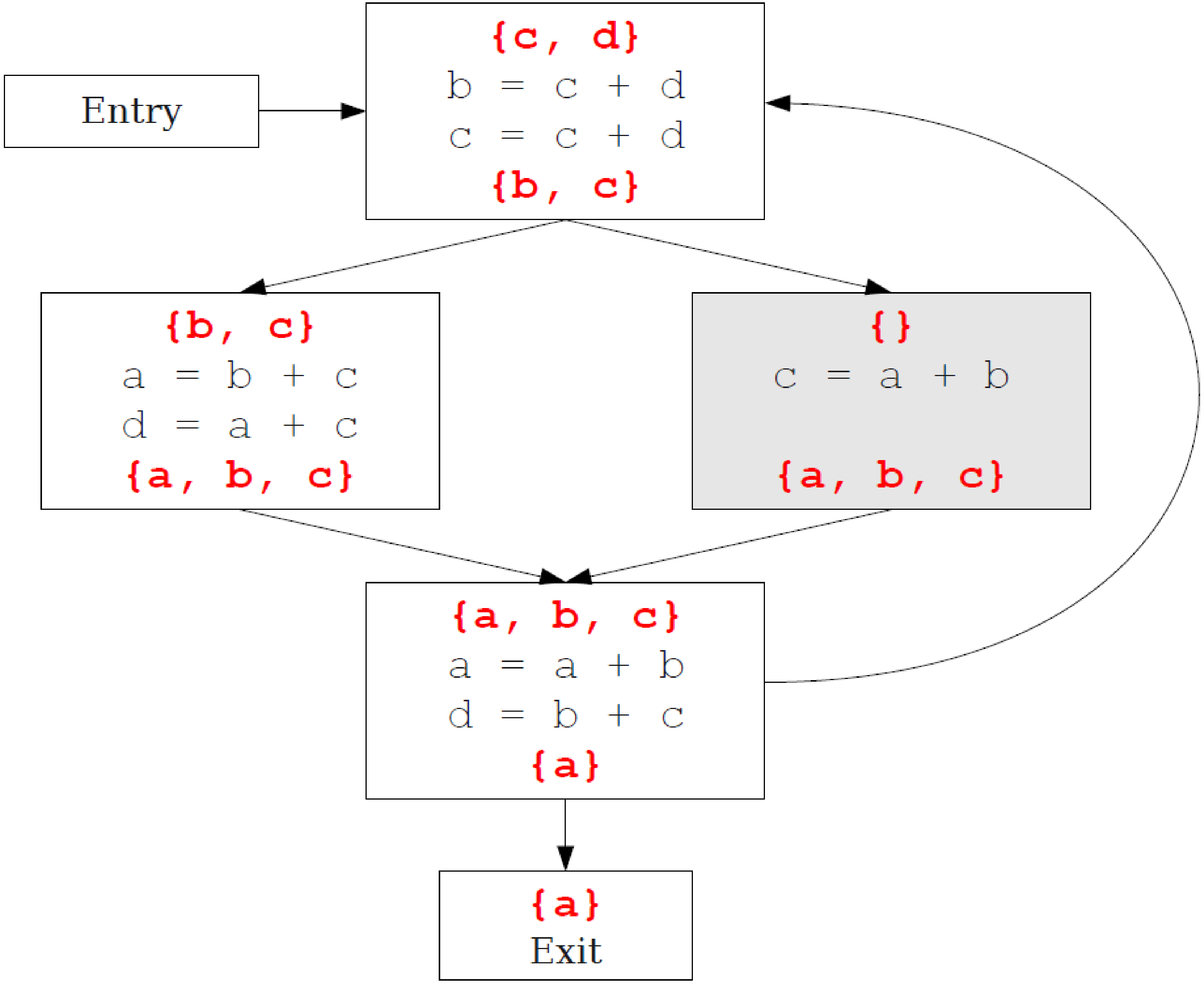
III-2. CFG avec boucles



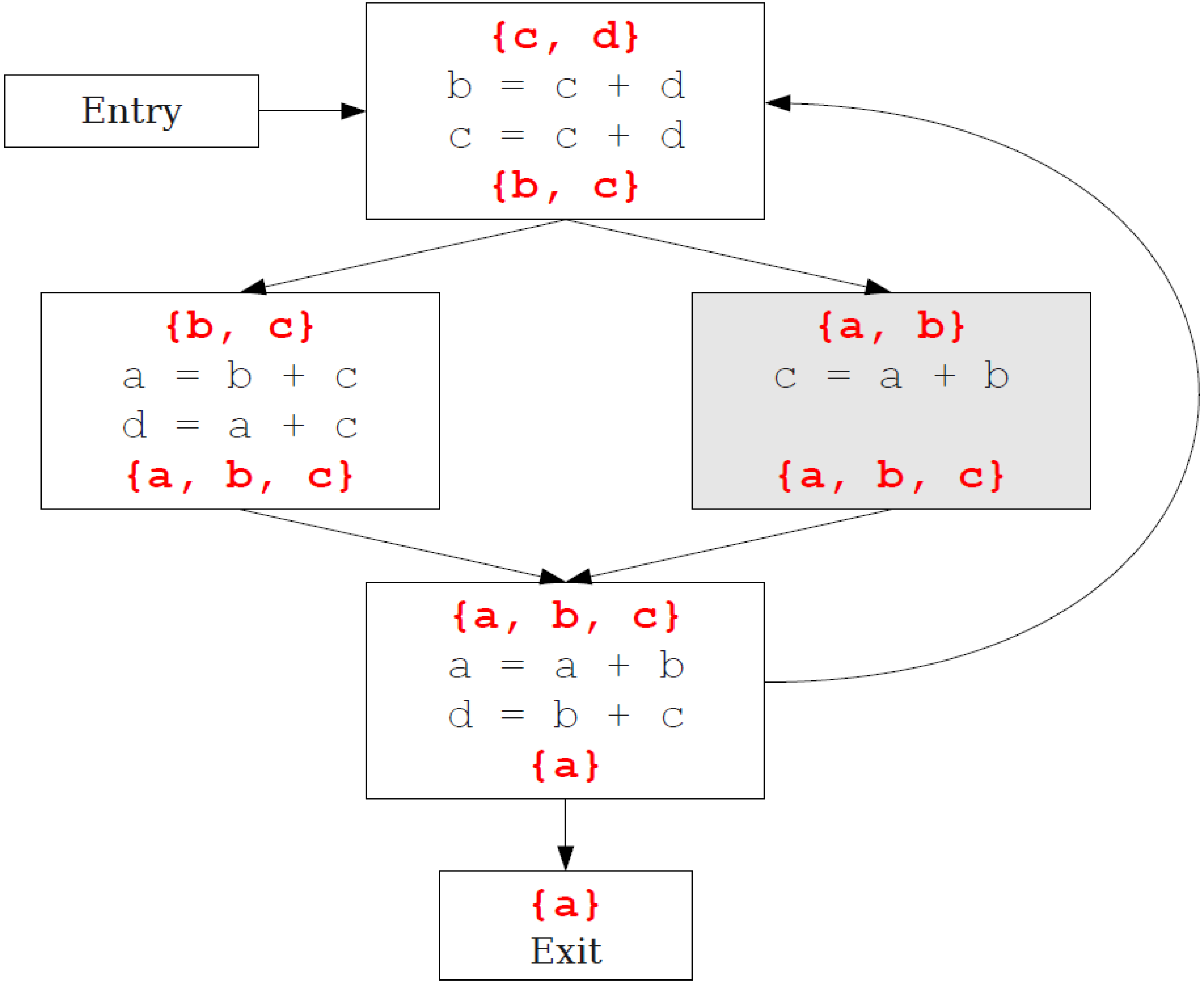
III-2. CFG avec boucles



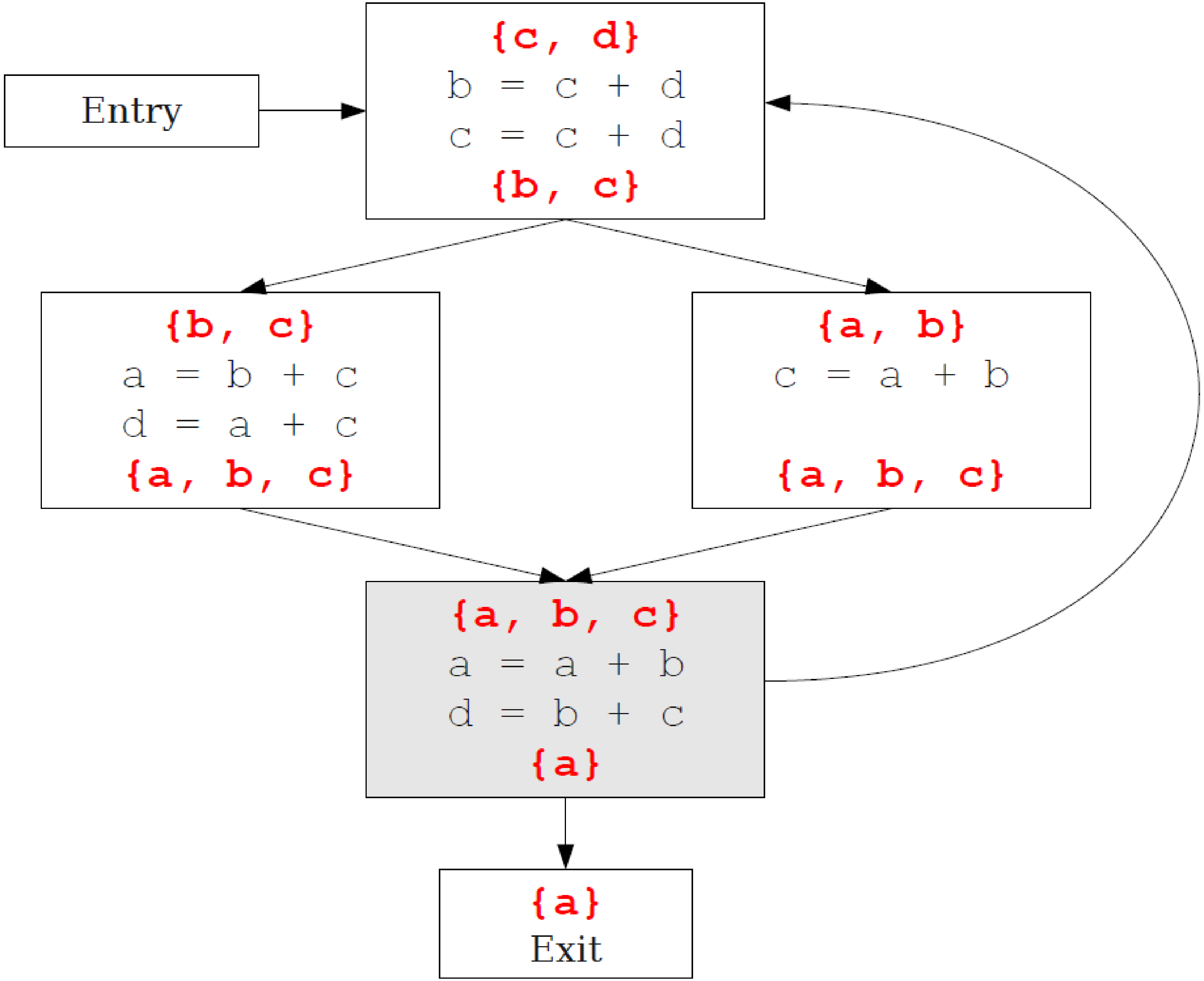
III-2. CFG avec boucles



