Compilation : B.A.BA
Analyse du flot de contrôle

Compilation Avancée, 2ème partie
Cours 1

Karine Heydemann
karine.heydemann@lip6.fr

Organisation de la 2ème partie

- Cours : Karine Heydemann ; TD-TP : Karine Heydemann
- Cours le 28 février puis le 21, 28 mars, puis les 18 et 25 avril.
- TD-TP le 2 mars, puis 23 et 30 mars, puis les 20 et 27 avril
- Support de cours mis en ligne : www-soc.lip6.fr/ñeydeman/
- TPs à rendre sous forme de projet à la fin de l'UE, avec étapes intermédiaires notées (pour vous forcer à avancer).

Les besoins

- Puissance de calcul + contraintes : consommation, taille de code, temps de réponse (dur/mou), fiabilité, portabilité, temps de conception.
- Les systèmes haute-performance sont complexes : architectures généralistes superscalaires, VLIW/DSP, GPU, multi-coeurs homogènes/hétérogènes
- Volume des applications
- Temps de mise au point d'une application/système
  $\Rightarrow$ le compilateur a un rôle clé pour produire rapidement et automatiquement un code exécutable satisfaisant les besoins/contraintes

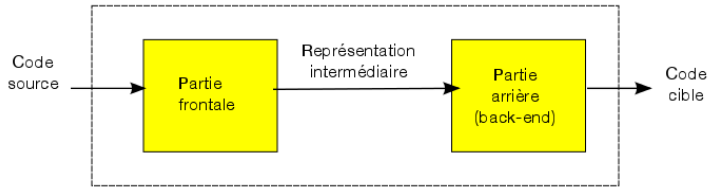
Comment produire un code performant pour une architecture donnée ? Comment tirer profit de la puissance de calcul offerte ? Est ce facile ?

Compilation

- Compilation : série de phases qui analysent séquentiellement une forme de programme et qui en synthétisent une nouvelle
- En général
  - L'entrée est un programme source vue comme une séquence de caractères
  - La sortie est un code objet qui peut être lié à d'autres, un code exécutable qui peut être chargé en mémoire pour être exécuté
- La traduction doit être
  - être correcte : respecter la sémantique du programme
  - cacher la complexité du matériel
  - produire un code efficace : exploiter les ressources tout en respectant les contraintes
- La structure des compilateurs n'a pas beaucoup changé depuis les années 50

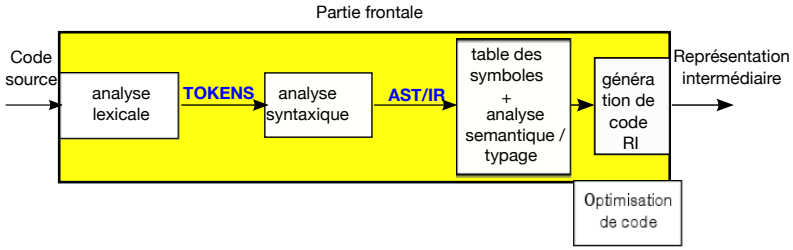
Structure d'un compilateur

- Partie dépendante du langage est le frontal/front-end
- Partie dépendante de la cible est la partie arrière/back-end
- NB : forme de programme en entrée (source, asm) et nouvelle forme de programme en sortie (C pre-processé, asm, objet, executable).



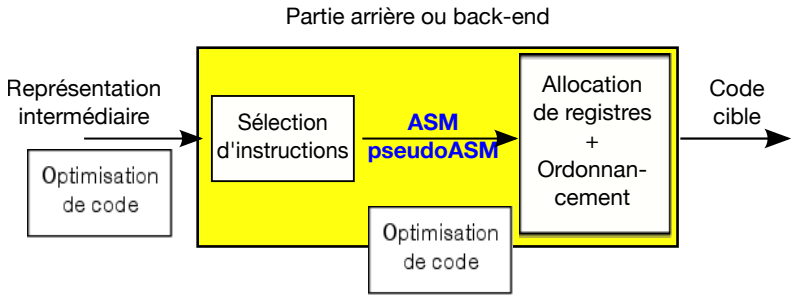
Frontal d'un compilateur

- Partie dépendante du langage
- Etapes clé
  - Analyse lexicale : analyse la séquence de caractères composant le code et produit une série de tokens
  - Analyse syntaxique : analyse la séquence de tokens et produit une représentation intermédiaire (IR) + table des symboles
  - Analyse sémantique : à partir de IR et table des symboles vérifie des proriétés sémantiques requises par le langage (ex : identifiant déclaré et utilisé de manière cohérente)
  - Génération de code intermédiaire (code cible directement si compilateur basique)



