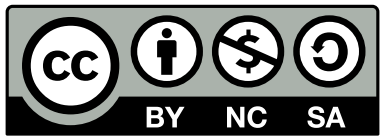
Chapitre 8: Optimisations

INF7641 Compilation

Jean Privat

Université du Québec à Montréal

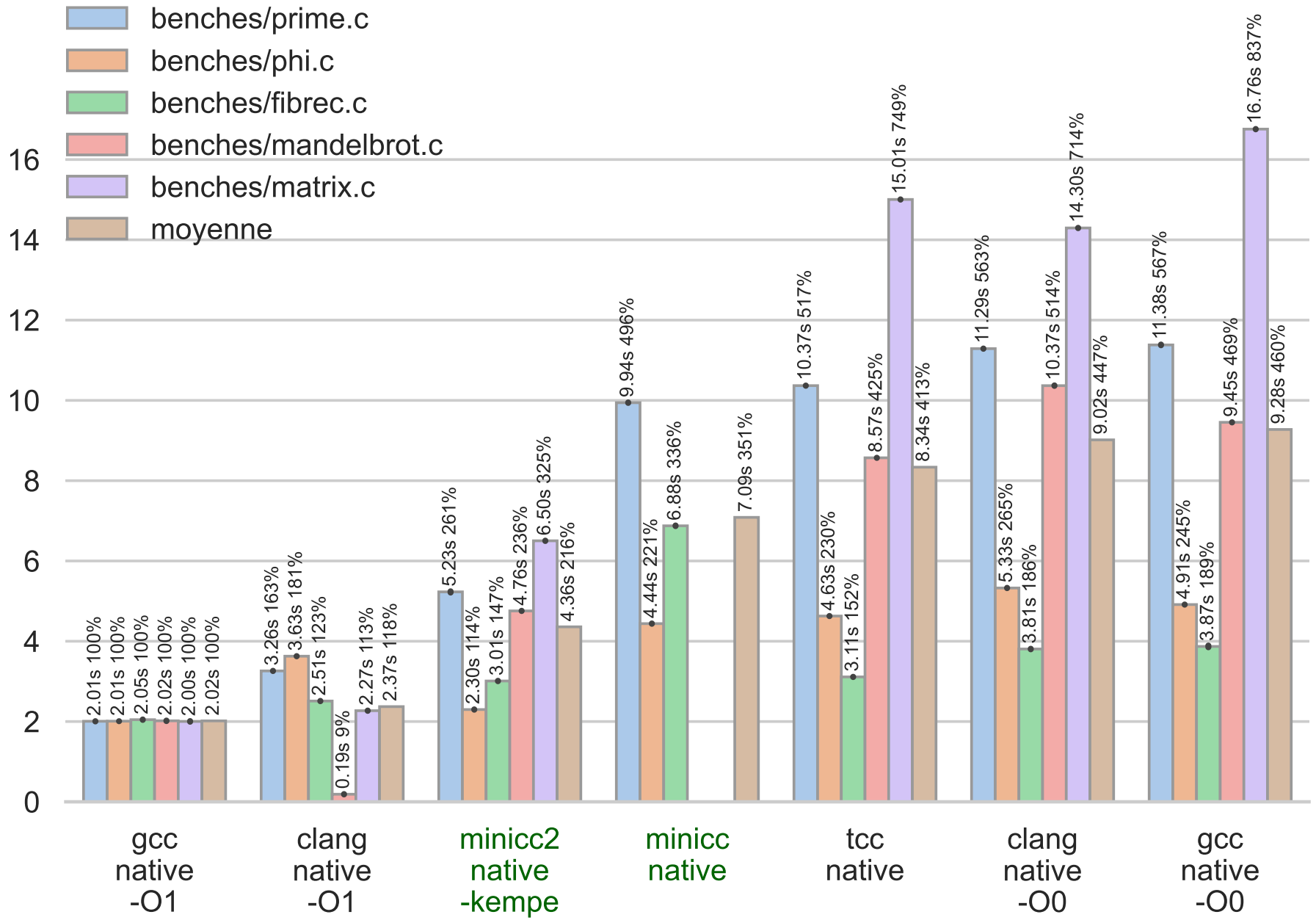
INF7641 Compilation
v251



Où on en est ?

- On a une représentation intermédiaire (IR)
- On sait faire quelques analyses (*dataflow*)

MiniCC2 -O0


benches/prime.c
benches/phi.c
benches/fibrec.c
benches/mandelbrot.c
benches/matrix.c
moyenne
0
2
4
6
8
10
12
14
16
2.01s 100%
2.01s 100%
2.05s 100%
2.02s 100%
2.00s 100%
2.02s 100%
3.26s 163%
3.63s 181%
2.51s 123%
0.19s 9%
2.27s 113%
2.37s 118%
5.23s 261%
2.30s 114%
3.01s 147%
4.76s 236%
6.50s 325%
4.36s 216%
9.94s 496%
4.44s 221%
6.88s 336%
7.09s 351%
10.37s 517%
4.63s 230%
3.11s 152%
8.57s 425%
15.01s 749%
8.34s 413%
11.29s 563%
5.33s 265%
3.81s 186%
10.37s 514%
14.30s 714%
9.02s 447%
11.38s 567%
4.91s 245%
3.87s 189%
9.45s 469%
16.76s 837%
9.28s 460%
gcc native -O1
clang native -O1
minicc2 native -kempe
minicc native
tcc native
clang native -O0
gcc native -O0

Aujourd'hui ?

IR $\rightarrow$ IR : MiniCC -O1

- Tant que pas satisfait: analyser & transformer IR

Difficultés

- Représentation de l'information
  IR et résultats d'analyses
- Maintient/recalcul de l'information
  Transformer = changer le code,
  donc possiblement invalider des analyses

Représentation de la satisfaction

- L'IR est plus joli ? Plus court ?

Aujourd'hui ?

IR $\rightarrow$ IR : MiniCC -O1

- Tant que pas satisfait: analyser & transformer IR

Difficultés

- Représentation de l'information
  IR et résultats d'analyses
- Maintient/recalcul de l'information
  Transformer = changer le code,
  donc possiblement invalider des analyses

Représentation de la satisfaction

- L'IR est plus joli ? Plus court ?
- Voire on mesure !

Plan

1. *Single static assignment* (SSA)
2. Interlude : dominance
3. Transformer en SSA
4. Nos premières optimisations

Single static assignment (SSA)

Rechability analysis

Chaînes *use-def*/*def-use*

- Utiles pour les optimisation
- Doit être recalculé souvent
- Information quadratique : defs $\times$ uses

SSA (*single static assignment*)

Situation idéale : une **seule affectation** (**définition**) par registre

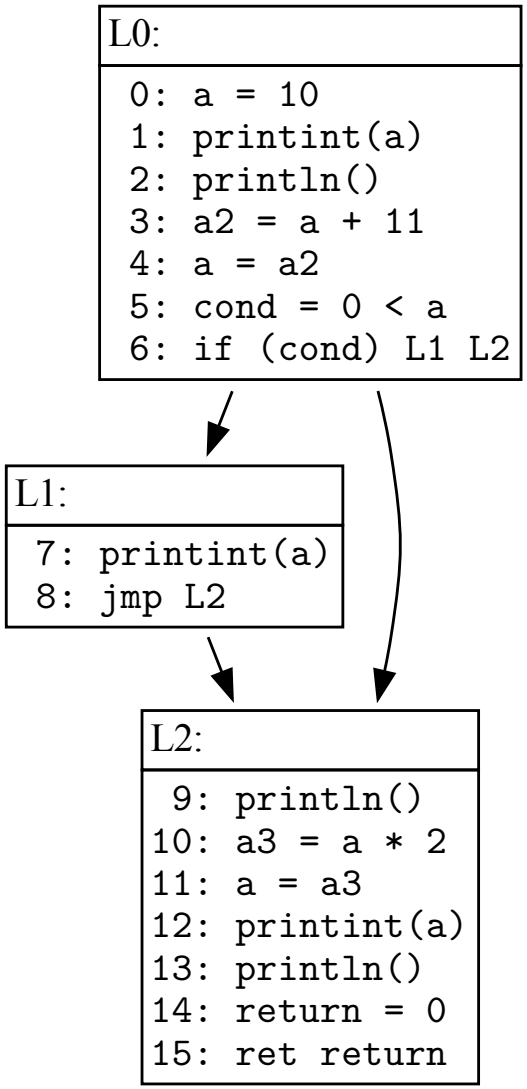
- C'est **statique** car c'est dans le code
- Mais cette affectation peut être exécutée plusieurs fois

Avantages

- Le *def* de ce registre est toujours connu (et unique)
- Les *uses* de ce registres sont **toutes** ses utilisations
- Pas besoin d'analyse *dataflow* pour les trouver
- Implémentation efficace possible pour mettre les uses en cache et accélérer les analyses et transformations

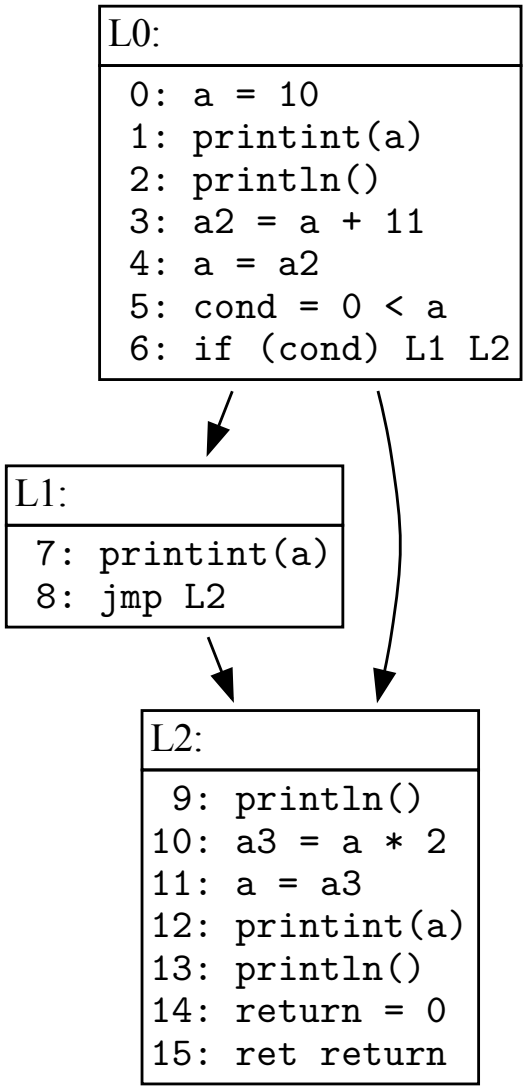
Un seule définition par registre. Mise en œuvre ?

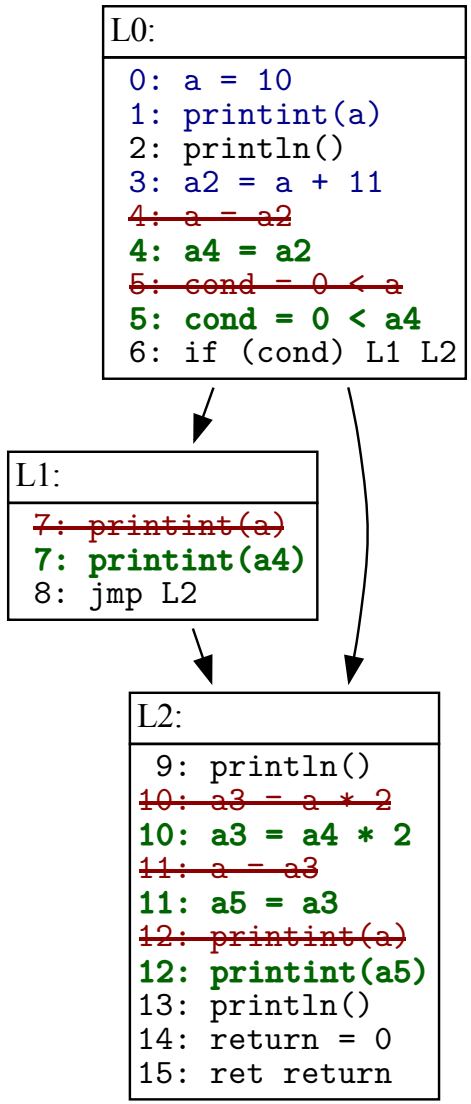
- Chaque affectation est associé à un nouveau registre
- $\rightarrow$ on renomme/numérote & maintient à jour les utilisations



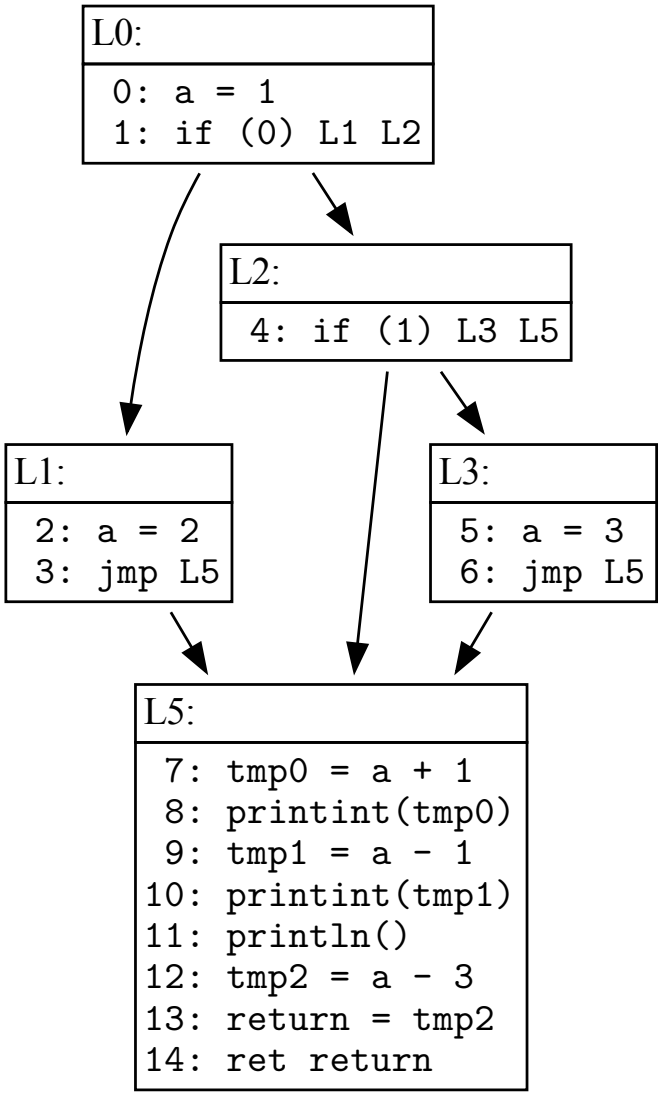
Un seule définition par registre. Mise en œuvre ?

- Chaque affectation est associé à un nouveau registre
- $\rightarrow$ on renomme/numérote & maintient à jour les utilisations



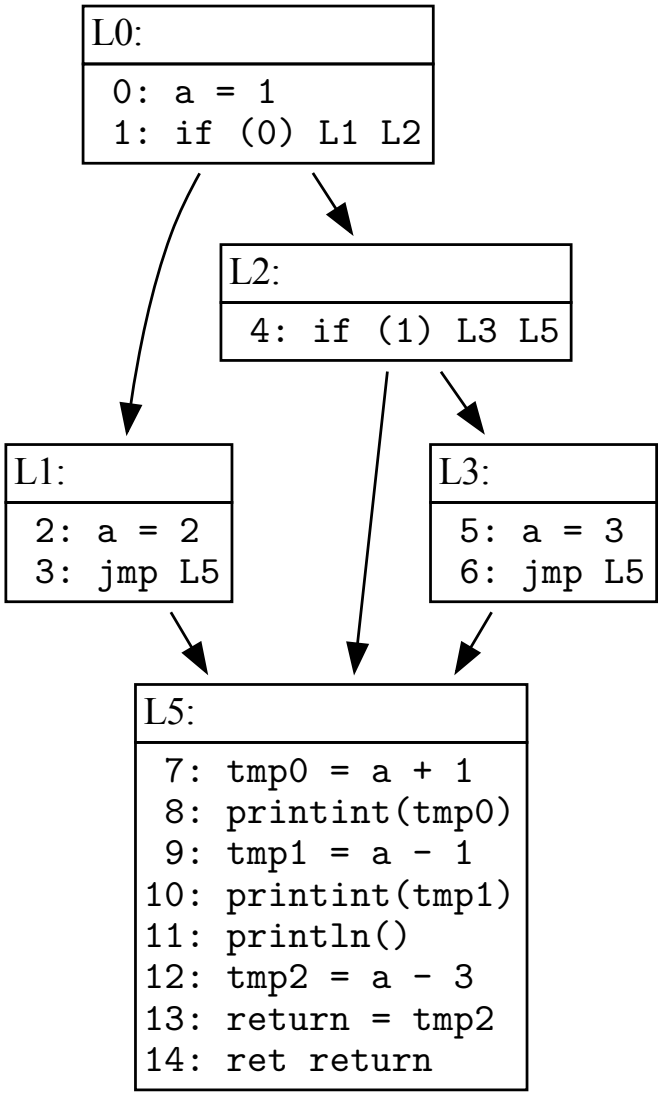


Problème : les jonctions. Comment renommer a ?

