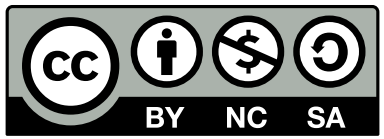
Chapitre 7: Représentation intermédiaire

INF7641 Compilation

Jean Privat

Université du Québec à Montréal

INF7641 Compilation
v251



MiniCC

- Une seule passe de compilation
- Une seule architecture cible
- Langage MiniC minimaliste
- Code généré très naïf
- Code Java très simple
- Des trucs *not yet implemented*

Comparons (en lignes de code selon cloc)

• minic seul (Java) :	628
• minic + `language_minic` (Java) :	12 281
• tcc0.9.27 (C) :	32 103
• jdk25 `jdk.compiler/` `hotspot/` (C++/Java) :	802 434
• llvm19 `clang/` `llvm/` (C/C++) :	4 232 888
• gcc14 `gcc/` (C/C++) :	4 961 248

Plan

Performance

Performance

- MiniC, MiniCC, tcc, gcc, clang (-O0, -O1)
- Natif, rars, qemu
- Pronostics ?

Performance

- MiniC, MiniCC, tcc, gcc, clang (-O0, -O1)
- Natif, rars, qemu
- Pronostics ? On a qu'à mesurer ?

Performance

- MiniC, MiniCC, tcc, gcc, clang (-O0, -O1)
- Natif, rars, qemu
- Pronostics ? On a qu'à mesurer ?

Avec quels programmes ?

- Manque à MiniC (entre autres): chaînes, mémoire, tableaux, structures, pointeurs, flottants, division, modulo, opérations binaires, test d'égalité, break, continue, goto, entrées, sorties (mieux que `printint`, `printbool` et `println`), graphisme...
- Que diable peut-on programmer avec un langage si pauvre ?

Performance

- MiniC, MiniCC, tcc, gcc, clang (-O0, -O1)
- Natif, rars, qemu
- Pronostics ? On a qu'à mesurer ?

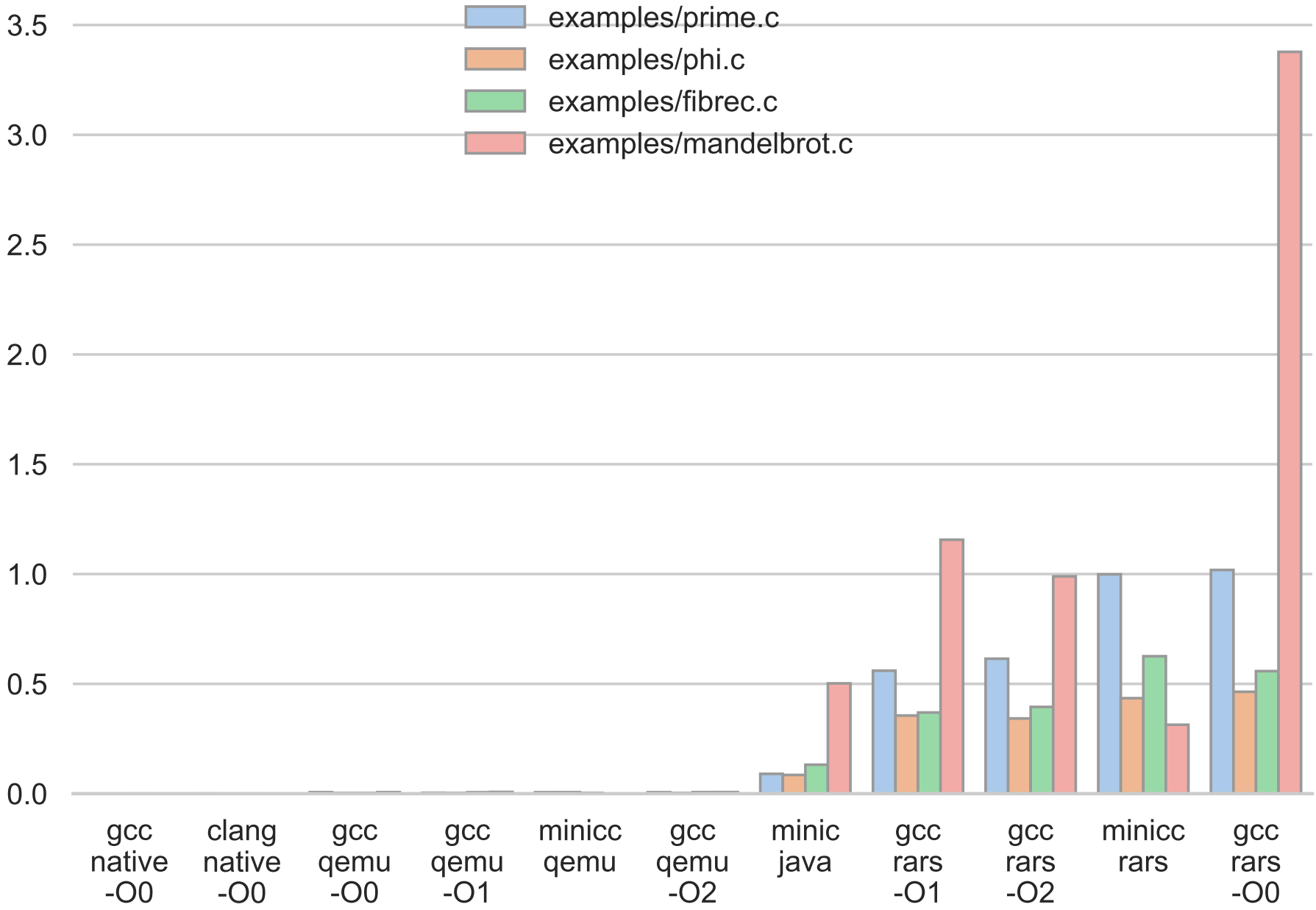
Avec quels programmes ?

- Manque à MiniC (entre autres): chaînes, mémoire, tableaux, structures, pointeurs, flottants, division, modulo, opérations binaires, test d'égalité, break, continue, goto, entrées, sorties (mieux que `printint`, `printbool` et `println`), graphisme...
- Que diable peut-on programmer avec un langage si pauvre ?

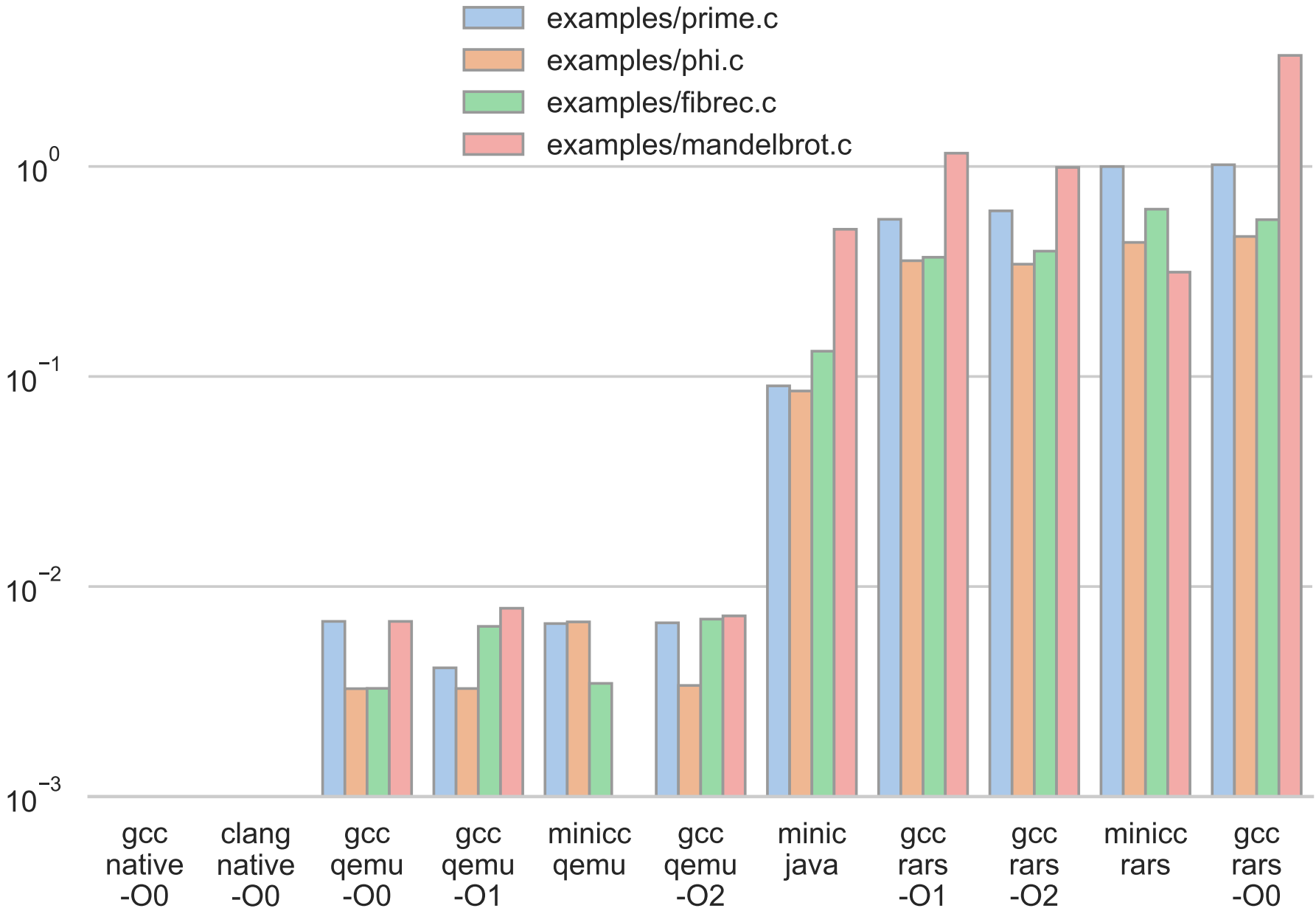
Notre banc d'essai

- `prime.c`: nombre de nombres premiers inférieurs à un nombre
- `phi.c`: indicatrice d'Euler (avec plus grand diviseur commun)
- `fibrec.c`: implémentation récursive de Fibonacci
- `mandelbrot.c`: approximation de l'ensemble de Mandelbrot (avec des entiers point fixes)

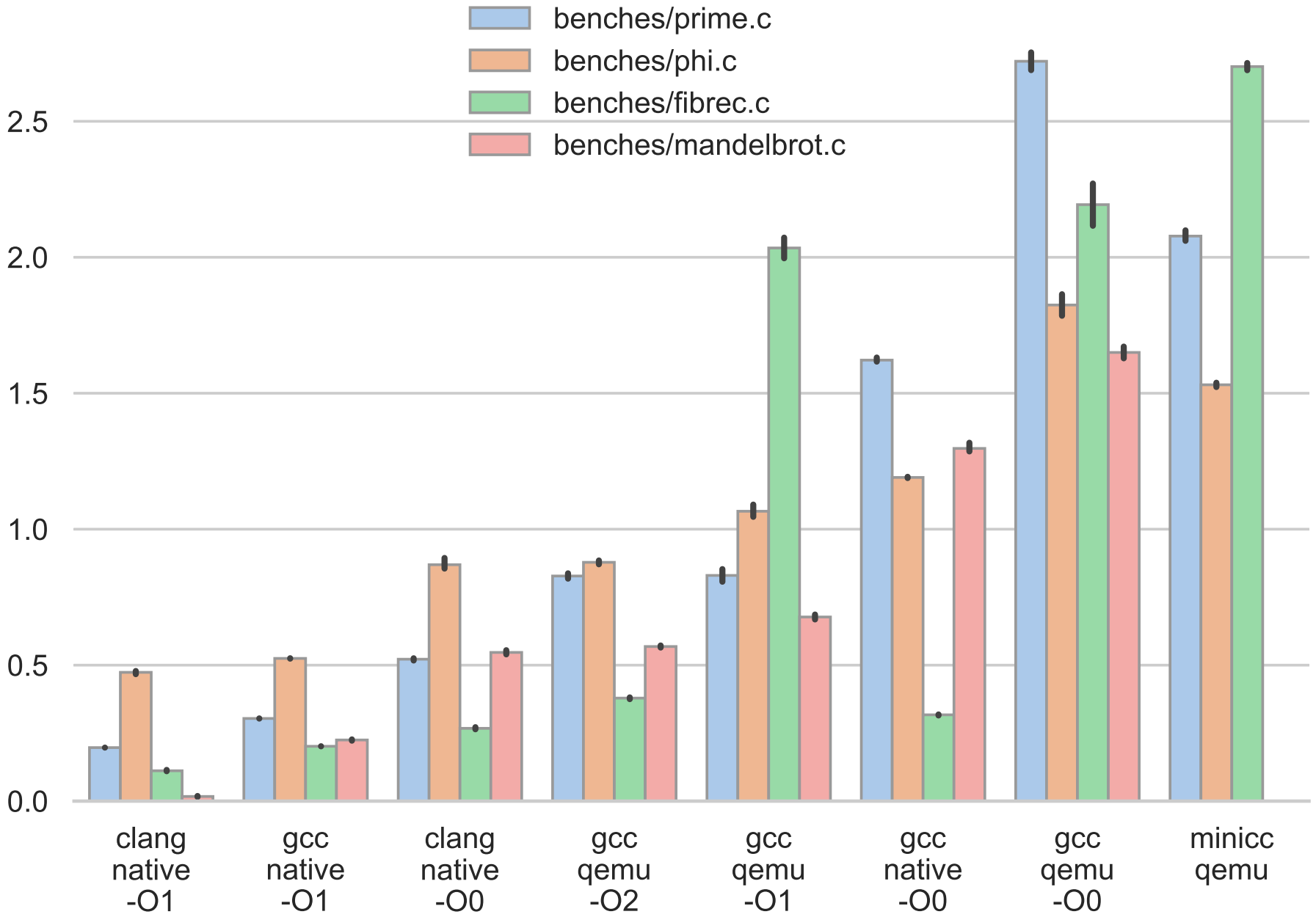
Temps en secondes (*usertime*) sur mon portable