Coeurs et back-end d'un compilateur

- Coeur = partie indépendante du langage et de la cible.
- Back-end :
  - Génération de code = sélection d'instructions dans le jeu d'instructions de la cible + implantation des données
  - Allocation de registres et ordonnancement des instructions



Génération de code performant

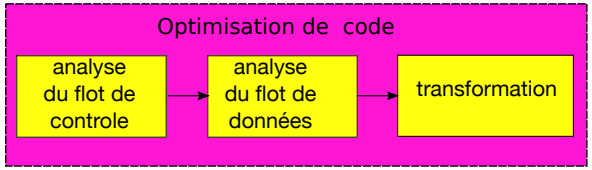
- Production de code performant, exploitant bien les ressources de la cible → les compilateurs appliquent des optimisations de code pour adapter les calculs à la microarchitecture cible.
- Optimisation ou plutôt application de transformation de code
- Transformation nécessite phases d'analyse préalables pour récupérer information sur le code
- Pas de code performant sans optimisation de code
- Pas de code performant sans prendre en compte l'architecture cible

Classification des optimisations

- Optimisations indépendantes de l'architecture cible
  - Réduction du nombre d'opérations statiques ou dynamiques/de leur coût
  - Elimination de sous-expression commune, propagation de copies, élimination du code inutile
- Optimisations dépendantes de l'architecture : besoin des caractéristiques de l'architecture cible, adaptation des calculs aux ressources, pour tirer profit des ressources disponibles
  - Ordonnancement des instructions
  - Allocation de registres
  - Optimisation pour la hiérarchie mémoire

Analyses de code et représentation du code

- Les analyses réalisées par un compilateur pour récupérer information sur le code sont de plusieurs natures :
  - Analyse du flot d'exécution i.e de la structure du code
  - Analyse du flot de données i.e utilisation des variables
  - Analyse d'alias i.e cibles potentielles des variables de type pointeur
- Différentes représentations du programme
  - Graphe de flot d'exécution : représente le code et les différents chemins d'exécution
  - Graphe de flot de données ou de dépendance : représente les dépendances entre les opérations du programme ou des données manipulées/calculées par le programme



Langage machine et jeu d'instructions

- Un programme de haut niveau $\rightarrow$ prog binaire = ensemble d'instructions compréhensibles par le processeur cible.
- Ces instructions sont dites en langage machine.
- Chaque processeur dispose d'un ensemble de traitements qui s'appuient sur son architecture (registres qu'il contient, organisation de la mémoire qu'il supporte, format des instructions qui a été défini en même temps que son architecture).

Jeu d'instructions

- La vue externe d'un processeur peut être définie par l'ensemble des instructions qu'il est capable de traiter
- Ces instructions sont stockées en mémoire et codées en binaire
- Le jeu d'instructions d'un processeur (ISA) est la donnée :
  1. de l'ensemble des instructions qu'il peut effectuer
  2. du codage de ces instructions en binaire

Qu'est ce qu'une instruction ?

- Instruction en langage machine = commande donnée au processeur
  - 1 traitement à effectuer
    - Opération mise en jeu (addition, opération logique, opération mémoire)
    - Opérandes : opérande(s) source(s), opérande destination (pas toujours)
  - Quelle sera la prochaine instruction à exécuter :
    - Modèle implicite séquentiel : la prochaine instruction est celle implantée en mémoire juste après.
      → l'adresse de la prochaine instruction à exécuter implicite dans ce cas.
    - Instructions spéciales pour spécifier l'adresse de la prochaine instruction sinon.