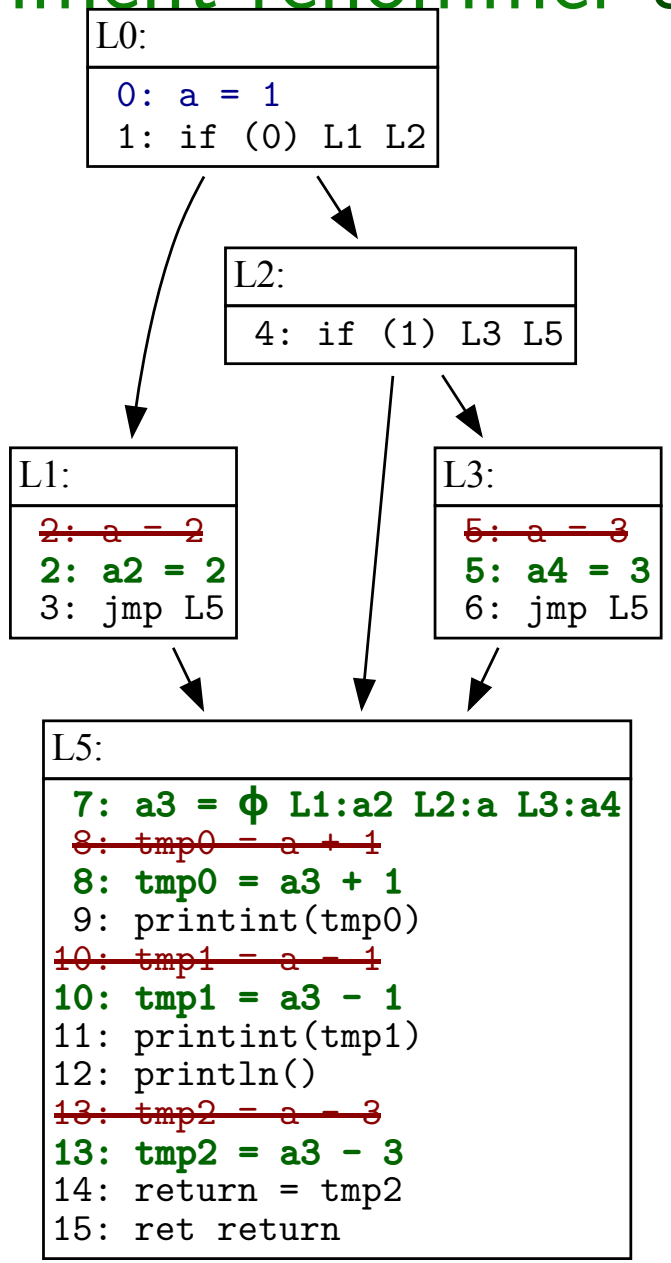


ssausedef.c: main()

Problème : les jonctions. Comment renommer a ?



ssausedef.c: main()



ssausedef.c: main()

Φ (phi)

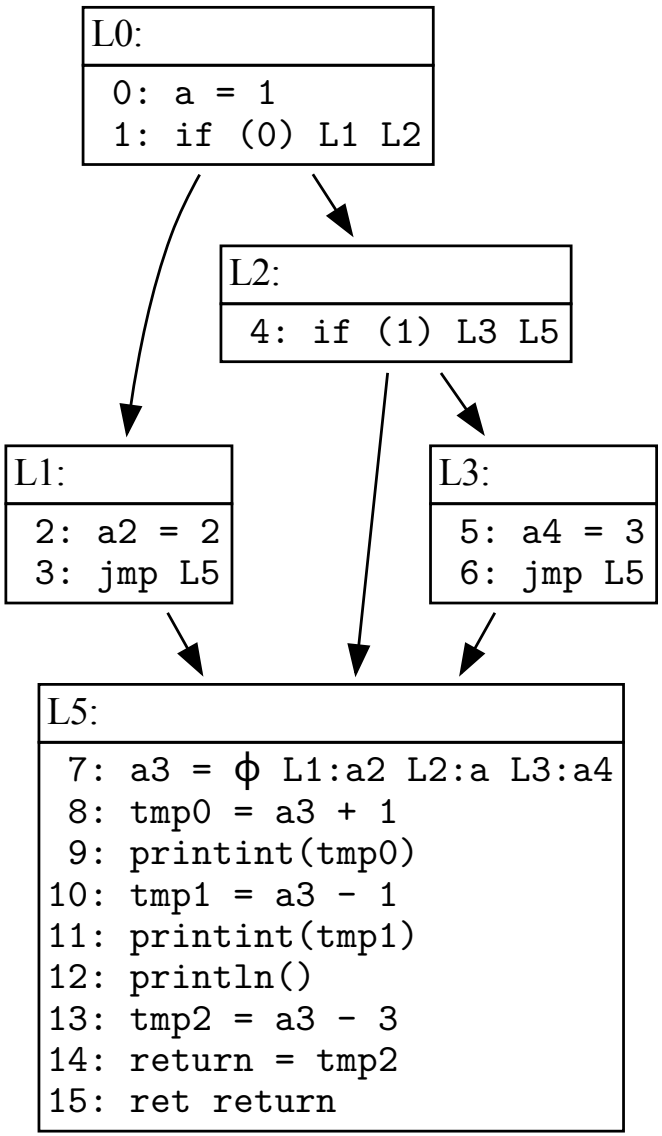
Instruction Φ

- Au début des blocs de base
- Combine les valeurs des différents arcs d'entrée
  - Chaque bloc d'entrée est associé à une opérande
- Dans un nouveau registre (car SSA)

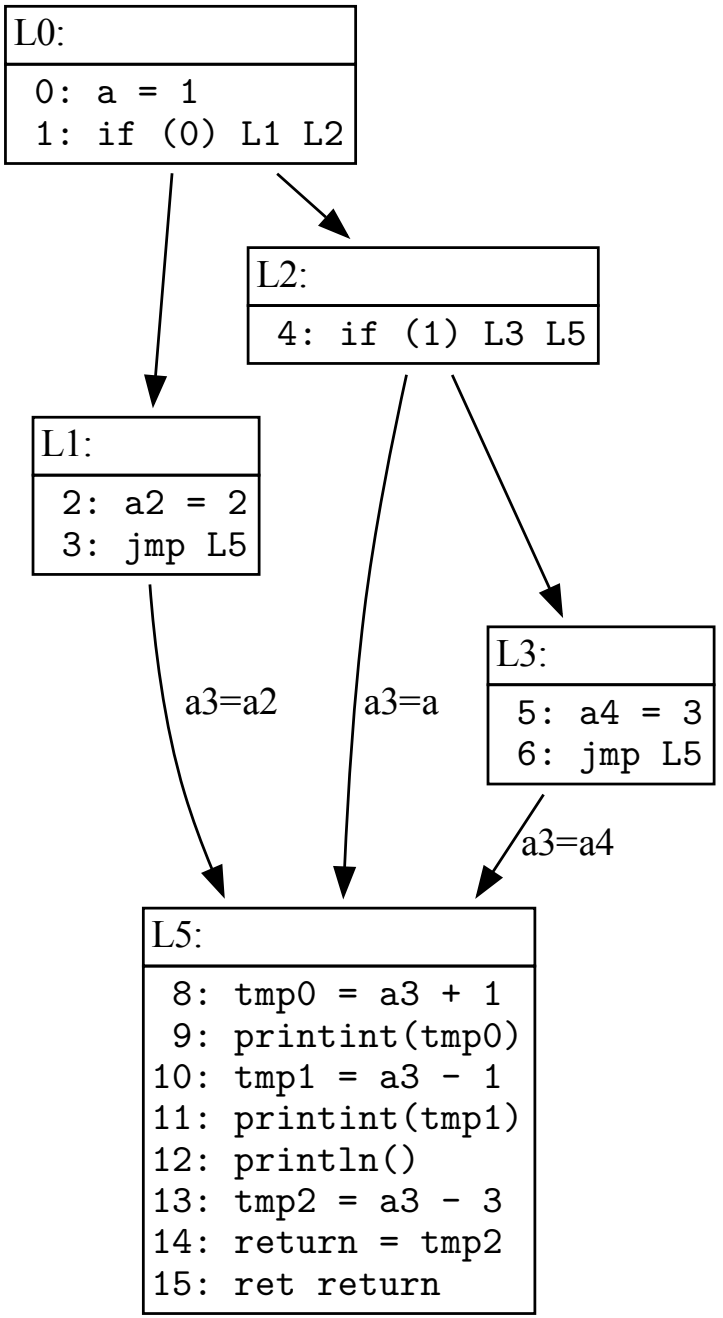
Sémantique de Φ

- Comme s'il y avait un `mv` sur chaque arc
- Attention, si plusieurs Φ,
  tous les `mv` sont à faire « en parallèle »

Sémantique du Φ



ssausedef.c: main()



Avantage et inconvénients de `phi`

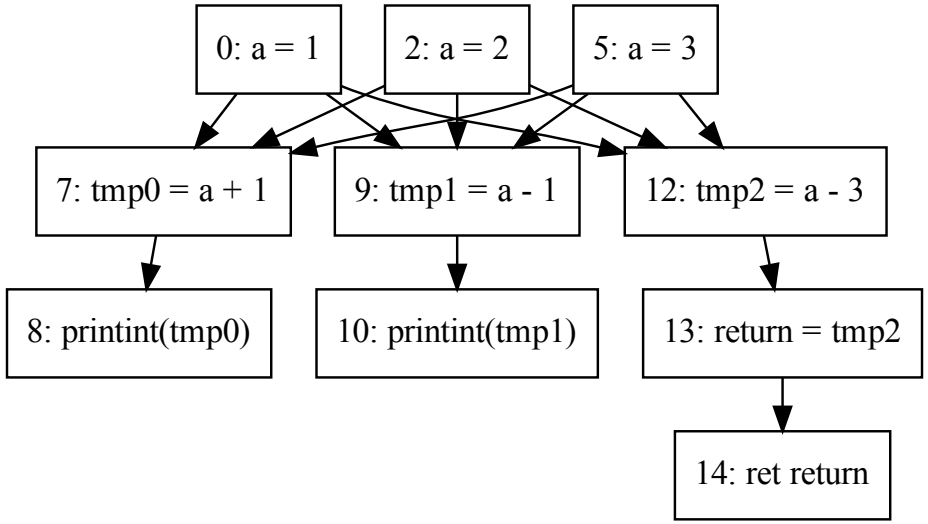
Avantages

- Φ se comporte à peu près comme n'importe quelle instruction
  - un `dst`, des `src`
- La représentation est SSA
  - Simplifie les chaînes *use-def* et *def-use*

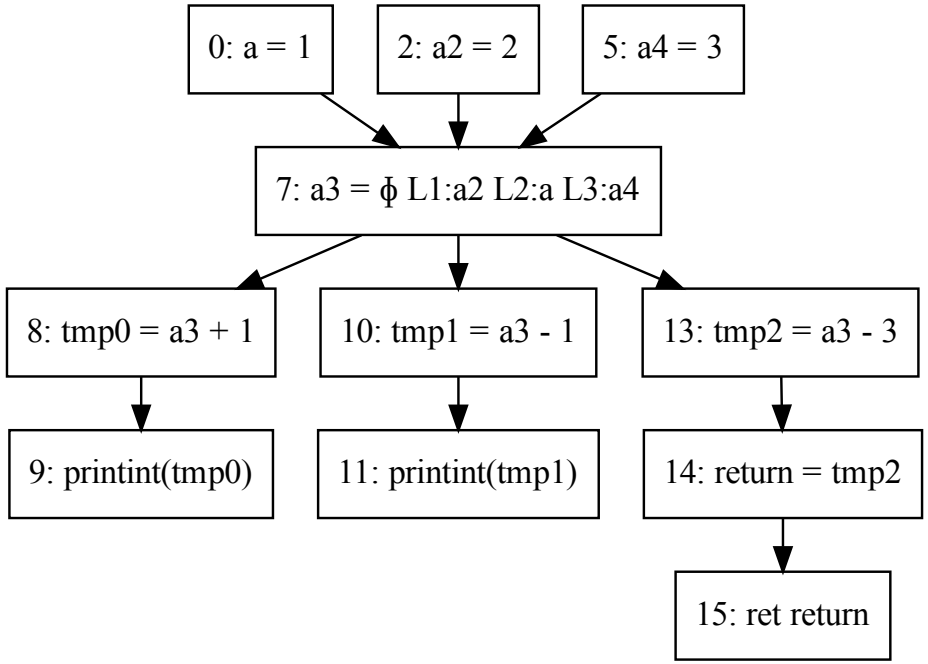
Inconvénients

- Pénible à mettre
- Pénible à enlever
- Modifier le CFG $\rightarrow$ maintenir la cohérence des Φ
- Plein de registres IR supplémentaires (l'allocateur doit être bon)

Simplification use-def



main



Transformer en SSA

Déterminer les blocs où les insérer les Φ

- Euh...

Renommer (numéroter) les registres

- Euh...

On est pas encore prêt

Quitter SSA : Enlever les Φ

Pour chaque bloc n du CFG avec des Φ

Concrétiser les arcs entrants de n

- Soit un bloc prédécesseur p avec $p \rightarrow n$
- On peut l'étendre si exclusif: $\texttt{succ}(p) = \{n\}$
- Sinon, insérer un nouveau bloc p'
  - $p \rightarrow p' \rightarrow n$, avec $\texttt{succ}(p') = \{n\}$

Mov

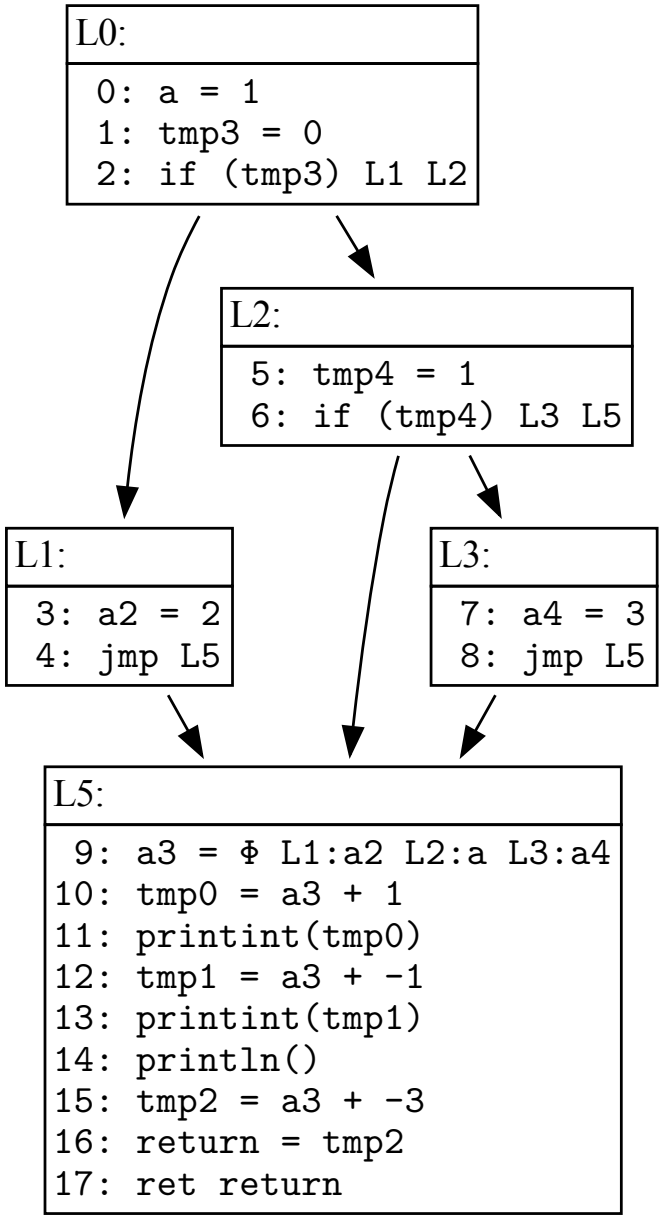
- Pour chaque $x = \Phi(p : y, ...)$
- Mettre un `x=y` à la fin de p

Sémantique parallèle

Attention, si plusieurs Φ

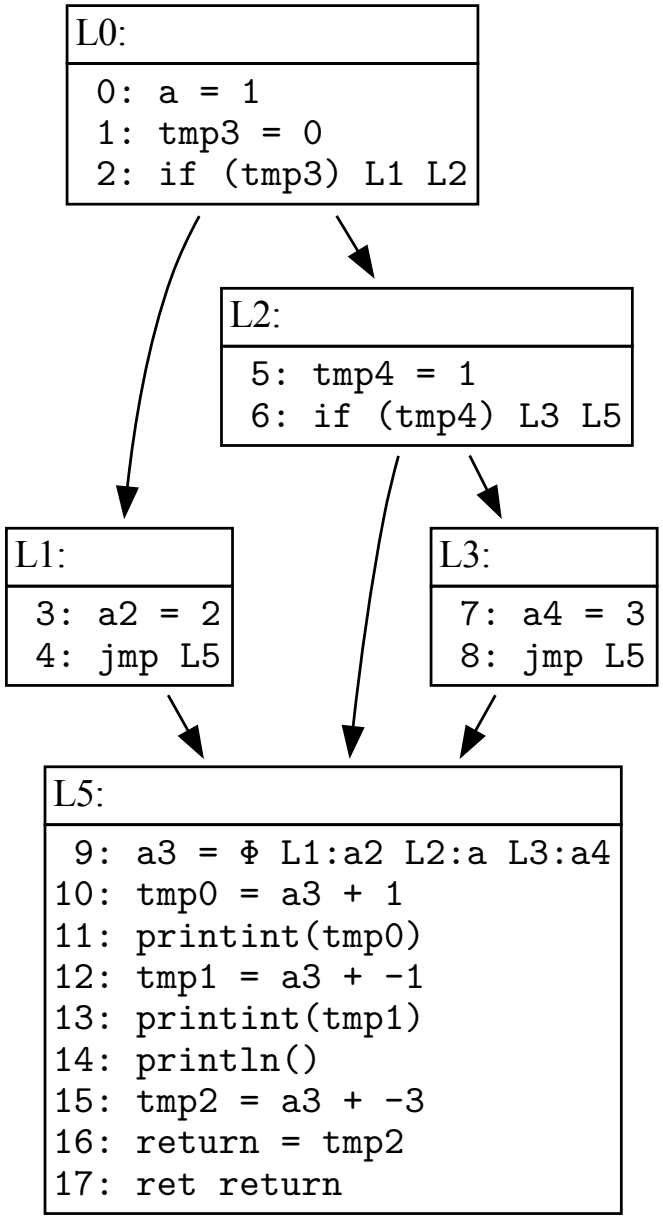
- Mettre les `mov` dans le bon ordre si dépendances
- Utiliser un registre temporaire supplémentaire si circularité
- $\rightarrow$ `PhiRemoval.java`

Exercice 1 : Enlever le Φ

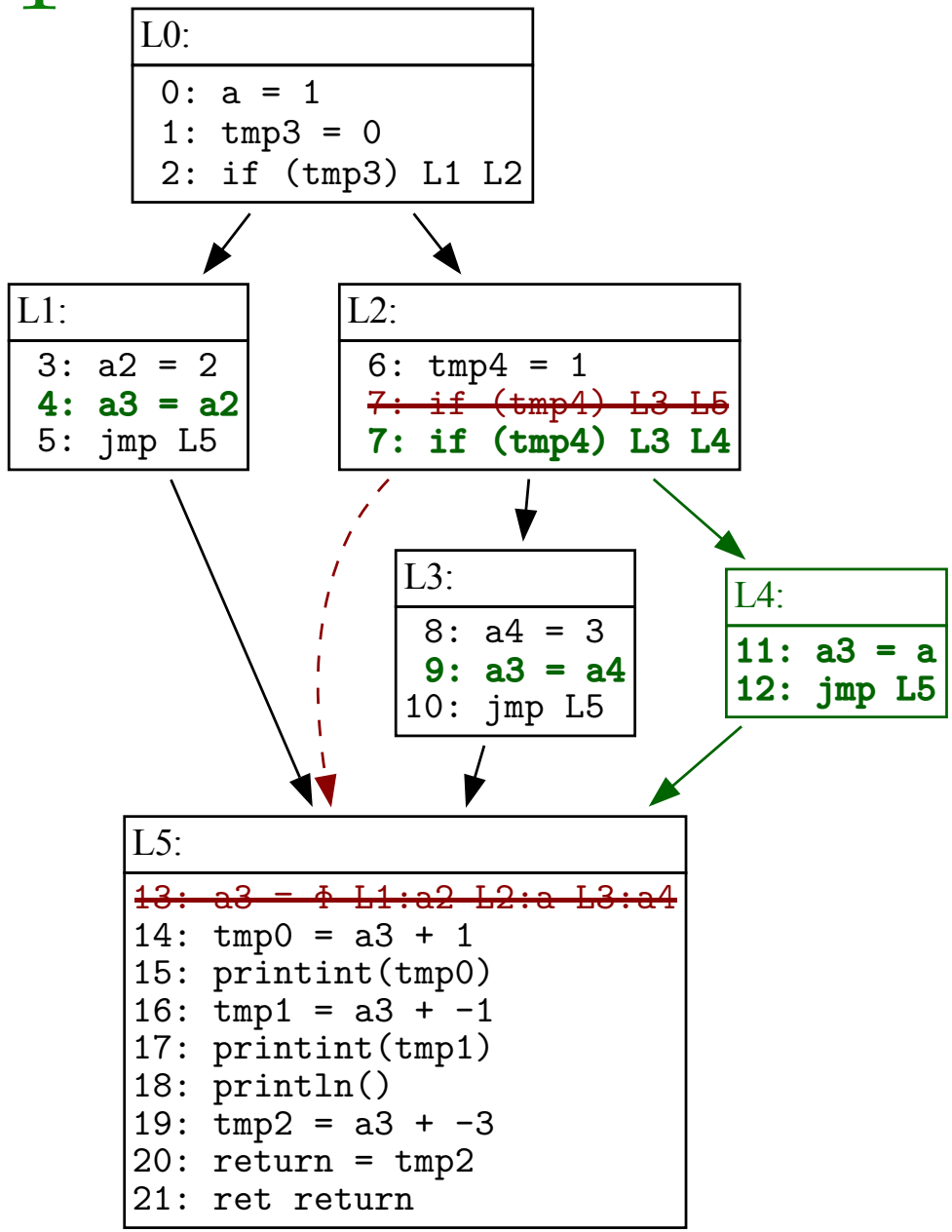


ssausedef.c: main()

