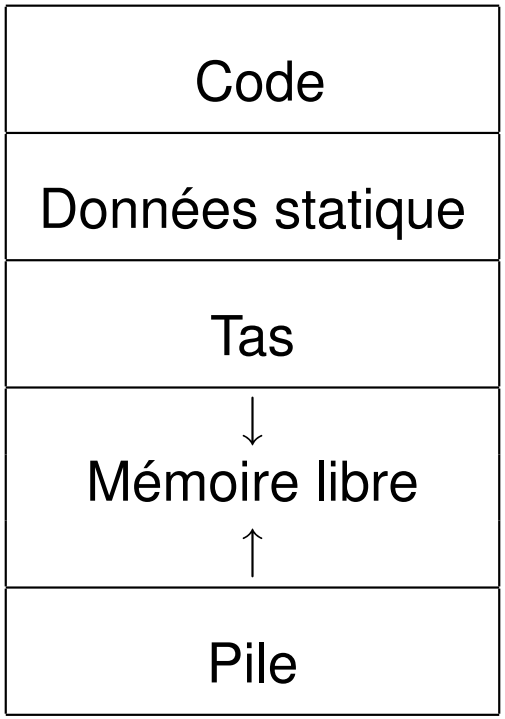
Environnement d'exécution et Optimisation 1

Sommaire

Organisation de la mémoire

Chaque programme s'exécute dans une portion de la mémoire attribuée par le système d'exploitation.



Quelques données sur la mémoire

- L'architecture de la machine propose différents niveaux de mémoire (registres, caches, mémoire vive, disque).
- Les mémoires les plus rapides sont en quantité limitées.
- Les mémoires les plus lentes sont chargés dans des mémoires plus rapides par bloc.
- Le compilateur devra prendre en compte la localité des données (mettre dans des endroits proches des données accédées dans un court laps de temps)

Principe

- Les nouvelles valeurs calculées au cours de l'exécution sont souvent allouées dans les tableaux d'activation sur la pile.
- Ce principe fonctionne si ce qui est alloué au cours de l'appel d'une procédure n'est plus accessible à la fin de la procédure.
- Ce principe ne permet pas de couvrir toutes les constructions de programme.
  - Lorsqu'une méthode crée dynamiquement des objets
  - Lorsqu'une fonction *f* renvoie en valeur une autre fonction, celle-ci peut faire référence à des variables locales de *f*, allouées dans son tableau d'activation qui est libéré à la sortie de *f*

Représentation des données structurées

- La manipulation dans la pile de données de taille variable est complexe.
  - Copier des valeurs peut alors être coûteux
- Manipuler des objets complexes via leur adresse est souvent plus commode
- Certaines instructions créent la structure
  Exemples:
  - Tableaux en CAML :
    Création explicite `[|1;2;3|]` ou via la commande `Array.create`
  - en C : initialisation des variables, utilisation de `malloc`
  - en Java : intruction `new`
- La valeur est ensuite manipulée par son adresse.
- Si une fonction renvoie une valeur structurée comme une adresse, il faut qu'elle soit allouée dans une zone persistante de la mémoire.

Allocation dynamique

- De nouveaux espaces mémoires créés dynamiquement au cours du programme; accessibles par le programme sans forcément être directement liés à une variable.
- La durée de vie de ces objets n'est pas limitée à la durée de vie des procédures : réservation de l'espace dans le tas.
- On distingue :
  - les données directes telles que les entiers qui seront allouées et manipulées dans la pile;
  - les données indirectes dont la manipulation se fera par l'intermédiaire d'une adresse dans la mémoire.

Allocation dynamique-remarques

- La manipulation d'objets via des adresses introduit une indirection (coûteuse) mais est plus uniforme.
- Elle est nécessaire lorsque le langage contient des fonctions polymorphes (les fonctions peuvent manipuler des données de taille variable à l'exécution).

Sommaire

Allocation/desallocation explicite

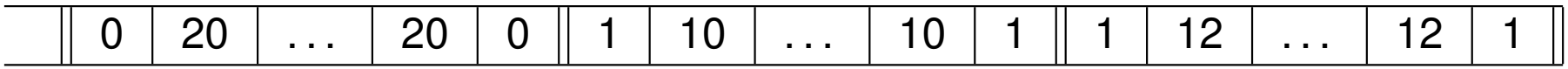
- Les langages comme Pascal ou C offrent des primitives (`new`/`dispose` en Pascal, `malloc`/`free` en C) pour allouer ou désallouer des parties de la mémoire.
- Le tas est organisé en une suite de blocs libres chaînés entre eux. Chaque bloc est formé d'une suite d'octets consécutifs en mémoire.
  - le bloc comporte une entête qui contient la taille du bloc en nombre d'octets.
  - les blocs libres sont chainés entre eux.
- Les blocs libres peuvent être organisés en plusieurs listes, par exemple en ayant des espaces réservés pour l'allocation de données de petite taille.

