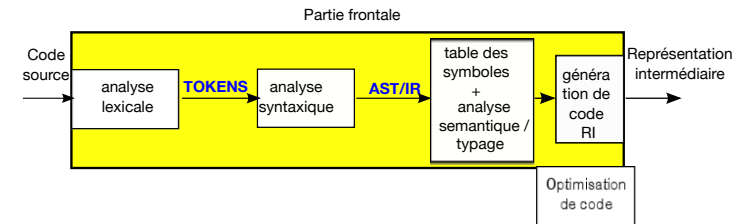


## Elimination de redondance Analyse du flot de données

Cours 3. UE Compilation Avancée

Karine Heydemann  
karine.heydemann@lip6.fr

## Code intermédiaire et optimisation



- ▶ La génération de code intermédiaire produit un code dit à 3 adresses, indépendant du langage source et cible
- ▶ Cette représentation intermédiaire est au cœur des compilateurs, varie d'un compilateur à un autre (niveau HIR/MIR/LIR)
- ▶ L'optimisation du code à ce niveau nécessite l'analyse du flot de contrôle mais aussi du flot de données
- ▶ L'élimination de redondance est appliquée à ce niveau.

## Code intermédiaire

### ▶ Code intermédiaire 3 adresses de la forme

- ▶  $z := op \ x \ y$
- ▶  $z := op \ x$
- ▶  $z := x \text{ (copie)}, z := a[i] \text{ (lecture)} \ a[i] := z \text{ (écriture)}$
- ▶ `si cond goto inst` (saut conditionnel)
- ▶ `goto inst` (saut inconditionnel)

```

(1) prod := 0
(2) i := 1
(3) t1 := 4 * i
(4) t2 := a[i]
(5) t3 := 4 * i
(6) t4 := b[t3]
(7) t5 := t2 * t4
(8) t6 := prod + t5
(9) prod := t6
(10) t7 := i + 1
(11) i := t7
(12) si i <= 20 goto (3)

debut
  prod := 0;
  i := 1;
  faire
    prod := prod + a[i] * b[i];
    i++;
  tant que i <= 20
fin
  
```

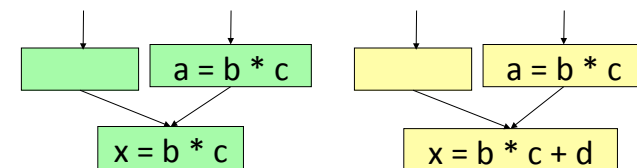
## Exemples de redondance

```

read(i)
j = i+1
k = i
n = k+1
  
```

```

i = 2
j = i*2
k = i+2
  
```



## Elimination de redondance

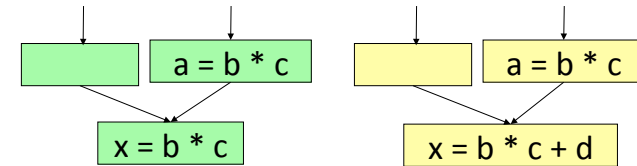
- ▶ Détermination que 2 calculs sont équivalents pour en éliminer un.
- ▶ Plusieurs types d'élimination de redondance
- 1. Value numbering
  - ▶ Associe un numéro symbolique aux calculs et identifie les expressions ayant le même numéro
- 2. Elimination de sous-expressions communes
  - ▶ Identifie que des expressions ont des opérandes avec le même nom
- 3. Propagation de constante/de copie
  - ▶ Identifie les variables avec valeur constante/copie d'une autre et utilise les constantes/la copie à la place des variables
- 4. Elimination de redondance partielle
  - ▶ Insère des calculs dans les chemins pour convertir de la redondance partielle en redondance complète

## Elimination de redondance

```
read(i)
j = i+1
k = i
n = k+1
```

```
i = 2
j = i*2
k = i+2
```

Les différentes approches détecteront les redondances dans quelques cas



## Value numbering

- ▶ Associe un numéro symbolique aux calculs et identifie les expressions/variables ayant le même numéro
  - ▶ Possible de faire de la propagation de constante en même temps
  - ▶ Indiquer si variable constante, associer la valeur le cas échéant
- ▶ Optimisation locale : appliquée à un bloc de base
- ▶ Optimisation super-local : appliquée à un superbloc (EBB cf. cours1)
- ▶ Optimisation globale : appliquée au graphe de flot de contrôle