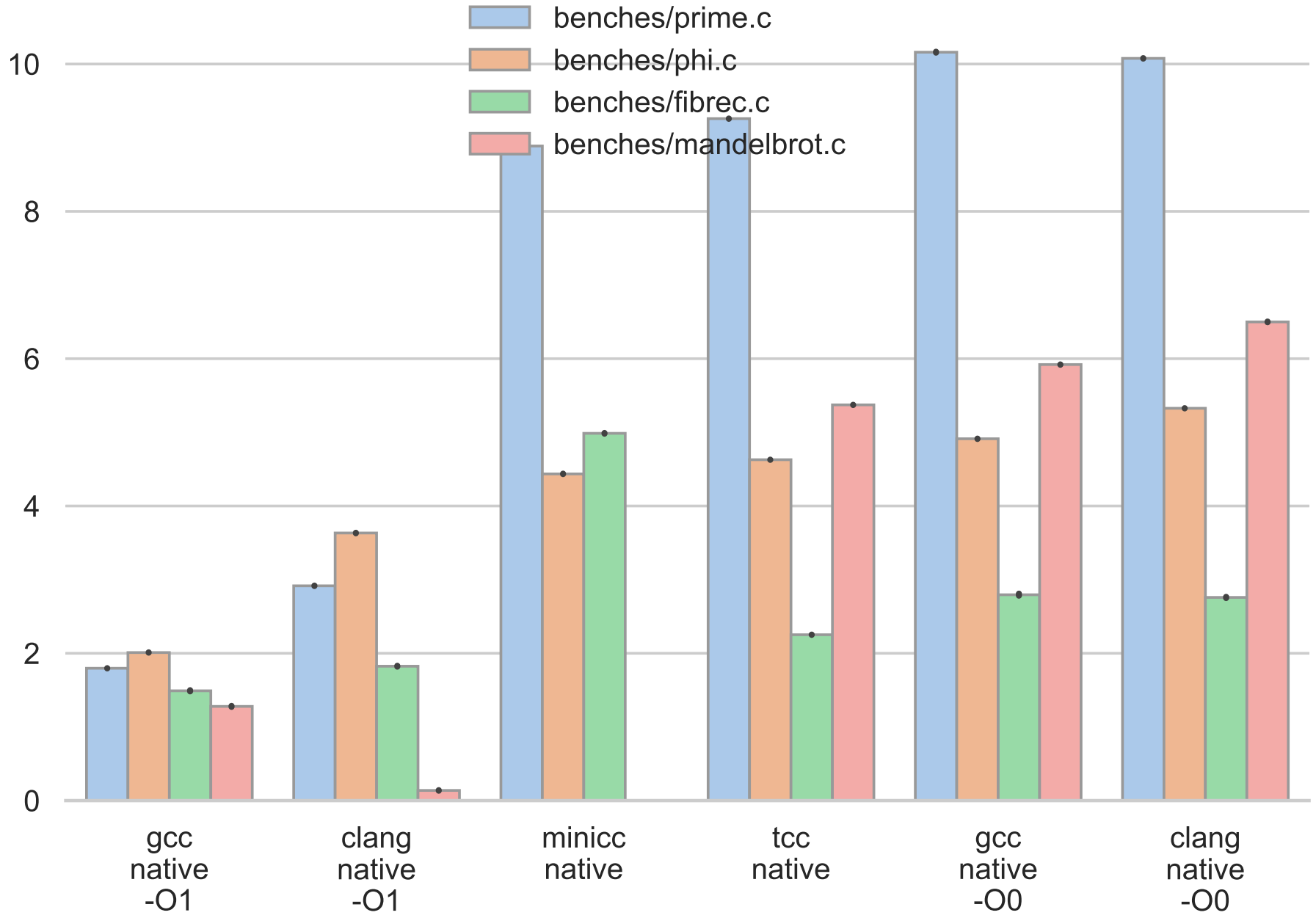
Échelle logarithmique



Comparer du comparable



RISC-V en métal (sur le starfive du lab)



C'est quoi mesurer la performance ?

Approche facile

- Exécuter un programme
- Compter les ressources consommées
  par exemple le temps

Quoi/comment mesurer ?

- Temps réel (horloge murale) → `time -f%E`
- Temps utilisateur (CPU consommé) → `time -f%U`
- Nombre de cycles processeurs utilisés → `linux perf`
- Nombre d'instructions machine exécutées → `linux perf`
- Profilage → `gcc -pg` et `gprof`
- Simulation → `valgrind` (support RISC-V en cours)

Mesurer c'est se tromper

Quoi tester ?

- Programmes de test utilisées
- Données/entrées de test utilisées
- $\rightarrow$ Solution: micro-benchmark + bancs de tests représentatifs

Environnement de test

- Architecture, micro-architecture processeur
- Carte-mère, RAM (qualité et quantité), configuration BIOS
- Version de l'OS
- Configuration de l'OS (dont `/proc` et `/sys`)
- Versions des bibliothèques

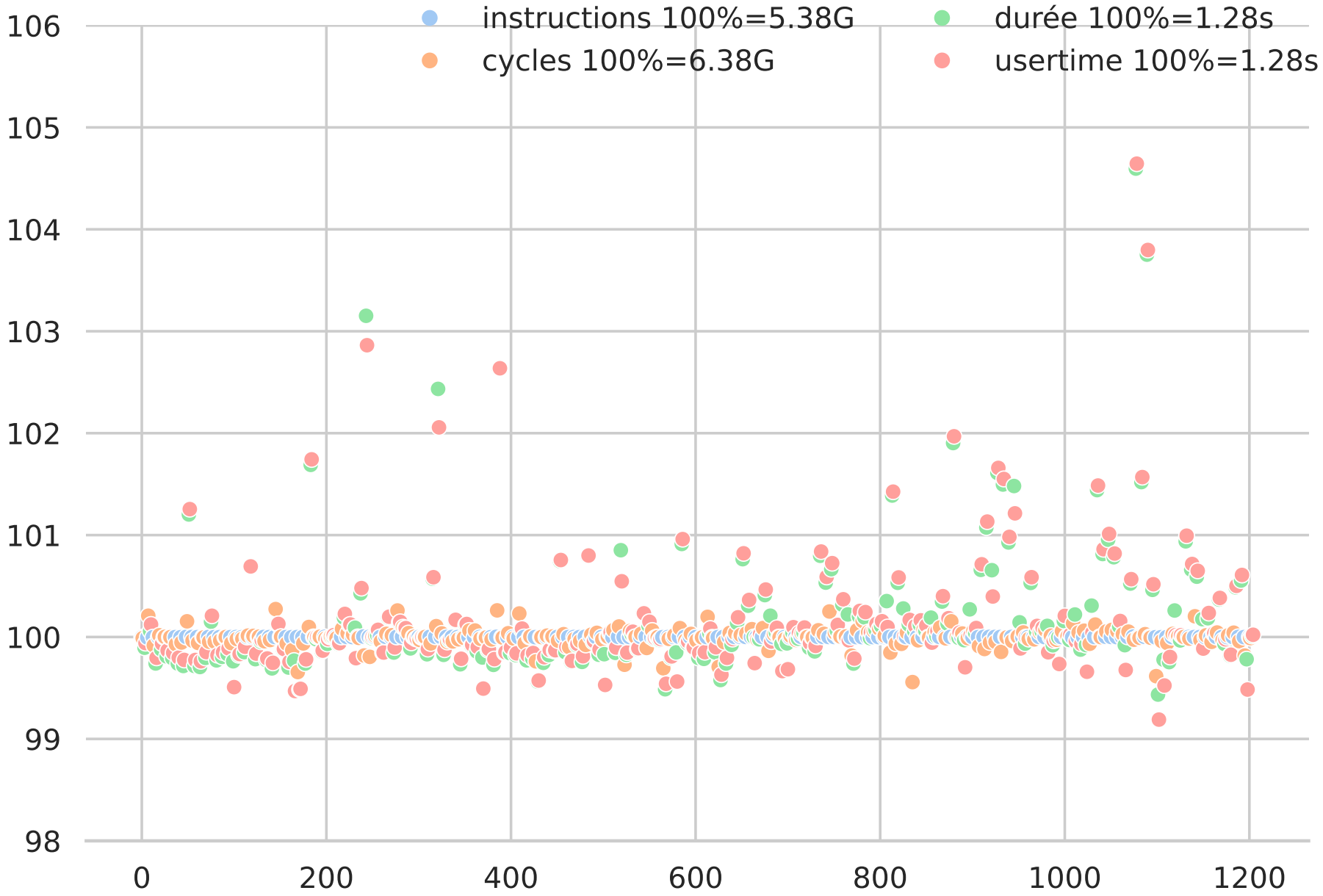
Bruit ?

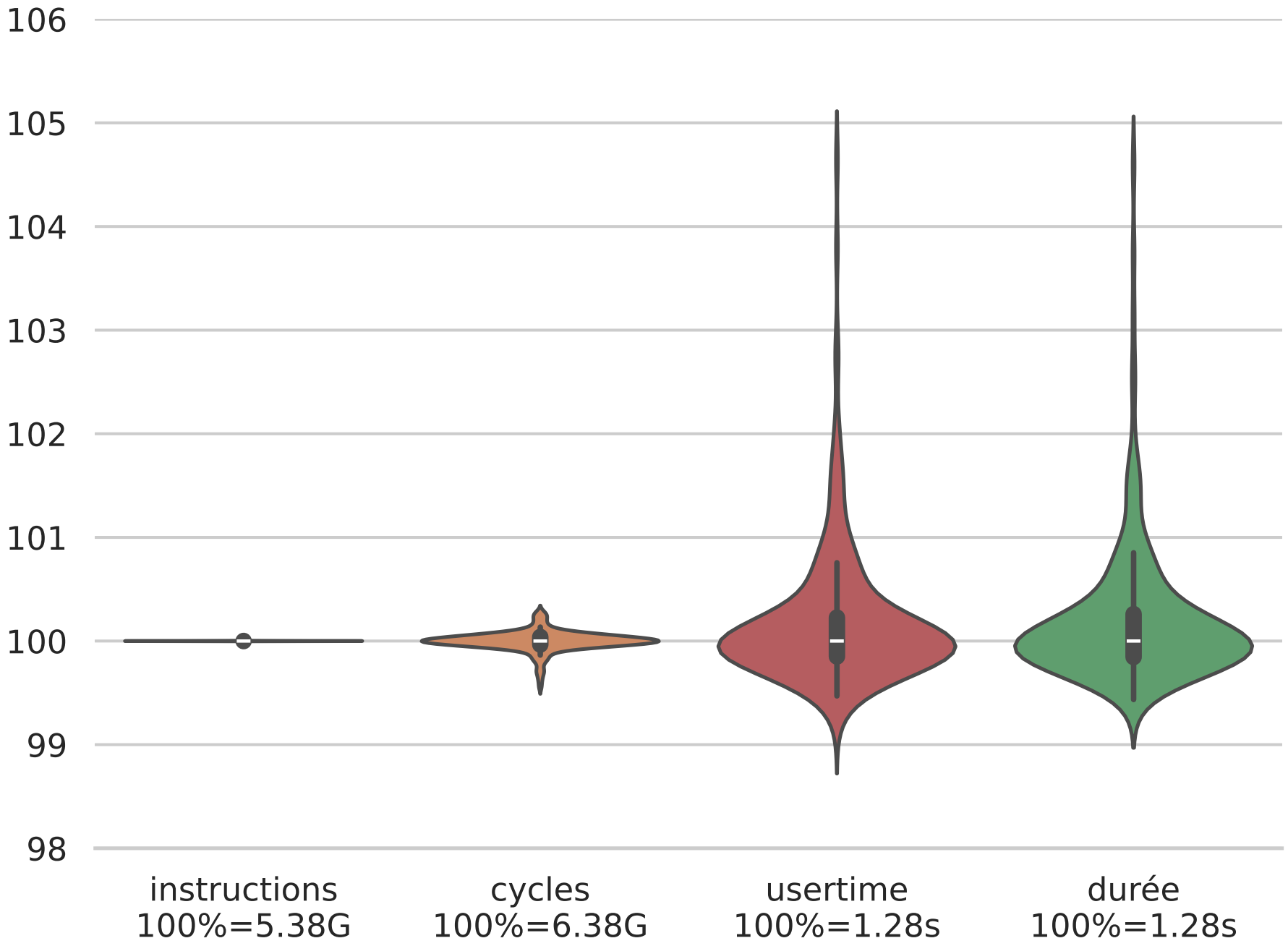
- OS, autres processus, autres VM, interruptions matérielles, *CPU throttling*
- Caches CPU: mémoire cache, TLB, prédicteur de branchement
- Coût du démarrage et de la terminaison

Solution ?

- Isoler au maximum ?
- Exécuter suffisamment longtemps ?
- Prendre le min, le max, la moyenne, autre ?

Bruit gcc -O0 benches/mandelbrot.c




106
105
104
103
102
101
100
99
98
instructions
100%=5.38G
cycles
100%=6.38G
usertime
100%=1.28s
durée
100%=1.28s

Qui mérite la performance ?

Qu'est-ce qu'on veut optimiser ?

Qui mérite la performance ?

Qu'est-ce qu'on veut optimiser ?

- Pour une plateforme spécifique ?
- Pour une micro-architecture spécifique ?
- Pour le cas moyen ? Pour le cas pire ?
  Comment les connaître ?
- Pour des données/entrées spécifiques ?
  Comment les connaître ?

Quelle est notre latitude ?

Respecter la sémantique du programme et langage

- Toujours produire un programme équivalent !
- Ça peut être subtil, le diable est dans les détails
- C'est mieux quand on a des algo prouvés
  (voire des compilateurs prouvés?)

Ressources limitées pour optimiser

- Temps et espace : de vrais programmes $\rightarrow$ passer l'échelle
- En particulier pour un JIT
- Temps de compilation de Linux $\approx$ 1h

Fleurs du tapis

- Une optimisation est-elle réellement bénéfique ?
- Ou est-ce simplement un effet de bord / dans le bruit ?
  - Réordonner les fonctions dans un binaire peut avoir un impact

Quelle est notre base théorique ?

Problèmes fondamentaux

- Souvent NP-difficiles, voire indécidables
- Problème de l'arrêt $\rightarrow$ Indécidable (Turing, 1936)
- Théorème de Rice, 1951 (corollaire B)
  $\rightarrow$ Toute propriété sémantique non triviale d'un programme est indécidable

Ce qu'on veut

- On veut idéalement du linéaire $\rightarrow$ Pour pouvoir passer l'échelle
- Se contenter d'heuristiques...
  - qui peuvent parfois dégénérer
    $\rightarrow$ l'optimisation fonctionne mal, voire est contre-productive
  - ou surprendre l'utilisateur
    $\rightarrow$ Un petit changement impacte grandement les optimisations

Compilation optimisante

Compilation optimisante

On veut générer du code machine plus efficace
... pour une certaine définition d'« efficace »

Approche globale

- Analyser le programme, et extraire toute la connaissance
- Puis générer de l'assembleur très performant d'un coup
- $\rightarrow$ Non, trop complexe !

Approche modulaire

- Des **passes** indépendantes qui analysent, transforment et optimisent le programme
- Itératif et incrémental
- Idéalement, faible couplage et forte cohésion des passes
- $\rightarrow$ Quelles passes (algorithmes) utiliser ? Dans quel ordre ?

Représentation des programmes

Choix de conception

- Des développeurs de compilateurs
- Quelles modélisations et structures de données utiliser ?
- Utilisation éventuelle de plusieurs représentations

Beaucoup de compromis. On la veut:

- Riche, simple, portable, malléable, maintenable et efficace
- Facile à générer (pour la construire)
- Facile à analyser (pour comprendre ce que fait le programme)
- Facile à transformer (pour compiler efficacement et sans bogues)
- Facile à traduire (vers du code machine par exemple)

Plusieurs représentations

Ce qu'on a à date

- Code source : des fichier texte, c'est notre entrée
- Code machine binaire : un tas d'octets, c'est notre sortie
- Assembleur : structure simple, à donner à l'assembleur
- AST : représentation syntaxique du programme source

Meilleur choix ?

Qu'est-ce qui est le plus **utile** (mieux pour optimiser) ?

Plusieurs représentations

Ce qu'on a à date

- Code source : des fichier texte, c'est notre entrée
- Code machine binaire : un tas d'octets, c'est notre sortie
- Assembleur : structure simple, à donner à l'assembleur
- AST : représentation syntaxique du programme source

Meilleur choix ?

Qu'est-ce qui est le plus **utile** (mieux pour optimiser) ?