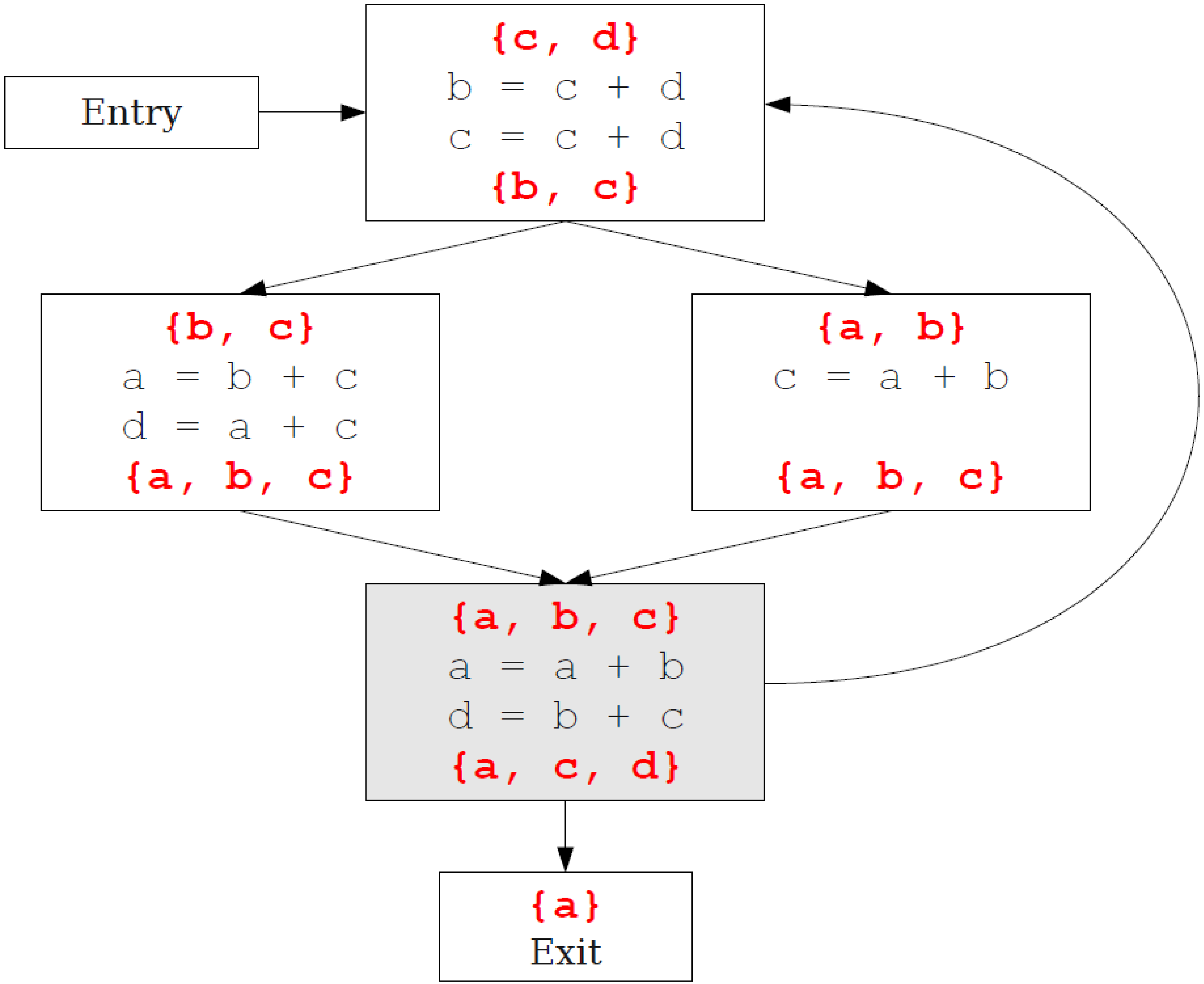
III-2. CFG avec boucles



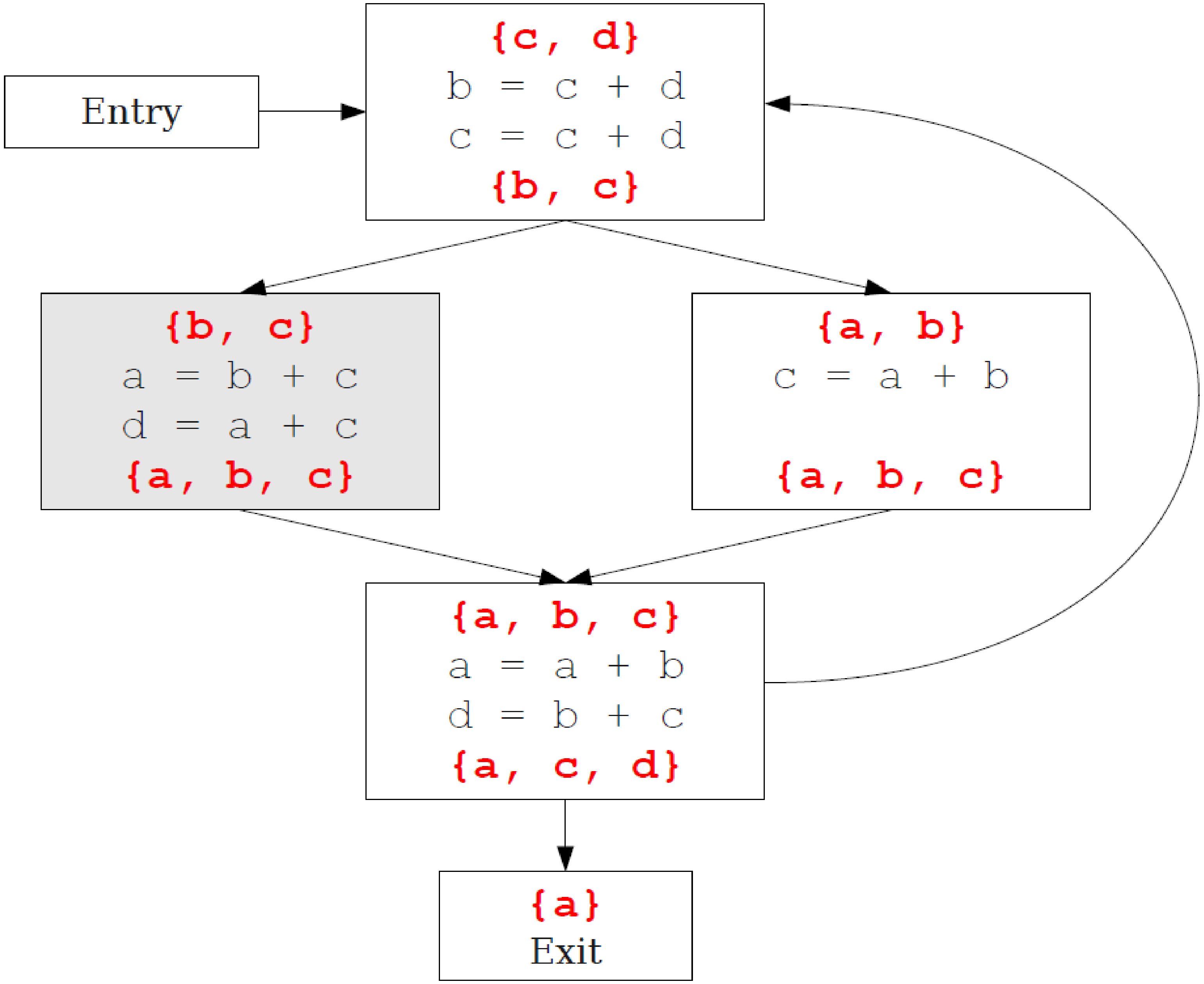
III-2. CFG avec boucles



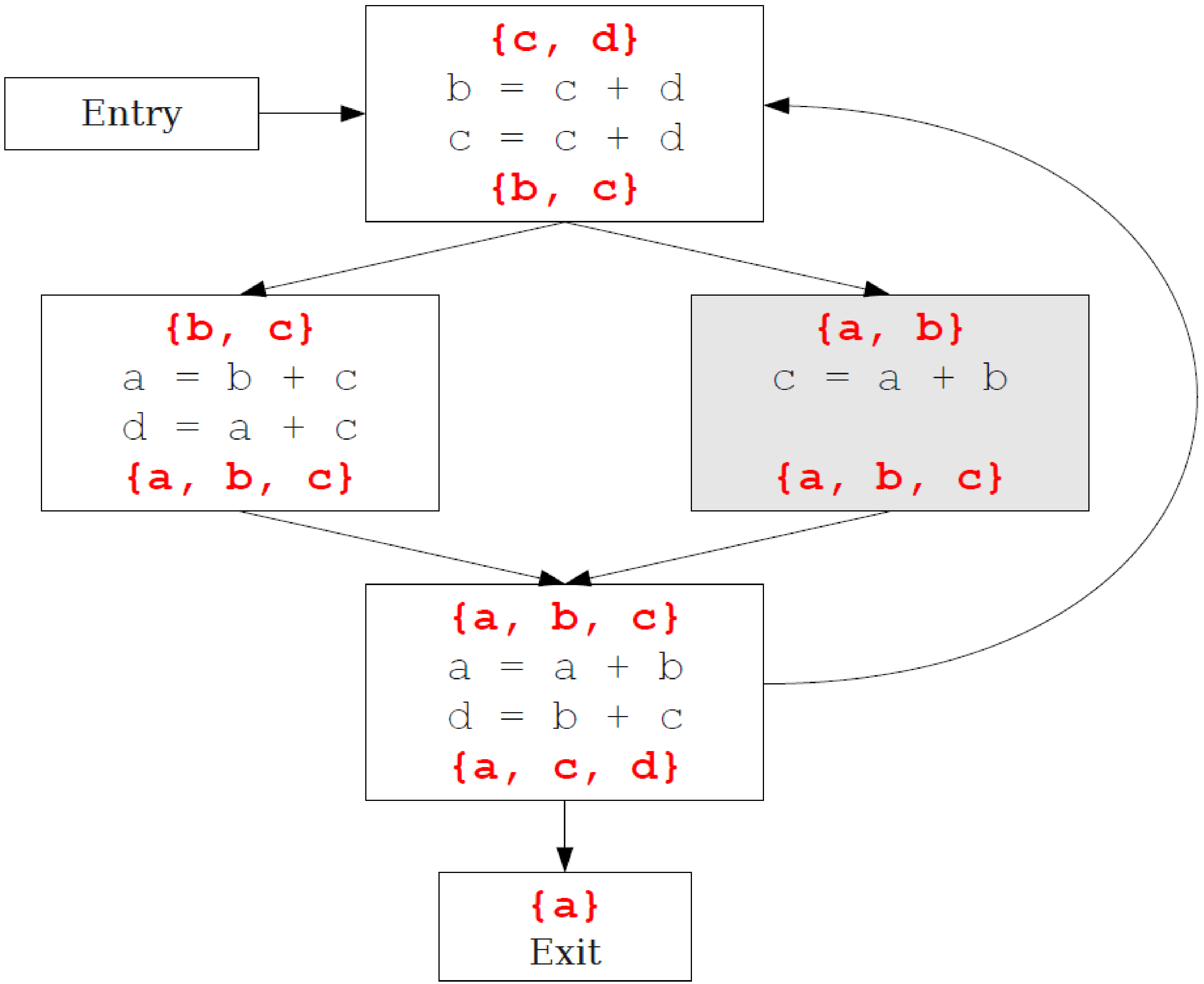
III-2. CFG avec boucles