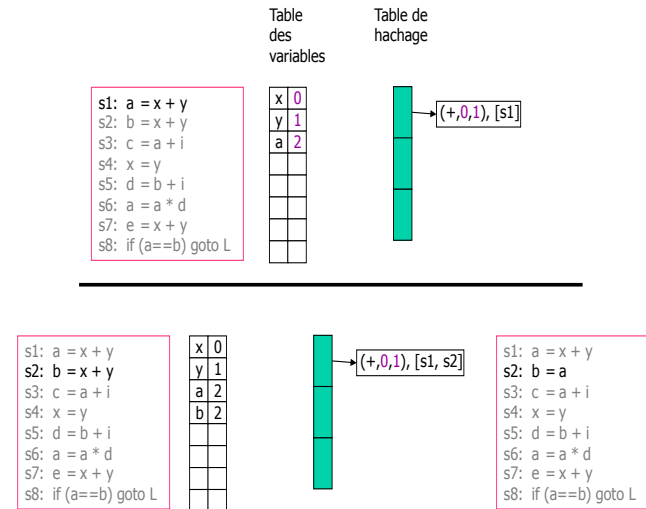
## Local value numbering (1)

- ▶ Scan linéaire des instructions (RI) d'un bloc de base
- ▶ Maintient d'une table associant à chaque variable  $x$  un numéro  $xv$ , si c'est une constante sa valeur
- ▶ Utilisation d'une table de hachage pour le déterminer le numéro des opérations/expressions et les instructions où elles apparaissent

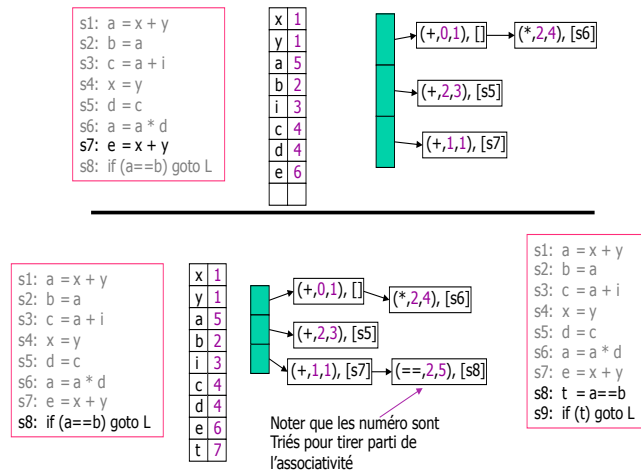
## Local value numbering (2)

- ▶ Pour chaque instruction  $\circ = \langle \text{op}, x, y \rangle$ , chercher dans la table les numéros  $x_v$  et  $y_v$  des opérandes, en donner un si n'existe pas
- ▶ Si l'instruction n'a pas d'opérande destination, créer un temporaire (exemple des conditions dans les branchements)
- ▶ Si  $x$  et  $y$  sont identifiés comme constantes, évaluer le résultat (si faisable) et remplacer l'opération par un chargement d'immédiat.
- ▶ Si  $x$  ou  $y$  constante, appliquer des simplifications algébriques le cas échéant.
- ▶ Chercher s'il existe un numéro pour l'opération finale, numéros d'opérande triés (utile si opération commutative) :
  - ▶ si oui remplacer l'opération avec la variable ayant le numéro trouvé, associer à  $\circ$  le même numéro.
  - ▶ sinon associer un nouveau numéro
- ▶ Enlever des opérations utilisant  $\circ$  dans la table de hachage