Exercice 1 : Enlever le Φ

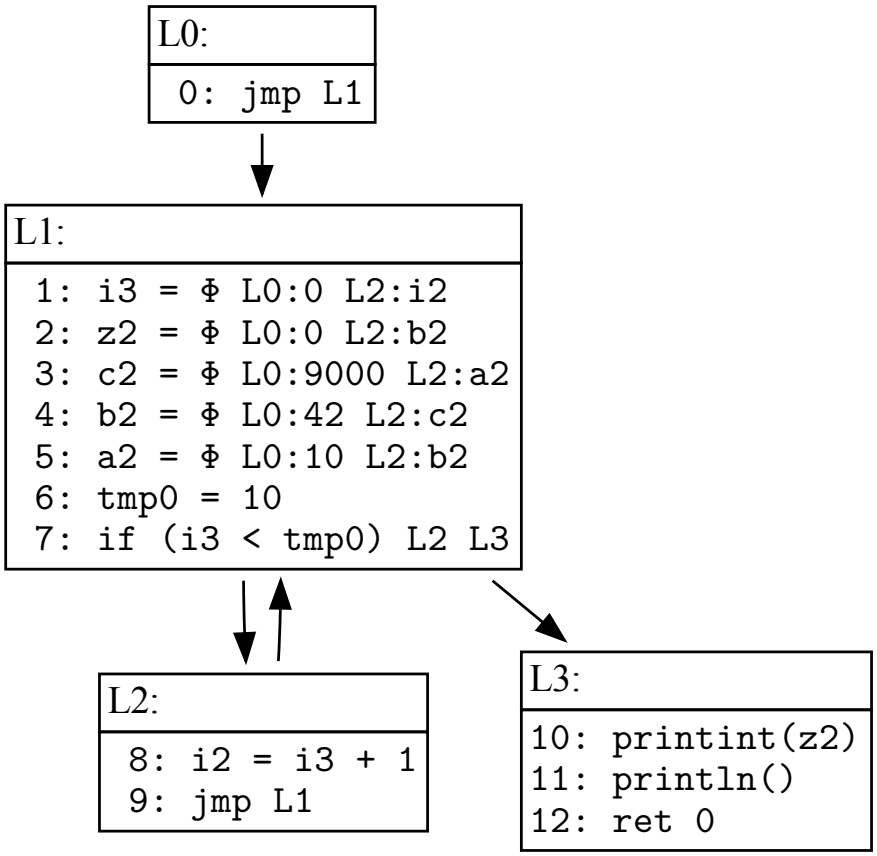


ssausedef.c: main()



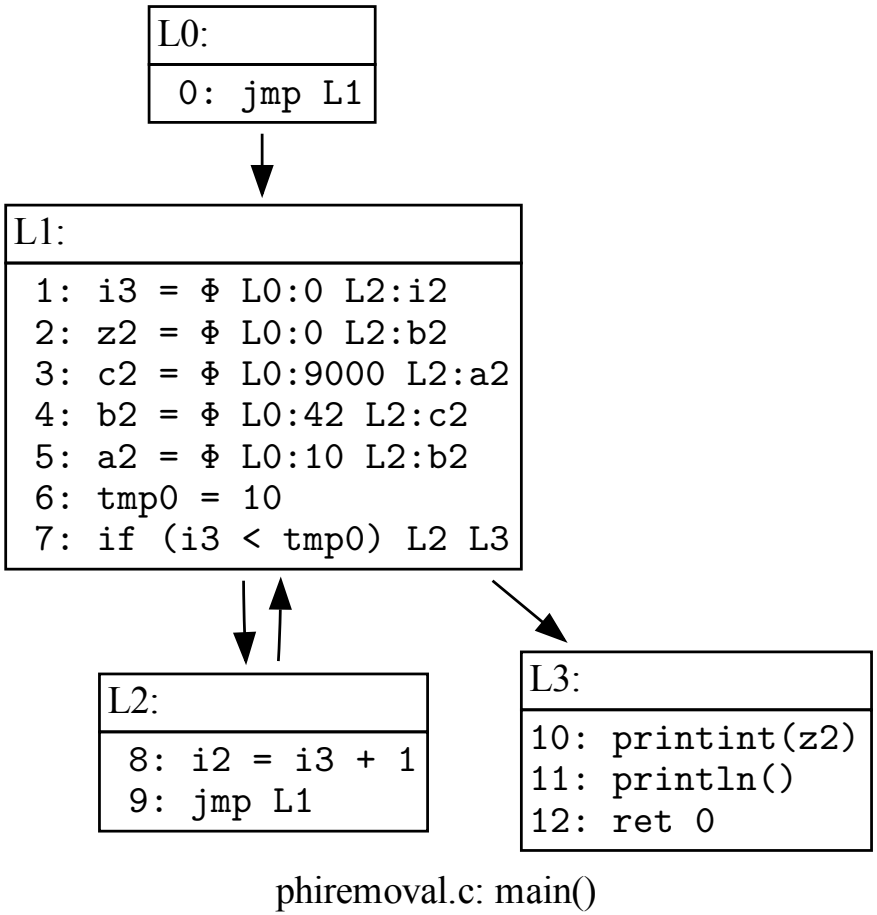
ssausedef.c: main()

Exercice 2 : Enlever les Φ

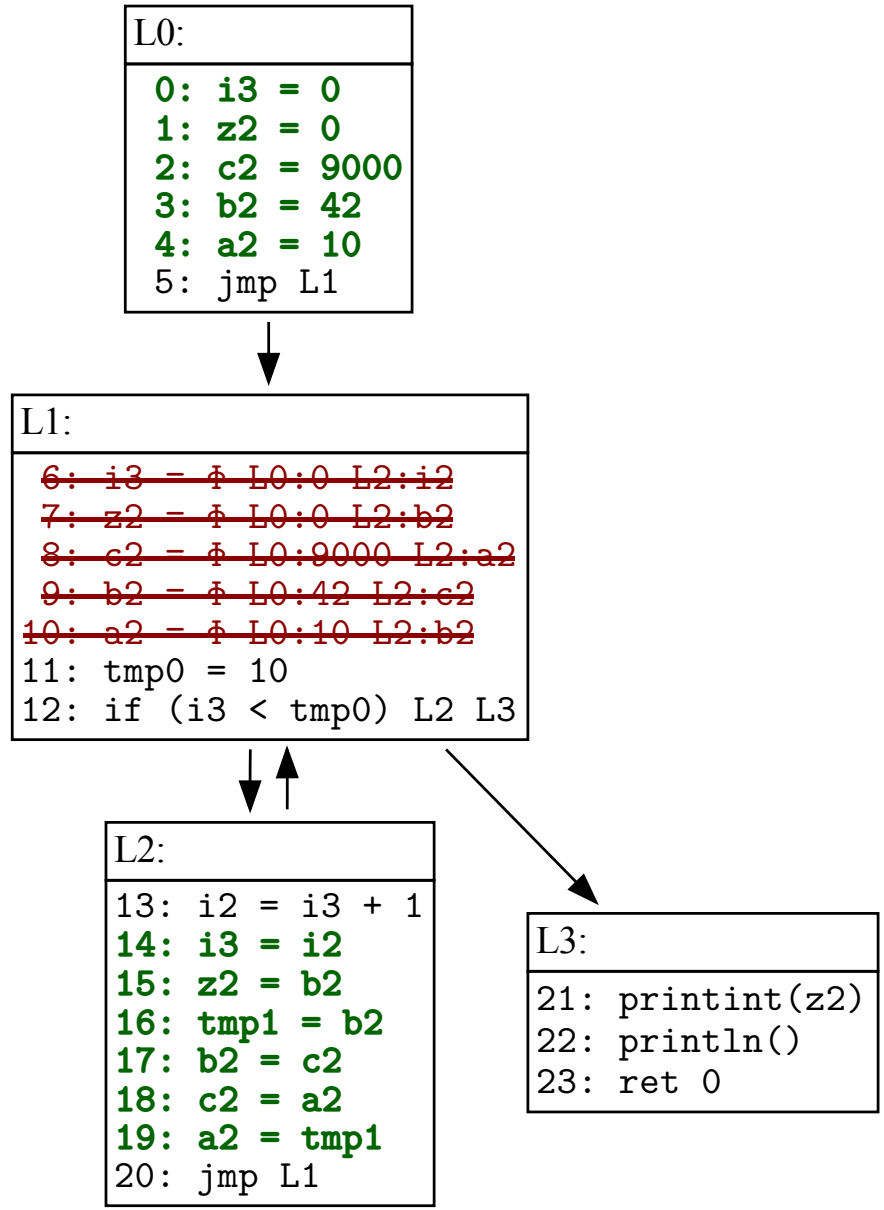


phiremoval.c: main()

Exercice 2 : Enlever les Φ



phiremoval.c: main()



phiremoval.c: main()

Interlude : dominance

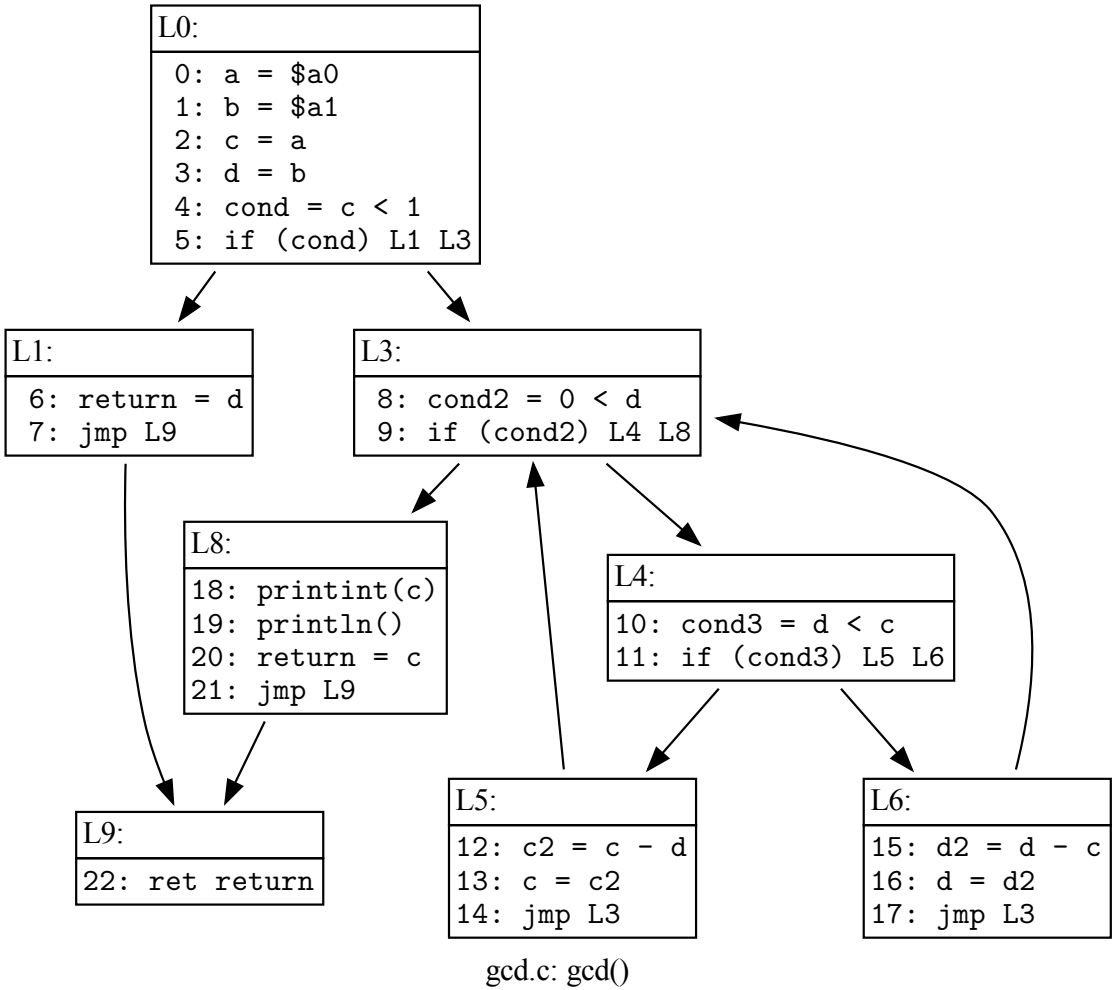
Dominance

- Dans un CFG (départ n_0), un nœud n_1 **domine** un nœud n_2
- (ou n_1 est un **dominateur** de n_2)
- Si : n_1 appartient à **tous** les chemins de n_0 à n_2

Vocabulaire

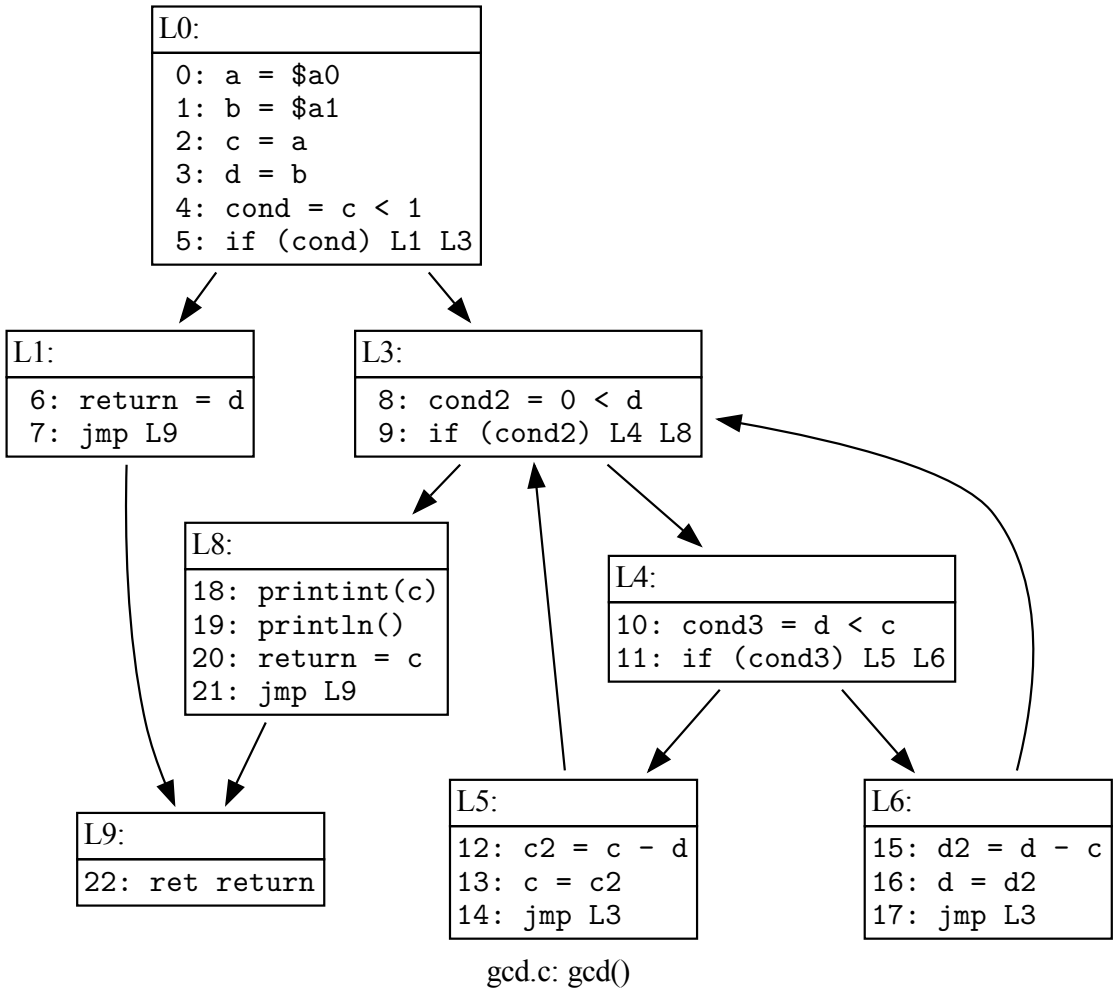
- dom (dominance) : relation d'ordre
  - réflexive, transitive, anti-associative
- sdom (domination stricte) : réduction réflexive de dom
  - n_1 sdom $n_2 \Leftrightarrow n_1 \neq n_2 \wedge n_1$ dom n_2
- idom (dominateur immédiat) : réduction transitive de sdom
  - $n_1 = \text{idom}(n_2) \Leftrightarrow n_1$ sdom $n_2 \wedge \nexists n_3, n_1$ sdom s_3 sdom n_2
  - Tous les nœuds, ont **un seul** dominateur immédiat
  - Sauf celui de départ
  - $\rightarrow$ On peut construire un arbre de dominance

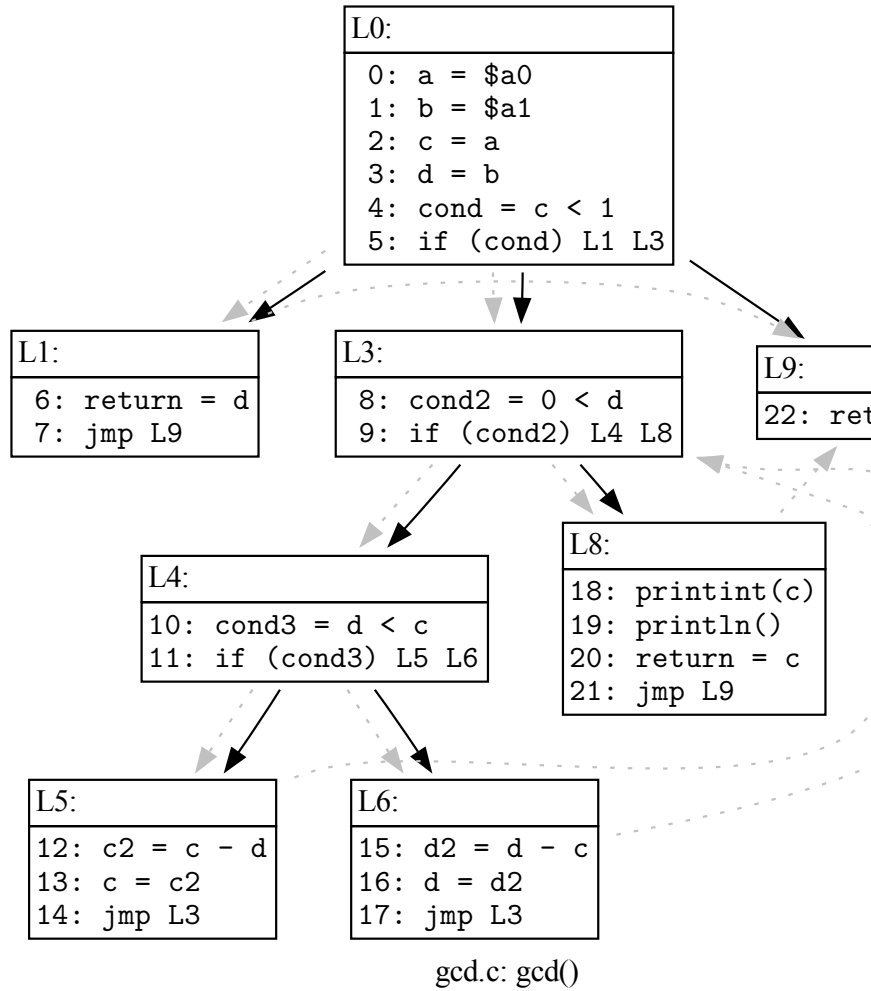
Exercice : Calculer la dominance



gcd.c: gcd()

Exercice : Calculer la dominance





Calcul de la dominance

Plusieurs algorithmes existent

- $O(m\alpha(m,n))$ Lengauer&Tarjan, 1979
  où α est l'inverse de la fonction d'Ackerman, croît très lentement
- $O(n^2)$ Cooper&Harvey&Kennedy, 2006