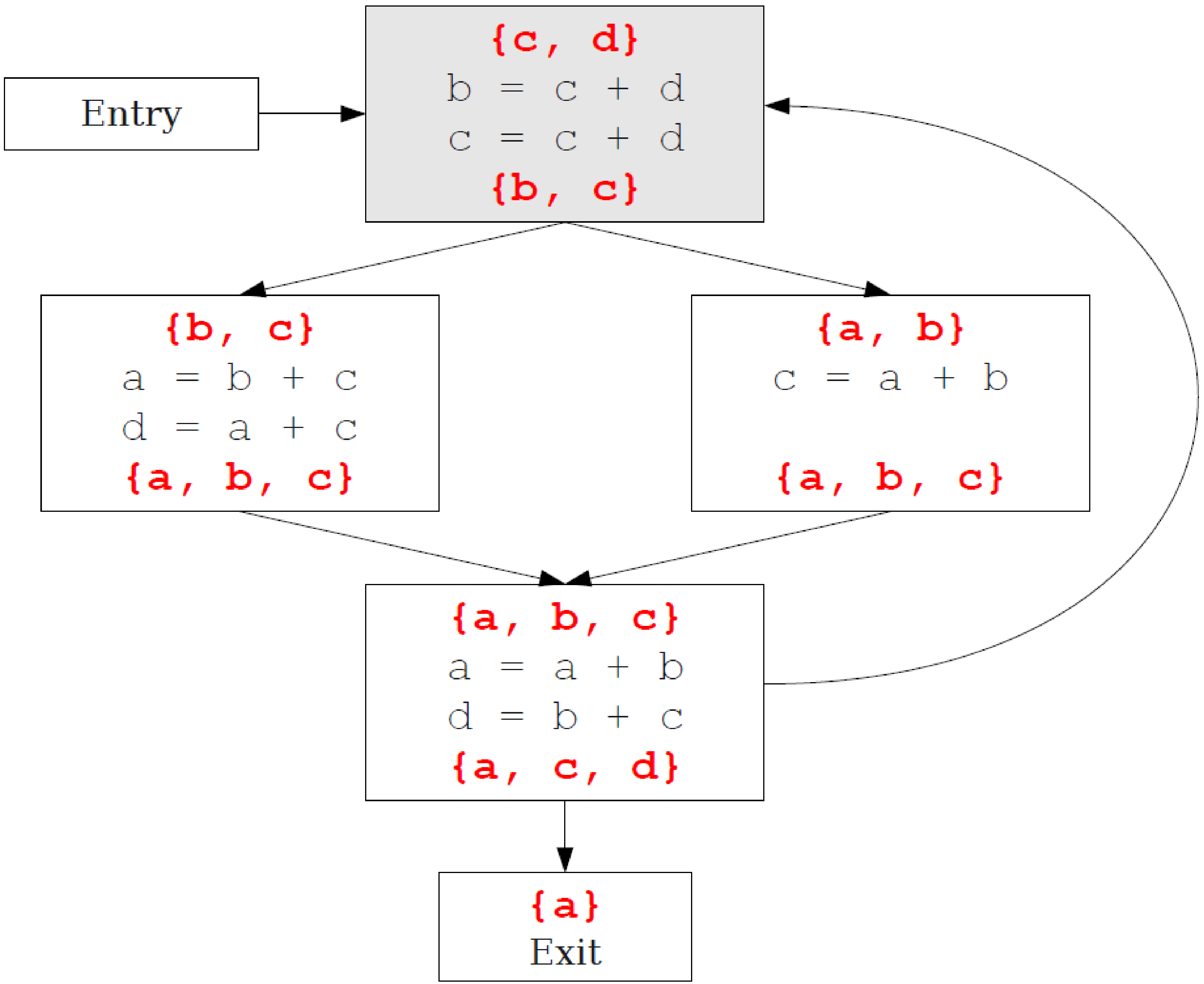
III-2. CFG avec boucles



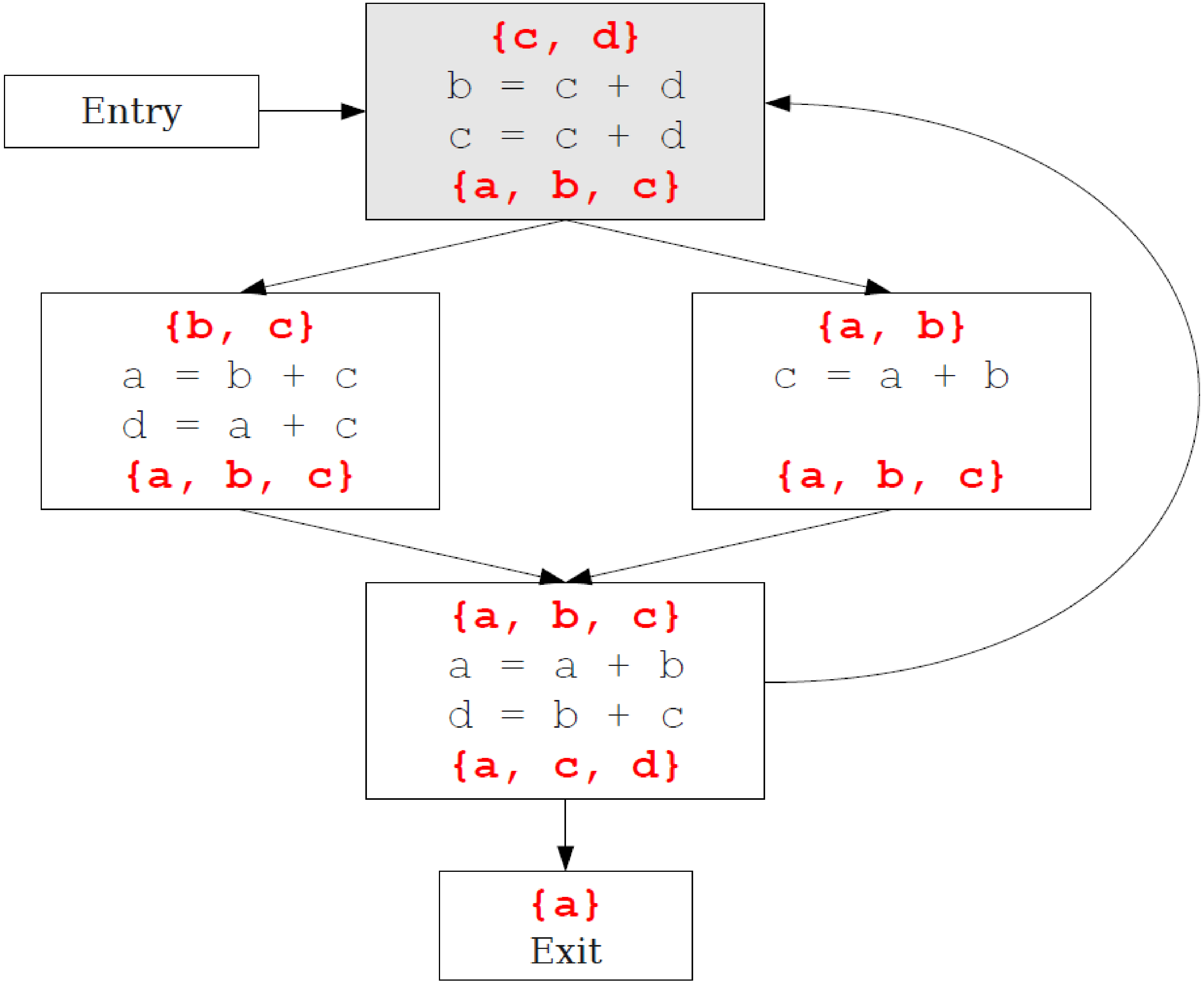
III-2. CFG avec boucles



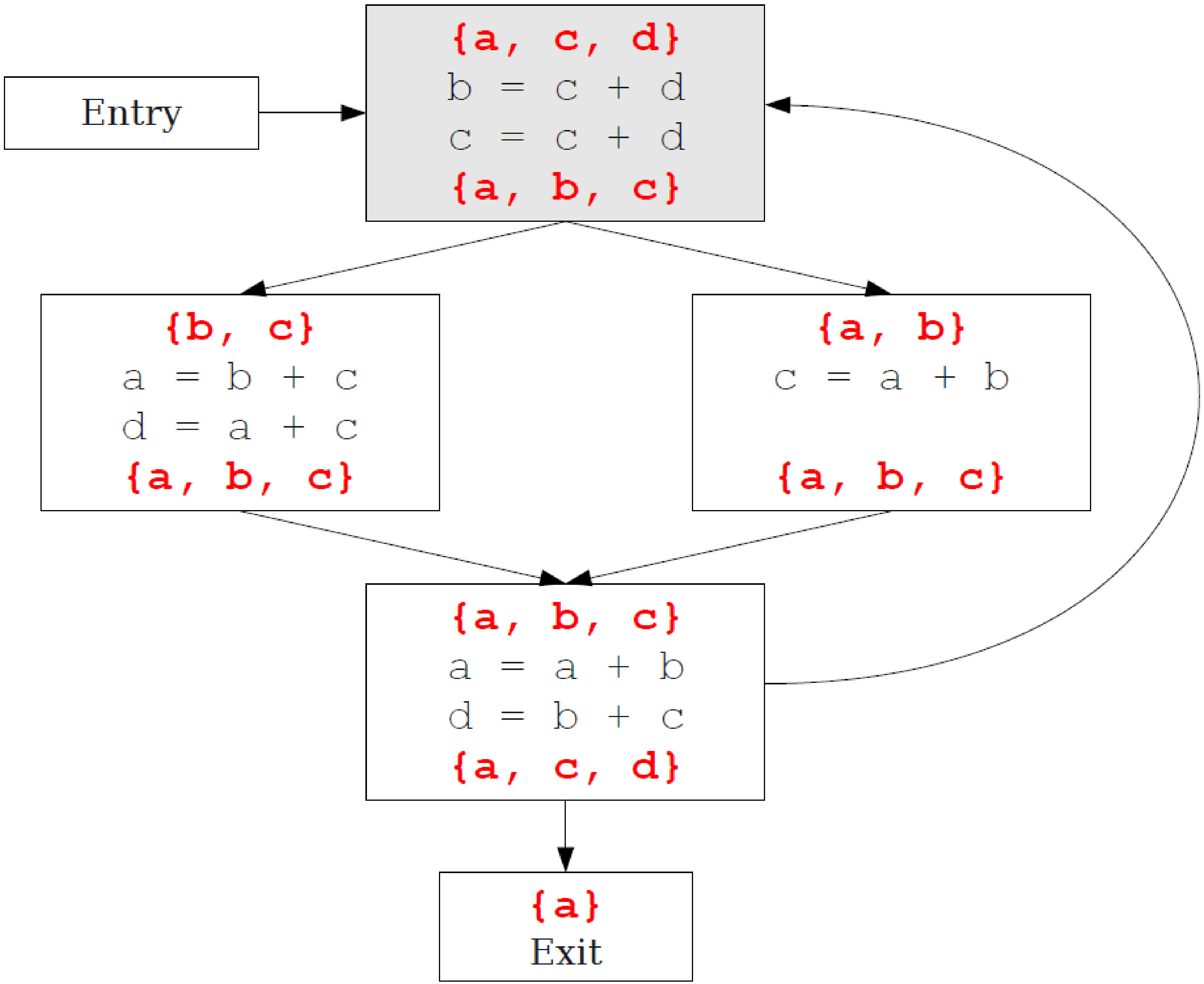
III-2. CFG avec boucles