## Exemple

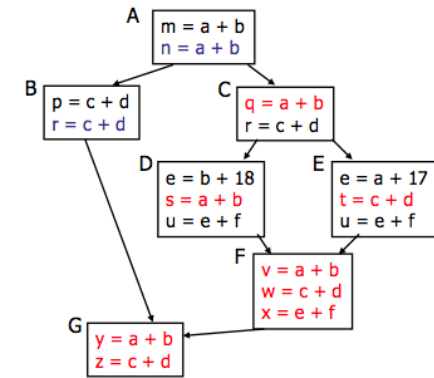


## Rappel : extended basic blocs



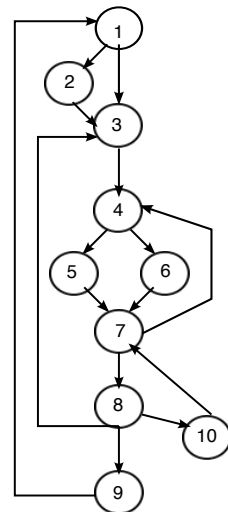
## Rappel : extended basic blocs

- ▶ Par bloc de base
  - élimination d'expressions
  - propagation des constantes
  - simplification algébrique
  - constant folding (evaluation)
- ▶ Aucun effet inter-bloc
- ▶ Manque des opportunités



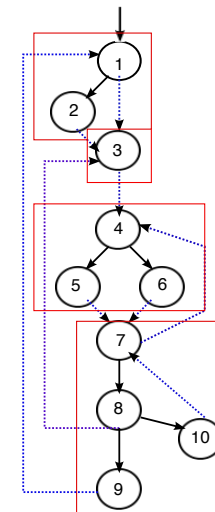
## Rappel : extended basic blocs

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



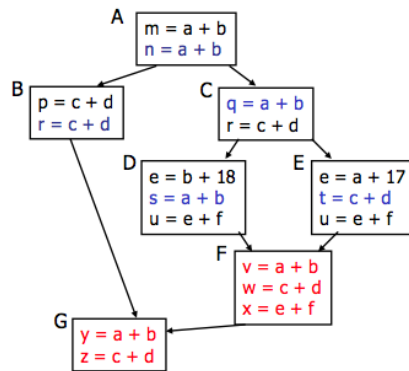
## Rappel : extended basic blocs

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



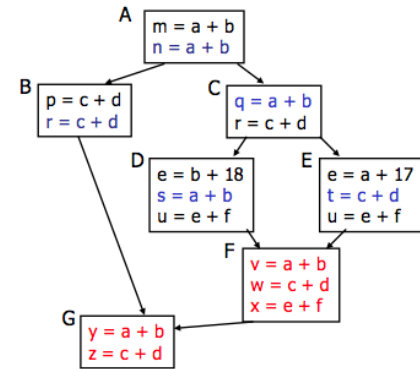
## Value numbering super-local

- Application aux superblocs d'un CFG
- Réutilise les calculs du bloc prédécesseur (unique)
- Tables finales de A = tables initiales pour B et C
- Tables finales de C = initiales pour D et E
- Calcul sur  $\{A, B\}$  et  $\{A, C, E\}$
- Ne fait rien pour F et G



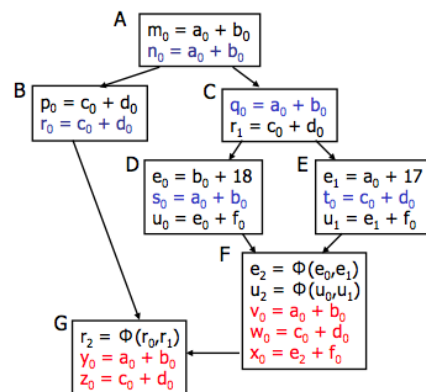
## Value numbering superlocal : extension ?

- Utiliser la sortie du bloc dominant immédiat pour F et G ?
- Soit C pour F et A pour G
- Pb : tables sont elles correctes en entrée de F et G ?