- Code source : travailler avec des chaînes de caractères ?
  - Exemple : préprocesseur, langage de macro
- Code machine binaire : bas niveau, dépendent de la plateforme
- Assembleur : pourquoi pas, mais dépendent de la plateforme
- AST : pourquoi pas, mais ça peut être complexe ou limitant
  - Nœuds facilement annotables
  - Besoin de transformation d'arbres
  - Exemple : remplacement/suppression de sous-arbres

Représentation intermédiaire

Représentation **utile**

Pour l'analyse et l'optimisation

- Structure de donnée en mémoire
- Bien conçue

Pour la sérialisation

- Fichier binaire : compact et facile à charger
- Fichier texte : lisible et modifiable par un humain
- $\rightarrow$ On parle alors de **langage intermédiaire**

Représentations classiques

- AST $\rightarrow$ Déjà vu
- À pile
- 3 adresses
- Formelle

Représentation intermédiaire

Représentation **utile**

Pour l'analyse et l'optimisation

- Structure de donnée en mémoire
- Bien conçue

Pour la sérialisation

- Fichier binaire : compact et facile à charger
- Fichier texte : lisible et modifiable par un humain
- $\rightarrow$ On parle alors de **langage intermédiaire**

Représentations classiques

- AST $\rightarrow$ Déjà vu
- À pile
- 3 adresses
- Formelle $\rightarrow$ Voir INF889J

Représentation/langage à pile

Stack-based representation/language

Pile de données

- Les données sont dans une pile
- Existe en typé et non typé
- Éventuellement, du stockage auxiliaire (variables locales par exemple)

Instructions qui manipulent la pile

- Les instruction agissent sur les sommets de la pile
- Les arguments sont empilés
- Les résultats sont dépiles
- Éventuellement, des opérandes statiques auxiliaires (constantes, symboles, numéros de variables locales)
- Ça ressemble à la notation **polonaise inverse**
  - `1 2 + 3 *` $\rightarrow$ `(1+2)*3`

Exemple : Bytecode Java

```
 0: iconst_0        17: iload_2
 1: istore_1        18: iload 4
 2: iconst_1        20: iadd
 3: istore_2        21: istore_2
 4: iconst_0        22: iload_3
 5: istore_3        23: iconst_1
 6: iload_3         24: iadd
 7: bipush 15       25: istore_3
 9: if_icmpge 29    26: goto 6
12: iload_1         29: getstatic #7 // System.out
13: istore 4        32: iload_1
15: iload_2         33: invokevirtual #13 // println:(I)V
16: istore_1        36: return
```

Langages à pile

Quelques langages à pile

- *Bytecode* (code-octet) Smalltalk-80(p596), Java
- WebAssembly
- PostScript(p508)

Avantages

- Interpréteur facile à implémenter
- Représentation très compacte

Mais analyses et transformations pas simples

- Déterminer les relations et dépendances entre instructions
- Changer l'ordre des instructions
- $\rightarrow$ Pas tant de recherche théorique sur le sujet

Code 3 adresses (C3A)

Données

- Des **registres** (ou **variables**) **virtuels** (ou **pseudo-**) En quantité illimité !
- Des valeurs immédiates (constantes)

Instructions (ou opérations)

- un opérateur
- (au plus) un registre résultat
- (au plus) deux opérandes (registre ou valeur immédiate)
- $\rightarrow$ 3 données (représentés par des adresses)
- $\rightarrow$ 4 choses, on appelle parfois ça un « *quad* »

En vrai

- Certaines instructions ont plus de résultats et/ou opérandes
- Ou des étiquettes ou des symboles (branchement et *call*)
- Ça ressemble à de l'assembleur maison

Exemple : smali (dex)

```
.registers 5
const/4 p0, 0x0
const/4 v0, 0x0
const/4 v1, 0x1
:goto_6
const/16 v2, 0xf
if-ge p0, v2, :cond_13
add-int/2addr v0, v1
add-int/lit8 p0, p0, 0x1
move v3, v1
move v1, v0
move v0, v3
goto :goto_6
:cond_13
sget-object p0, Ljava/lang/System;->out:Ljava/io/PrintStream;
invoke-virtual {p0, v0}, Ljava/io/PrintStream;->println(I)V
return-void
```

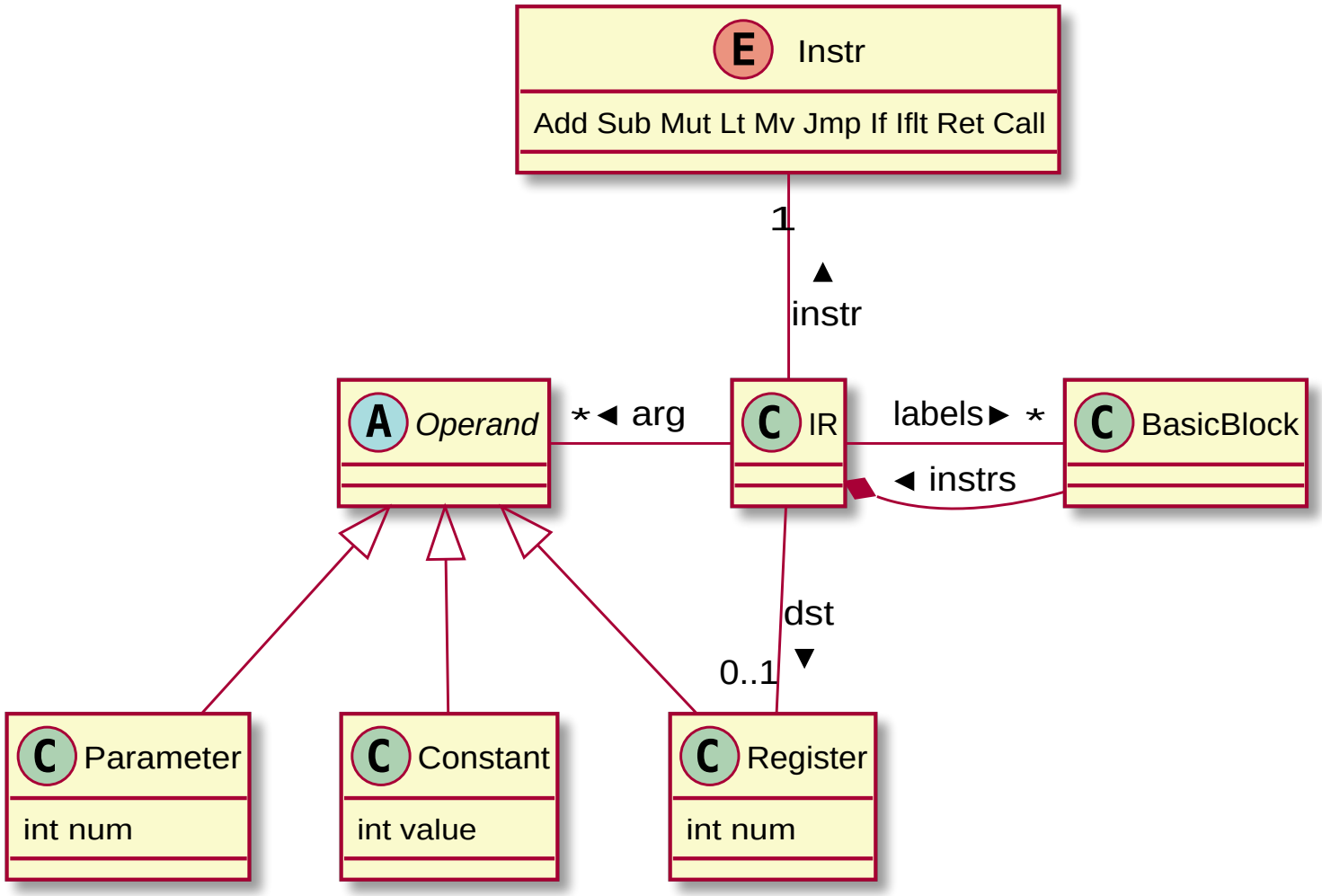
Code dit « 3 adresses » : Usage

- La plupart des compilateurs dans la vrai vie
- C'est raisonnablement facile à analyser et transformer
- C'est raisonnablement facile à traduire en assembleur ou code machine

MiniCC2 - Compilateur optimisant

MiniCC2 : Représentation intermédiaire

- Code 3 adresses, Simpliste, Homogène: une seule classe `IR`
  $\rightarrow$ Voir package `minic.ir`



Bloc de base (*basic block*)

- Séquence d'instructions
- Sans branchement entrant ou sortant intermédiaire
- Une seule entrée : à la première instruction
- Une seule sortie : à la dernière instruction, appelée **terminateur**

Astuce de représentation

- Un bloc de base à de bonnes propriétés
- Relations et dépendances entre instructions triviales
  $\rightarrow$ Optimisations **locales** faciles
- Optionnel, mais utile
  - Au pire, si on aime pas les blocs: un bloc = une instruction
  - On se contentera d'optimisations plus globales

MiniCC

- La dernière instruction est un Jmp, If, Iflt ou Ret
- Les autres instructions sont les autres
- Et `Call`?

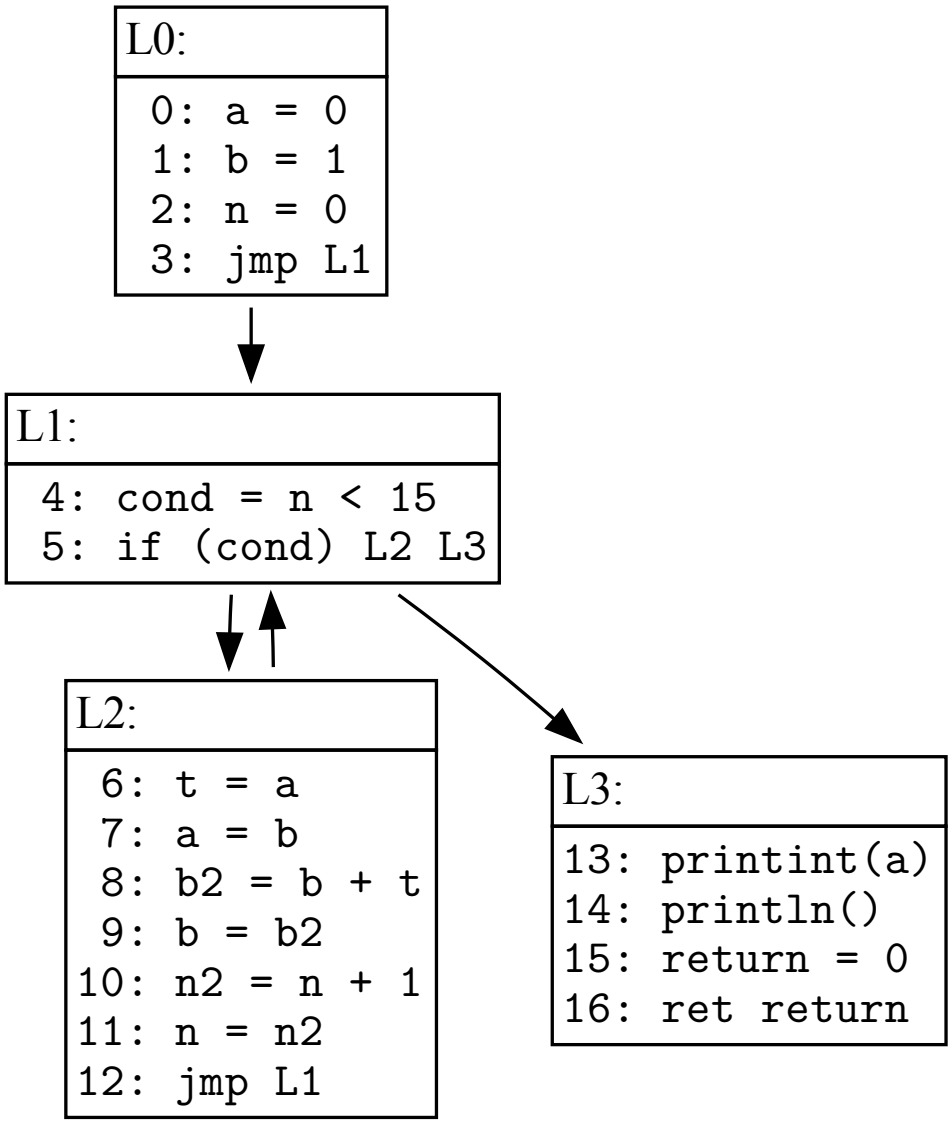
Graphe de flux de contrôle

- CFG : *Control flow graph*
  - Attention, parfois *context-free grammar*
- Graphe orienté
- Sommets = les blocs de base
- Arcs = les étiquettes du terminateur de chaque bloc

Simplification habituelle

- Un seul bloc d'entrée
- Un seul bloc de sortie

Exemple



fibonacci.c: main()

AST $\rightarrow$ IR

Comme MiniCC

Visiter et construire en une passe

- Classe `MiniCC2.java`
- `visitExpr` retourne une `Operand`
- Associer les variables à des registres
- Créer des registres supplémentaires au besoin

Blocs de base et CFG

- Les créer directement lors de la visite
- Alternative : créer du code IR linéaire et extraire les blocs

IR $\rightarrow$ Asm

Génération de code IR linéaire

- Faire disparaître les blocs
- Voire certains branchements!

Allocation des registres

- Associer les registres IR à des registres machine (ou des places en mémoire)
- Dépendant de la plateforme
- $\rightarrow$ On y reviendra

Sélection des instructions

- Associer les instructions IR à des instructions machine
- Dépendant de la plateforme

IR $\rightarrow$ IR : c'est là que la magie a lieu

Analyses du programme

- Trouver des **morceaux** de **sens** au programme
- Algorithmes, structures de données et mathématiques...
- $\rightarrow$ Sure (*sound*) : sur-approximation

Transformations incrémentales

- Intra-procédurales
  - Au niveau d'un bloc de base (locale)
  - Au niveau d'une boucle
  - Au niveau d'une fonction (un CFG)
- Inter-procédurales
  - Au niveau de toute l'unité de compilation (fichier source)
  - À l'édition de liens (*link-time optimization*, LTO)
  - Au chargement ? Durant l'exécution ?
- $\rightarrow$ Plus tard...

Analyses de flux de données

Data-flow « à l'ancienne »

- *Data-flow analysis* (DFA?)
  - Ne pas confondre avec *Deterministic finite automaton* (DFA!)
- Technique générale d'analyse $\rightarrow$ Voir `Dataflow.java`

Prend en compte le flux de contrôle (*flow sensitive*)

- Ordre des instructions
- Arcs du CFG
- $\rightarrow n$.prev et n.next pour chaque instruction (nœud) n

Prend en compte la sémantique de chaque instruction

- Quelle information devient vraie ? $\rightarrow n$.gen
- Quelle information devient fausse ? $\rightarrow n$.kill
- Information $=$ ensemble fini (de prédicats)

Résultat

- Quelle information est vraie avant et après : $\rightarrow n$.in et n.out

Exemple : *Reaching definition*

- Les affectations de registres (définitions)
  qui **peuvent** atteindre chaque instruction **future**
- Voir `ReachingDefinition.java`

Identifier l'information cherchée

- Ensemble considéré :
  Toutes les instructions qui affectent un registre (`dst != null`)
- Ensembles calculés à chaque instruction :
  Les affectations **passées** qui **peuvent** se rendre

Identifier les gen et kill de chaque instruction

- gen = l'instruction courante si elle affecte un registre
  $\emptyset$ sinon
- kill = toutes autres les instructions qui affectent le même registre destination

Et puis?

- Circuler l'information dans le CFG jusqu'à la stabilité
  $\rightarrow$ Converge vers le plus petit point fixe

Détails théoriques

- État : information n.in et n.out pour chaque instruction n
- État initial E_0 : n_0.in et n_0.out à $\emptyset$ pour chaque instruction n
- Fonction « faire un tour » : $E_{k+1} = \text{tour}(E_k)$
  - Pour chaque instruction n
    - $n_{k+1}.\text{in} \leftarrow \bigcup_{p \in n.\text{prev}} p_k.\text{out}$
    - $n_{k+1}.\text{out} \leftarrow (n_{k+1}.\text{in} \setminus n.\text{kill}) \cup n.\text{gen}$
- Point fixe
  - On s'arrête quand il n'y a plus de changement
  - C'est-à-dire quand $E_{k+1} = \text{tour}(E_k) = E_k$
- Preuves : terminaison ? le plus petit ? sûr (*sound*) ?