Calcul de la dominance

Plusieurs algorithmes existent

- $O(m\alpha(m,n))$ Lengauer&Tarjan, 1979
  où α est l'inverse de la fonction d'Ackerman, croît très lentement
- $O(n^2)$ Cooper&Harvey&Kennedy, 2006

Mais un simple dataflow fonctionne

- Chercher les dominateurs d'un bloc
- Ensemble :

Calcul de la dominance

Plusieurs algorithmes existent

- $O(m\alpha(m,n))$ Lengauer&Tarjan, 1979
  où α est l'inverse de la fonction d'Ackerman, croît très lentement
- $O(n^2)$ Cooper&Harvey&Kennedy, 2006

Mais un simple dataflow fonctionne

- Chercher les dominateurs d'un bloc
- Ensemble : les blocs
- Ensemble de chaque bloc : ses dominateurs
- Sens : en avant (le dominateur est **avant** le dominé)
- Fusion : intersection (sur **tous** les chemins)
- Gen: le bloc lui-même
- Kill: rien
- $\rightarrow$ `Dominance.java`

Frontière de dominance

- Un nœud f appartient à la **frontière** de n si :
- Il existe un prédécesseur p de f, pour qui **tous les chemins** depuis l'entrée n_0 passent par n
- Et, il **existe un chemin** vers f depuis n_0 qui ne passe pas par n
- $\rightarrow$ f est hors mais à la limite de la zone de dominance de n

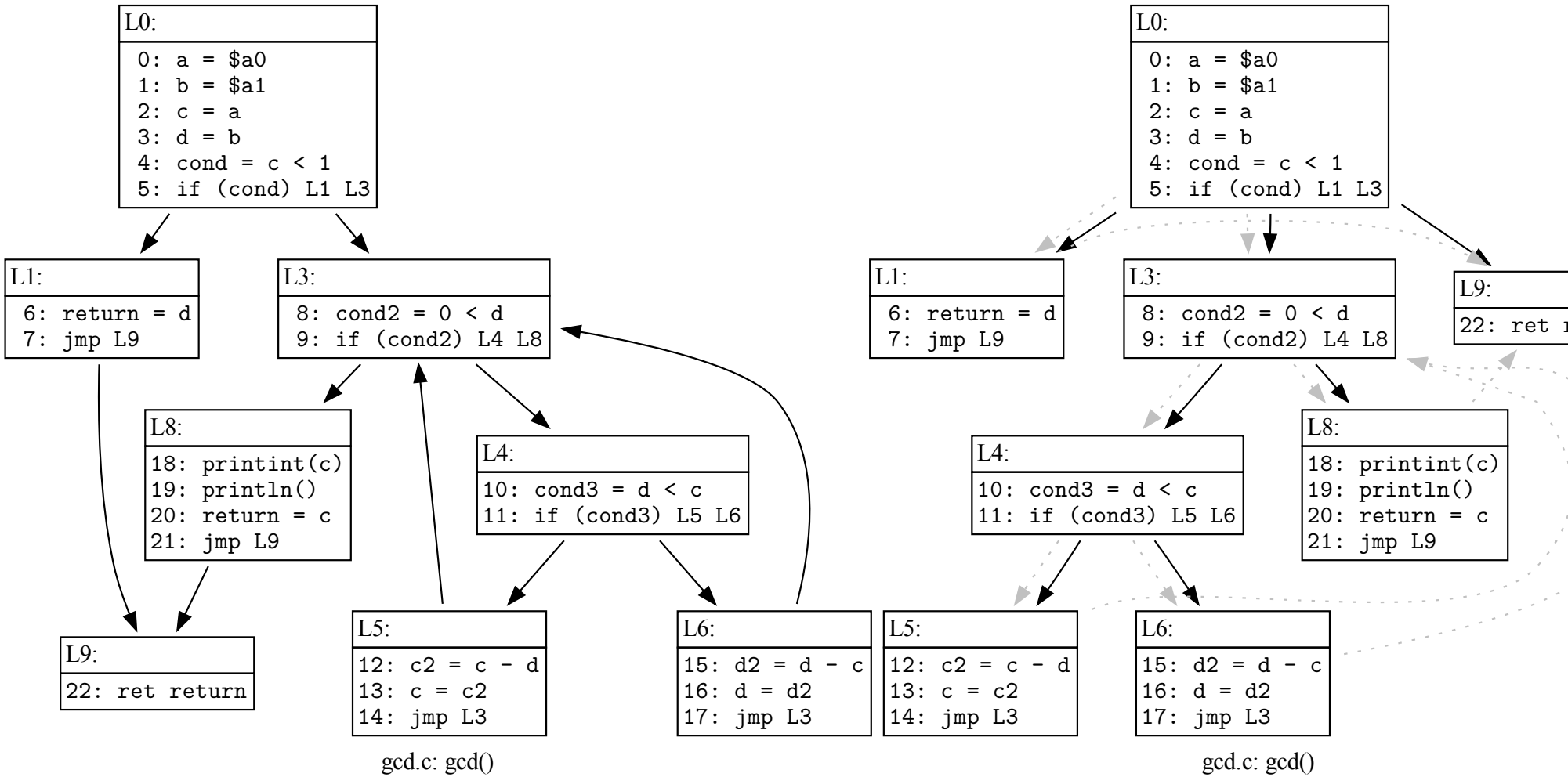
Plus formellement ?

- $f \in \mathsf{df}(n)$
  - f pas strictement dominé par n
  - f successeur d'un sommet dominé par n
- $\mathsf{df}(n) = \{f | \neg(n \text{ sdom } f) \wedge \exists p \in \mathsf{pred}(f), n \text{ dom } p\}$

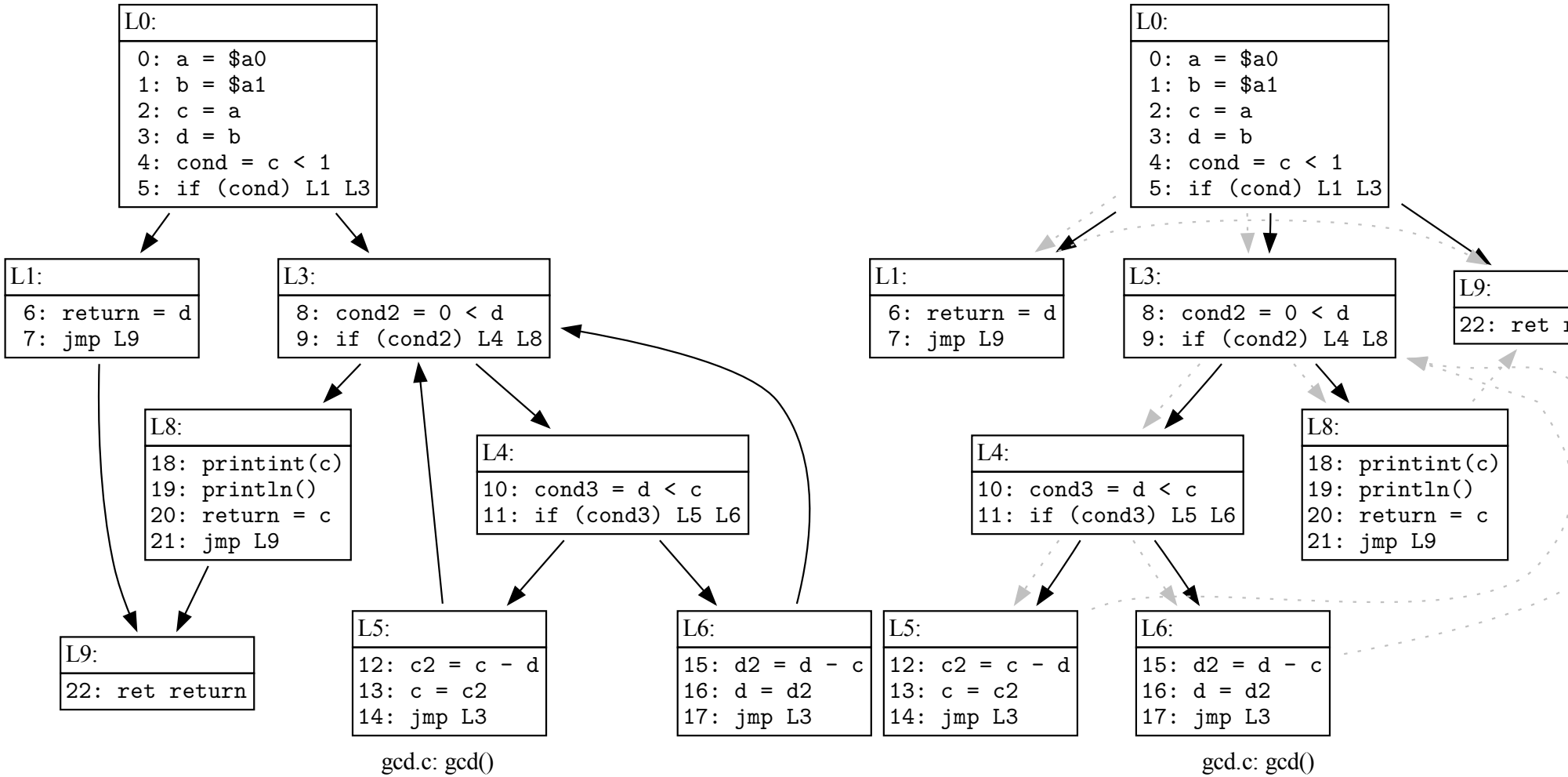
Frontière étendue

- La frontière d'un ensemble de nœuds est l'union des frontières de ces nœuds
- $\mathsf{df}(N) = \bigcup_{n \in N} \mathsf{df}(n)$

Exercice : déterminer les frontières

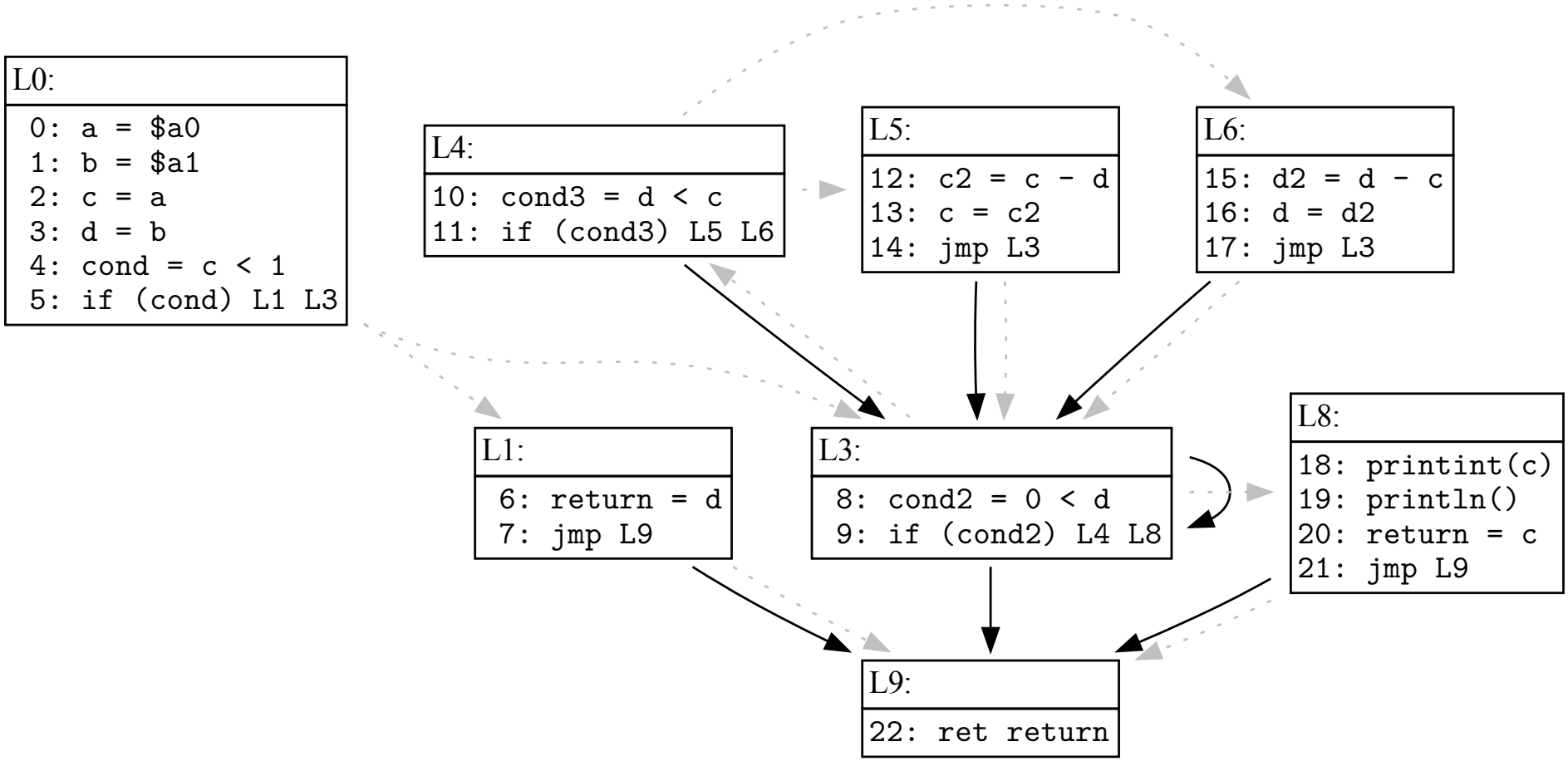


Exercice : déterminer les frontières



L0:[] L1:[L9] L3:[L3, L9] L4:[L3] L5:[L3] L6:[L3] L8:[L9] L9:[]

Relation de frontière de dominance



gcd.c: gcd()

Transformer en SSA

Placer les instructions Φ

- Soit N l'ensemble des blocs qui ont une définition de r
- Il faut placer un $r = \Phi(...)$ à chaque nœuds de $\mathsf{df}(N)$
- Mais ! Ça ajoute une nouvelle définition de r !
- Donc on itère
  - $\mathsf{df}_1(N) = \mathsf{df}(N)$
  - $\mathsf{df}_{i+1}(N) = \mathsf{df}(N \cup \mathsf{df}_i(N))$
  - Jusqu'au point fixe : $\mathsf{df}^+(N)$

Algorithme d'insertion de Φ

pour *chaque registre* r **faire**
- Liste $\leftarrow \emptyset$;
- **pour** *chaque nœud* n *affectant* r **faire**
  - Ajouter n à Liste ;
- **tant que** *Liste* $\neq \emptyset$ **faire**
  - Retirer n de Liste ;
  - **pour** *chaque* $f \in df(n)$ **faire**
    - Ajouter une Φ dans f pour r (sauf si déjà un) ;
    - Ajouter f à Liste (sauf si on a déjà traité f) ;

Renommage des registres

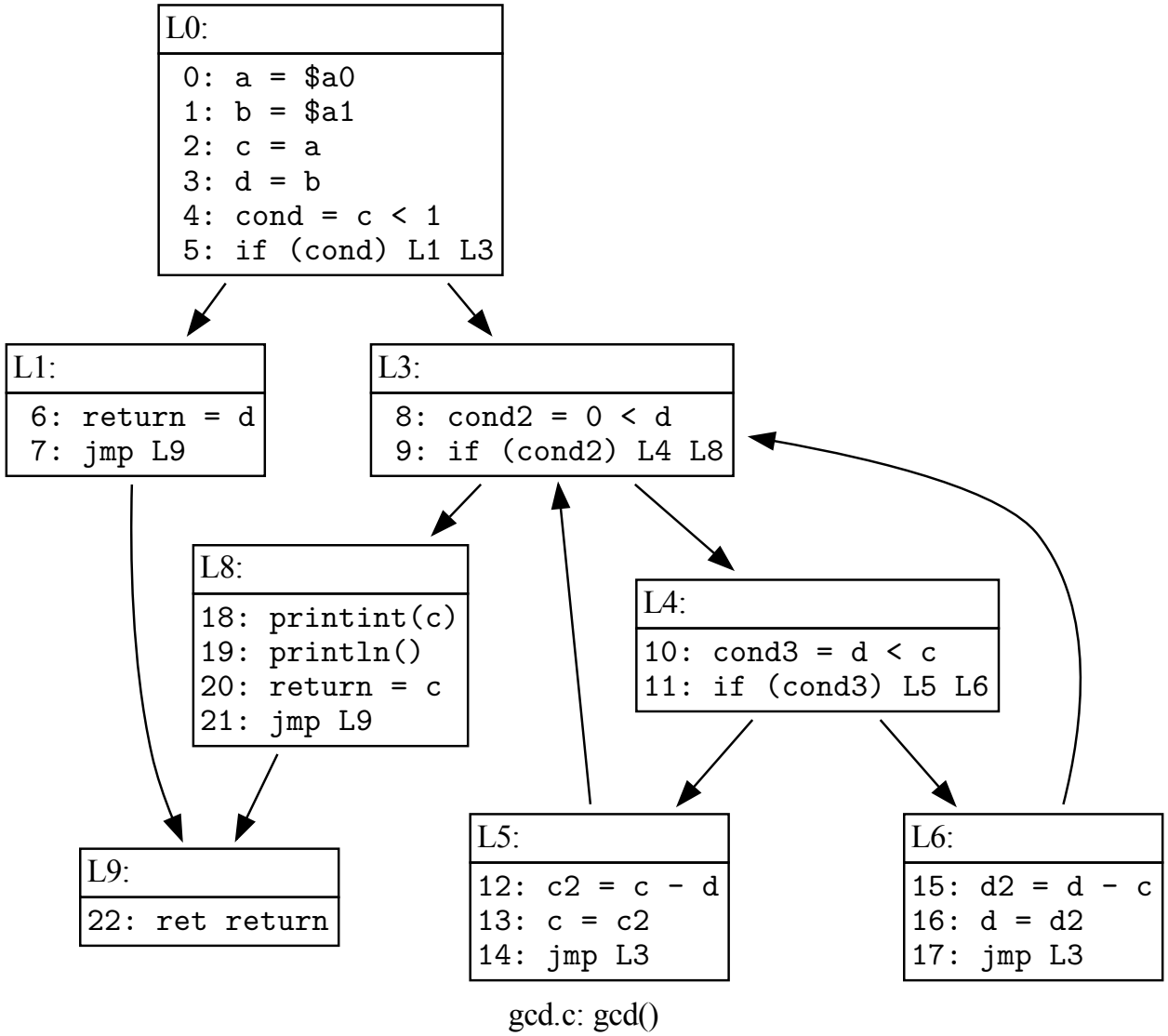
- Pour chaque registre, on conserve une pile de noms
  - L'historique des noms
- On visite en profondeur le CFG
  - Remplacer les opérandes par les noms les plus récents
  - Créer et empiler des nouveaux noms pour les résultats
  - Mettre à jour les entrées des Φ

pour *chaque registre* r **faire**
  pile[r] = nouvelle pile vide;

renommer(nœud d'entrée du CFG) ;