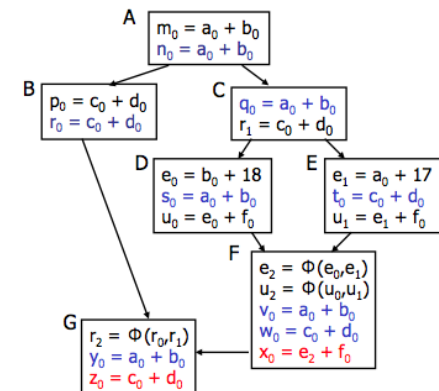
## Single Static Assignment

- Solution : indiquer les variables,  $a_i$  valeur de la  $i$ ème définition de  $a$
- 1ère utilisation (sans def) ou 1ère définition (sans utilisation antérieure) indiquée avec 0
- A la réunion de 2 noeuds, redéfinir une variable définie dans chacune des branches  
⇒ Utilisation de la fonction  $\phi$ , c'est la forme SSA
- ⇒ Un nom correspond à une et une seule définition/valeur



## Dominator Value Numbering Technique

- Utiliser la forme SSA
- Pour un bloc  $x$ , utiliser sortie de  $IDom(x)$
- Expressions redondantes et disponibles en F et G trouvées !



## Value numbering : bilan

- ▶ Algorithme local, étendu aux superblocs
- ▶ DVNT : utilisation de la forme SSA et des dominants immédiats
- ▶ Propagation le long des arcs avant seulement
- ▶ Pas d'approche globale

⇒ analyse de flot de données fournit une solution globale à l'analyse des expressions redondantes

⇒ Allons voir l'analyse de flot de données !

## Analyse globale des expressions disponibles

- ▶ But : utiliser l'analyse de flot de données pour trouver les sous-expressions communes
- ▶ Idée : calculer les expressions disponibles (*available expressions*) au début de chaque bloc de base
- ▶ Evite de ré-évaluer une expression disponible en utilisant une opération de copie

## C'est quoi une expression disponible ?

- ▶ Une expression  $e$  est **définie** en un point  $p$  si sa valeur est calculée au point  $p$  (site de définition)
- ▶ Une expression  $e$  est **tuée** (killed) au point  $p$  si un de ses opérandes est défini au point  $p$  (site de disparition)
- ▶ Une expression  $e$  est **disponible** au point  $p$ , si tout chemin menant à  $p$  contient une définition de  $e$  et que  $e$  n'est pas tuée entre cette définition et  $p$

## Ensemble des expressions disponibles

Pour chaque bloc de base  $b$

- ▶  $AVAILin(b)$  = ensemble des expressions disponibles en entrée de  $b$
- ▶  $KILL(b)$  = ensemble des variables tuées/définies par  $b$
- ▶  $NKILL(b)$  = ensemble des expressions de  $AVAILin(b)$  non tuées par  $b$ , calculé à partir de  $AVAILin(b)$  et  $KILL(b)$
- ▶  $DEF(b)$  = ensemble des expressions définies par  $b$  et non tuées dans  $b$  après leur définition
- ▶  $AVAILout(b)$  = ensemble des expressions disponibles en sortie de  $b$  qui vaut  $NKILL(b) \cup GEN(b)$

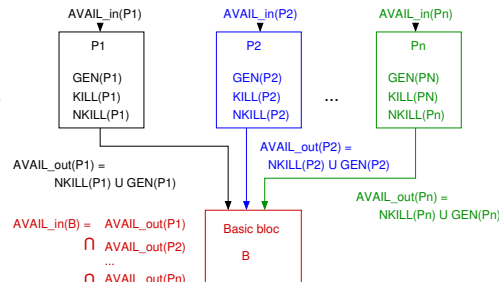
## Calcul des expressions disponibles

Pour chaque bloc de base  $b$

$$\text{AVAILin}(b) = \bigcap_{x \in \text{Pred}(b)} \text{AVAILout}(x)$$

- C'est l'intersection des expressions disponibles en sortie des prédécesseurs (assurance de leur "disponibilité")

- Cela donne un ensemble d'équations à résoudre : c'est un problème de flot de données