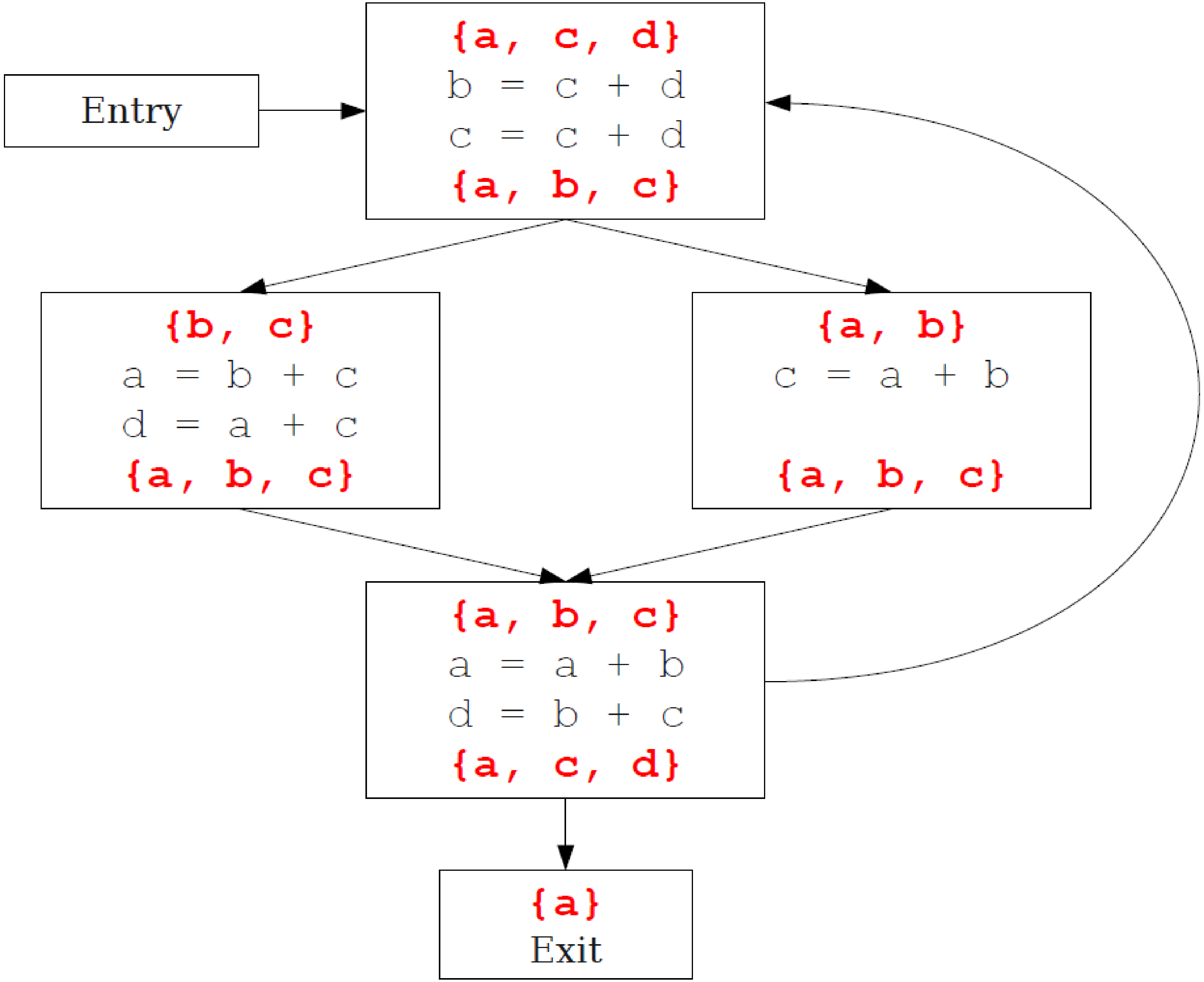
III-2. CFG avec boucles



III-2. CFG avec boucles



III-2. CFG avec boucles



III- Optimisations globales

Résumé des différences avec l'optimisation locale:

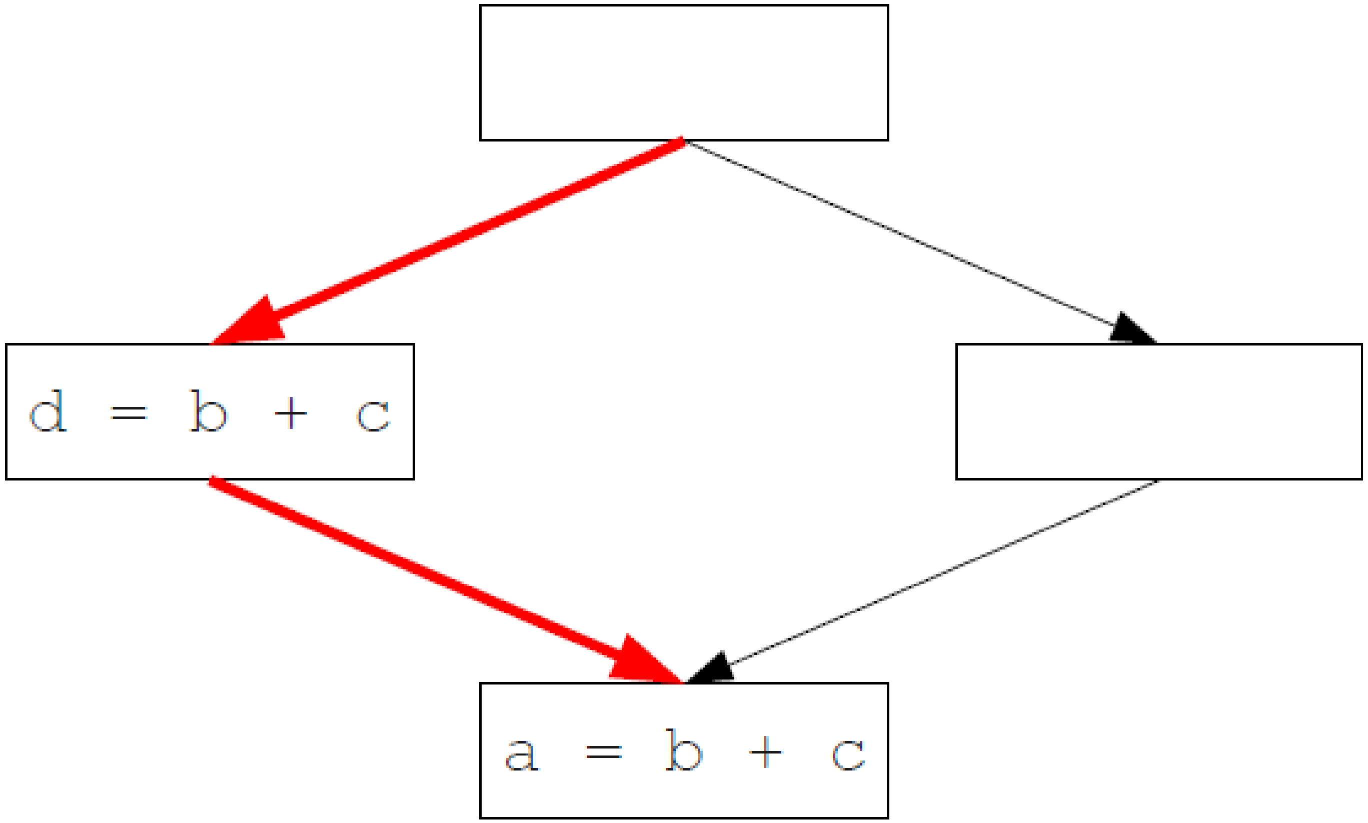
- Elle doit être capable de gérer plusieurs prédécesseurs/successeurs pour un bloc de base.
- Elle doit être capable de gérer plusieurs chemins à travers le graphe de flot de contrôle, et peut itérer plusieurs fois pour calculer la valeur finale (mais l'analyse doit encore se terminer!)
- Elle doit pouvoir affecter à chaque bloc de base une valeur par défaut raisonnable avant l'analyse.

III-3. Élimination de redondances partielles

- Un calcul dans un programme est dit **redondant** s'il calcule une valeur déjà connue.
- Les sous-expressions courantes sont un exemple de redondance.
- Le code invariant dans une boucle est un autre exemple.
- Normalement, tous les optimiseurs des compilateurs ont une logique pour essayer d'éliminer la redondance.

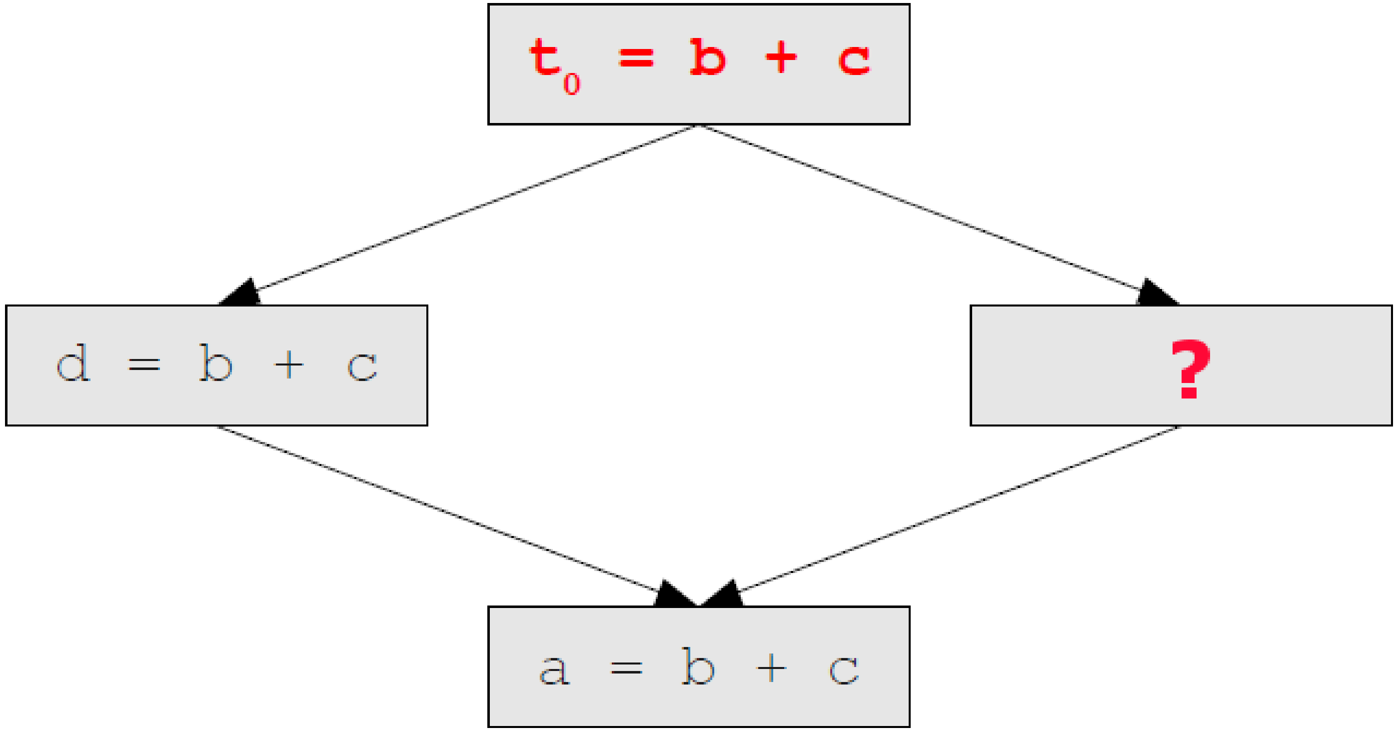
III-3. Élimination de redondances partielles

Un calcul est partiellement redondant si sa valeur n'est connue que sur certains chemins qui l'atteignent.



III-3. Élimination de redondances partielles

- Une expression est dite "**anticipée**" à un point de programme s'il est garantie que l'expression va-t-être utilisée après ce point.
- Bien que tous les chemins du programme n'aient pas besoin directement d'une expression, ils peuvent anticiper l'expression.