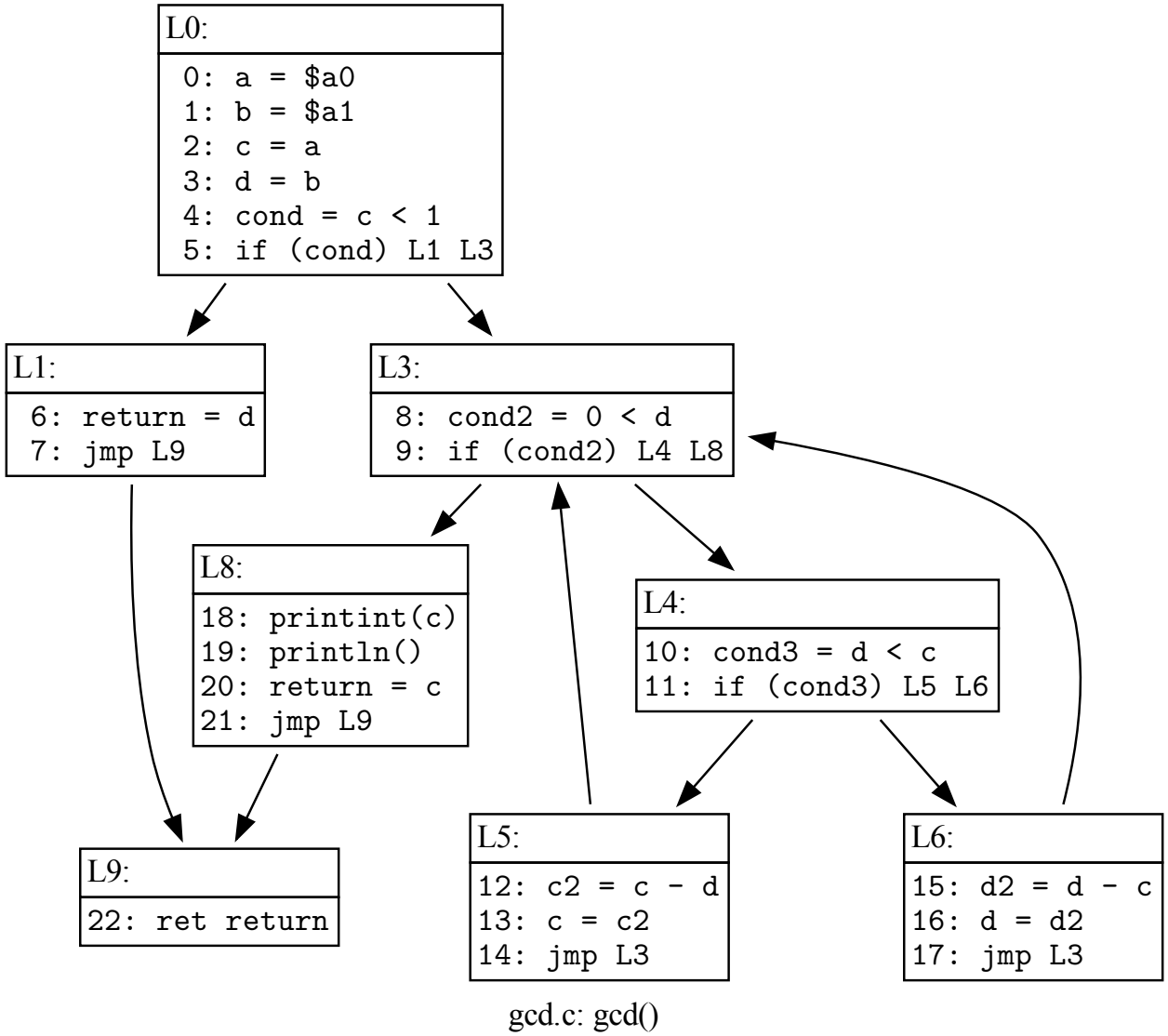
```
fun renommer(n: Nœud)
    pour chaque instruction i de n faire
        pour chaque opérande r de i faire
            Remplacer r par tête(pile[r]) ;
        pour le résultat r de i faire
            Creer un nouveau nom r' ;
            Empiler r' dans pile[r] ;
            Remplacer r par r' ;
    pour chaque successeur s de n faire
        pour chaque instruction r=Φ(...) de s faire
            Associer tête(pile[r]) pour l'arc n → s ;
        si s pas encore visité alors
            renommer(s) ;
    Dépiler (restaurer) les piles ;
```

Exercice : Renommer c, d et return

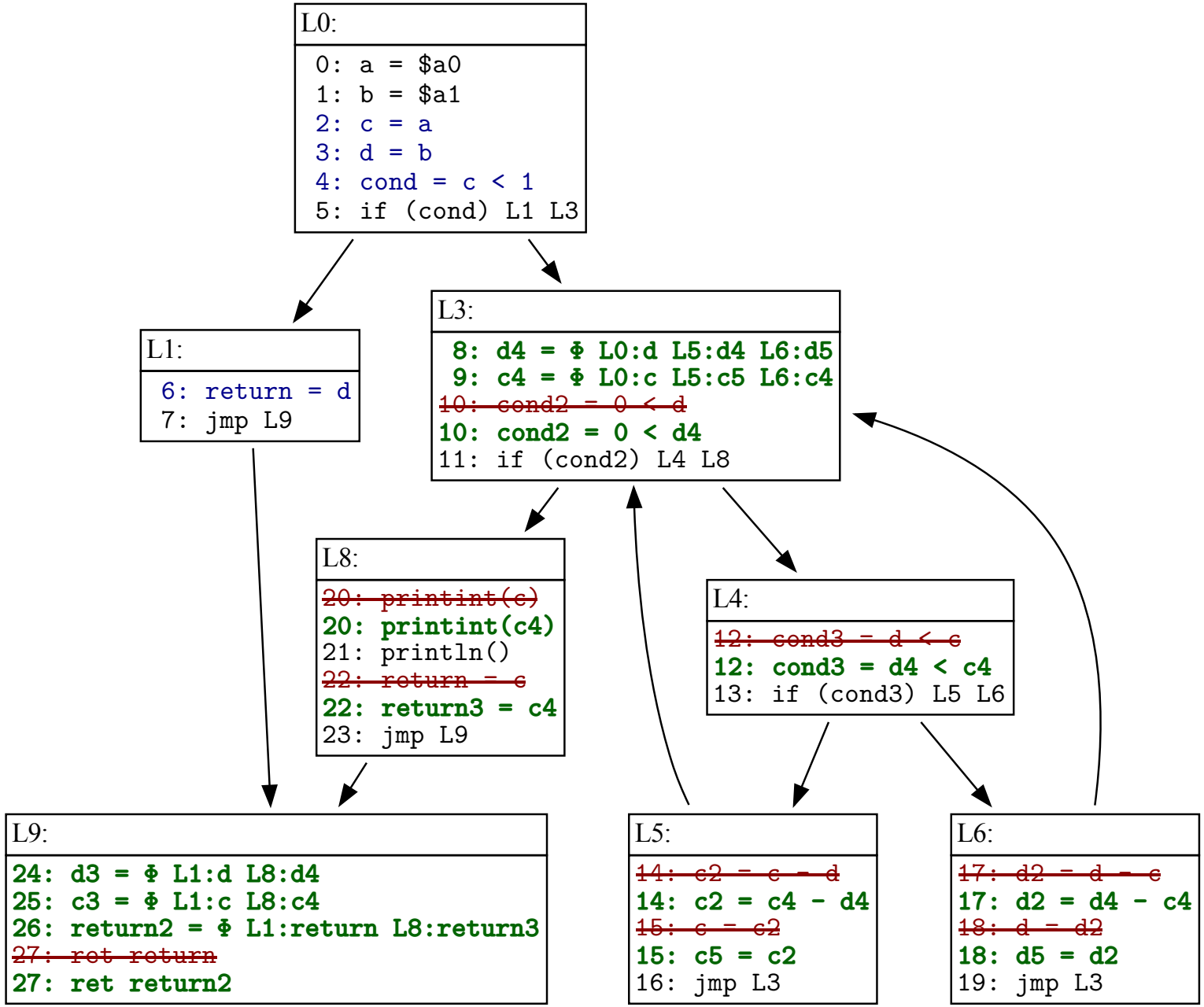


Où sont les Φ ?

Exercice : Renommer c, d et return



Où sont les Φ ? Tous dans L3 et L9


L0:
0: a = $a0
1: b = $a1
2: c = a
3: d = b
4: cond = c < 1
5: if (cond) L1 L3
L1:
6: return = d
7: jmp L9
L3:
8: d4 = Φ L0:d L5:d4 L6:d5
9: c4 = Φ L0:c L5:c5 L6:c4
10: cond2 = 0 < d
10: cond2 = 0 < d4
11: if (cond2) L4 L8
L8:
20: printint(c)
20: printint(c4)
21: println()
22: return = c
22: return3 = c4
23: jmp L9
L4:
12: cond3 = d < c
12: cond3 = d4 < c4
13: if (cond3) L5 L6
L9:
24: d3 = Φ L1:d L8:d4
25: c3 = Φ L1:c L8:c4
26: return2 = Φ L1:return L8:return3
27: ret return
27: ret return2
L5:
14: c2 = c - d
14: c2 = c4 - d4
15: c = c2
15: c5 = c2
16: jmp L3
L6:
17: d2 = d - c
17: d2 = d4 - c4
18: d = d2
18: d5 = d2
19: jmp L3

gcd.c: gcd()

SSA: résumé

- Une **seule définition** par registre
- Instruction **magique** Φ pour combiner les valeurs possibles
- La définition **domine** les utilisations
  - (sauf, bien sûr, les opérandes des nœuds Φ)
- $\rightarrow$ `SSA.java`

Nos premières optimisations

Quelques optimisations faciles

Déjà faciles

- *Constant folding* : `x=1+2` $\rightarrow$ `x=3`
- *Algebraic simplification* : `x*1` $\rightarrow$ `x`
- *Strength reduction* : `y=x*8` $\rightarrow$ `y=x<<3`

Plus faciles avec SSA

- *Constant propagation* : `y=1; z=x+y` $\rightarrow$ `z=x+1`
- *Copy propagation* : `y=x; z=y+1` $\rightarrow$ `y=x; z=x+1`
- *Algebraic combinaison* : `y=x+1; z=y+1` $\rightarrow$ `y=x+1; z=x+2`
- *Common subexpression elimination*
  `x=b*c+d; y=b*c+e` $\rightarrow$ `t=b*c; x=t+d; y=t+e`
- *Dead code elimination* : supprimer variables inutilisées

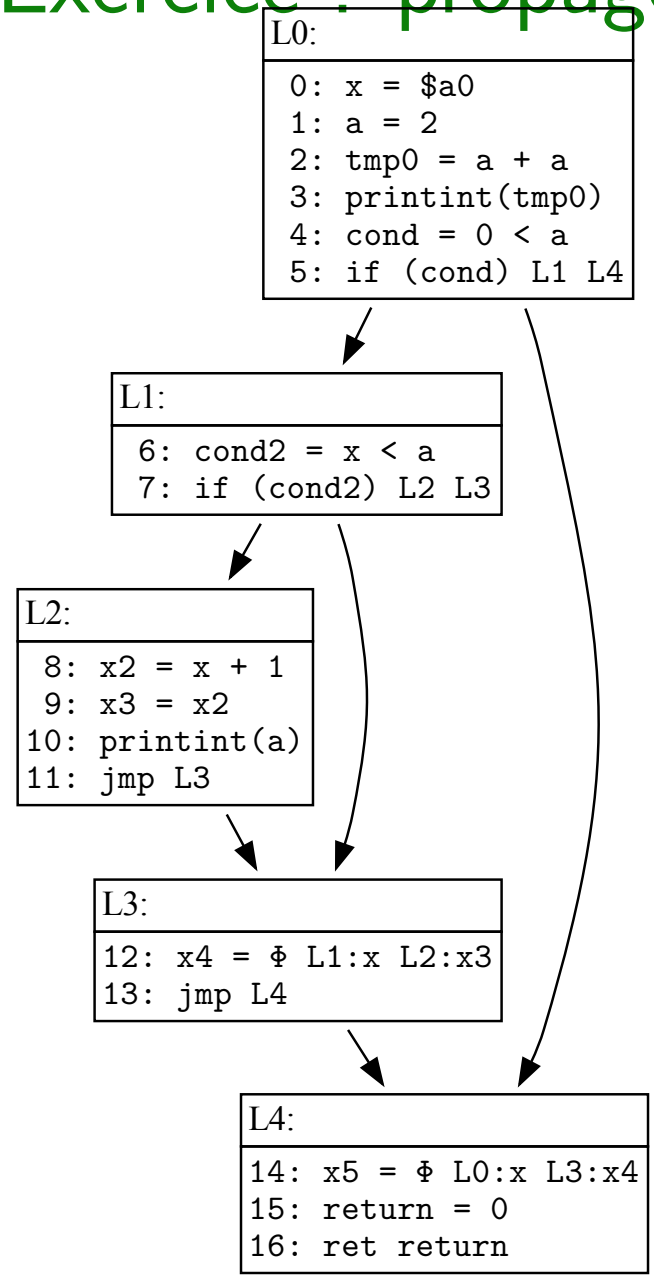
Sparse Conditional Constant Propagation (SCCP)

- Wegman&Zadeck, 1991
- Analyse *dataflow* « moderne »
- Propage les valeurs et calculs constantes
- Élimine les branches conditionnelles mortes
- Profite de SSA
- Rapide

Alternative : à la main, plus lent et moins efficace

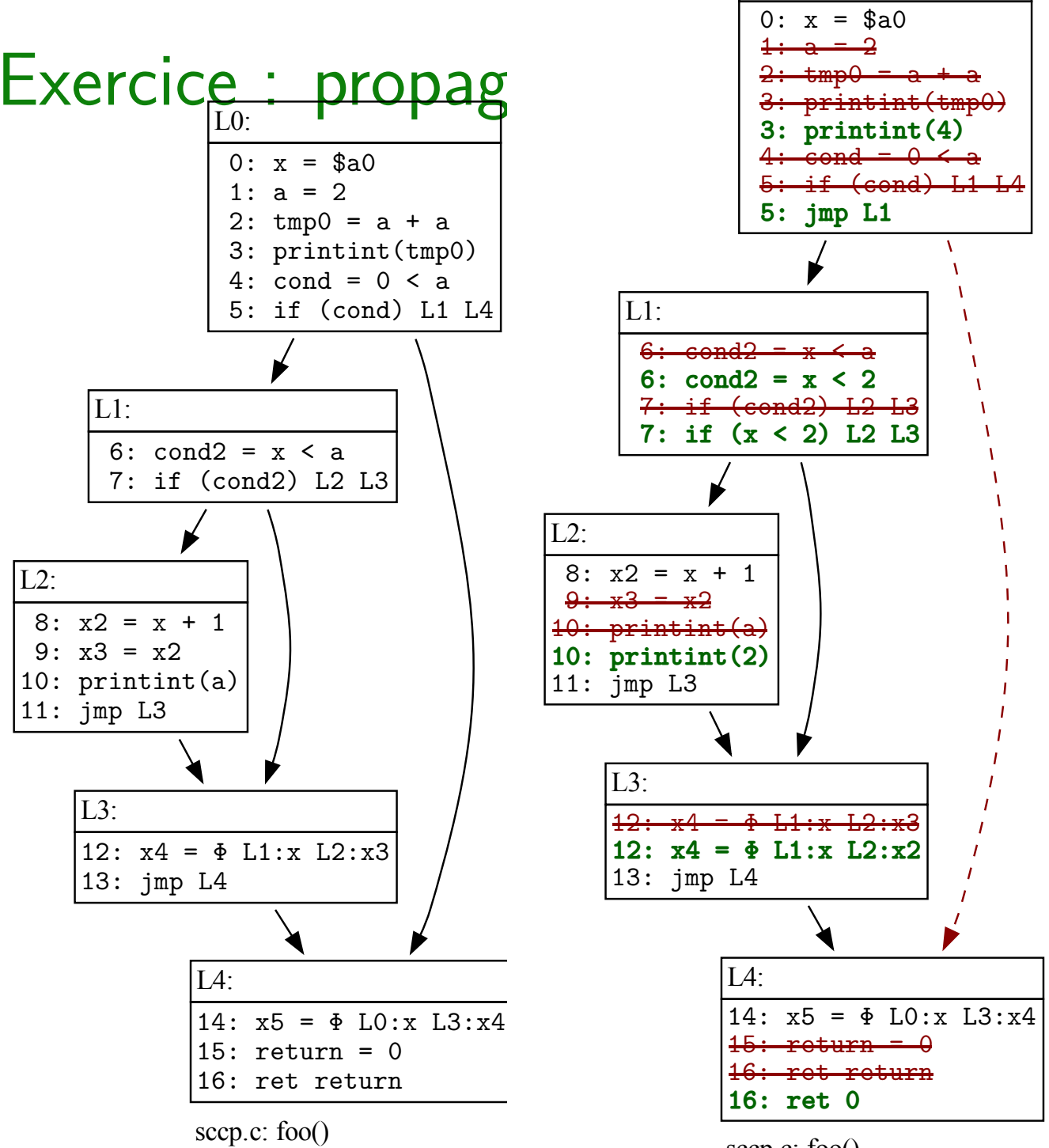
- Itérer, simplifier, propager
- Voir `ConstantPropagation.java` et `DeadCodeElimination.java`

Exercice : propager les constantes

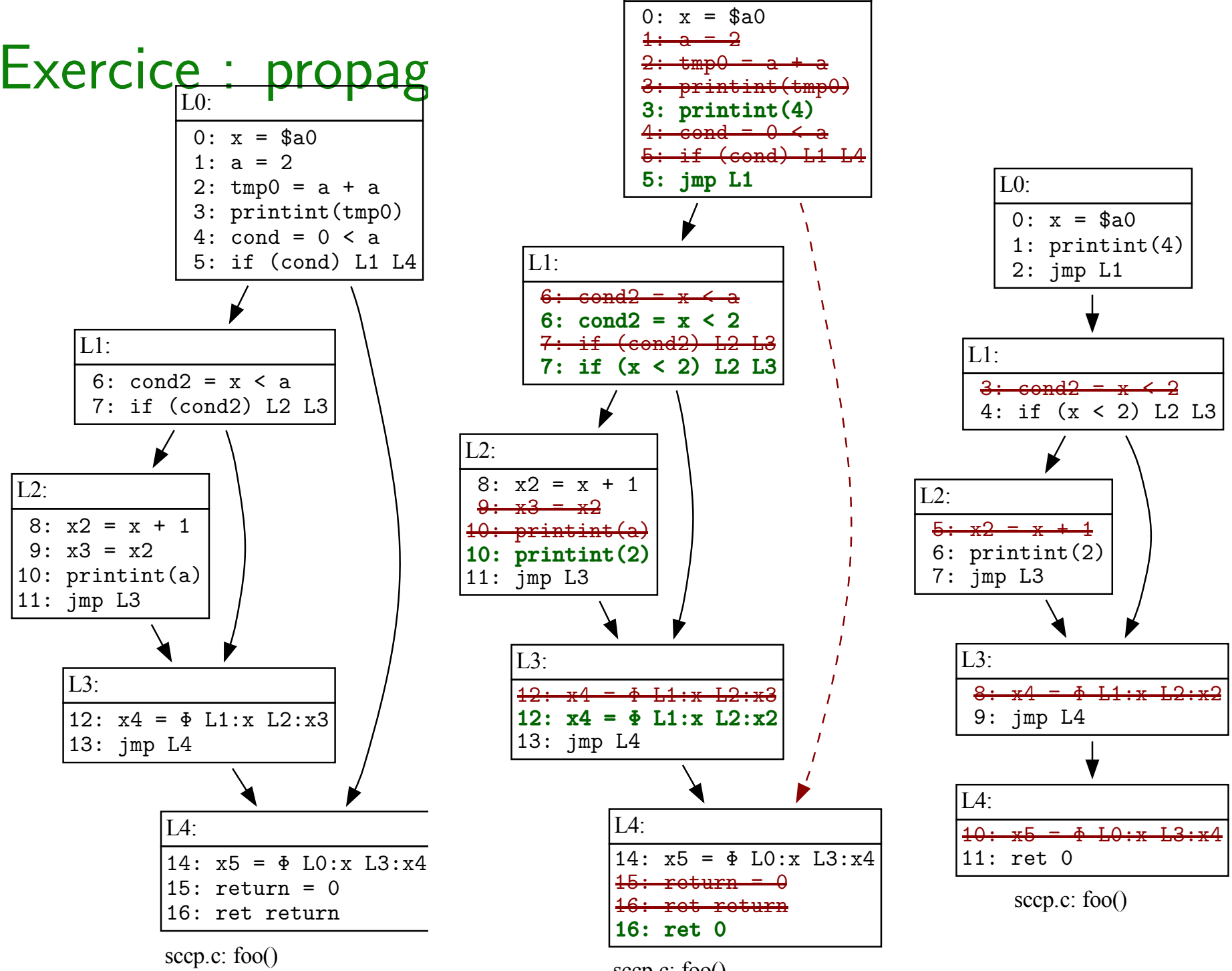


sccp.c: foo()