Algorithme liste de travail (un peu plus efficace)

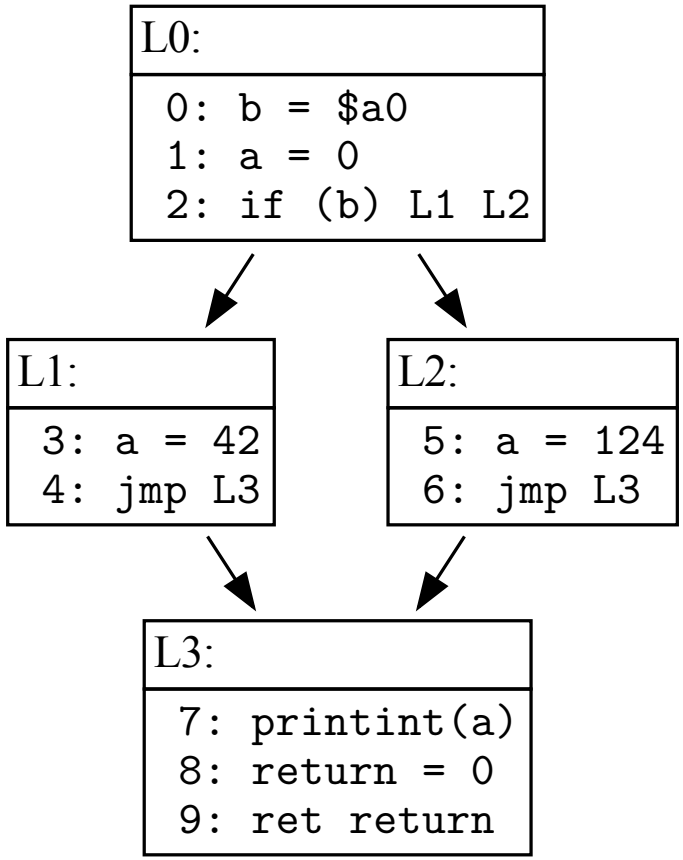
pour $n \in$ *nœuds* **faire**
    n.out $\leftarrow \emptyset$;
liste $\leftarrow$ nœuds ;
tant que *liste* $\neq \emptyset$ **faire**
    retirer n de liste ;
    si $n.prev = \emptyset$ **alors**
        out $\leftarrow n$.gen ;
    sinon
        n.in $\leftarrow \bigcup_{p \in n.\text{prev}} p$.out ;
        out $\leftarrow (n.\text{in} \setminus n.\text{kill}) \cup n.\text{gen}$;
    si *out* $\neq n.out$ **alors**
        n.out $\leftarrow$ out ;
        liste $\leftarrow$ liste $\cup$ n.next ;

Exercice : *Reaching definition*

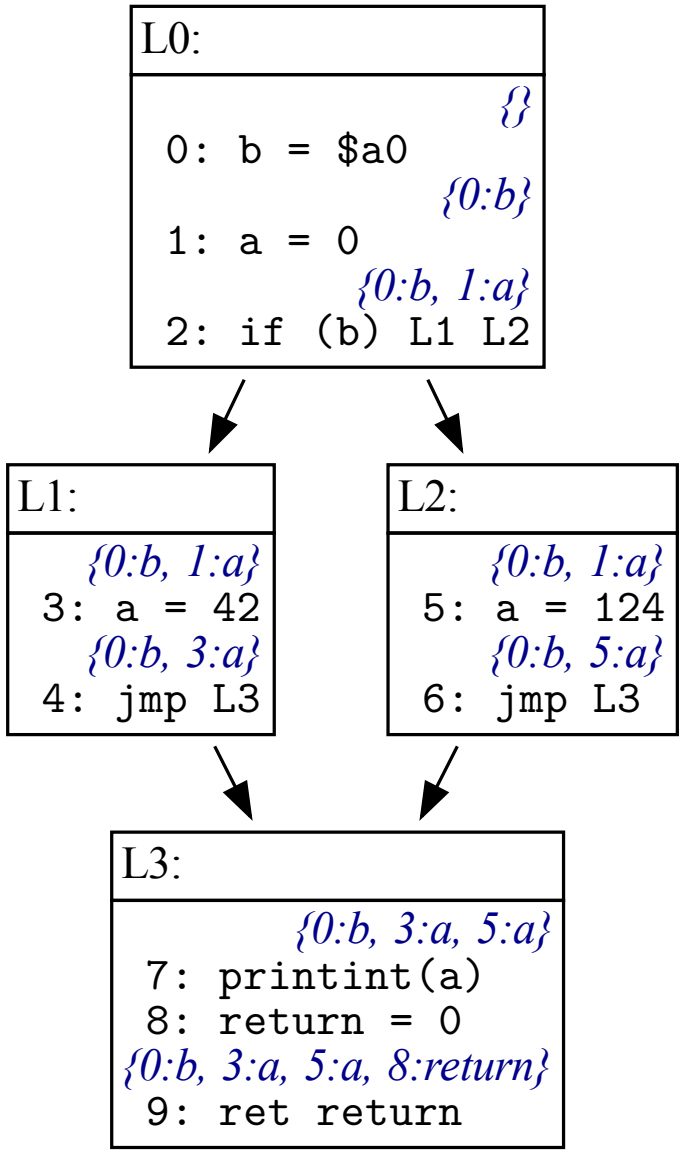
```
int foo(bool b) {
    int a = 0;
    if (b) {
        a = 42;
    } else {
        a = 124;
    }
    printint(a);
}
```

- Étape 1 : faire le CFG
- Étape 2 : déterminer n.gen et n.kill
- Étape 3 : initialiser n.out
- Étape 4 : faire circuler !

Exercice : Calculer *reaching definition*



reaching_definition.c: foo()



reaching_definition.c: foo()

Chaînes use-def

Use-def

- Pour chaque affectation d'un registre IR
- L'ensemble des instruction qui peuvent l'utiliser

Def-use

- Pour chaque utilisation d'un registre IR
- L'ensemble des instructions qui peuvent l'affecter

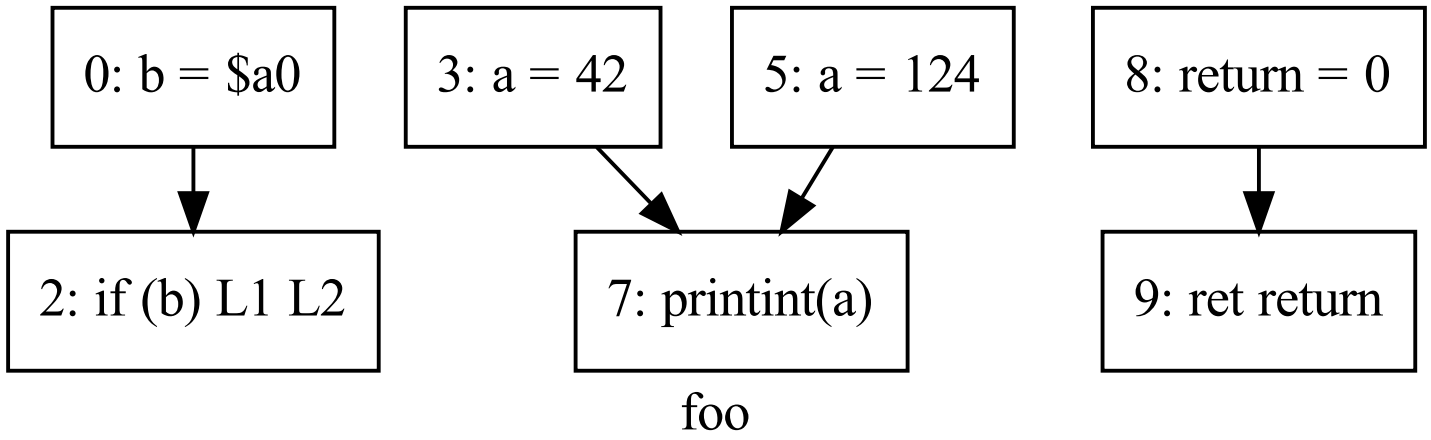
Utilité

- De nombreuses optimisations
- Mais, doit être recalculé ou maintenu

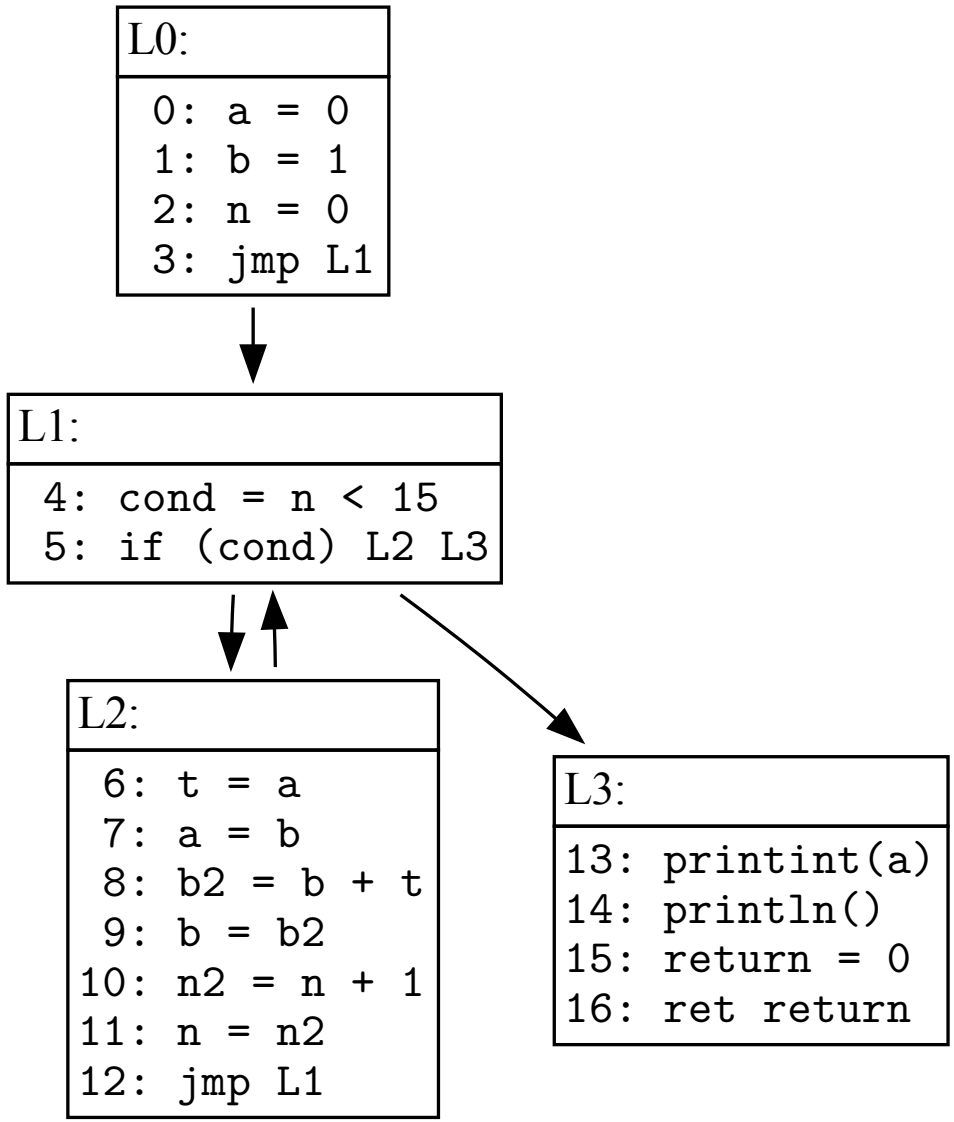
Exercice

- Calculer les chaînes use-def et def-use
- (Revenir à la diapo précédente)