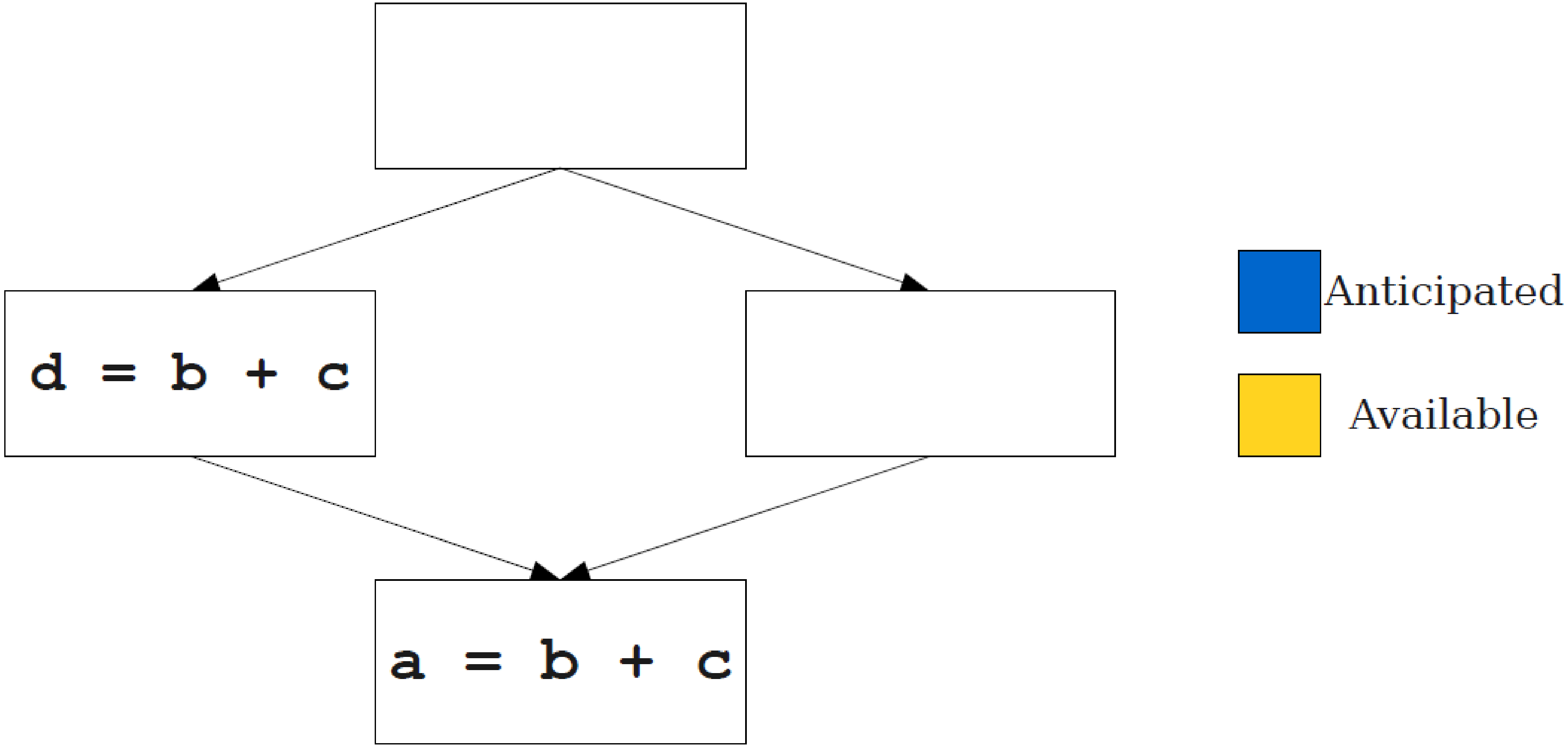


III-3. Élimination de redondances partielles

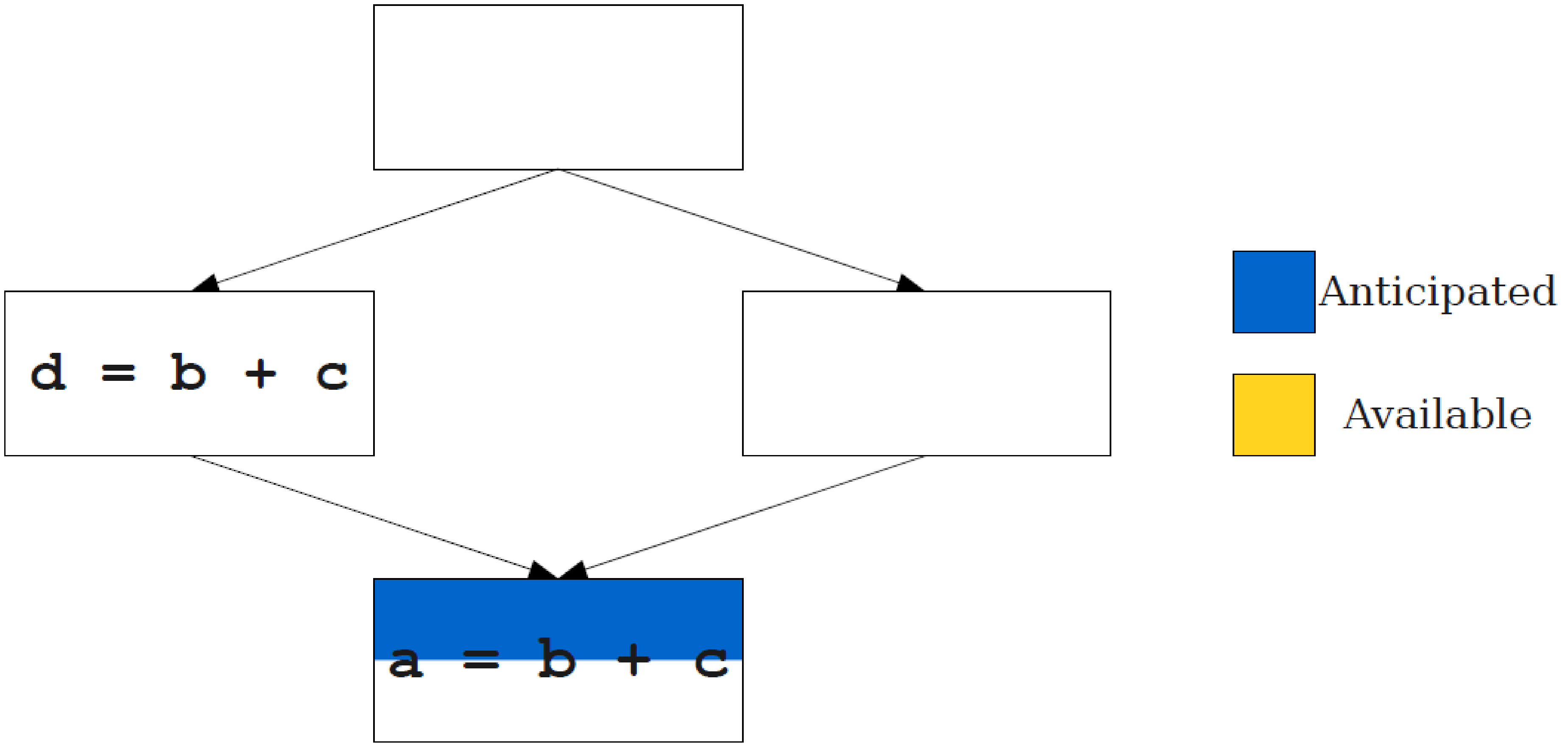
Idée: Rendre l’expression disponible partout où elle est anticipée

- Exécuter une analyse pour localiser les endroits où l’expression est anticipée.
- Exécuter une seconde analyse pour localiser les endroits où l’expression est disponible.
- Placer l’expression aux premiers emplacements où l’expression est anticipée mais non disponible.

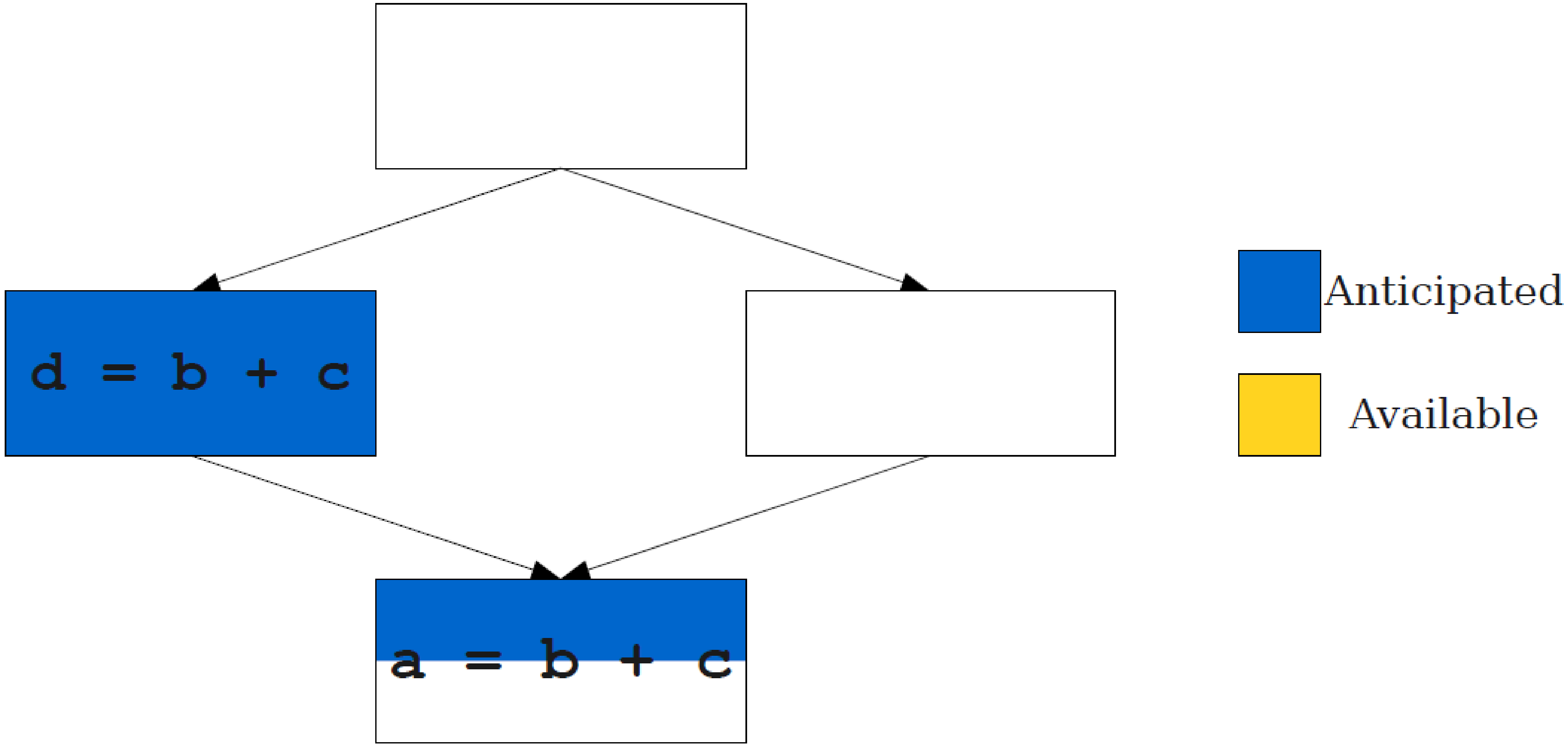
III-3. Élimination de redondances partielles



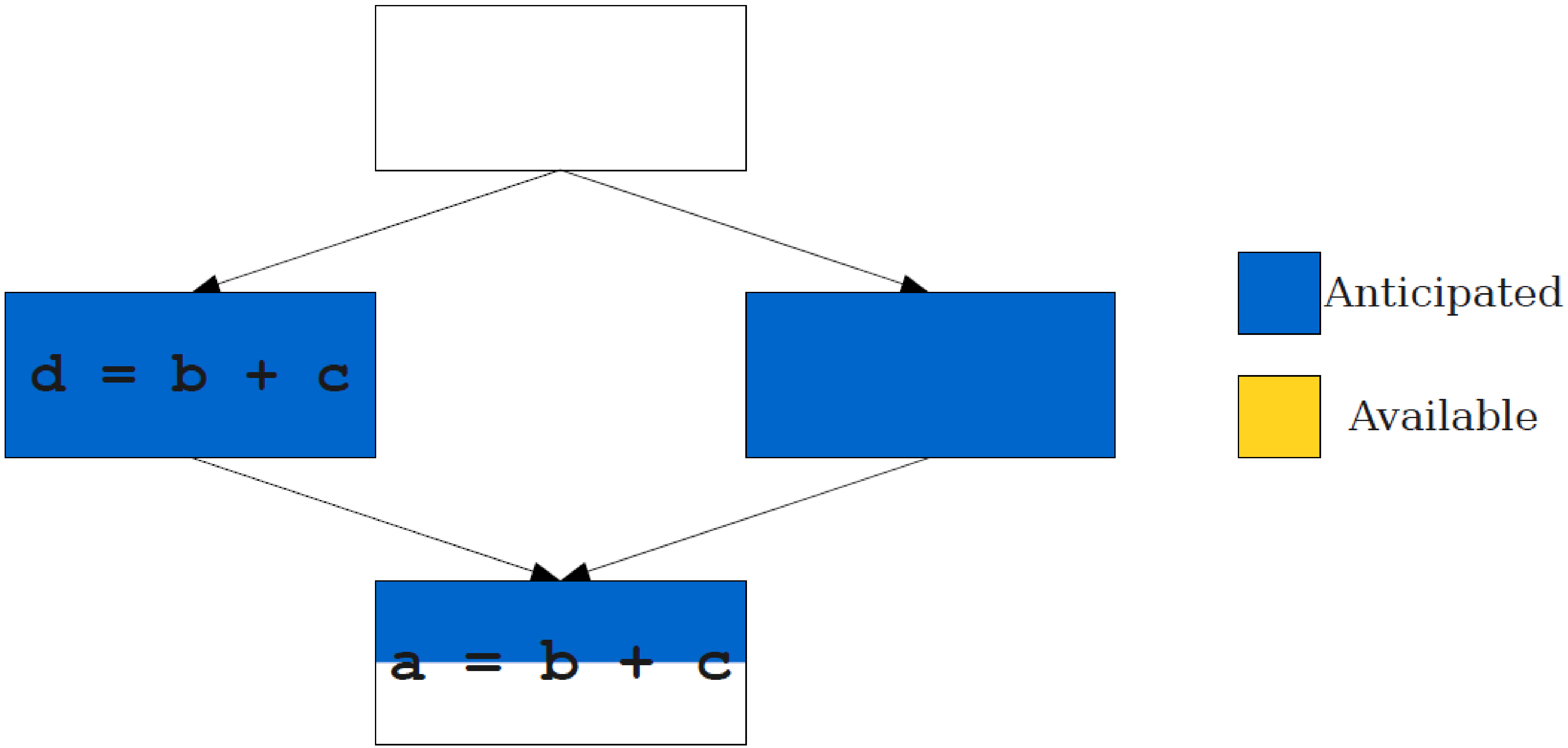
III-3. Élimination de redondances partielles