Exercice : propag



Exercice : propag



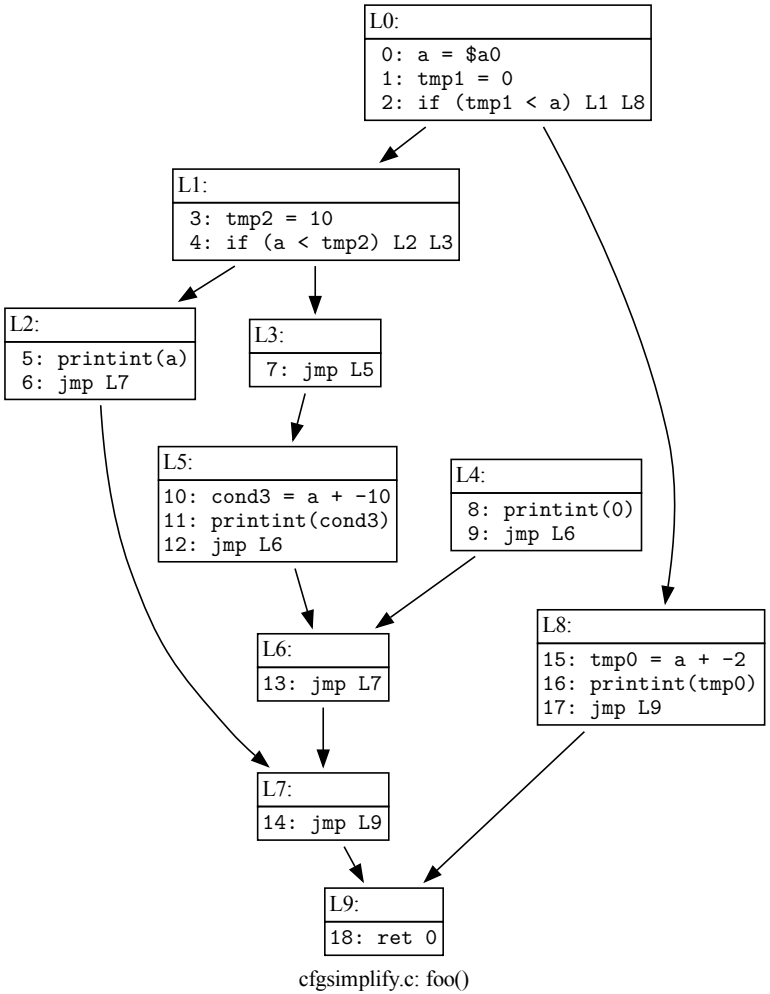
Simplifications du CFG

- Supprimer bloc sans prédécesseur
  - $\rightarrow$ Voir `CFGPrune.java`
- Fusionner bloc avec son successeur unique
  - Si ce successeur a un seul prédécesseur
  - $\rightarrow$ Voir `CFGMerge.java`
- Supprimer bloc qui contient juste un Jmp
  - $\rightarrow$ Voir `CFGFoldJump.java`
- Supprimer Φ des blocs avec un seul prédécesseur
  - $\rightarrow$ Voir `CFGPrune.java` aussi

Note: Pénible avec SSA, car il faut maintenir les Φ

Exercice : Simplifier le CFG

original

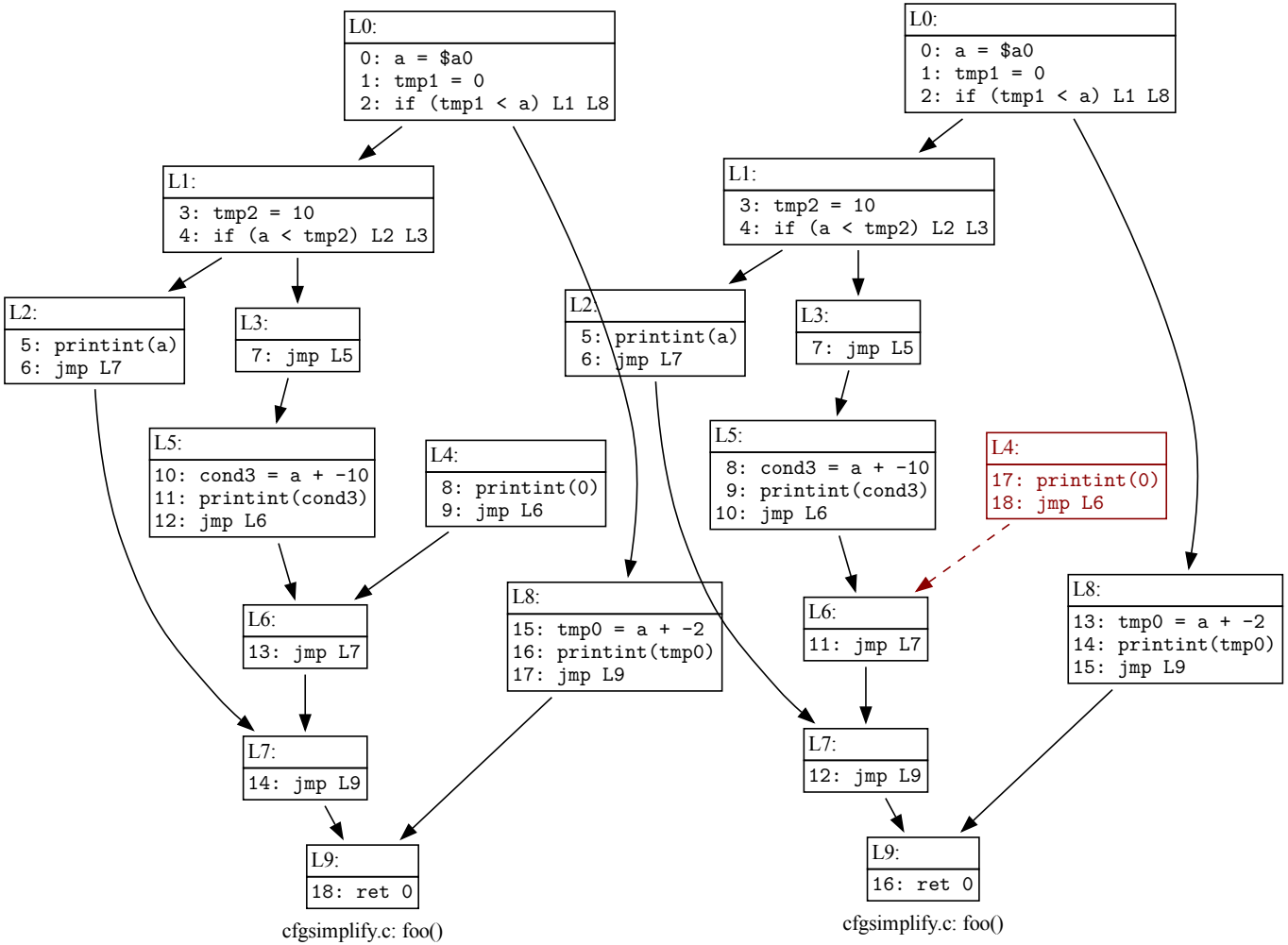


cfgsimplify.c: foo()

prune

Exercice : Simplifier le CFG

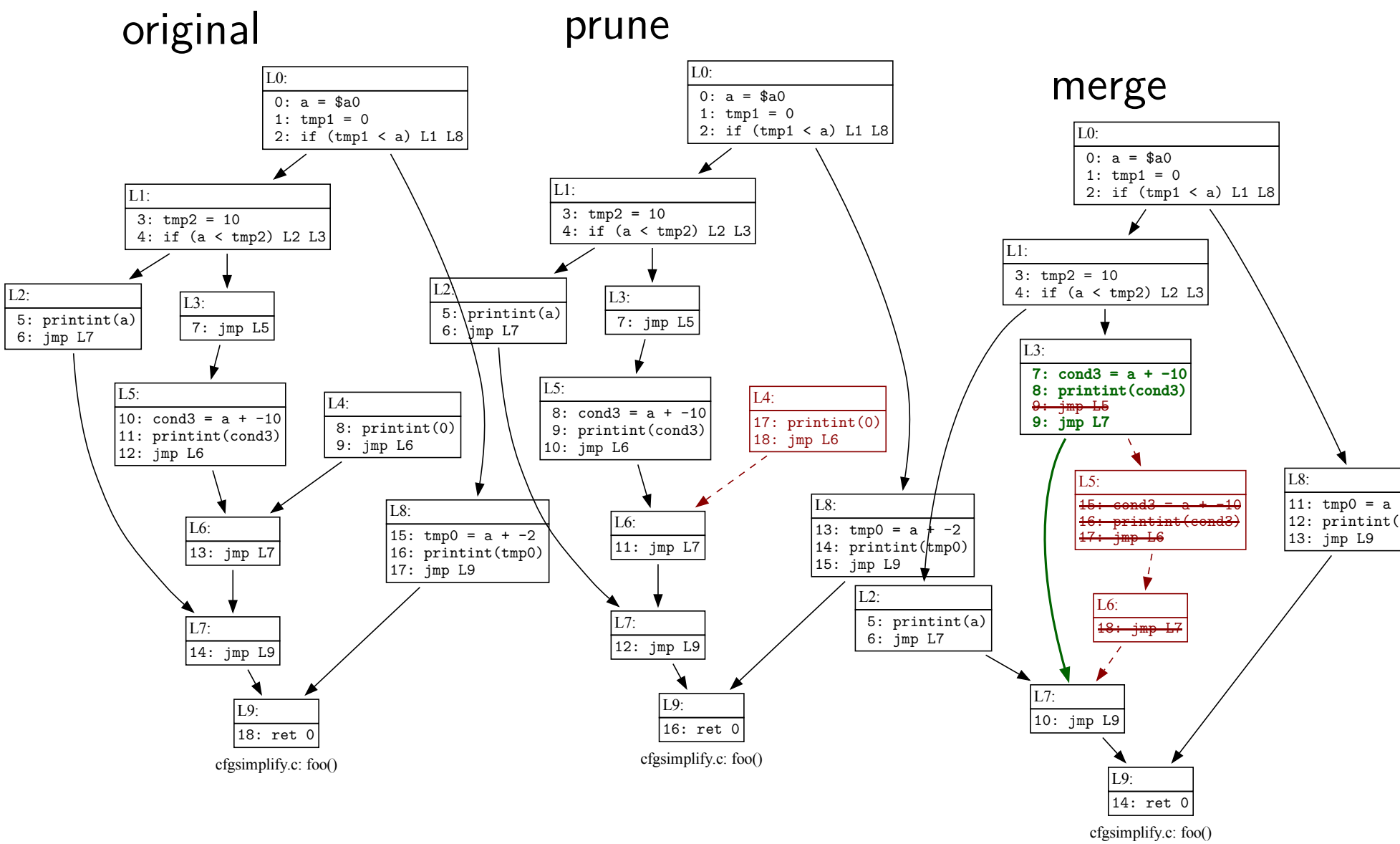
original



prune

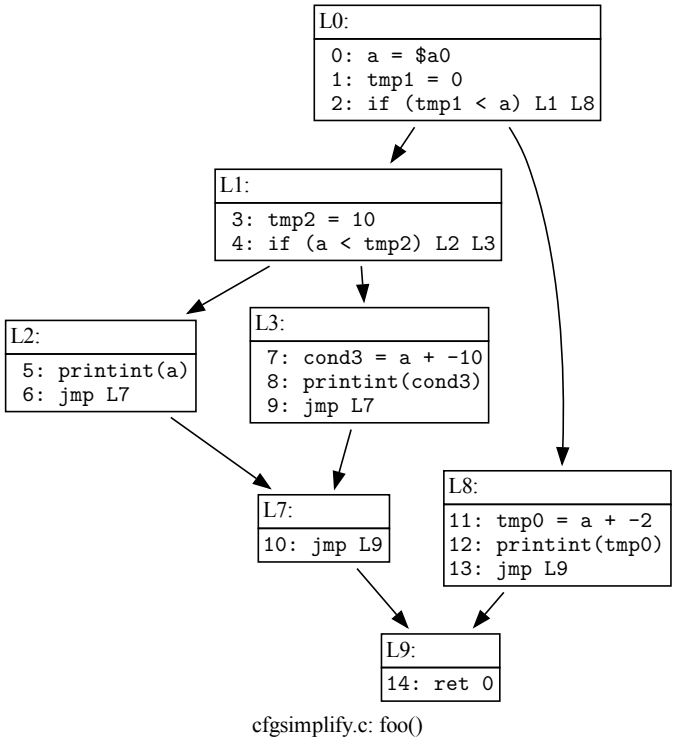
merge

Exercice : Simplifier le CFG



Exercice : Simplifier le CFG

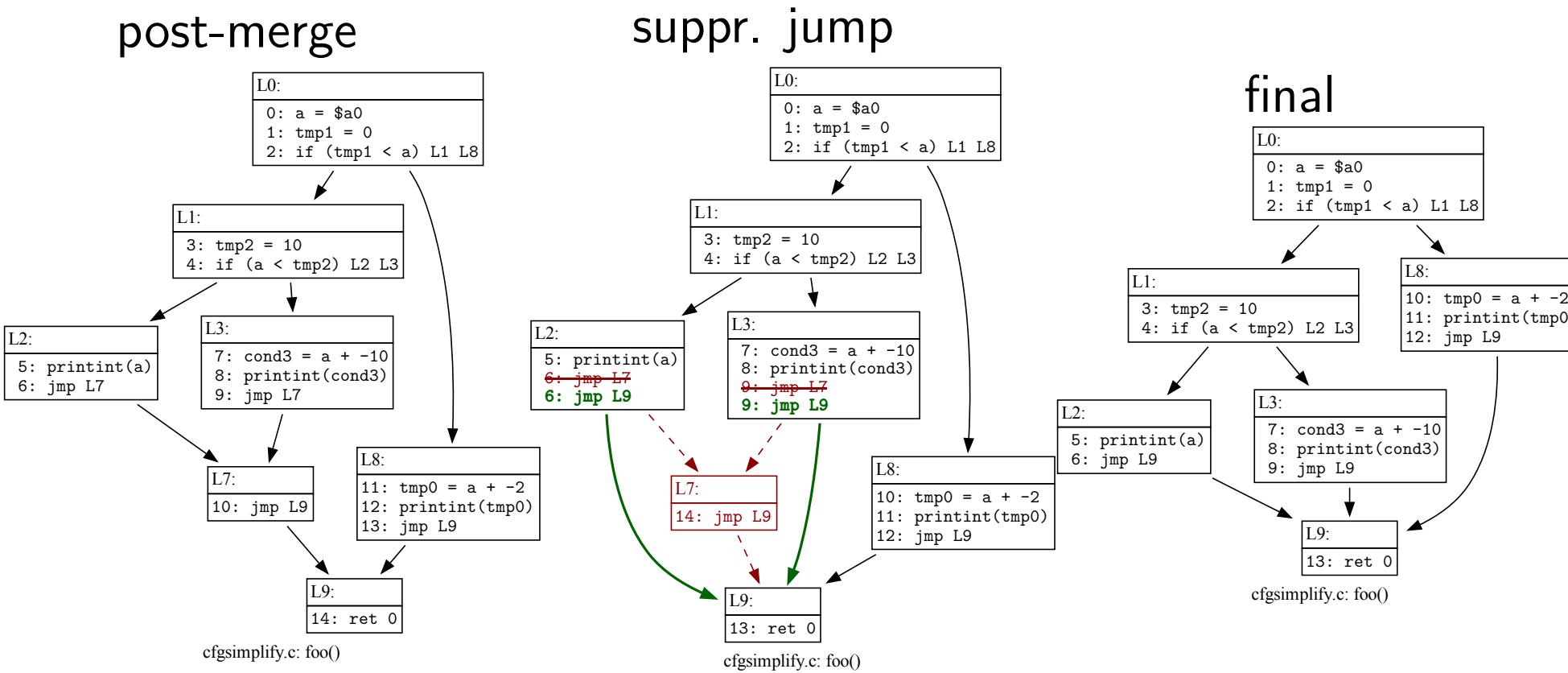
post-merge



cfgsimplify.c: foo()

suppr. jump

Exercice : Simplifier le CFG



Common subexpression elimination, CSE

Expressions disponibles (*available expressions*)

- Analyse dataflow
- Détermine les expressions déjà calculées
- Déjà vu

Numérotation des valeurs (*value numbering*)

Griggs&Cooper&Simpson,1997

- Technique alternative
- S'intéresse aux valeurs plutôt qu'aux formes syntaxiques
- Basée sur SSA

Gain et pertes : CSE

Common subexpression elimination, CSE

Expressions disponibles (*available expressions*)

- Analyse dataflow
- Détermine les expressions déjà calculées
- Déjà vu

Numérotation des valeurs (*value numbering*)

Griggs&Cooper&Simpson,1997

- Technique alternative
- S'intéresse aux valeurs plutôt qu'aux formes syntaxiques
- Basée sur SSA

Gain et pertes : CSE

- Réduit la taille du code
- Réduit le temps d'exécution
- Un registre à faire vivre plus longtemps

Élimination de code mort

Code mort (on y revient)

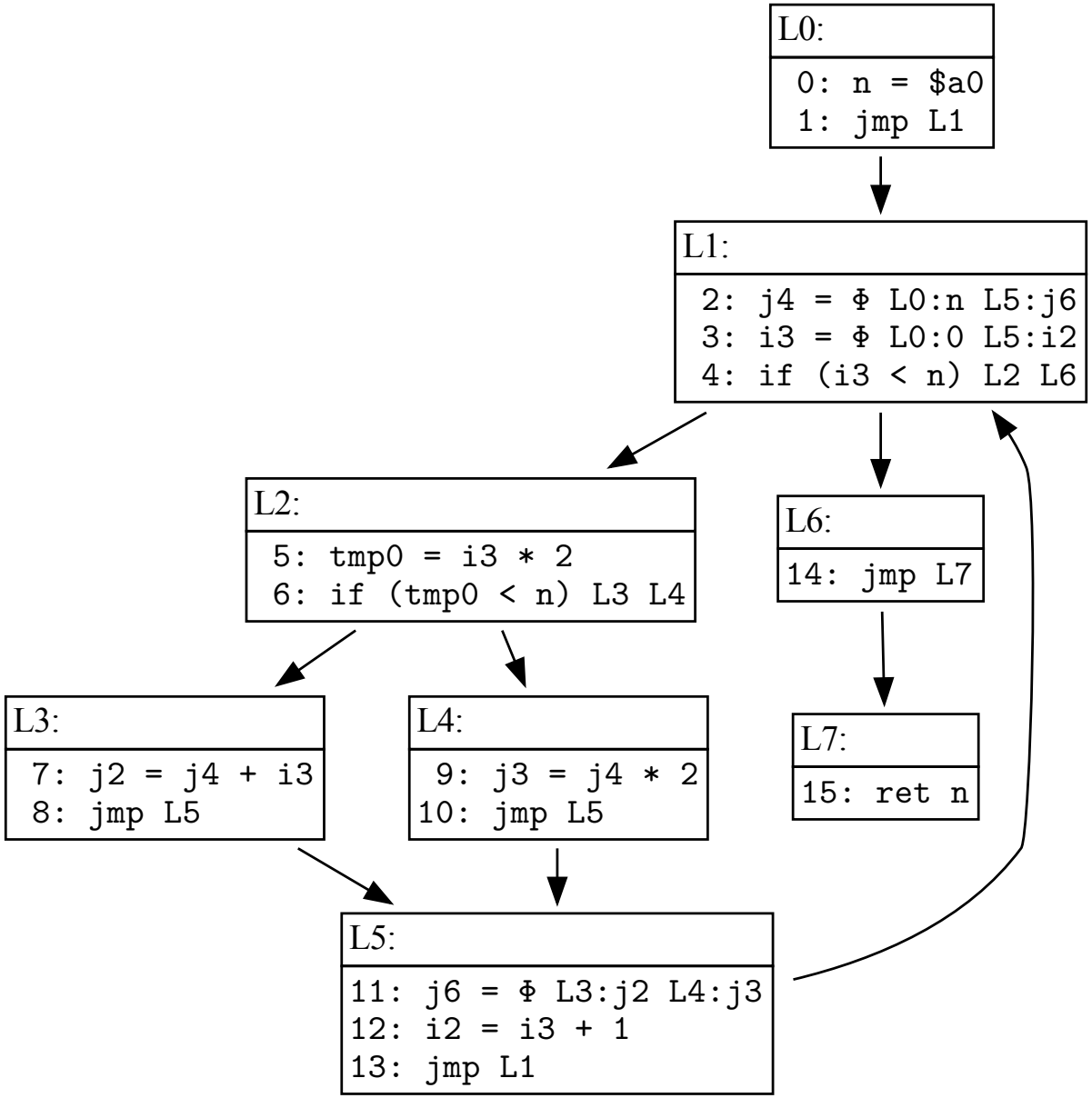
- Registre inutilisé en opérande
- Supprimer les opérations **sans effet de bord** (pures) qui le calcule
  - `mov` ? arithmétique ? `return` ? `jmp` ? `if` ? `call` ?