Les différentes classes d'instructions MIPS

Il y a 4 classes d'instructions :

- Les instructions arithmétiques et logiques (entière et flottante)
- Les instructions de transfert/accès mémoire
- Les instructions de rupture de séquence
- Les instructions système - macro-instructions

Instructions arithmétiques et logiques (1)

Ces instructions utilisent l'ALU pour réaliser le calcul.

- instructions arithmétiques : `add, addi, sub, mult, ...`
  - `add $2, $3, $4` :
    `$2 ← $3 + $4`
  - `addi $2, $3, 0x25` :
    `$2 ← $3 + 0x00000025`
  - `mult ($0), $4, $2` :
    `($hi, $lo) ← $4 * $2`
- instructions logiques : `or, ori, and, andi, xor, ...`
  - `or $2, $3, $4` :
    `$2 ← $3 |$4`
  - `andi $2, $3, 0xFF00` :
    `$2 ← $3 & 0x0000FF00`
  - `xor $4, $2, $3` :
    `$2 ← $3 ^ $4`

Instructions arithmétiques et logiques (2)

Ces instructions utilisent l'ALU pour réaliser le calcul.

- instructions de décalage : `sll, srl, sra, sllv, srav, srlv`
  - à gauche `sllv $2, $3, $4` :
    `$2 ← $3 « $4`
  - à droite 'signé' avec immédiat `sra $2, $3, 4` :
    `$2 ← $3 » 4`
- instruction d'affectation de registre : `lui, mfhi, mflo`
  - `lui $2, 0xABCD` :
    `$2 ← 0xABCD0000`

Transferts mémoire

- Ces instructions lisent ou écrivent des valeurs en mémoire, en indiquant :
  - l'adresse mémoire = contenu d'un registre + un immédiat
  - le registre contenant la valeur à écrire (écriture) ou où écrire la valeur lue en mémoire (lecture)
  - la taille du mot à lire/écrire via le code de l'opération.
- Les instructions en MIPS :
  - lecture réalisant `Rdst ← MEM[Rsrc + Immediat]` :
    - d'un mot `lw $4, 0($3)` :
      `$4 ← MEM[$3 + 0]`
    - d'un demi mot `lh $4, 2($3)`
    - d'un octet `lb $4, 12($3)`
  - écriture réalisant `Rsrc2 → MEM[Rsrc1 + Immediat]` :
    - d'un mot `sw $4, 0($3)` :
      `$4 → MEM[$3 + 0]`
    - d'un demi-mot `sh $4, 2($3)`
    - d'un octet `sb $4, 12($3)`

Rupture de séquence

- Ces instructions indiquent quelle est l'adresse de la prochaine instruction.
- Il y a deux types de saut :
  - les sauts inconditionnels qui ont toujours lieu :
    - `j etiquette_inst`
    - `jal ma_fonction`
    - `jr $31`
  - les sauts conditionnels qui sont effectués si et seulement une condition est vérifiée :
    - `beq $0, $4, etiquette_inst`
    - `bne $0, $4, etiquette_inst`
    - `bltz $4, etiquette_inst`

Macro-instructions

- Instructions non codées en binaire directement
- Correspondent à une ou plusieurs instructions du jeu d'instructions.
- Traduites en ces une ou plusieurs instructions au moment du codage.
  - `bnz $4, etiquette_inst`
  - `move $2, $4`
  - `li $2, 0x12345679`
  - `la $2, etiquette`

Exécution d'une instruction

L'exécution d'une instruction nécessite plusieurs étapes :

- Lire l'instruction en mémoire
- Décoder l'instruction (quelle opération, quels opérandes)
- Exécuter l'instruction (réaliser le traitement correspondant à l'instruction)
- Déterminer l'adresse de la prochaine instruction