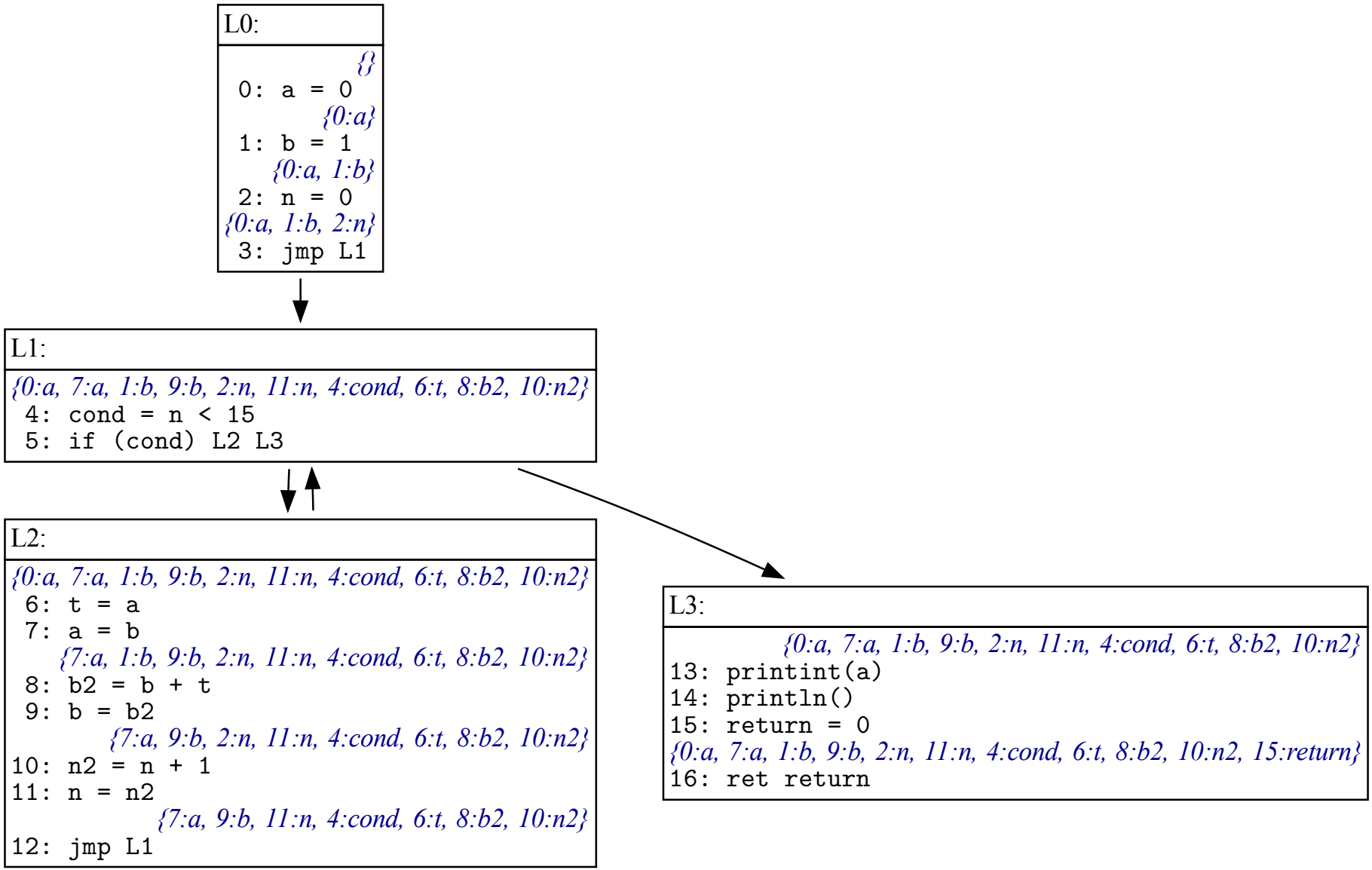
Chaînes use-def



Même chose pour fibonacci.c

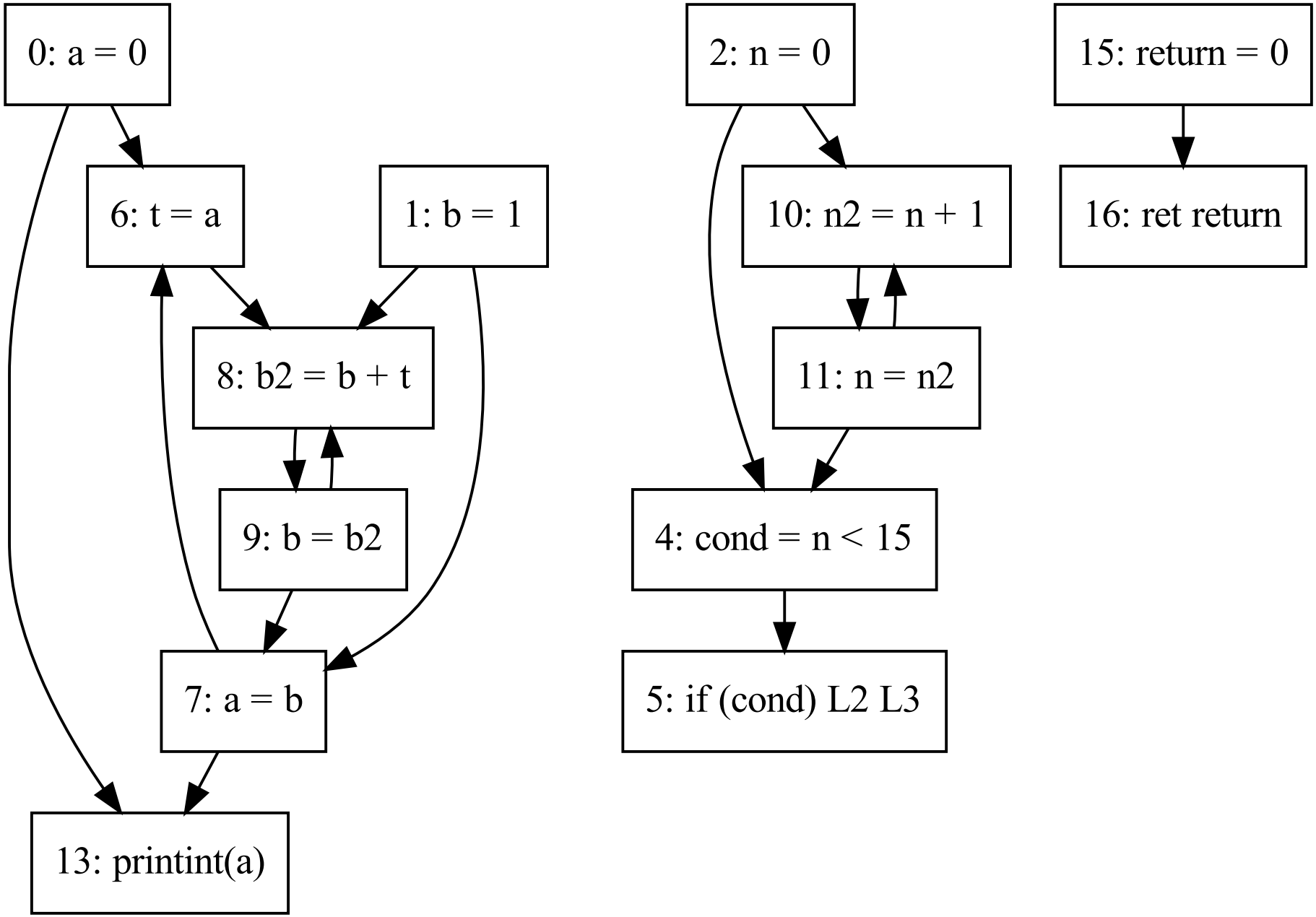


fibonacci.c: main()



fibonacci.c: main()

Chaînes use-def pour fibonacci.c



Implémentation Dataflow

Représenter des ensemble

- Utiliser des *bitsets*
- $\rightarrow$ Compact et efficace
- Voir `Dataflow.java`

Déterminer l'ordre de parcours

- Visiter les prédécesseurs avant les successeurs
- $\rightarrow$ Converge plus vite

Travailler sur le CFG (et non le graphe des instructions)

- Une passe (linéaire) pour déterminer les gen/kill globaux de chaque bloc à partir des gen/kill des instructions
- Faire circuler sur les blocs du CFG jusqu'au point fixe
- Une passe (linéaire) pour calculer les in/out de chaque instruction de chaque bloc

Doit (*must*) / peut (*may*)

Déterminer de l'information **possible**/**éventuelle**

- Celle qui est vraie sur **au moins** un des chemins
- C'est l'algo présenté (avec les unions ensemblistes)

Déterminer de l'information **certaine**/**nécessaire**

- Celle qui est vraie sur **chacun** des chemins
- $\rightarrow$ Faire l'intersection (au lieu de l'union) pour calculer n.in
- $\rightarrow$ Initialiser n.out à l'ensemble plein
- Note: on cherche alors le plus grand point fixe

Exemple : *Available expression*

- Les calculs **nécessairement** déjà calculés dans le **passé**
  - $\rightarrow$ `AvailableExpressions.java`
- Optimisation : *Common subexpression elimination*
  - Avantage:

Exemple : *Available expression*

- Les calculs **nécessairement** déjà calculés dans le **passé**
  - → `AvailableExpressions.java`
- Optimisation : *Common subexpression elimination*
  - Avantage: Pas besoin de les calculer à nouveau !
  - Inconvénient: Il faut maintenir un registre de plus
  - → `CommonSubexpressionElimination.java`

Dataflow

- Ensemble :

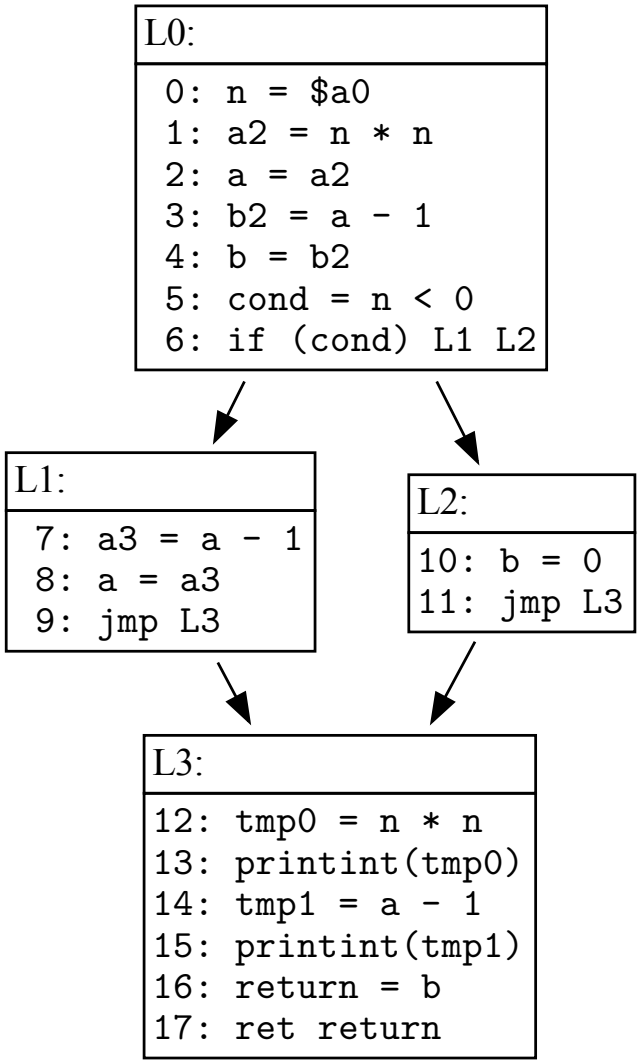
Exemple : *Available expression*

- Les calculs **nécessairement** déjà calculés dans le **passé**
  - → `AvailableExpressions.java`
- Optimisation : *Common subexpression elimination*
  - Avantage: Pas besoin de les calculer à nouveau !
  - Inconvénient: Il faut maintenir un registre de plus
  - → `CommonSubexpressionElimination.java`

Dataflow

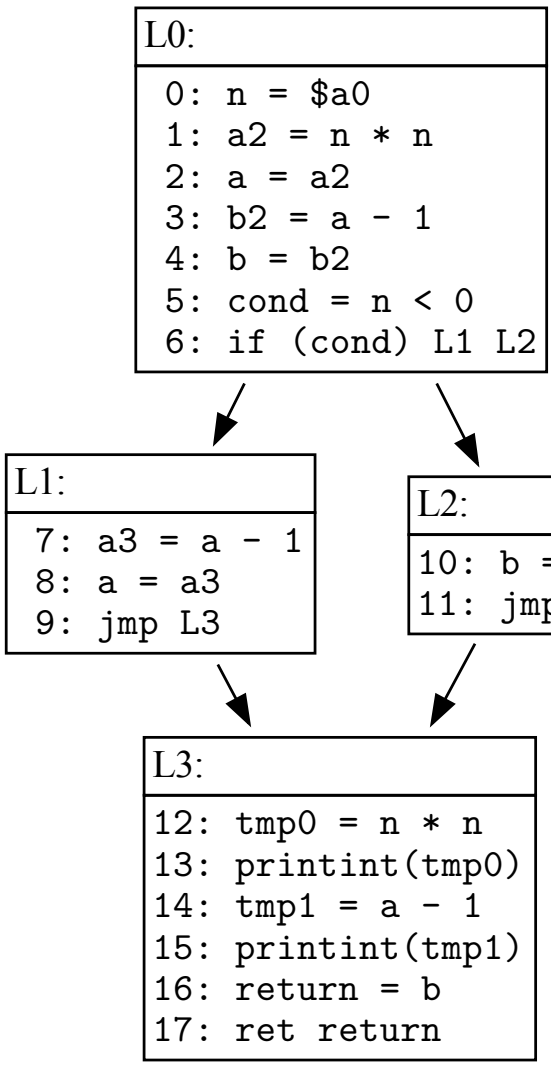
- Ensemble : les expressions arithmétiques (sans effet de bord)
- Ensemble à chaque instruction : toutes les expressions **nécessairement** disponibles
- Fusion : **doit**
- Gen(i) : l'expression courante i
  (si l'instruction i est un expression arithmétique)
- Kill(i) : toutes les expressions arithmétiques qui utilisent i.dst en argument

Exercice : *Available expression*



available2.c: foo()

Exercice : *Available expression*



available2.c: foo()

available2.c: foo()

En avant et en arrière

En avant (*forward*)

- Déterminer de l'information par rapport au **passé**
- C'est l'algo présenté

En arrière (*bakward*)

- Déterminer de l'information par rapport au **futur**
- $\rightarrow$ Parcourir le CFG à l'envers

Exemple : *Live-variable analysis*

- Un registre IR est **vivant** à une instruction s'il est **possiblement** utilisé dans une instruction **subséquente**
- Optimisation : on peut les *réutiliser* dès qu'on en a plus besoin
- `LiveVariables.java`

Dataflow

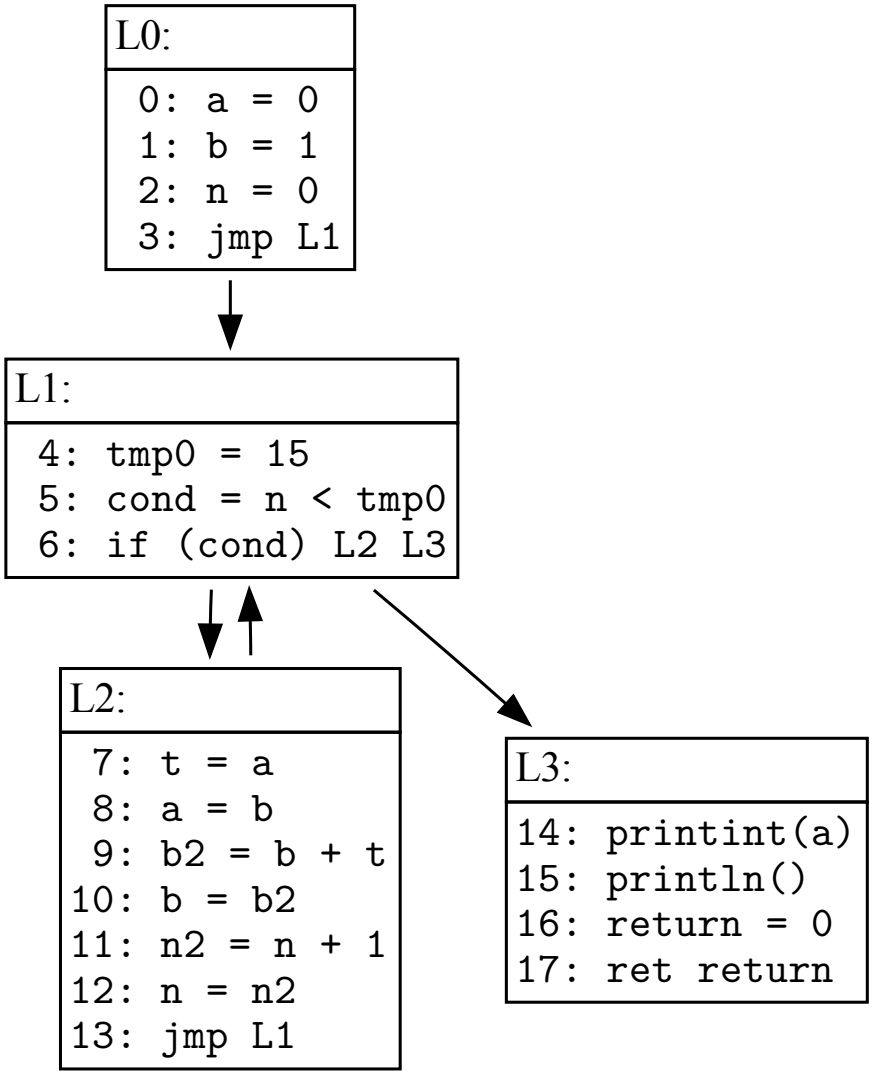
- Ensemble :