Élimination de code mort

Code mort (on y revient)

- Registre inutilisé en opérande
- Supprimer les opérations **sans effet de bord** (pures) qui le calcule
  - `mov` ? arithmétique ? `return` ? `jmp` ? `if` ? `call` ?
  - `mov` et arithmétique : pures
  - `return` et `call` : effet de bord
  - `jmp` et `if` : gardent le CFG cohérent
- Et répéter jusqu'à stabilité
- $\rightarrow$ `DeadCodeElimination.java`

Exercice : éliminer le code mort



dce.c: foo()

Un peu simpliste ?

- Un registre peut être utilisé mais pas très utile quand même
- Dépendances entre registres inutiles
  - `x = y+5; y = x-5;` (où `x` et `y` pas utilisés par ailleurs)
- Boucles et conditions inutiles
  - `while(i<10) { i=i+1; }` (où `i` pas utilisé par ailleurs)

Dépendance de contrôle (*control dependency*)

Définition

- Un bloc n a une **dépendance de contrôle** envers le bloc f si
- Il existe un successeur s de f, pour qui tous **les chemins** vers la sortie n_ω passent par n
- Et, il **existe un chemin** de f vers n_ω qui ne passe pas par n
- $\rightarrow n$ contrôle l'exécutabilité de f

... ça nous rappelle quelque chose, non?

Dépendance de contrôle (*control dependency*)

Définition

- Un bloc n a une **dépendance de contrôle** envers le bloc f si
- Il existe un successeur s de f, pour qui tous **les chemins** vers la sortie n_ω passent par n
- Et, il **existe un chemin** de f vers n_ω qui ne passe pas par n
- $\rightarrow n$ contrôle l'exécutabilité de f

... ça nous rappelle quelque chose, non?

Post-dominance

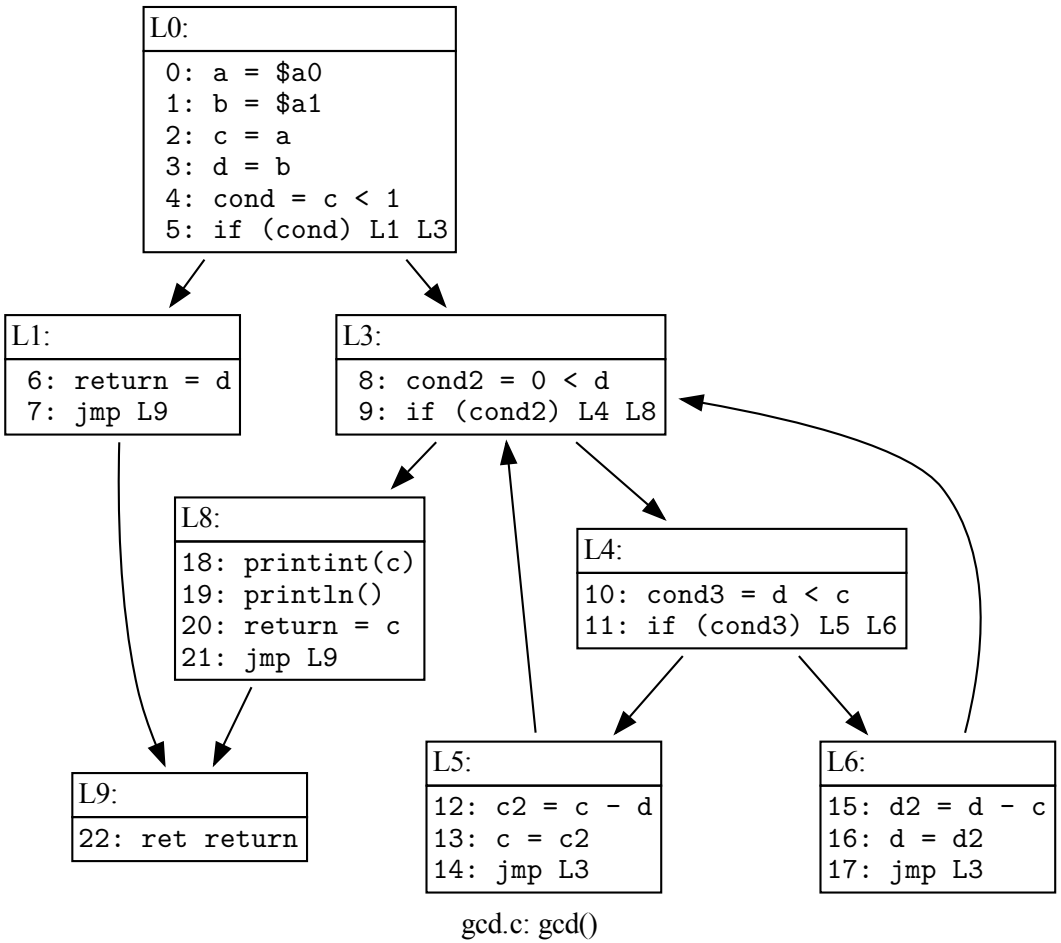
Notion duale de la dominance

- Dans un CFG (sortie n_ω), un nœud n_1 **post-domine** un nœud n_2
- (ou n_1 est un **post-dominateur** de n_2)
- Si : n_1 appartient à **tous** les chemins de n_2 à n_ω.

En fait, c'est pareil mais à l'envers

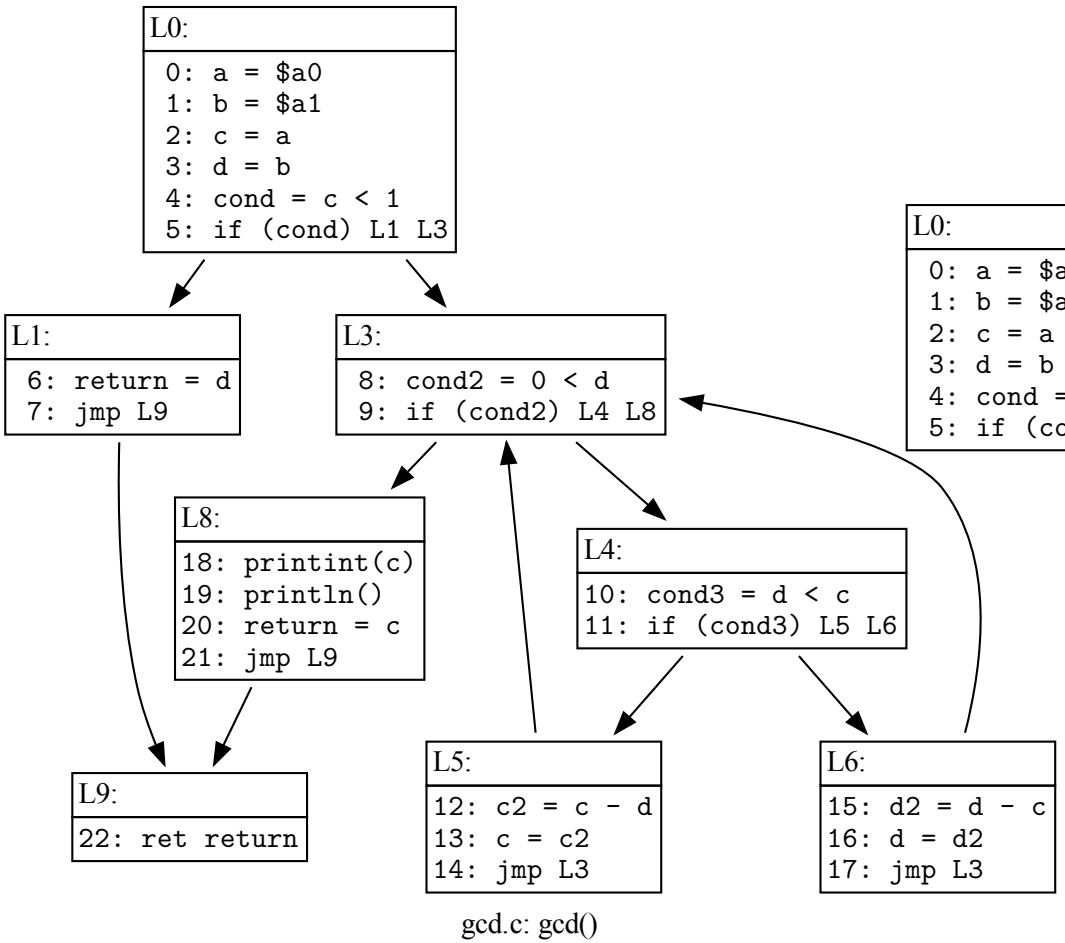
- On travaille sur le CFG dont on a inversé les arcs
- Ou on fait le *dataflow* en mode **arrière**
- Attention: inverser l'arbre de dominance ne suffit pas
- relation d'ordre (pdom), version stricte (spdom), version arbre (ipdom)
- et une frontière de post-dominance (fpd) !
- $\rightarrow$ `PostDominance.java`

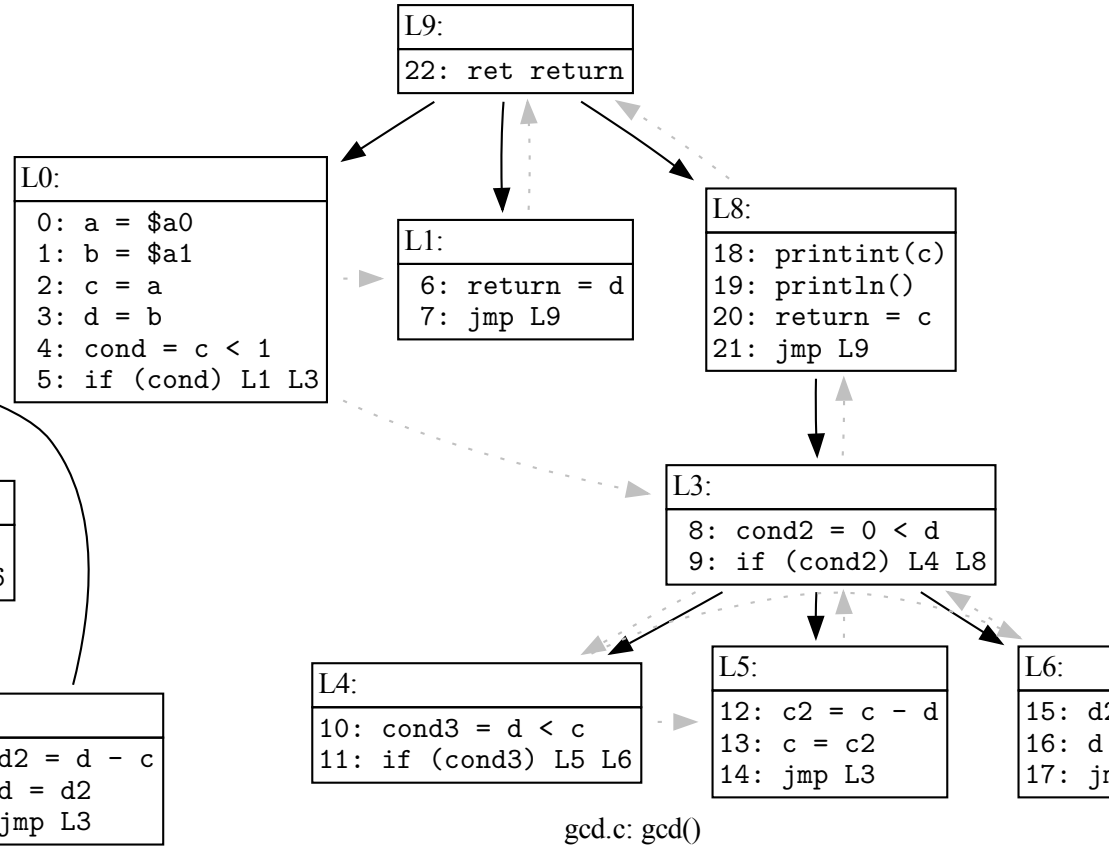
Exercice : Calculer la post-dominance



gcd.c: gcd()

Exercice : Calculer la post-dominance





Frontière de post-dominance

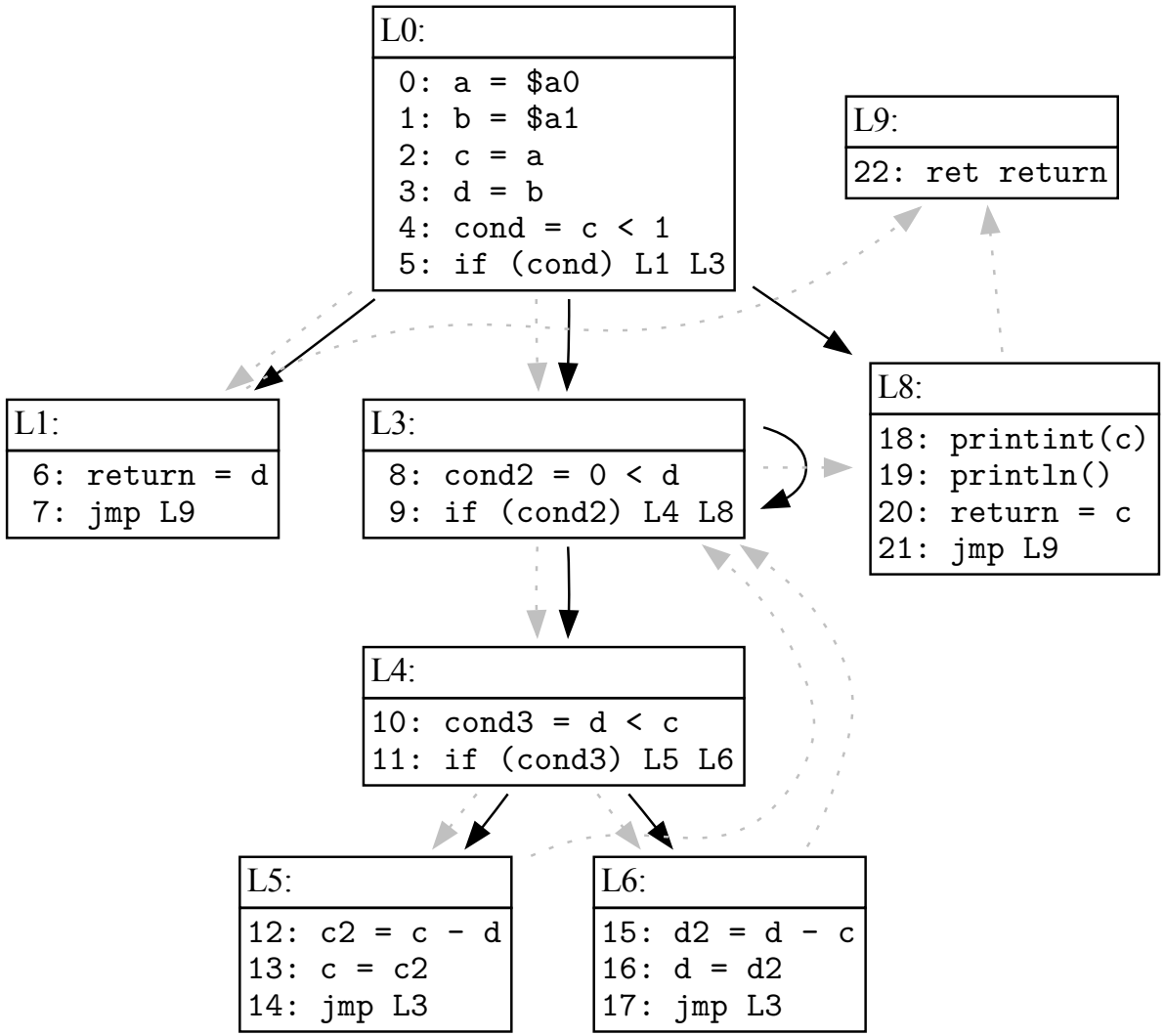
- Un bloc n a une **dépendance de contrôle** envers le bloc f si
- f appartient à la **frontière de post-dominance** de n
- $\rightarrow$ Il existe s, $f \rightarrow s$ tel que n pdom s et $\neg(n$ spdom $f)$

Exercice : postes frontières

- Diapo précédente

Exercice : postes frontières

L0:[] L1:[L0] L3:[L0, L3] L4:[L3] L5:[L4] L6:[L4] L8:[L0] L9:[]



gcd.c: gcd()

Élimination agressive de code mort (ADCE)

Les registres sont **inutiles** (morts), sauf preuve du contraire

Sont **utiles**

- Les opérandes et les blocs des opérations utiles
- `call`, `ret`
- Les opérations qui calculent des registres **utiles**
- Les `if` de frontière de post-domination d'un bloc utile
  - Les aiguillages doivent fonctionner...
- Les blocs entrants d'un Φ utile
  - Car les `mov` du Φ *sont sur* les arcs entrants