Exemple de code assembleur MIPS

Comporte un ensemble de directives, d'instructions et d'étiquettes.

```
  . file  1 "test_01a.c"
  .section  .mdebug.abi32
  .previous
  .gnu_attribute  4,  1
  . abicalls
  . text                    #debut code
  . align   2
  . globl   main
  .set   nomips16
  .ent   main         #decl fonction
  .type main, @function
main:
  .frame  $fp,24,$31
  .mask 0x40000000,-4
  .fmask  0x00000000,0
  .set   noreorder
  .set   nomacro
```

```
.set   nomacro
addiu $sp,$sp,-24
sw  $fp,20($sp)
move $fp,$sp
 li   $2,2
sw  $2,8($fp)
lw   $2,8($fp)
move $sp,$fp
lw   $fp,20($sp)
addiu $sp,$sp,24
j  $31
nop       #delayed slot

.set   macro
.set   reorder
.end  main
.size  main, .-main
.ident   "GCC: (GNU) 4.5.2"
```

Représentation du code bas niveau

- Une fichier assembleur peut être représenté comme une liste de lignes de type `directive`, `label` ou `instruction` (cf. TP associé à ce cours)
- Une ligne est soit une directive, une instruction ou une étiquette
- Une instruction comporte entre autre un type, une opération, des opérandes sources et potentiellement un destination
- Un opérande est soit un registre, un label, un immédiat ou une expression
- Une fonction est alors un sous-ensemble de lignes d'un fichier
- Le compilateur manipule une structure appelée graphe de flot de contrôle (CFG) représentant le code d'une fonction (à construire en TP)
- Séquence d'instructions souvent manipulée appelée le bloc de base (à délimiter en premier)

Bloc de base : définition et calcul

- Bloc de base : séquence d'instructions comportant un seul point d'entrée (1er inst) et un seul point de sortie
- Entête : première instruction d'un bloc de base
- Algorithme de construction des blocs de base
  1. Déterminer les entêtes
  2. Déterminer les BBs
- Remarque : la notion de BB existe à tous les niveaux (source, IR, asm)

Bloc de base : calcul

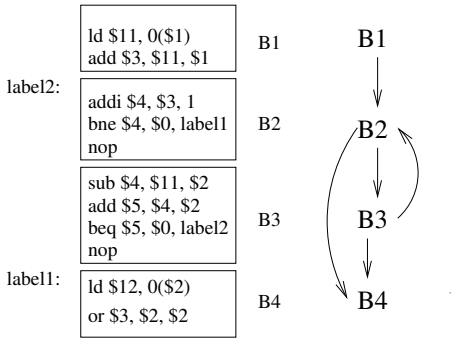
1. Déterminer les entêtes
   - La première instruction du code d'une fonction est une entête
   - Toute instruction cible d'un saut est une entête
   - Toute instruction qui suit un saut (ou dernière instruction des delayed slots) est une entête
2. Déterminer les BBs
   - Pour chaque entête, le BB correspondant est constitué de toutes les instructions qui la suivent jusqu'à l'entête suivante, non comprise (ou la fin de la fonction).

CFG : graphe de contrôle de flot

- Les liens entre les BB traduisent le flot de contrôle du programme
- CFG : graphe reflétant le flot de contrôle et manipulé par le compilateur
- Les noeuds sont des blocs de base
- Il y a un arc entre deux blocs BB1 et BB2 si
  - il y a un saut de BB1 vers BB2
  - si BB2 suit BB1 dans le l'ordre du programme et BB1 ne se termine pas par un saut inconditionnel (forme j etiquette)
- Calculé pour les fonctions en général, CFG inter-procédural possible aussi.
- Remarque : la notion de CFG existe à tous les niveaux (source, IR, asm)

CFG d'une fonction : exemple

```
  ld  $11, 0($1)
  add $3, $11, $1
label2:
  add $4, $3, 1
  bne $4, $0, label1
  nop
  sub $4, $11, $2
  add $5, $4, $2
  beq $5, $0, label2
  nop
label1:
  ld  $12, 0($2)
  or  $3, $2, $2
```



CFG d'une fonction : exemple

```
  ld  $11, 0($1)
  add $3, $11, $1
label2:
  add $4, $3, 1
  bne $4, $0, label1
  nop
  sub $4, $11, $2
  add $5, $4, $2
  beq $5, $0, label2
  nop
label1:
  ld  $12, 0($2)
  or  $3, $2, $2
```