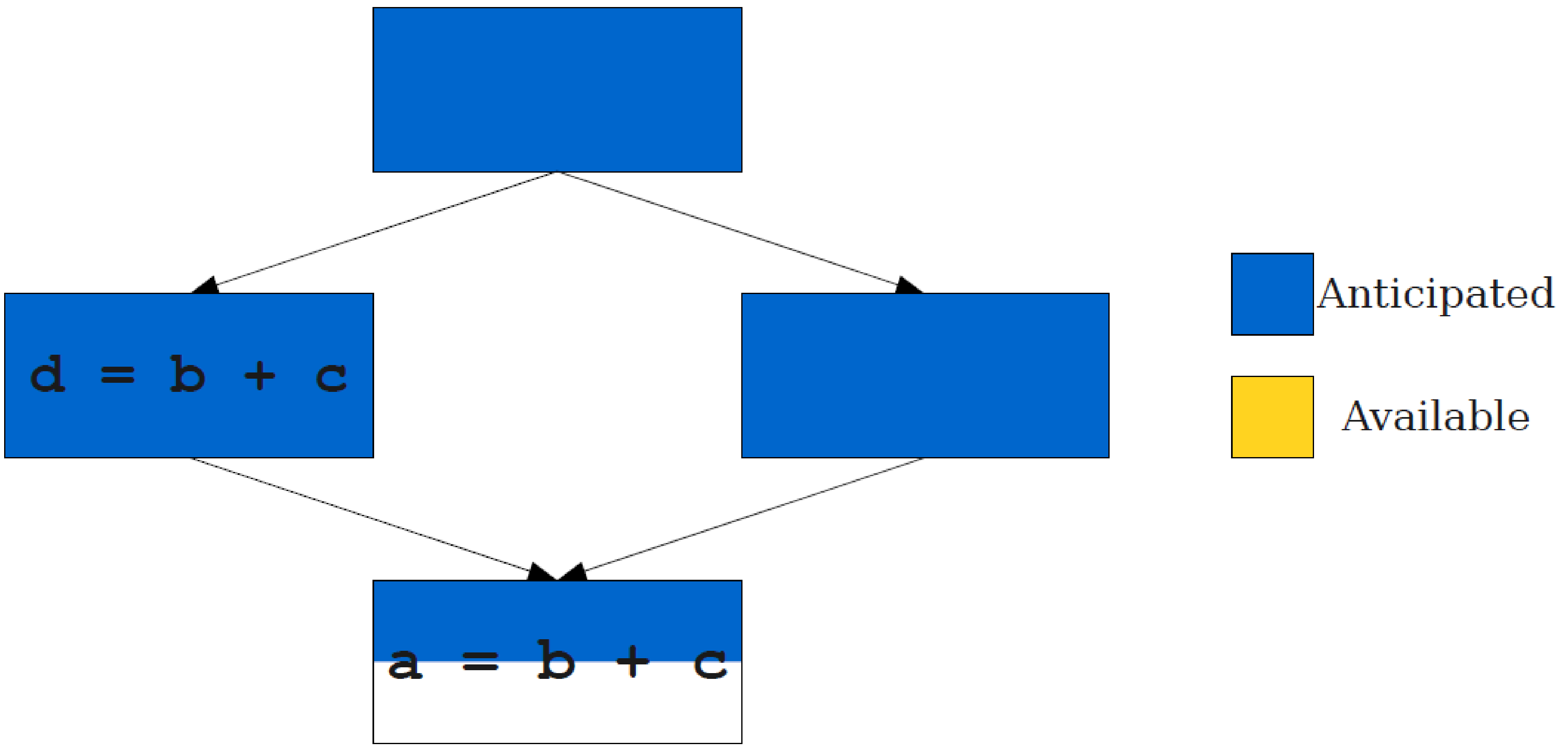
III-3. Élimination de redondances partielles



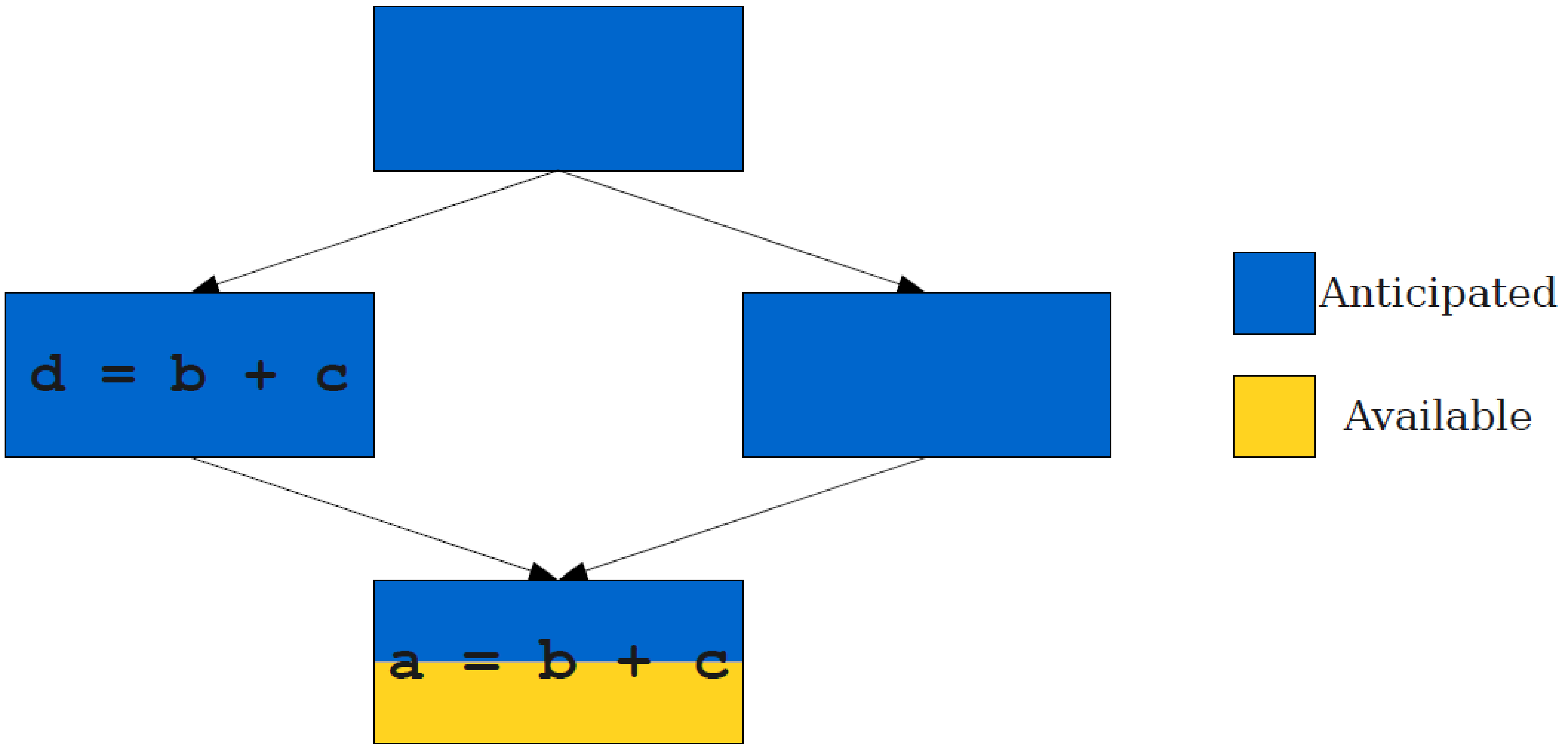
III-3. Élimination de redondances partielles