## Calcul de GEN, KILLED et NKILL

Pour chaque bloc de base  $b = i_1, i_2, \dots, i_k$

$$\text{DEF}(b) := \emptyset$$

$$\text{KILLED} := \emptyset$$

For  $j$  from  $k$  to  $1$

//  $i_j$  de la forme  $x := \text{op } y \ z$

If  $(y \notin \text{KILLED} \text{ and } z \notin \text{KILLED})$  Then

$$\text{DEF}(b) := \text{DEF}(b) \cup (\text{op}, y, z)$$

$$\text{KILLED} := \text{KILLED} \cup \{x\}$$

$$\text{NKILL}(b) := \{\text{AVAILin}(b)\}$$

For each  $e$  expression  $\in \text{NKILL}(b)$

For each  $v$  variable  $\in e$

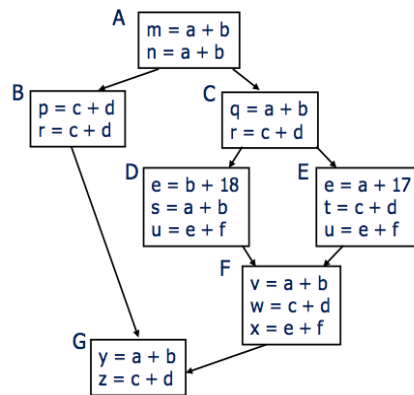
If  $v \in \text{KILLED}$  Then

$$\text{NKILL}(b) := \text{NKILL}(b) \setminus \{e\}$$

## Calcul global pour un CFG

```

r = root(G)
worklist = empty_FIFO
push(r, worklist)
while (worklist ≠ ∅)
  b := pop(worklist)
  compute AVAILin(b)
  compute AVAILout(b)
  If AVAILout(b) changed
  Then
    push(worklist, ∪_{s ∈ Succ(b)} s)
    
```



## Expressions disponibles dans un CFG

- On a donc les expressions disponibles au début de chaque bloc
- On peut donc optimiser le bloc pour utiliser des copies des expressions plutôt que les recalculer
- Le calcul des expressions disponibles est un exemple d'analyse de flot de données
- Il en existe d'autres, avec des calculs similaires (cela ne vous rappelle rien ?)

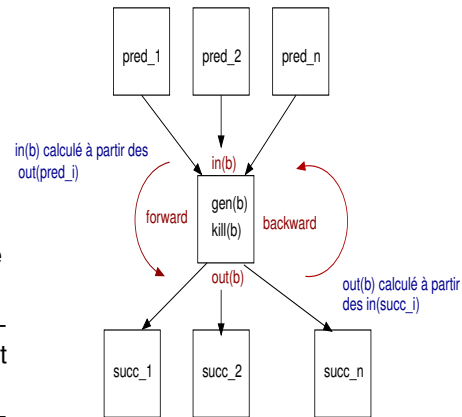
## Analyse de flot de données dans un CFG

Calcul d'un ensemble d'informations pour chaque bloc  $b$

- ▶  $in(b)$  information en entrée de  $b$
- ▶  $out(b)$  information en sortie de  $b$
- ▶  $gen(b)$  information engendrée par  $b$
- ▶  $kill(b)$  information tuée par  $b$

Equations pour chaque bloc reliant  $in(b)$  et  $out(b)$  utilisant  $gen(b)$  et  $kill(b)$

Sens de l'analyse : avant ou arrière



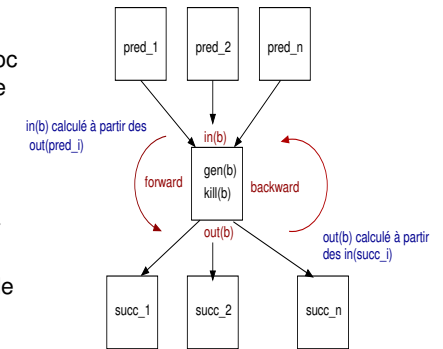
## Analyse de flot de données en pratique

Si analyse avant :

- ▶  $out(b) = F(in(b), gen(b), kill(b))$
- ▶ Commencer à la racine du CFG
- ▶ Ajouter les successeurs d'un bloc  $b$  traité à la working liste lorsque  $out(b)$  a changé (pb des boucles).