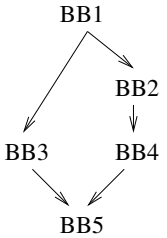
CFG d'une fonction : exemple

```
  ld  $11, 0($1)
  add $3, $11, $5
  addi $4, $3, 1
  bne $4, $0, label1
  nop
  j  label2
  nop
label1:
  sub $4, $11, $2
  add $3, $4, $2
  sw $3, 0($1)
  j  label3
  nop
label2:
  ld  $12, 0($2)
  or  $3, $4, $12
  sw $3, 0($1)
label3:
```

CFG d'une fonction : exemple

```
  ld  $11, 0($1)
  add $3, $11, $5
  addi $4, $3, 1
  bne $4, $0, label1
  nop
  j  label2
  nop
label1:
  sub $4, $11, $2
  add $3, $4, $2
  sw $3, 0($1)
  j  label3
  nop
label2:
  ld  $12, 0($2)
  or  $3, $4, $12
  sw $3, 0($1)
label3:
```

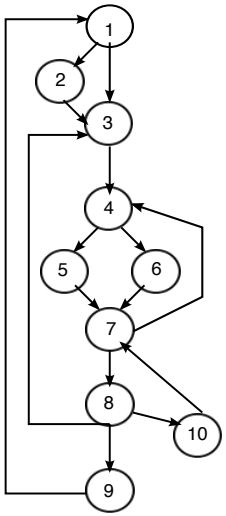
```
ld $11, 0($1)
add $3, $11, $5
addi $4, $3, 1
bne $4, $0, label1
nop                          BB1
j label 2
nop                          BB2
label1:
sub $4, $11, $2
add $3, $4, $2
sw $3, 0($1)
j label3
nop                          BB3
label2:
ld $12, 0($2)
or $3, $4, $12
sw $3, 0($1)                 BB4
label3:
                             BB5
```



Relation de dominance

- Relation de dominance dans un graphe : noeud `d` domine le noeud `i` si tous les chemins de l'entrée à `i` passent par `d`
- On note `d dom i` la relation
- La relation est reflexive, transitive et antisymetrique
- Calcul des dominants : `a` domine `b` si
  - `a` = `b`
  - `a` est l'unique prédécesseur immediat de `b`
  - `b` a plusieurs prédécesseurs immédiats `c` et $\forall$ `c` $\in$ `Pred(b)`, `c` $\neq$ `a` et `a dom c`

Dominance : exemple

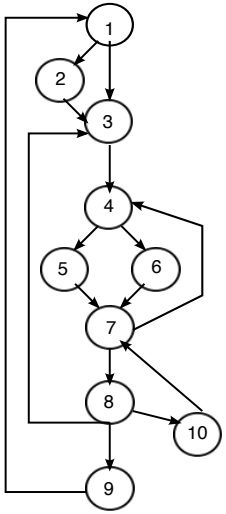


noeud	dominateurs
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Relation de dominance immédiate

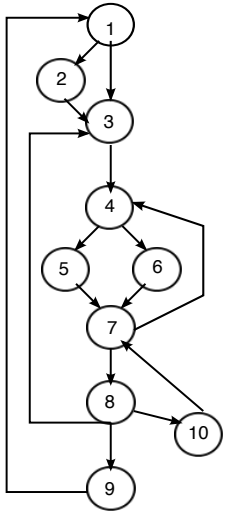
- Relation de dominance immédiate notée `idom` :
  `a idom b` avec `a` $\neq$ `b` ssi
  `a dom b` et $\nexists$ `c`, `a dom c` et `c dom b`.
- Le dominateur immédiat est unique (s'il existe).
- La relation de dominance immédiate forme un arbre de noeuds d'un CFG dont la racine est le noeud d'entrée et les arcs représentent la relation de dominance immédiate
- Les chemins dans cet arbre indiquent alors toutes les dominances