Exemple : *Live-variable analysis*

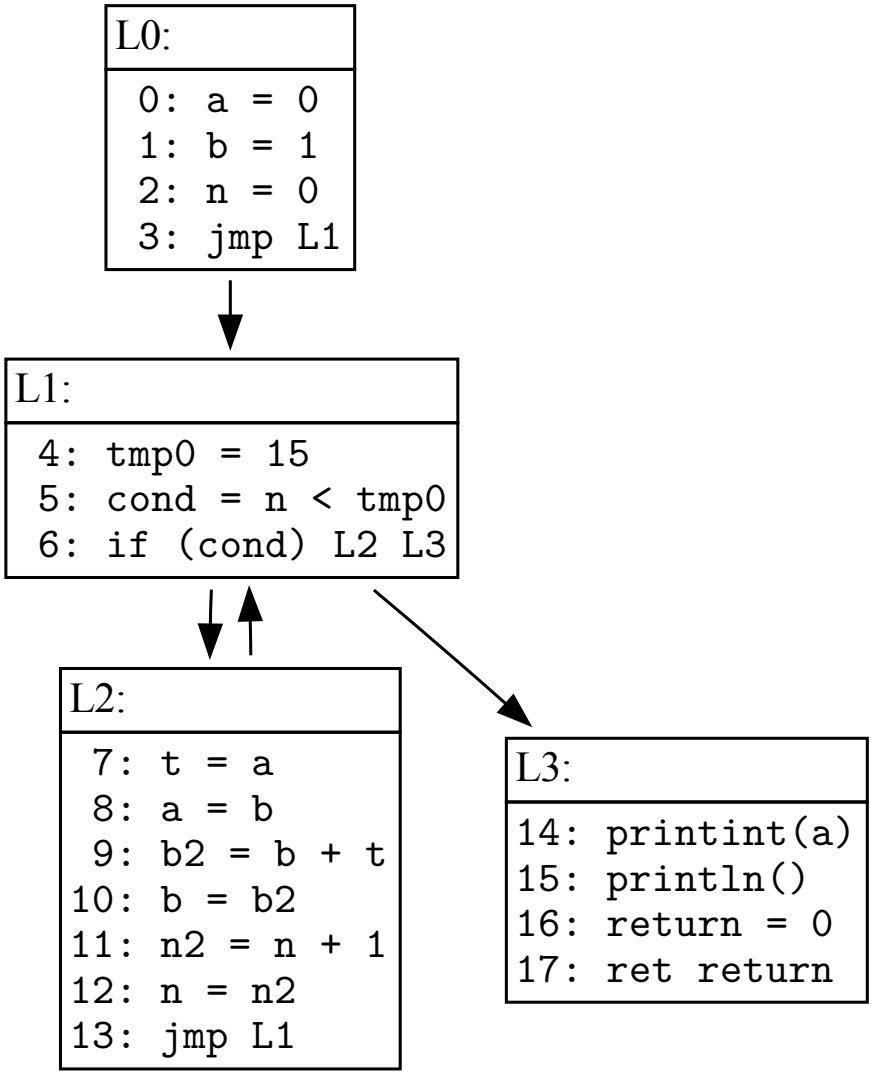
- Un registre IR est **vivant** à une instruction s'il est **possiblement** utilisé dans une instruction **subséquente**
- Optimisation : on peut les *réutiliser* dès qu'on en a plus besoin
- `LiveVariables.java`

Dataflow

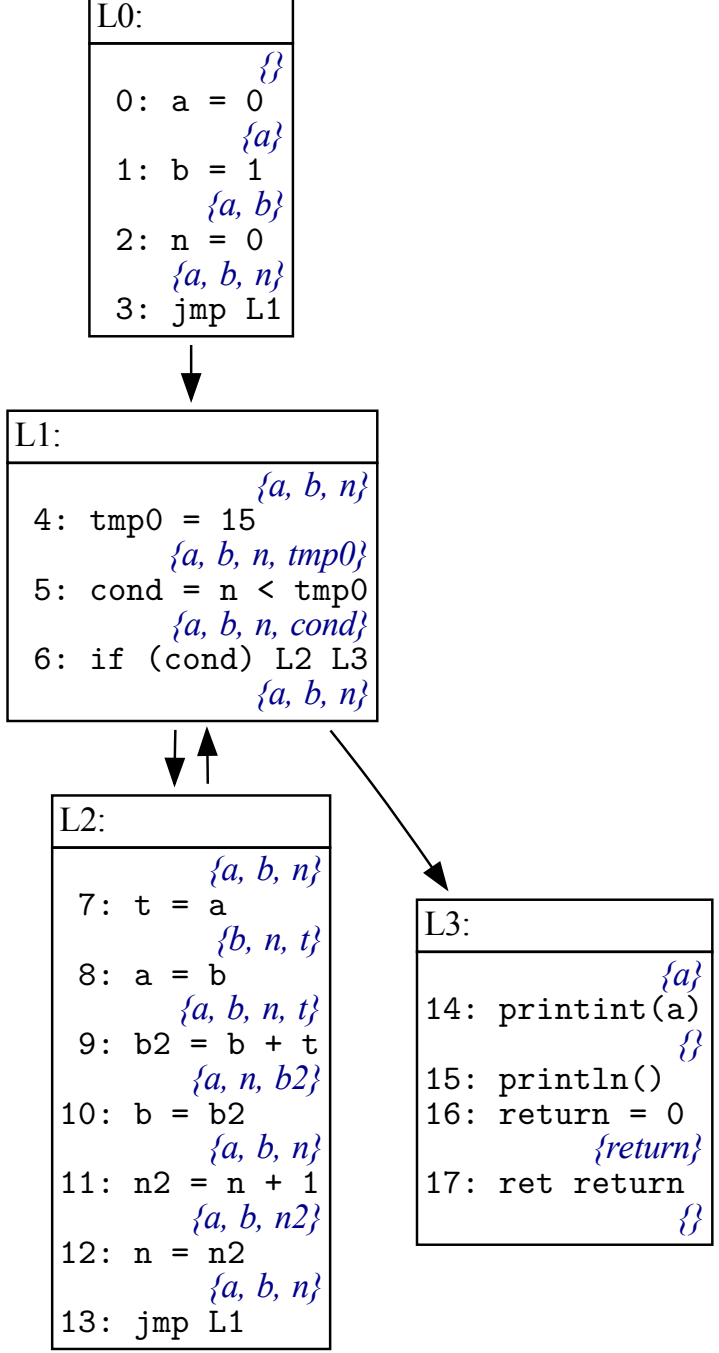
- Ensemble : les registres IR
- Ensembles calculés à chaque instruction : les registres vivants
- Sens : en arrière (*backward*), car information sur le « futur »
- Fusion : union (*may*), car c'est une possibilité
- Gen(i) : les registre sources de l'instruction i
- Kill(i) : le registre destination de l'instruction i



fibonacci.c: main()



fibonacci.c: main()



fibonacci.c: main()

En arrière et doit

- On combine **en arrière** et **doit**
- Déterminer
  - de l'information **certaine**
  - sur tous les chemins **futurs**

Exemple : *Very busy expressions*

- Les calculs qui vont **nécessairement** être faits dans le **futur**
  - $\rightarrow$ `VeryBusyExpressions.java`
- Optimisation : *code hoisting* (remontée de code?)
  - On peut les faire maintenant une fois pour toute
- Avantages

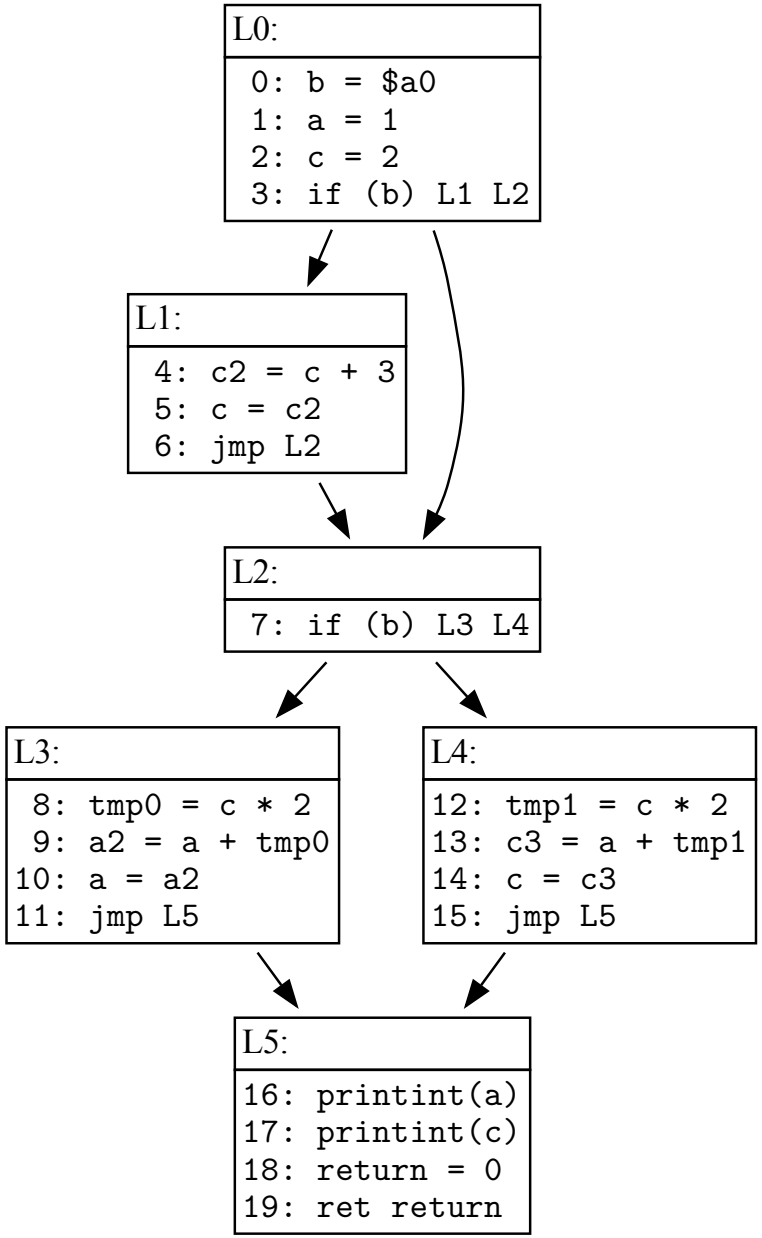
En arrière et doit

- On combine **en arrière** et **doit**
- Déterminer
  - de l'information **certaine**
  - sur tous les chemins **futurs**

Exemple : *Very busy expressions*

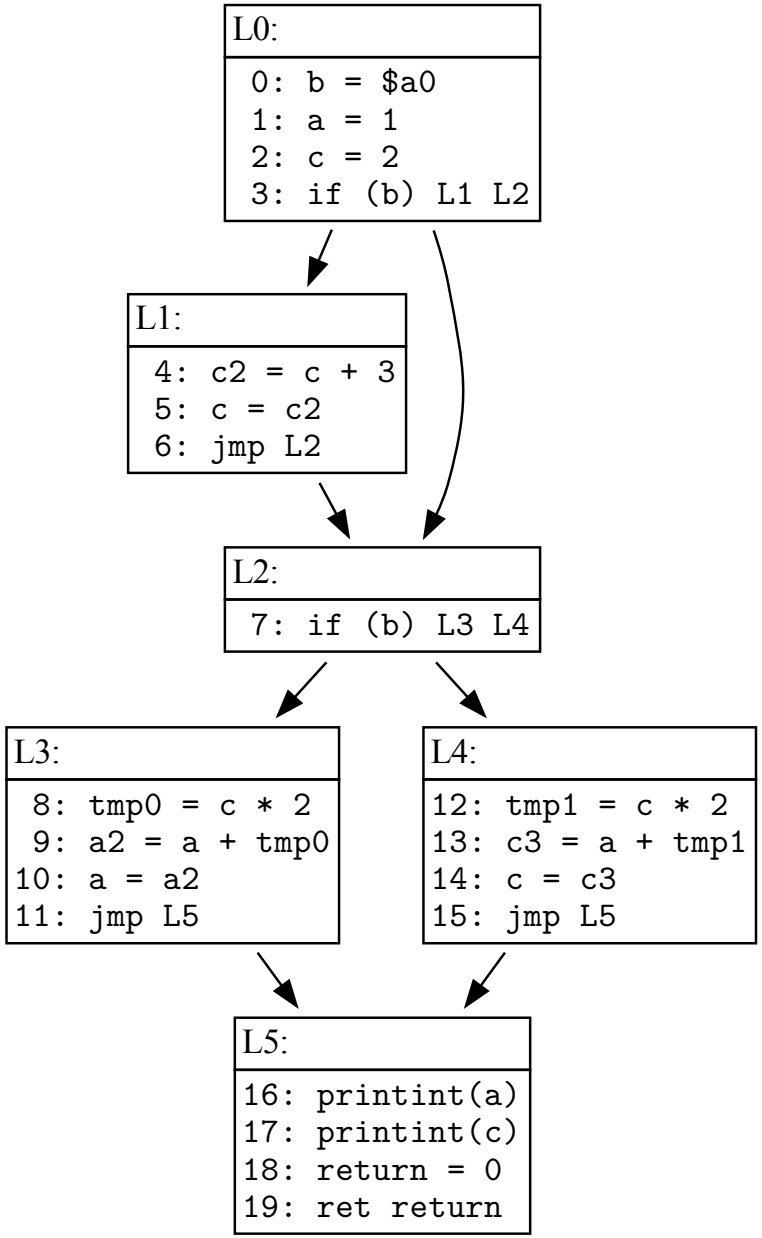
- Les calculs qui vont **nécessairement** être faits dans le **futur**
  - → `VeryBusyExpressions.java`
- Optimisation : *code hoisting* (remontée de code?)
  - On peut les faire maintenant une fois pour toute
- Avantages
  - Code plus petit
  - Peut sortir les instructions des boucles
- Inconvénient
  - Coûter un registre à maintenir plus longtemps

Exercice : *Very busy expressions*

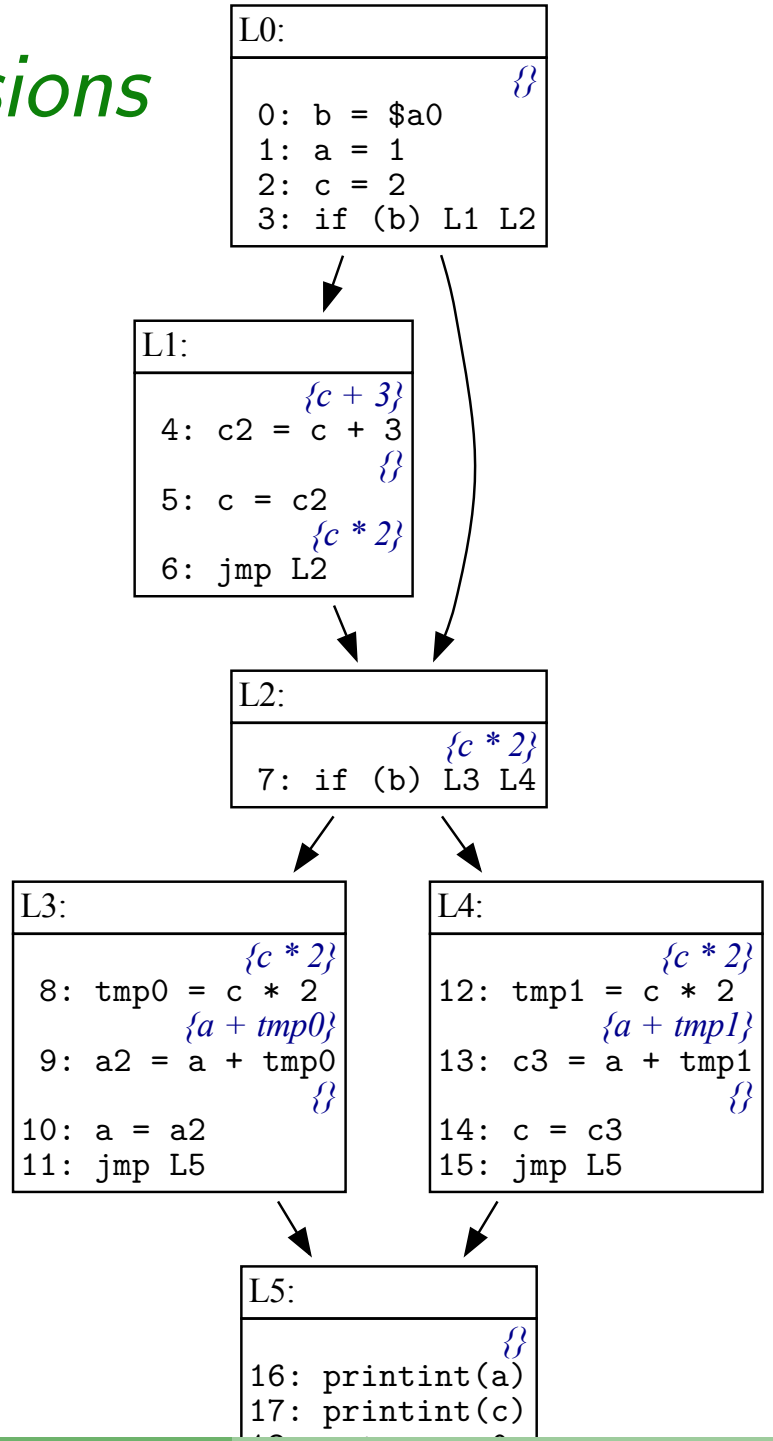


verybusy.c: foo()

Exercice : *Very busy expressions*



verybusy.c: foo()



Plus d'analyses ?