Si analyse arrière/backward

- ▶  $in(b) = F(out(b), gen(b), kill(b))$
- ▶ Commencer avec le(s) bloc(s) de sortie du CFG
- ▶ Ajouter les prédécesseurs d'un bloc  $b$  traité à la WL lorsque  $in(b)$  a changé (pb des boucles).



## Analyses de flot de données dans un CFG

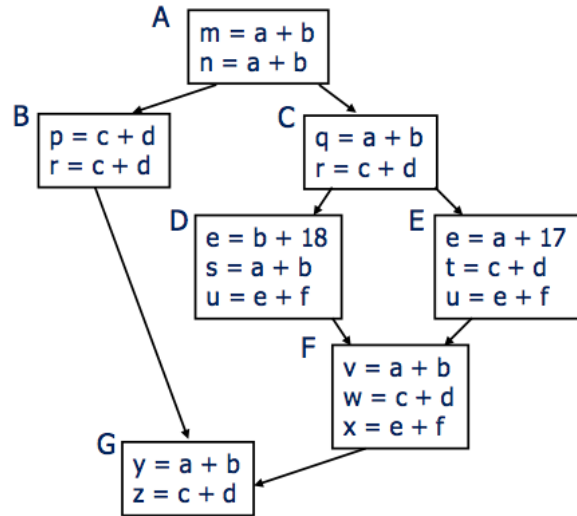
- ▶ Reaching definition/definitions atteignant un bloc :
  - ▶ Une définition  $d$  d'une variable  $v$  **atteint** une opération  $i$  si  $i$  lit  $v$  et qu'il y a un chemin de  $d$  à  $i$  le long duquel  $v$  n'est pas redéfinie
  - ▶ Trouve tous les points de définition possibles pour une variable utilisée dans une expression
- ▶ Variable vivante
  - ▶ Une variable  $v$  est **vivante** en un point  $p$  s'il existe un chemin jusqu'à la sortie le long duquel est lue avant d'être potentiellement redéfinie
  - ▶ Trouve toutes les variables dont on a potentiellement besoin de la valeur
- ▶ Very Busy Expressions :
  - ▶ Une expression  $e$  est dite **"very busy"** en un point  $p$  si  $e$  est évaluée et utilisée le long de tous les chemins quittant  $p$ , et la valeur produite donnera le même résultat
  - ▶ Signifie que l'on peut calculer  $e$  au point  $p$

## Analyses de flot de données dans un CFG

- ▶  $gen(b)$  et  $kill(b)$  définis spécifiquement pour une analyse
- ▶ Expression disponible (AvailExpr) :
  - ▶ Analyse avant et sure
  - ▶  $in(b) = \bigcap_{p \in Pred(b)} out(p)$
- ▶ Reaching definition/Definitions atteignant un bloc (RD) :
  - ▶ Analyse avant et potentielle
  - ▶  $in(b) = \bigcup_{p \in Pred(b)} out(p)$
- ▶ Variable vivante (Liveness)
  - ▶ Analyse arrière et potentielle
  - ▶  $out(b) = \bigcup_{s \in Succ(b)} in(s)$
- ▶ Very Busy Expressions (VBE) :
  - ▶ Analyse arrière et sure
  - ▶  $out(b) = \bigcap_{s \in Succ(b)} in(s)$



## Analyses de flot de données dans un CFG



A vous de "jouer" !

## Analyses de flot de données et optimisation

- ▶ Permettent l'application d'optimisations
  - ▶ Propagation de constante (RD)
  - ▶ Code hoisting (VBE)
  - ▶ Dead code elimination (Liveness)
  - ▶ Elimination d'expression commune (Avail Expr)
- ▶ Indépendantes de la cible
- ▶ Utiles (nécessaires !) aussi après application de certaines optimisations (déroulage de boucle)

## Propagation de constantes

```

int main(){
  int a = 5;
  int b = 7;
  int c, d;
  init_data(&c, &d);
  c = d + b;
  d = c - a;
  use_data(&c, &d);
  return 0;
}
  