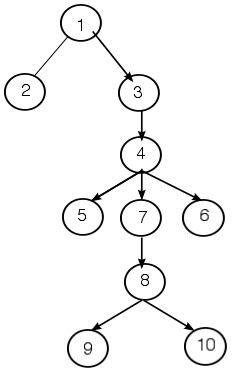
Dominance immédiate et arbre de dominance



noeud	dominateurs	idom
1	1	
2	1, 2	
3	1, 3	
4	1, 3, 4	
5	1, 3, 4, 5	
6	1, 3, 4, 6	
7	1, 3, 4, 7	
8	1, 3, 4, 7, 8	
9	1, 3, 4, 7, 8, 9	
10	1, 3, 4, 7, 8, 10	

Dominance immédiate et arbre de dominance





Calcul des dominants dans un CFG

```
Input : N : set of Node, Pred, Succ : Node -> set of Node, r : Node
Output : Domin : Node -> set of Node

D, T : set of Node ; n, p, s : Node ; WorkingList : List of Node

change := true
Domin(r) = {}                  /* traitement de la racine */
WorkingList.push_back(r) /*    initialisation    liste des BB a traiter */
for each n in N - {r} do /*    initialisation    pour tous les autres noeuds */
  Domin(n) := N

do while (WorkingList <> empty)    /* iteration tant liste non vide */
 change := false
 n := Head_and_Remove(WorkingList) /* premier element qui est enleve*/
 T := N
 for each p in Pred(n) do  /* considerer tous les predecesseurs de n */
   T  +:= Domin(p)          /* intersection des dominants des predecesseur*/
 D := T + {n}               /* ajout du noeud lui-meme */
 if D <> Domin(n) then    /* detection de changement */
      Domin(n) := D
      change := true

 if change = true              /* si changement ajout des successeurs de n */
   for each s in Succ(n) do
       WorkingList.push_back(s)
od
return Domin
```

Relation de post-dominance

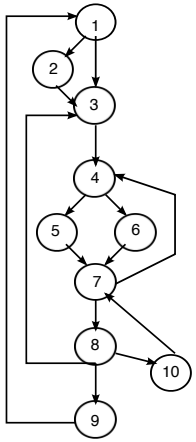
- Relation de post-dominance dans un graphe : noeud `d` post-domine le noeud `i` si tous les chemins de `i` à la sortie passent par `d`
- On note `d pdom i` la relation de post-dominance
- La relation est reflexive, transitive et antisymetrique
- Calcul des post-dominants : `a` post-domine `b` si
  - `a` = `b`
  - `a` est l'unique successeur immediat de `b`
  - `b` a plusieurs successeurs immédiats `c` et $\forall$`c` $\in$ `Succ(b)`, $c \neq$ `a` et `a pdom c`

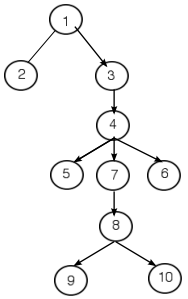
Relation de post-dominance immédiate

- Relation de post-dominance immédiate notée `idom` :
  `a ipdom b` avec $a \neq b$ ssi
  `a pdom b` et $\nexists$ `c`, `a pdom c` et `c pdom b`.
- Le post-dominateur immédiat est unique (s'il existe).
- La relation de post-dominance immédiate forme un arbre de noeuds d'un CFG dont la racine est le noeud de sortie et les arcs représentent la relation de post-dominance immédiate
- Les chemins dans cet arbre indiquent alors toutes les post-dominances

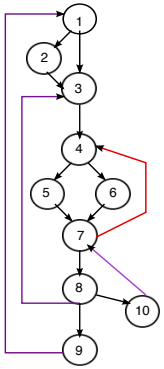
Arcs retour

- On appelle un arc retour `BBy` $\rightarrow$ `BBx` dans un graphe `G = (N, E)` un arc tel que `BBx dom BBy`