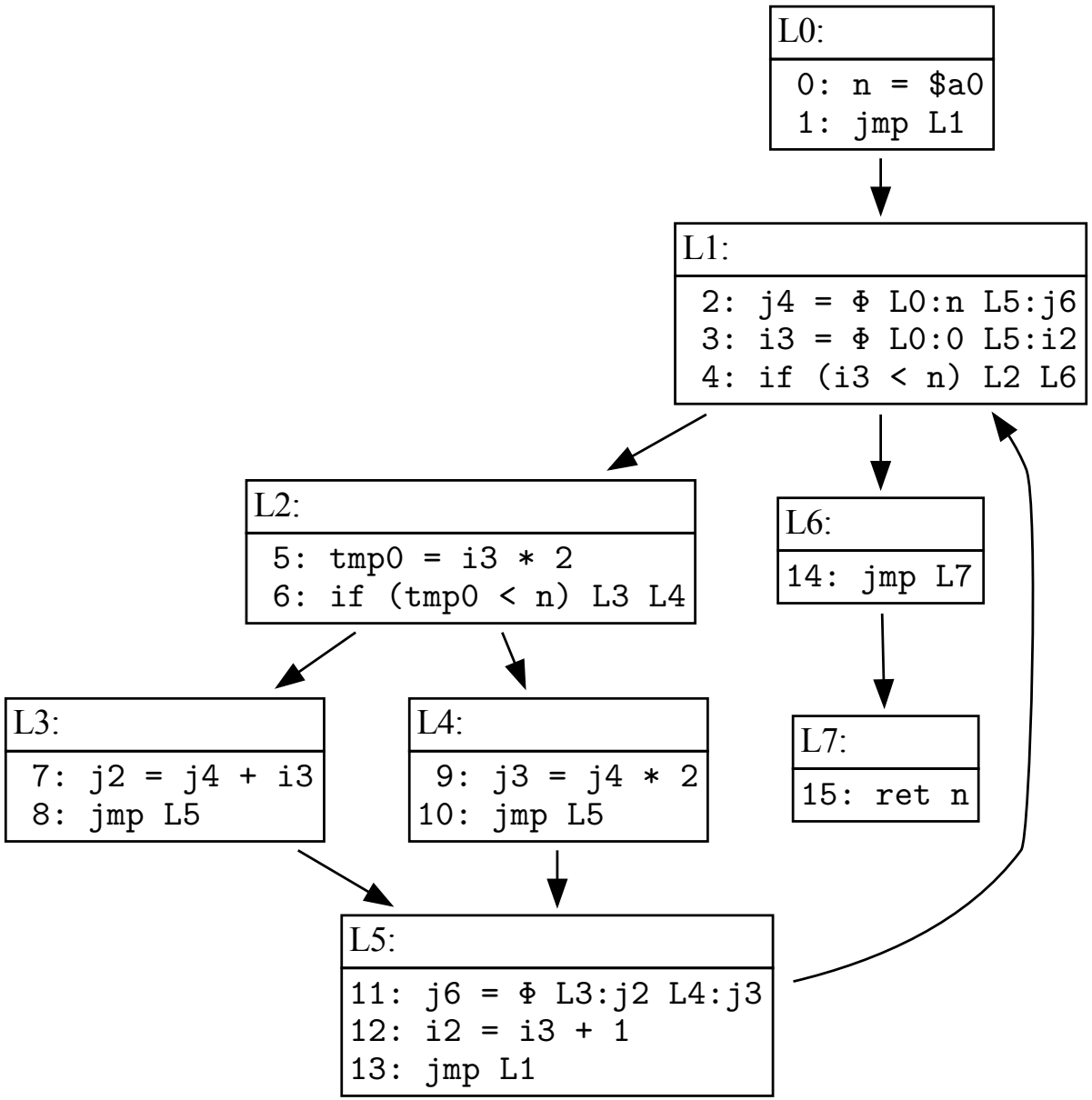
Supprimer l'inutile

- Tout ce qui n'est pas marqué utile
- On garde quand même les `jmp`
  - La simplification du CFG s'en occupera
- Un `if` inutile est remplacé par un `jmp`

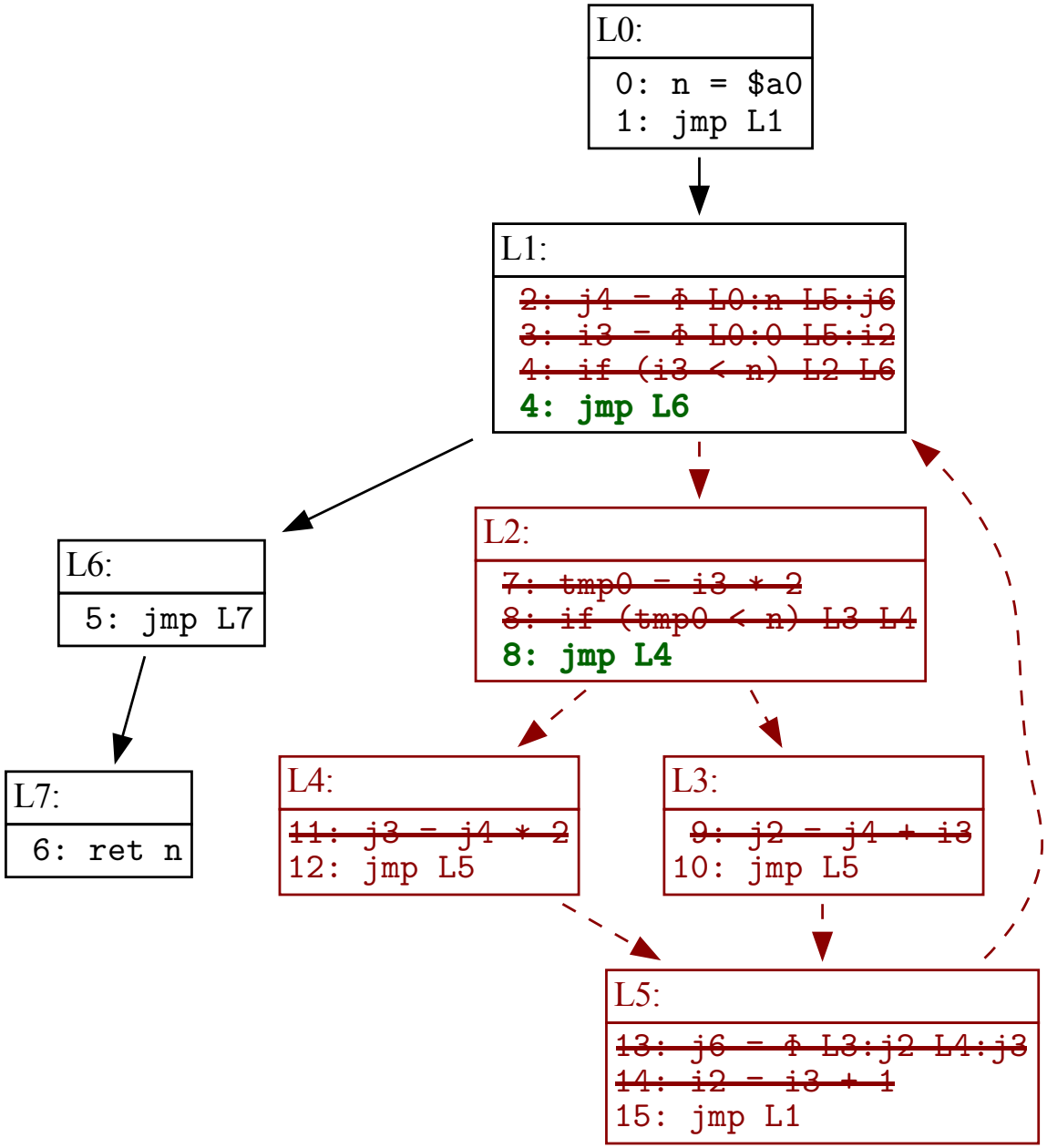
Implémentation

- Une simple liste de travail pour ramasser les trucs utiles
- Une itération pour éliminer
- $\rightarrow$ `AggressiveDeadCodeElimination.java`

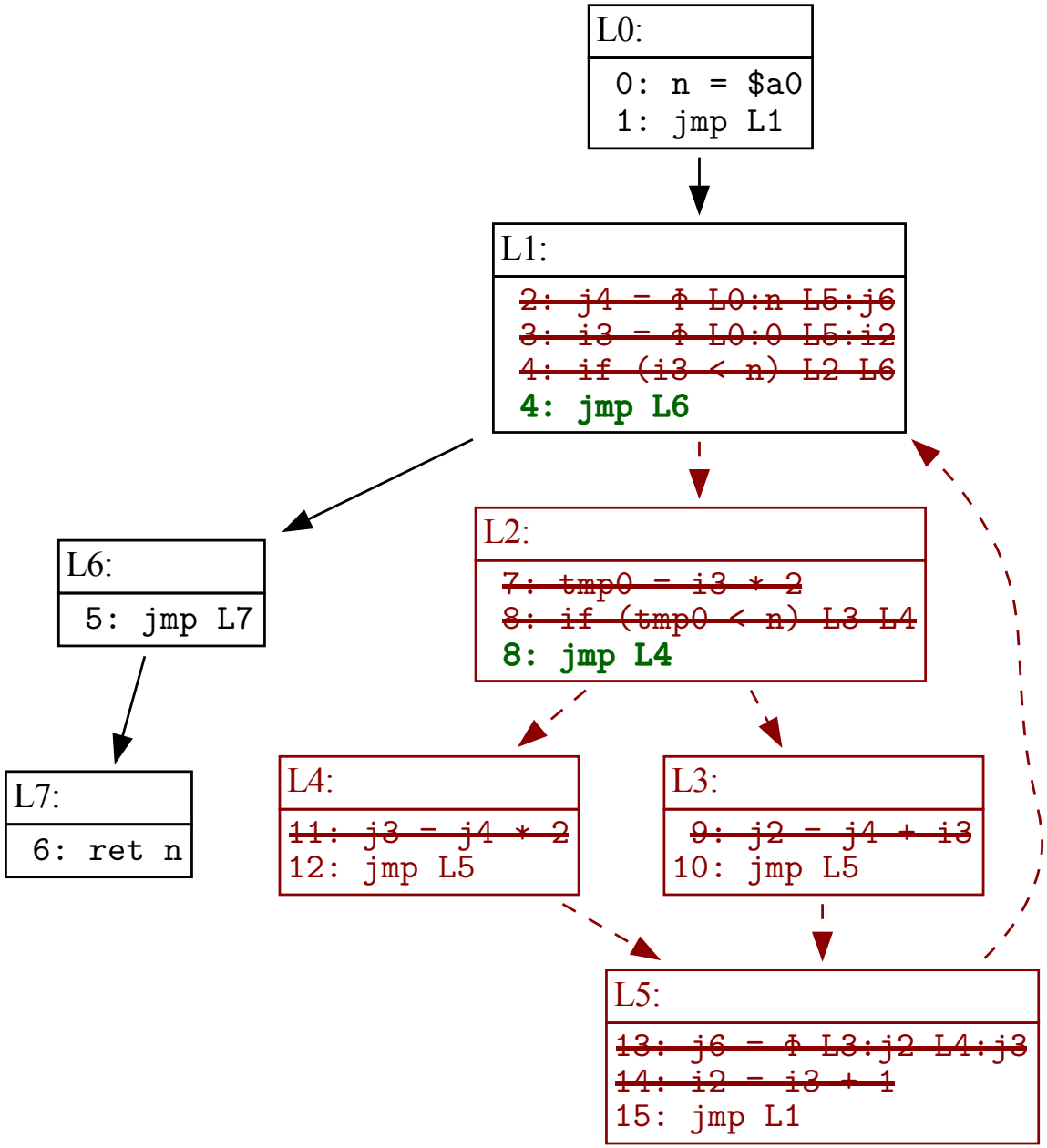
Exerice : ADCE



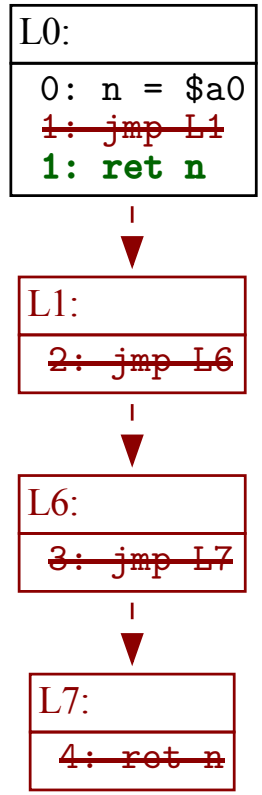
dce.c: foo()


L0:
0: n = $a0
1: jmp L1
L1:
2: j4 = ϕ L0:n L5:j6
3: i3 = ϕ L0:0 L5:i2
4: if (i3 < n) L2 L6
4: jmp L6
L6:
5: jmp L7
L2:
7: tmp0 = i3 * 2
8: if (tmp0 < n) L3 L4
8: jmp L4
L7:
6: ret n
L4:
11: j3 = j4 * 2
12: jmp L5
L3:
9: j2 = j4 + i3
10: jmp L5
L5:
13: j6 = ϕ L3:j2 L4:j3
14: i2 = i3 + 1
15: jmp L1

dce.c: foo()



dce.c: foo()



dce.c: foo()

Remontée de code (*code hoisting*)

Very busy expression (déjà vu)

- Quelles opérations seront nécessairement faites plus tard ?
- ... ou vraisemblablement ? (on prend un chance!)
- $\rightarrow$ Les remonter

Invariants de boucle

- Sortie du code des boucles
- Une autre fois...

Gains et pertes

Remontée de code (*code hoisting*)

Very busy expression (déjà vu)

- Quelles opérations seront nécessairement faites plus tard ?
- ... ou vraisemblablement ? (on prend un chance!)
- $\rightarrow$ Les remonter

Invariants de boucle

- Sortie du code des boucles
- Une autre fois...

Gains et pertes

- Réduit la taille du code
- Réduit éventuellement le temps d'exécution
- Les opérandes deviennent peut-être inutiles plus tôt
- Un registre à faire vivre plus longtemps

Rematérialisation (*rematerialization*, *remat*)

- Contre-mesure !
- Recalculer une valeur plutôt que la stoker dans un registre

Pourquoi ?

Rematérialisation (*rematerialization*, *remat*)

- Contre-mesure !
- Recalculer une valeur plutôt que la stoker dans un registre

Pourquoi ?

- Un recalcul est moins cher qu'un registre *spillé*

Contraintes

- Calculs : les opérandes doivent être encore disponibles
- Les initialisations à un immédiat sont de bons candidats
- Doit être intégré avec l'allocation de registres

Code sinking

- Descendre une instruction à un bloc successeur
- Facile en SSA !

Gains

Code sinking

- Descendre une instruction à un bloc successeur
- Facile en SSA !

Gains

- Un registre à faire survivre moins longtemps
- L'instruction sera peut-être même pas exécutée

Pertes

- Augmentation de la durée de vie des opérandes
- Déplacement par erreur dans un boucle
  - Calcul exécuté plusieurs fois pour rien
- Pression sur le parallélisme processeur
  - Définition juste avant les utilisations, pas forcément optimal

Ordonnancement des instructions (*instruction scheduling*)

Objectif 1 : Réduire la pression sur les registres

Objectif 2 : Profiter du parallélisme d'instructions des processeurs modernes : **pipelines**

- $\rightarrow$ Prévenir les **bulles** (*bubble* ou *pipeline stall*)
- $\rightarrow$ Réordonner les instructions

Contraintes

- Besoin de connaître les détails micro-architecturaux (très dépendant de la plateforme)
  - Souvent par rétro-ingénierie : on mesure en boite-noire et on construit des modèles plus ou moins représentatifs
  - `-march` et `-mtune`
- À faire vers la fin du processus de compilation

Peephole (à la lorgnette? au judas?)

McKeeman,1965

Cherche et substitue des patterns

- Localement
- Sur un petit nombre d'instructions
- Très nombreuses et spécifiques (plus de 1000 chez LLVM)

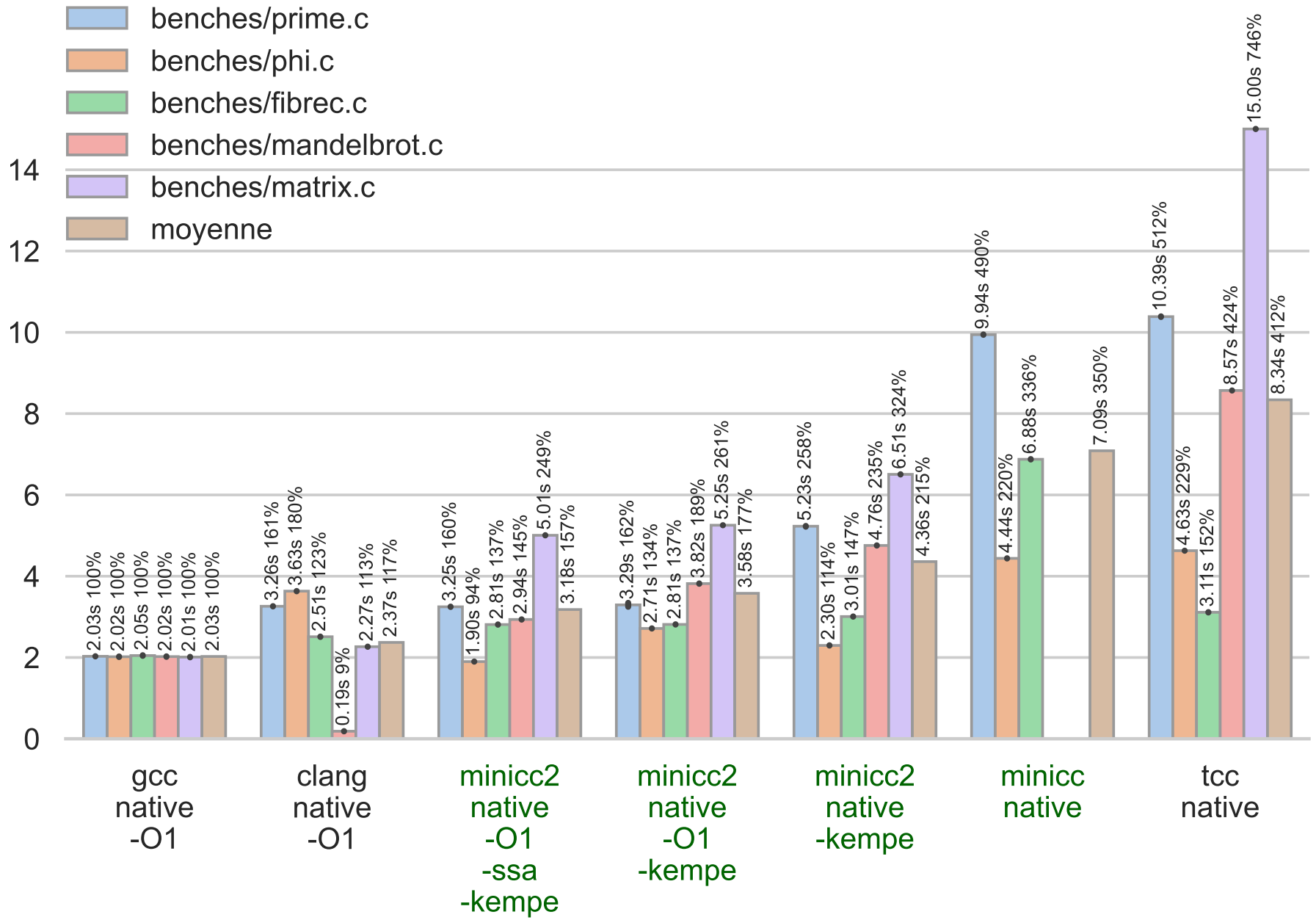
Exemples indépendant de la plateforme

- *Constant folding* (`x=40+2` $\rightarrow$ `x=42`)
- *Strength reduction* (`x=y*8` $\rightarrow$ `x=y<<3`)

Exemples Dépendant de la plateforme

- Très utile en CISC : beaucoup d'instructions qui font plusieurs trucs d'un coup
- `a=a+1` $\rightarrow$ `INC a`
- `b=a*8;d=b+c;e=d+42` $\rightarrow$ `LEA e, [c + a*8 + 42]`

MiniCC2 -O1


benches/prime.c
benches/phi.c
benches/fibrec.c
benches/mandelbrot.c
benches/matrix.c
moyenne
0
2
4
6
8
10
12
14
2.03s 100%
2.02s 100%
2.05s 100%
2.02s 100%
2.01s 100%
2.03s 100%
3.26s 161%
3.63s 180%
2.51s 123%
0.19s 9%
2.27s 113%
2.37s 117%
3.25s 160%
1.90s 94%
2.81s 137%
2.94s 145%
5.01s 249%
3.18s 157%
3.29s 162%
2.71s 134%
2.81s 137%
3.82s 189%
5.25s 261%
3.58s 177%
5.23s 258%
2.30s 114%
3.01s 147%
4.76s 235%
6.51s 324%
4.36s 215%
9.94s 490%
4.44s 220%
6.88s 336%
7.09s 350%
10.39s 512%
4.63s 229%
3.11s 152%
8.57s 424%
15.00s 746%
8.34s 412%
gcc native -O1
clang native -O1
minicc2 native -O1 -ssa -kempe
minicc2 native -O1 -kempe
minicc2 native -kempe
minicc native
tcc native

La prochaine fois

- Les boucles