Unified Framwork « moderne », plus général

- Marlowe&Ryder, 1990
- Treillis, au lieu d'ensembles
- Fonction de transfert riches, au lieu de gen/kill statiques

Interprétation abstraite ← la vraie affaire

- Cousot&Cousot, 1977
- Avec plus de treillis
- Et des correspondances de Gallois
- Et des preuves

Plein d'autres

- Analyses symboliques → On formalise l'exécution
- Analyses dynamiques → On raisonne sur des traces
- → INF889A Analyse de programmes pour la sécurité logicielle

Allocation de registres

Allocation de registres

- Associer des registres IR à des registres machine
- Objectif : Minimiser le nombre de registres machine nécessaires
  - Et espérer ne pas avoir besoin de la pile
- Contrainte : si deux registres IR sont utiles en même temps
  - Ils ne peuvent être associé au même registre machine

Spilling (déversement? vidage?)

- Allouer un registre IR en mémoire au lieu d'un registre machine (habituellement dans la pile)
- Besoin de réserver de la place dans la pile
- RISC : Besoin d'instructions de sauvegarde et de chargement et de réserver quelques registres pour la manipulation
- CISC : Besoin d'utiliser des modes d'adressage plus coûteux

Via l'AST et l'analyse de portée (*scope analysis*)

- Comme MiniCC
- Alternativement, via un arbre d'expressions, qu'on peut reconstruire depuis les blocs de base

Variable

- Associée du début jusqu'à la fin de sa portée
- Indépendamment de son utilisation réelle

Résultat intermédiaire

- Associé de leur calcul (unique) jusqu'à leur utilisation (unique)
- Astuce : Algorithme Sethi–Ullman, 1970
  $\rightarrow$ Évaluer en premier l'opérande qui nécessite le plus de registres

Linear scan

Poletto et Sarkar, 1999

Algo rapide

- 1: Linéariser le CFG (ordre arbitraire)
- 2: Calculer les registres vivants (*live variables*)
- 3: Déterminer l'intervalle de vie de chaque registre
- 4: Parcourir les intervalles chronologiquement
  - Libérer les registres machine des intervalles finis
  - Assigner un registre machine libre au début d'un intervalle

→ `RegisterAllocationLinearScan.java`

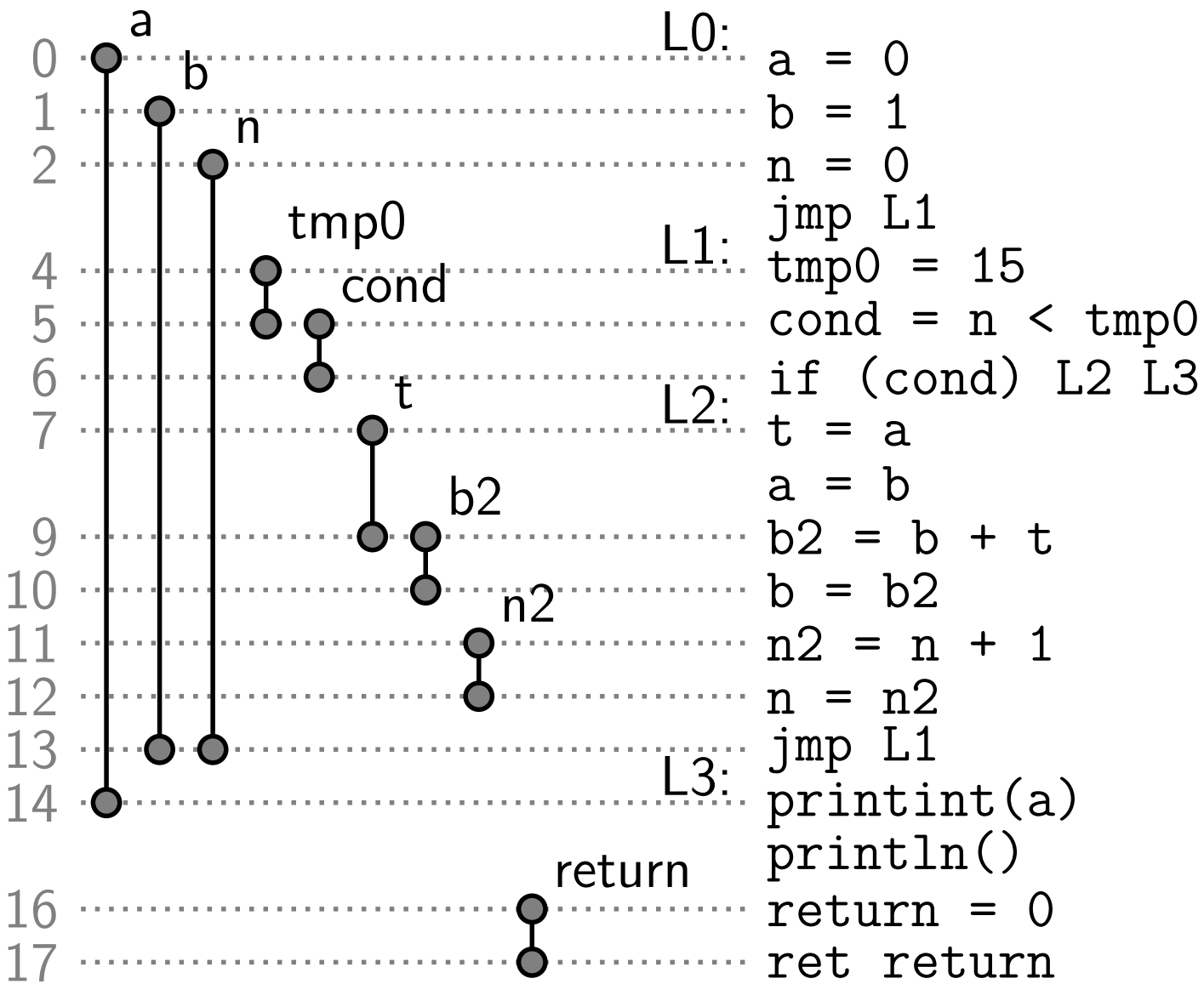
Inconvénients

- Ne prend pas en compte les « trous » dans l'intervalle de vie
- Dépendant de la linéarisation choisie

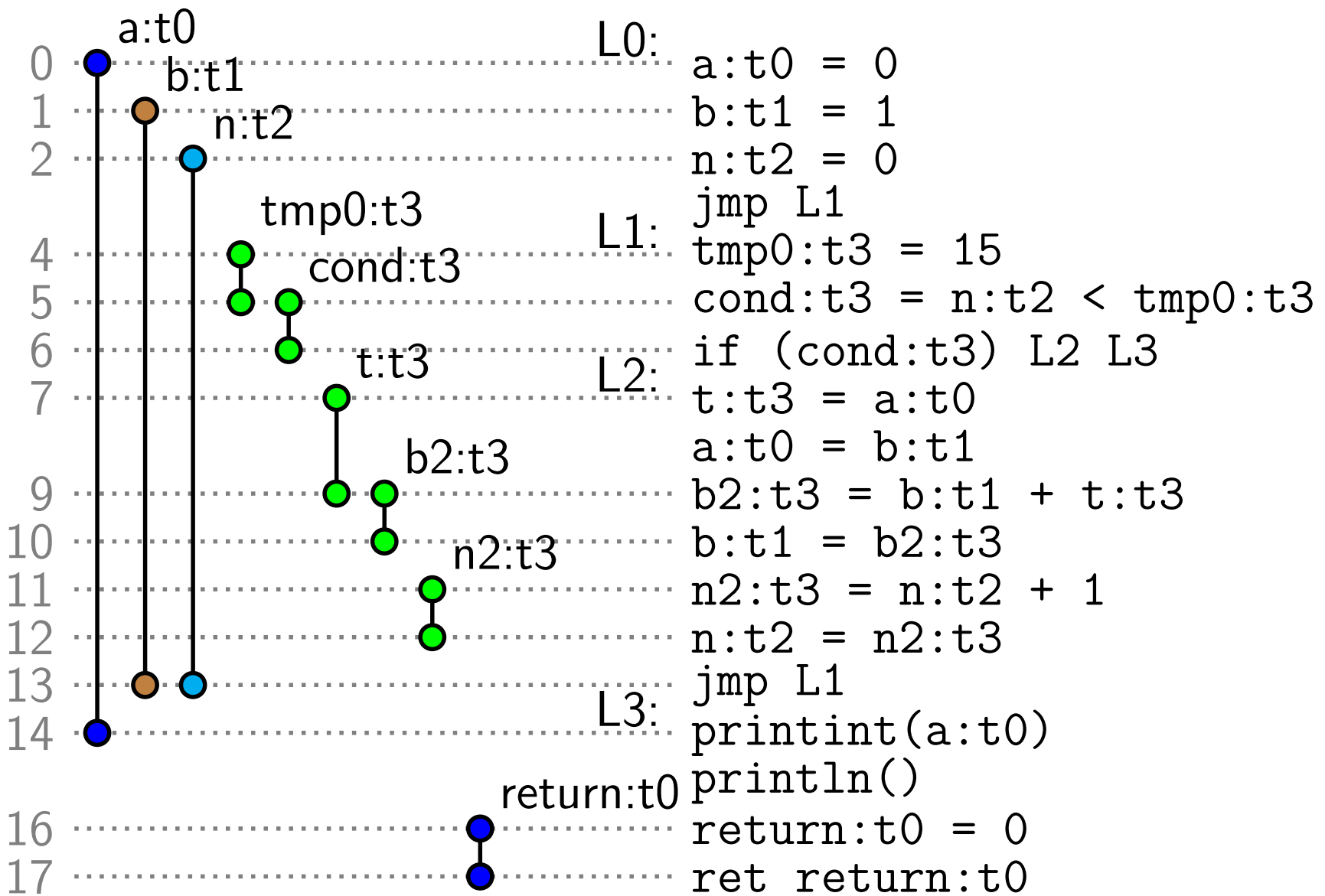
Exercice : sur fibonacci

- 9 registres IR à associer → 7 registres machine `t0` à `t6`

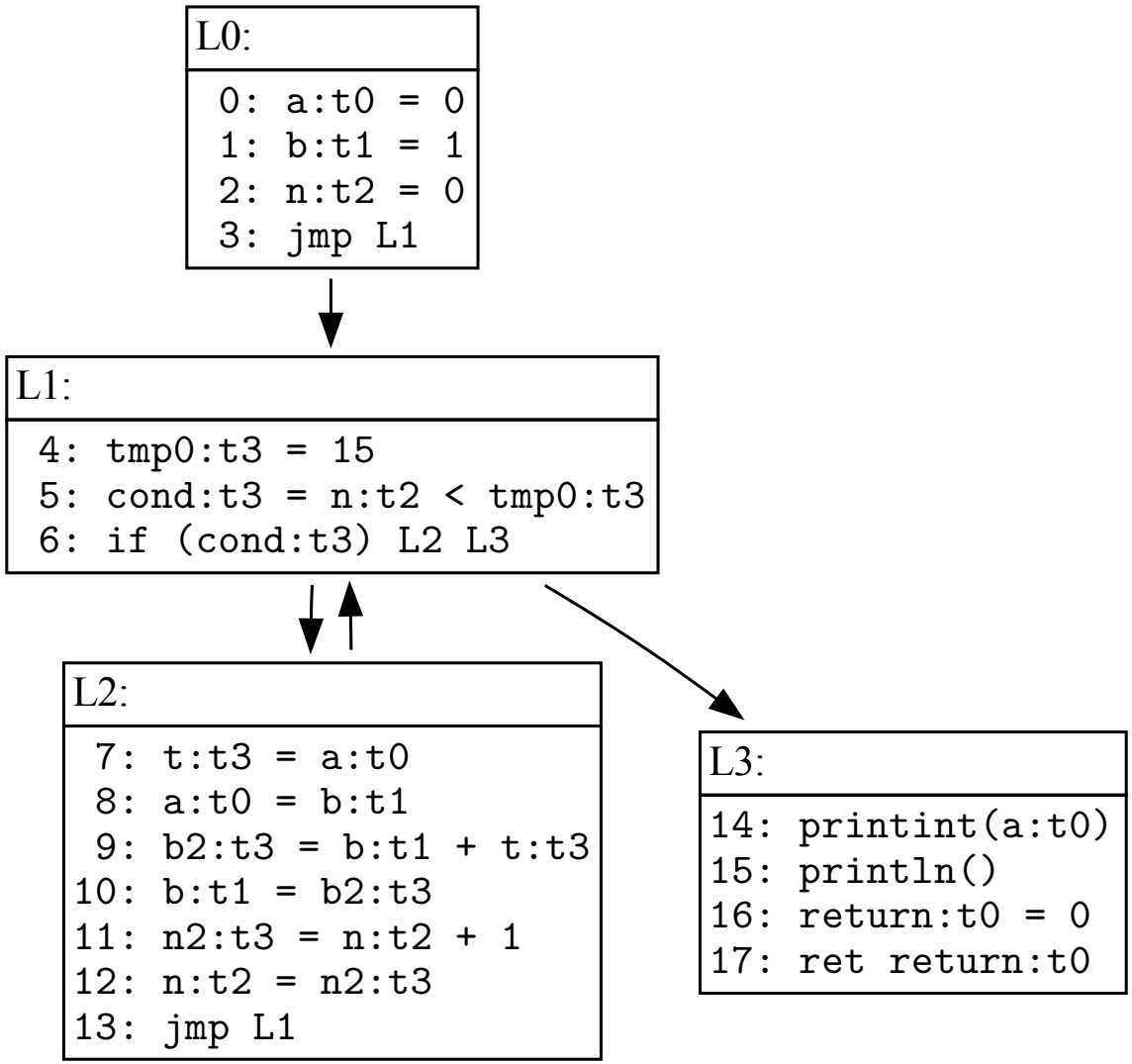
Intervalles



Allocation terminée



Allocation terminée



fibonacci.c: main()

Coloration

Graphe d'interférence

- Graphe non orienté
- Sommets : Les registres IR
- Arrêtes : Entre registres vivants à une même instruction

Coloration

- Attribuer une **couleur** à chaque sommet
- Tel que deux sommets adjacents soient de **couleur** différente
- Borner (ou minimiser) le nombre de couleurs