Boucles naturelles

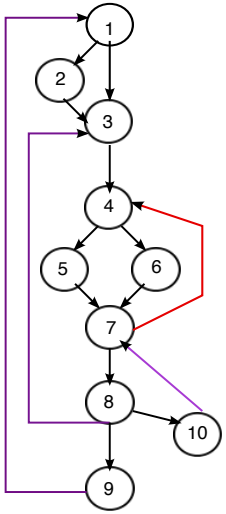


- Une boucle naturelle induite par un arc retour `BBy` $\rightarrow$ `BBx` est le sous-graphe `sG = (sN,sE)` de `G` avec
  - `BBx` est l'unique entête de la boucle
  - `sN` = ensemble avec `BBx` + les noeuds de `N` depuis lesquels `BBy` est accessible sans passer par `BBx`
  - `sE` = ensemble des arcs de `E` connectant les blocs de base `sN`

Boucles naturelles

- Une boucle naturelle induite par un arc retour `BBy` $\rightarrow$ `BBx` est le sous-graphe `sG = (sN,sE)` de `G` avec
  - `BBx` est l'unique entête de la boucle
  - `sN` = ensemble avec `BBx` + les noeuds de `N` depuis lesquels `BBy` est accessible sans passer par `BBx`
  - `sE` = ensemble des arcs de `E` connectant les blocs de base `sN`
- On peut construire le sous-graphe associé à un arc retour en partant de `BBy` et en ajoutant ses prédécesseurs jusqu'à `BBx`.
- Un arc avec un bloc prédécesseur ne définit pas forcément une boucle naturelle.

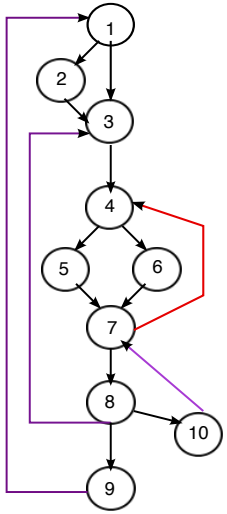
Arc retour et boucles naturelles : exemple



- On peut construire le sous-graphe associé à un arc retour en partant de `BBy` et en ajoutant ses prédécesseurs jusqu'à `BBx`.

arc	boucle naturelle
$10 \rightarrow 7$	
$7 \rightarrow 4$	
$8 \rightarrow 3$	
$9 \rightarrow 1$	

Arc retour et boucles naturelles : exemple

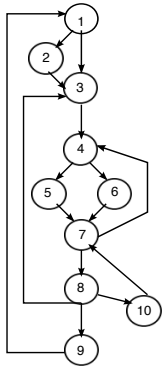


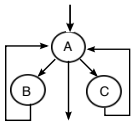
- On peut construire le sous-graphe associé à un arc retour en partant de `BBy` et en ajoutant ses prédécesseurs jusqu'à `BBx`.

arc	boucle naturelle
$10 \rightarrow 7$	7, 8, 10
$7 \rightarrow 4$	4, 5, 6, 7, 8, 10
$8 \rightarrow 3$	tous sauf 1, 2, 9
$9 \rightarrow 1$	tous

Boucles (naturelles) internes et imbrications

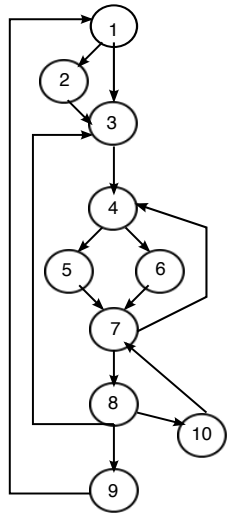
- Si deux boucles n'ont pas le même bloc en-tête : imbriquées ou disjointes
- S'il n'y a aucun arc arrière entre ses blocs autre que l'arc avec l'entête = boucle interne
- Si en-tete commune on ne peut rien dire sans connaitre le code (une seule ou deux boucles).





Extended basic blocs

- Blocs de base étendu/superblocs
- Un seul point d'entrée `p`
- Plusieurs points de sortie
- A un forme d'arbre de BB
- Seul `p` a $|\texttt{Pred(p)}| \geq 1$ dans le CFG
- Utilisés dans certaines optimisations



Extended basic blocs

- Blocs de base étendu/superblocs
- Un seul point d'entrée `p`
- Plusieurs points de sortie
- A un forme d'arbre de BB
- Seul `p` a $|\texttt{Pred(p)}| \geq 1$ dans le CFG
- Utilisés dans certaines optimisations