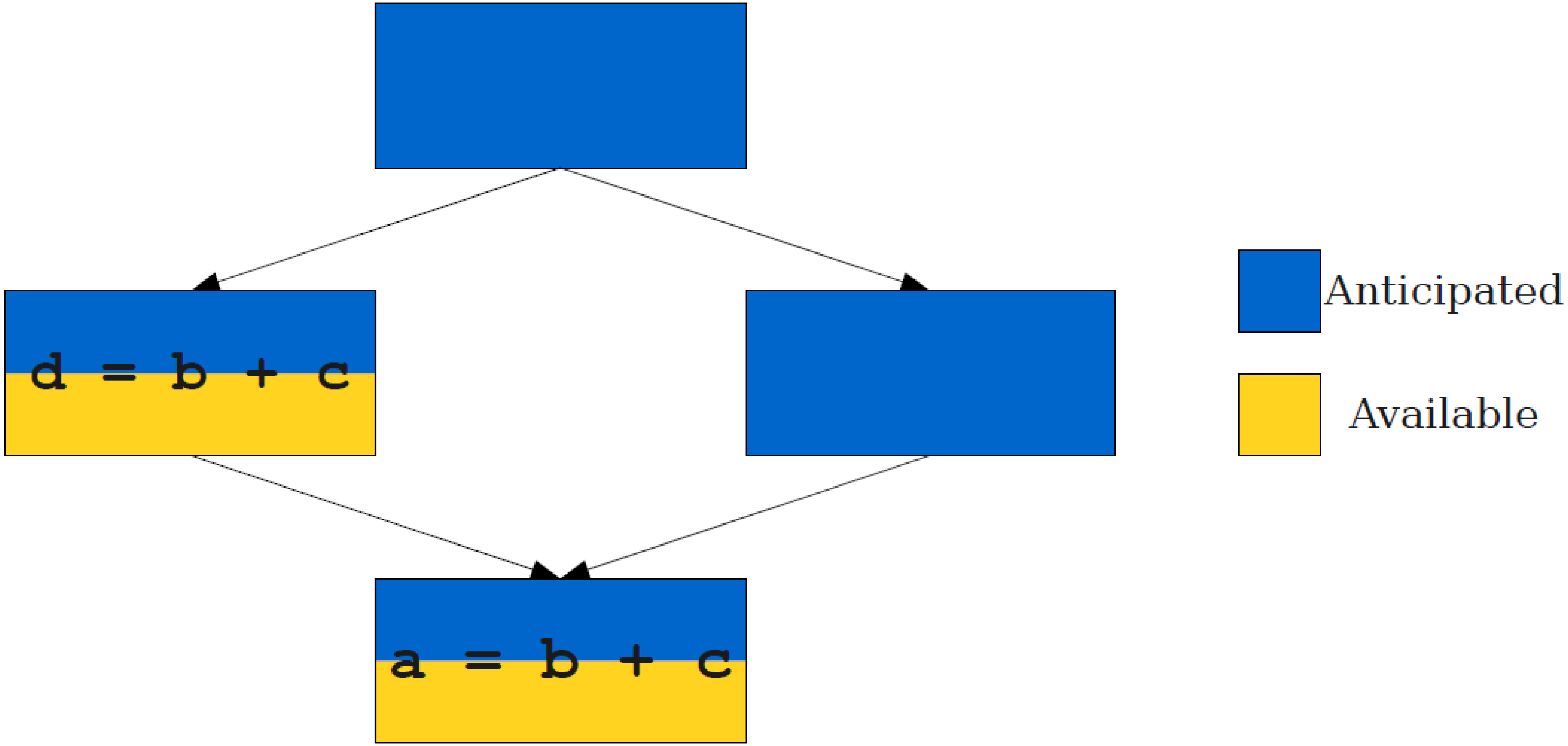
III-3. Élimination de redondances partielles



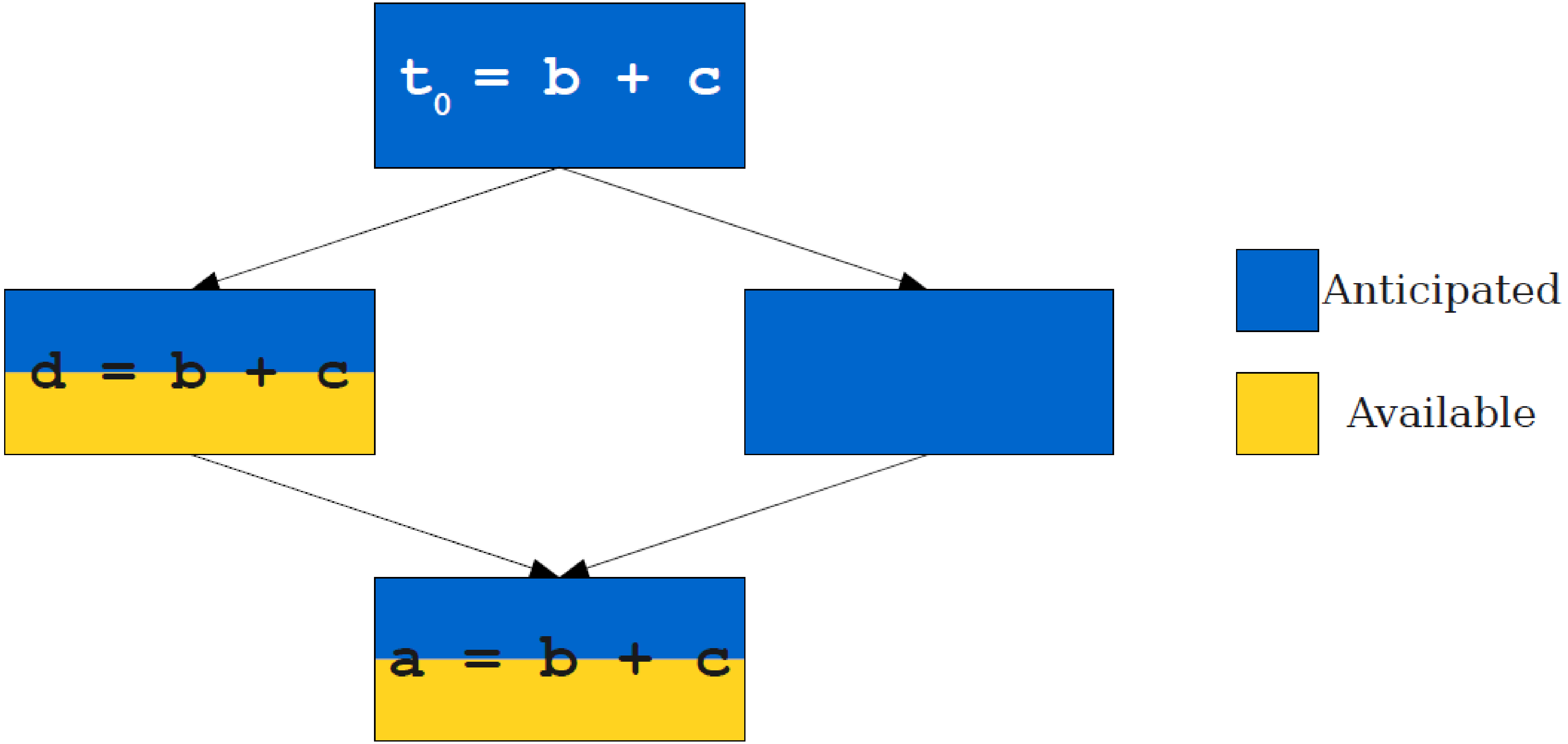
III-3. Élimination de redondances partielles



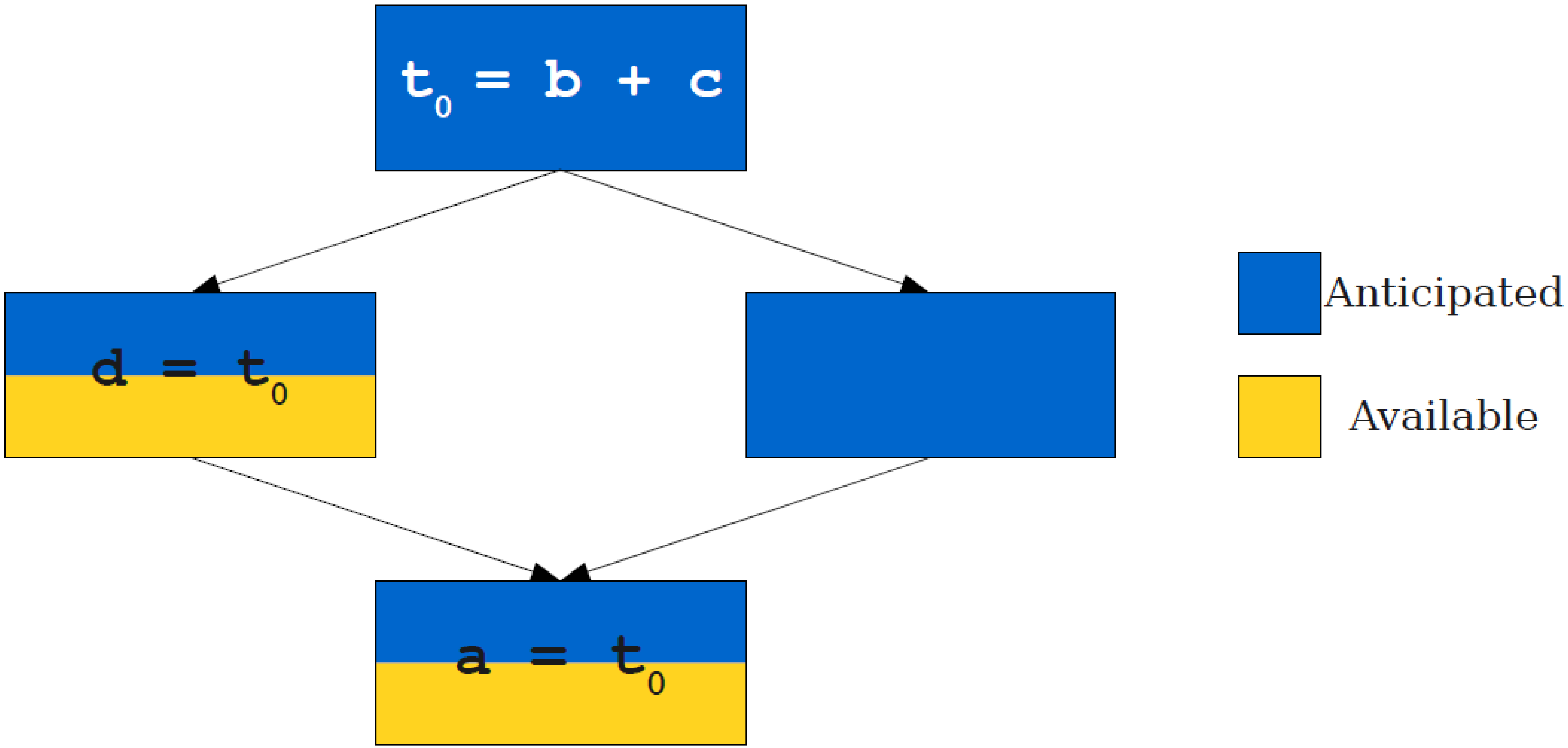
III-3. Élimination de redondances partielles



III-3. Élimination de redondances partielles



III-3. Élimination de redondances partielles



III-3. Élimination de redondances partielles