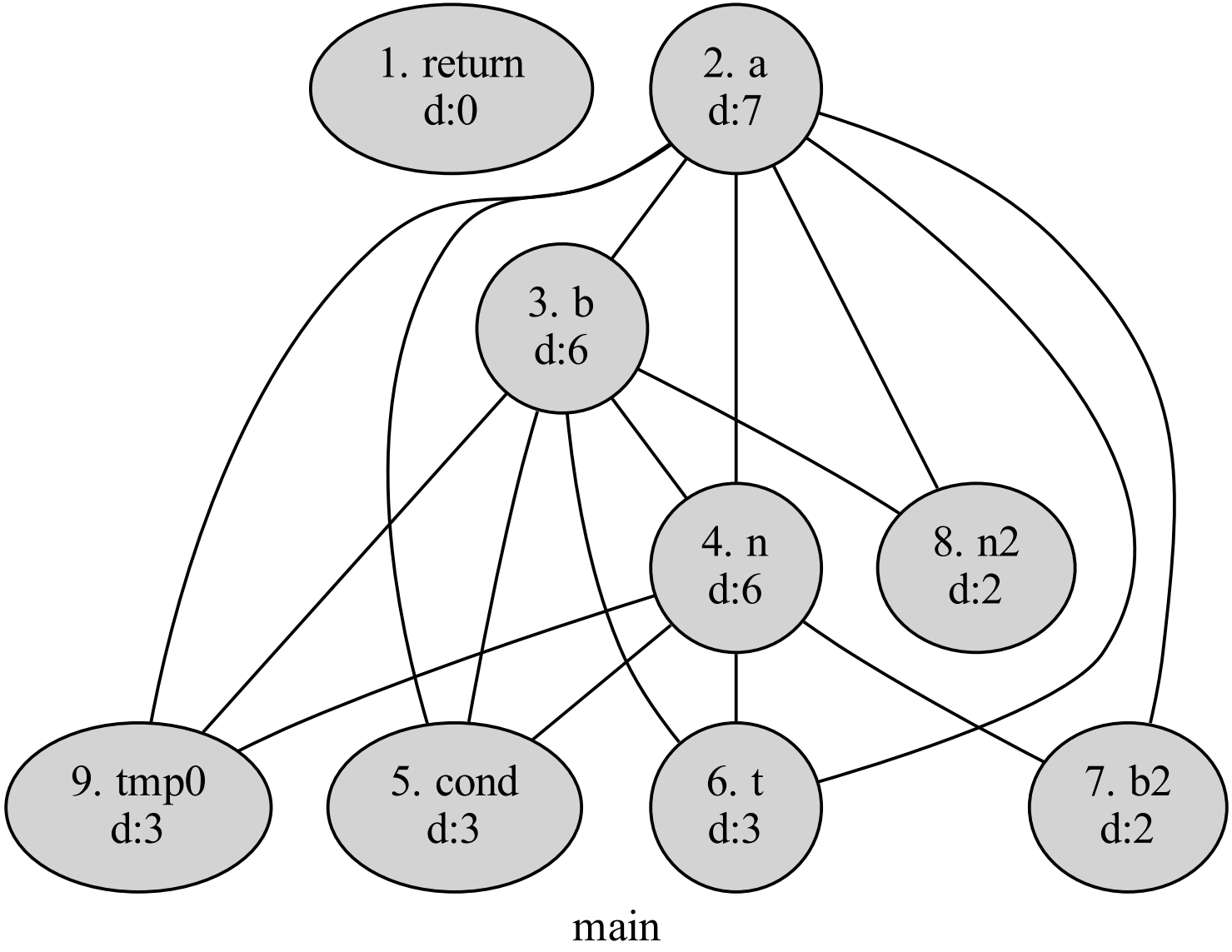
Problème

- La coloration, c'est NP-Difficile (oups)

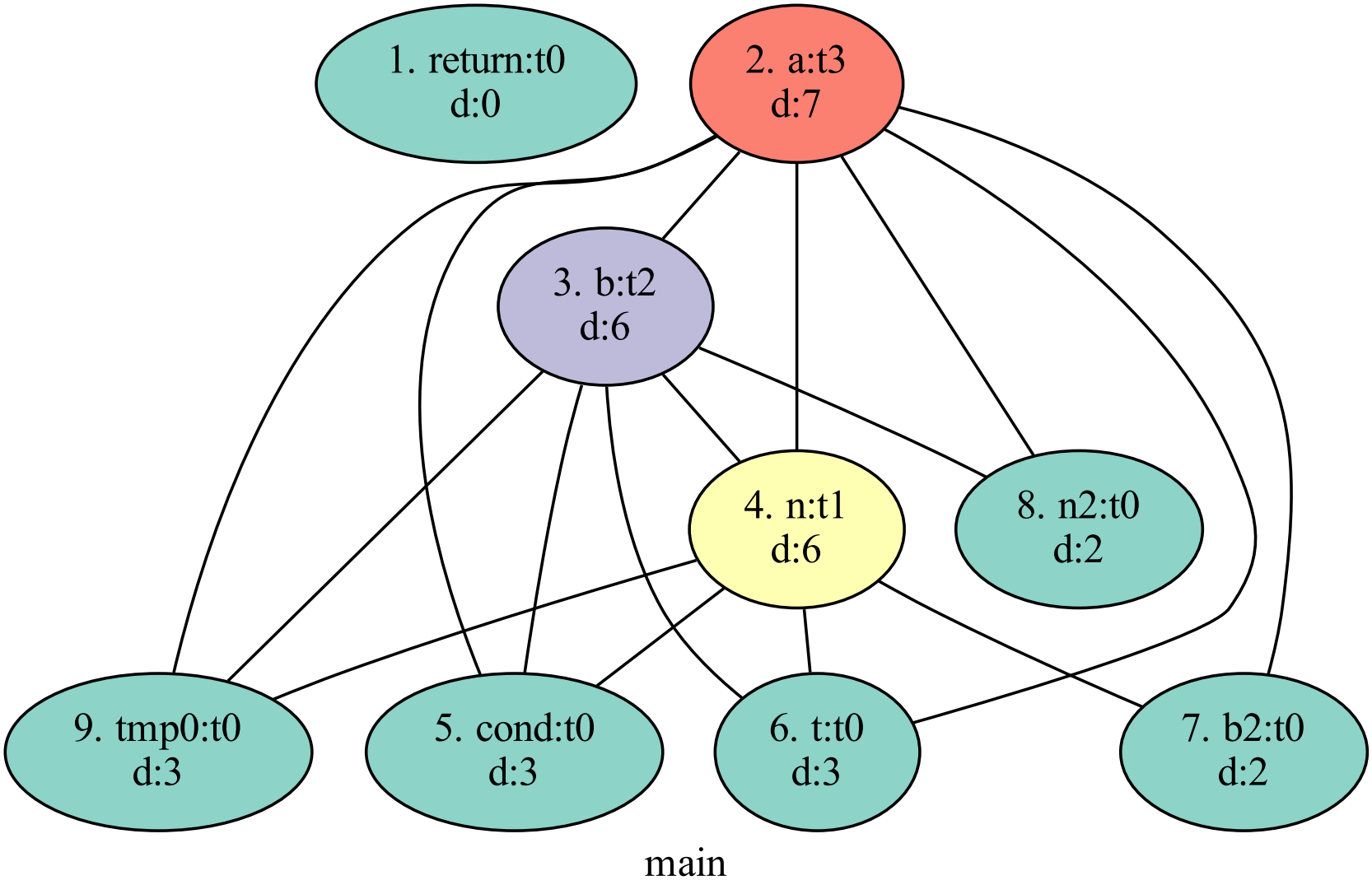
Heuristique de Kempe (1879)

Données : $G = (V, E)$ un graphe
$p \leftarrow$ Pile vide ;
tant que G *non vide* **faire**
    Choisir $v \in V$ de degré minimal ;
    Retirer v de G, et toutes ses arêtes ;
    Empiler v dans p ;
tant que p *non vide* **faire**
    $v \leftarrow$ dépiler p ;
    Ajouter v à G, avec ses arêtes ;
    Colorer v par rapport à ses voisins actuels ;

Exercice coloration



Voilà


1. return:t0
d:0
2. a:t3
d:7
3. b:t2
d:6
4. n:t1
d:6
8. n2:t0
d:2
9. tmp0:t0
d:3
5. cond:t0
d:3
6. t:t0
d:3
7. b2:t0
d:2
main

Petits détails

Registre sauvés

- Utiliser un registre sauvé (s) ou non (t) ?
- Si le registre IR est vivant à au moins un `call`,
  - Alors utiliser un registre sauvé
  - $\rightarrow$ Colorier deux ensembles disjoints

Contraintes machines / d'ABI

Certains registres spécifiques doivent parfois être utilisés

- Contraintes de l'architecture. Eg. `XLAT` = `AL=[AL+RBX]`
- Contrainte d'ABI. Eg. `a0` pour le retour.
- $\rightarrow$ Pré-allouer ces registres
- $\rightarrow$ MiniCC2 (hack)
  - Allouer seulement `s` et `t`
  - Réserver `a` pour les détails des arguments et retours

Coalescence (*coalescing*)

Coalescence opportuniste

- Après coloration
- Si un `mov` entre deux registres IR
  mais associés au même registre machine
  $\rightarrow$ Supprimer le `mov`

Coalescence forcée

- Avant coloration
- Deux registres pas en conflit,
  mais connectés pas des `mov`
  $\rightarrow$ Unifier les deux registres, et supprimer les `mov`

Émission de code

Émission de code

- Générer de l'assembleur ou du code machine
- Très dépendant de la plateforme

Sélection d'instructions

- Quelles instructions machines pour quelles instructions IR ?
- Problème complexe : ce n'est pas du 1 pour 1
  - Surtout en CISC

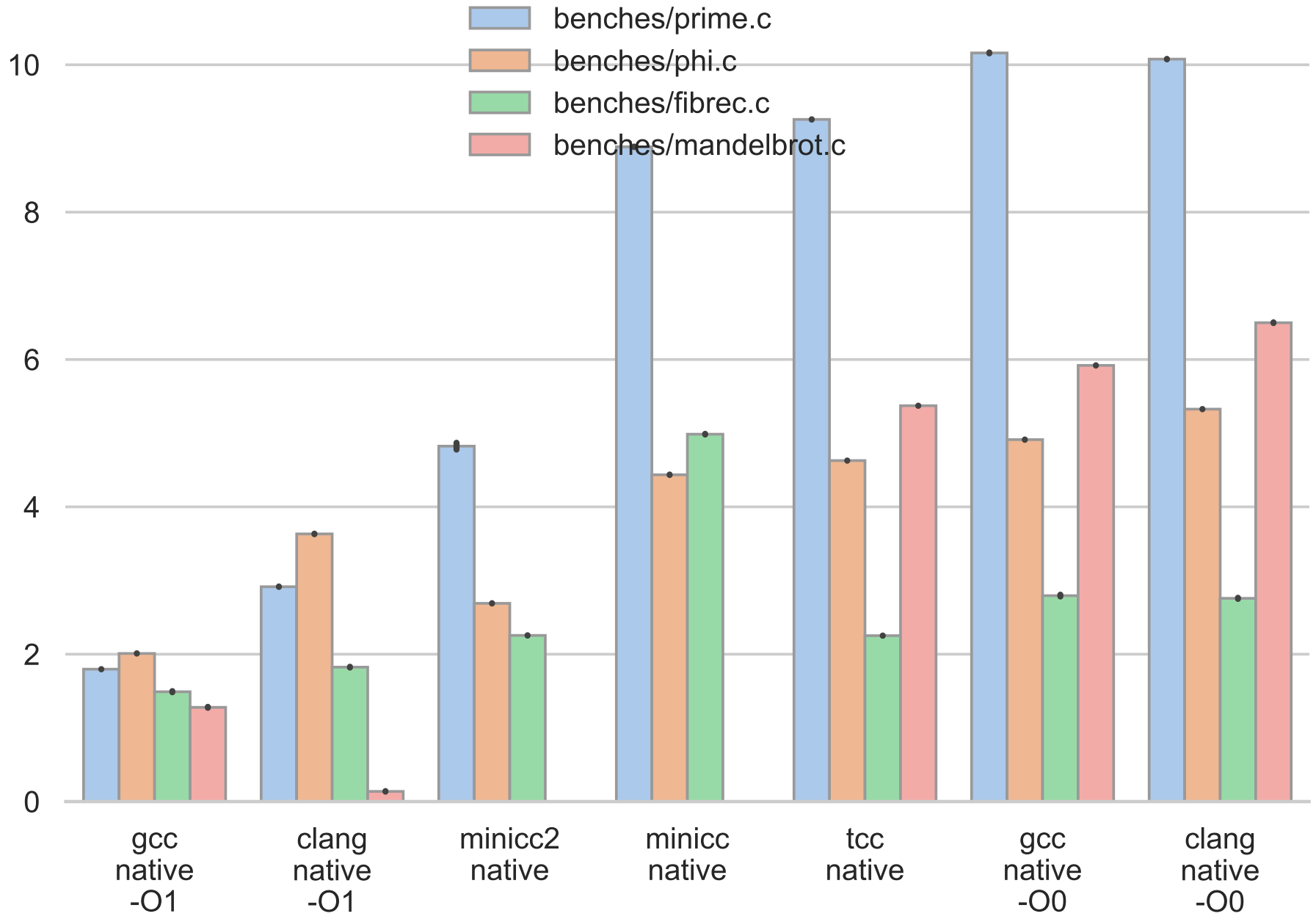
Contraintes d'architecture

- Limites et adressages des instructions machines
- `add` vs. `addi`, taille des valeurs immédiates

MiniCC2

- `Legalize.java` → applique les contraintes machines (sur l'IR)
- `AsmEmitter.java` → traduction un-pour-un naïve (IR → asm)

MiniCC2 de base (-O0?)



La vrai vie

GCC

- GCC internals
- GCC options d'optimisations

Ses représentations intermédiaires

- GENERIC : AST pour le C/C++ et autres langages
- GIMPLE (ou *Tree*) : C3A hiérarchique, indépendante de la plateforme
- RTL (*Register Transfer Language*), C3A hiérarchique, dépendante de la plateforme
- Assembleur : émit depuis le RTL via des règles de *pattern matching*

Voir ?

- `gcc -fdump-tree-all -fdump-rtl-all` (et plein d'autres options)

LLVM/Clang

Ses représentations intermédiaires

- Clang internals : AST pour le C/C++
- Langage LLVM : C3A, indépendante de la plateforme, stable et public
- Machine IR : C3A, dépendante de la plateforme

Voir ?

- `clang -mllvm -print-changed` (et plein d'autres options)
- `opt -print-changed` (et plein d'autres options)
- `llc -print-changed` (et plein d'autres options)

JDK

- javac doc
- HotSpot doc

Ses représentations intermédiaires

- javac : AST Java
- Bytecode Java : langage à pile, indépendante de la plateforme, stable et public
- HotSpot C1 : HIR (*high IR*) : C3A, indépendante de la plateforme
- HotSpot C1 : LIR (*low IR*) : C3A, dépendante de la plateforme
- HotSpot C2 : IR : représentation « sea of nodes »

Voir ?

- `javac -verbose` mais pas d'info plus précise
- `java -XX:+UnlockDiagnosticVMOptions \`
  `-XX:+LogCompilation`
  et `-Xcomp` pour forcer la compilation

Nit

- Projet Nit

Ses représentations intermédiaires

- Juste l'AST. On le transforme et on le simplifie
  C'est pas toujours facile → Sujet de recherche ?
- Génère du C (utilisé comme macro-assembleur portable)
  On pourrait générer quelque chose de mieux (llvm?) → Sujet de recherche ?

Voir ?

- L'AST mérite plus d'amour → Sujet de recherche ?
- Le C généré est lisible

Pharo

- Projet Pharo

Ses représentations intermédiaires

- Coté image: AST $\rightarrow$ IR à pile $\rightarrow$ Bytecode
- Coté vm : Bytecode $\rightarrow$ code machine natif (template)
- projet Druid : Bytecode $\rightarrow$ IR C3A $\rightarrow$ code machine natif

Voir ?

- Tout est vivant et inspectable

La prochaine fois

- Optimisations (pour vrai)