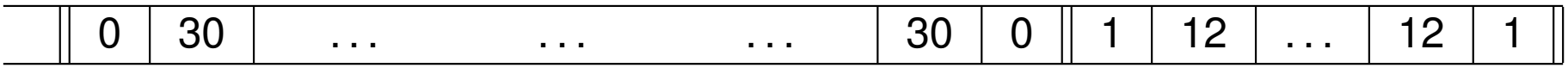
Allocation explicite

- Pour allouer de la mémoire, on cherche un bloc de taille suffisante en parcourant la liste des blocs libres.
- Si on ne trouve pas de bloc adéquat. il faut demander au système un morceau de mémoire supplémentaire.
- Si on trouve un bloc trop grand, on en réserve la partie nécéssaire et on garde le reste dans la liste des blocs libres.
- Il peut être judicieux de choisir le bloc le plus petit qui convient.
- Garder la localité est aussi utile.

Libération explicite

- Le morceau libéré est ajouté à la liste des blocs libres.
- Il est utile de fusionner des blocs libres consécutifs pour récupérer des espaces plus grands.
- On peut garder sur chaque bloc, en tête et en fin, un bit indiquant s'ils sont libres + la taille du bloc.
- Il suffit de mettre à jour localement si l'un des blocs adjacents est libre.





Allocation/desallocation explicite

- L'allocation/ désallocation explicite rend la programmation plus lourde.
- On risque d'utiliser inutilement la place par des données inaccessibles
- On risque au contraire de faire disparaître des données accessibles et de provoquer des erreurs difficilement détectables ensuite suivant la manière dont cet espace est réalloué.
- De telles erreurs posent des problèmes de sécurité.

Sommaire

Allocation/desallocation implicite

- Des langages tels que LISP, ML ou JAVA ont choisi la voie de l'allocation dynamique implicite.
- L'exécution va être amenée à allouer de la mémoire
  - `new` en Java
  - constructeurs, fonctions comme résultat en Caml
- L'environnement de programmation s'occupe également de récupérer la mémoire.
- C'est le mécanisme de Garbage collector (ramasse-miette, glaneur de cellules).

Remarque

- Le GC s'appuie sur le typage fort des langages qui restreint statiquement l'accès à la mémoire (via des variables déclarées ou bien des champs de ces variables).
- Dans un langage comme C, l'arithmétique de pointeur autorise a priori l'accès à n'importe quelle partie de la mémoire.
- Des outils d'analyse de la mémoire "faibles" permettent de détecter des erreur potentielles.

Sommaire

Compteur de référence

Chaque cellule dynamiquement allouée comporte un compteur indiquant combien de pointeurs peuvent y accéder.

- `new`(p) la cellule a un compteur initialisé à 1
- Une affectation entre pointeurs a la forme `t = u`
  - `t` s'évalue en une valeur gauche *x*
  - *x* initialement a pour valeur droite une cellule *m*
  - `u` s'évalue en une valeur droite qui est une cellule *n*
- On décrémente le compteur de *m* et on incrémente celui de *n*