```

définition de a et b

utilisation de a et b

pas de modification de a et b par la fonction : on peut les remplacer par leur valeur

## Elimination des sous-expressions communes

Une expression (ou sous-expression) est redondante en un point du programme si elle a déjà été calculée en un point précédent du programme

```

int main(){
  int a, b, c, d;
  init_data(&a, &b, &c, &d);
  b = a + 2 + 3*d;
  c = 4*b;
  if (b>c)
    b = 1;
  d = a + 2 + 3*d;
  use_data(&a, &b, &c, &d);
  return 0;
}
  
```

même calcul, a et d non modifiées sur l'ensemble des chemins entre les 2 instructions

=> réutilisation du calcul en utilisant un registre ou variable tmp.

```

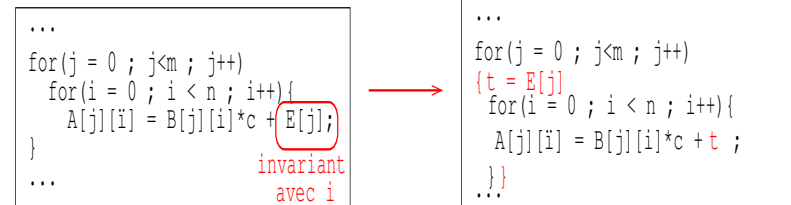
int main(){
  int a, b, c, d;
  init_data(&a, &b, &c, &d);
  tmp = a + 2 + 3*d; b = tmp;
  c = 4*b;
  if (b>c)
    b = 1;
  d = tmp;
  use_data(&a, &b, &c, &d);
  return 0;
}
  
```

## Optimisations pour les boucles

- ▶ C'est là que l'exécution passe le plus de temps potentiellement
- ▶ Normal de chercher à les optimiser en premier (loi d'Amdhal)
- ▶ Les optimisations précédentes sont utiles pour éliminer des calculs dans le corps d'une boucle
- ▶ Il en existe d'autres spécifiques aux boucles : celles qui reposent sur les variables d'induction et celles qui cherchent à éliminer les calculs invariants

## Extraction des calculs invariants

- ▶ Un calcul/une expression qui, dans une boucle, ne change pas d'une itération à une autre est dit **invariant**.
- ▶ Il peut être extrait du corps de la boucle.



## Variable d'induction

- ▶ Variable d'induction = variable incrémentée d'un pas constant à chaque tour de boucle
- ▶ Variable d'induction primaire  $i$  = une seule affectation et de la forme  $i = i + c$  ou  $i = i - c$  avec  $c$  invariant de boucle
- ▶ Variable d'induction secondaire  $j$  = une seule affectation et de la forme  $j = a*i$  ou  $j = b+i$  avec  $i$  variable d'induction et  $a, b$  des invariants
- ▶ Détermination des variables d'induction primaires puis secondaires dans les boucles
- ▶ Utilisation pour optimiser les calculs au sein de la boucle

```

k = n;
i = 1;
while (i <= n){
  j = 2*i + 1;
  a[i] = b[k] + a[j];
  k = k - 1;
  i = i + 1;
}

```

$i$  et  $k$  sont des variables d'induction primaires

$j$  est une variable d'induction secondaire de la famille de  $i$

## Réduction de force

La détermination des variables d'induction permet de réduire le coût de certains calculs dans une boucle en les remplaçant par des calculs moins coûteux.

```

k = n;
i = 1;
while (i <= n){
  j = 2*i + 1;
  a[i] = b[k] + a[j];
  k = k - 1;
  i = i + 1;
}

```

```

j = 1;
k = n;
i = 1;
while (i <= n){
  j = j + 2;
  a[i] = b[k] + a[j];
  k = k - 1;
  i = i + 1;
}

```

expression	réduction	type de données
$X*2$	$X + X$	int, float, double,...
$X^2$	$X * X$	int, float, double,...
$i*2^n$	$i \ll n$	int
$\text{len}(s1.s2)$	$\text{len}(s1) + \text{len}(s2)$	string