Compteur de référence-récupération

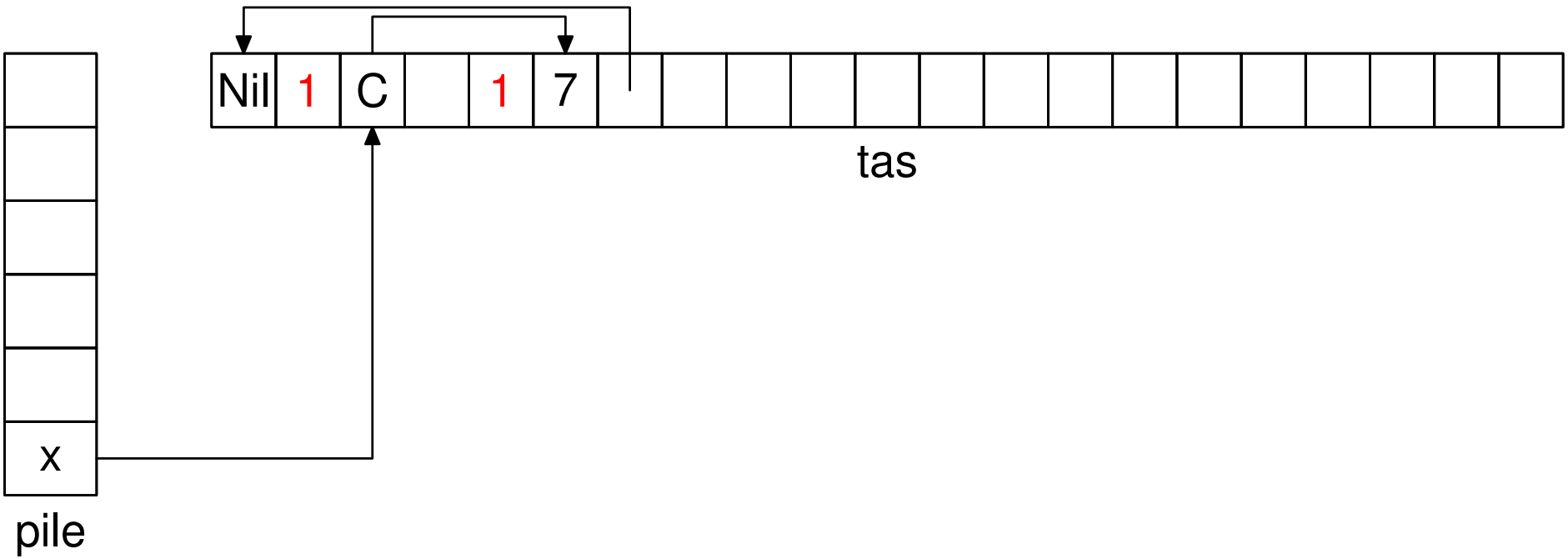
- Les cellules dont le compteur est zéro peuvent être récupérées.
- Il est possible que deux cellules pointent mutuellement chacune sur l'autre et gardent des compteurs à 1 (même si elles ne sont plus accessibles).
- Cette méthode est coûteuse en temps de calcul.
- La récupération de la mémoire se fait au fur et à mesure.

Compteur de référence : exemple

```
type list = Nil | Cons of cellule
and cellule = {value : int; mutable next : list}
let x = Cons {value=7; next=Nil}
in let y = Cons {value=9; next=x}
in let t = Cons {value=10; next = y}
in let Cons z = x in z.next<-y;;
```

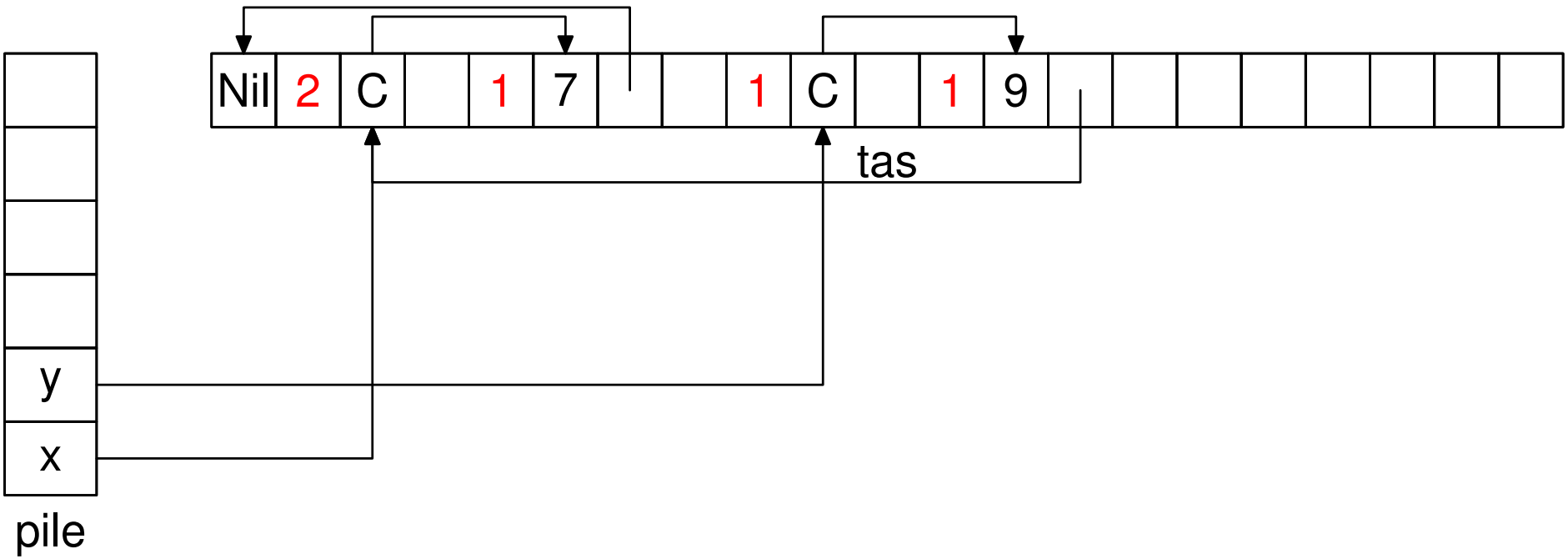
Etape 1

```
let x = Cons {value=7; next=Nil}
```



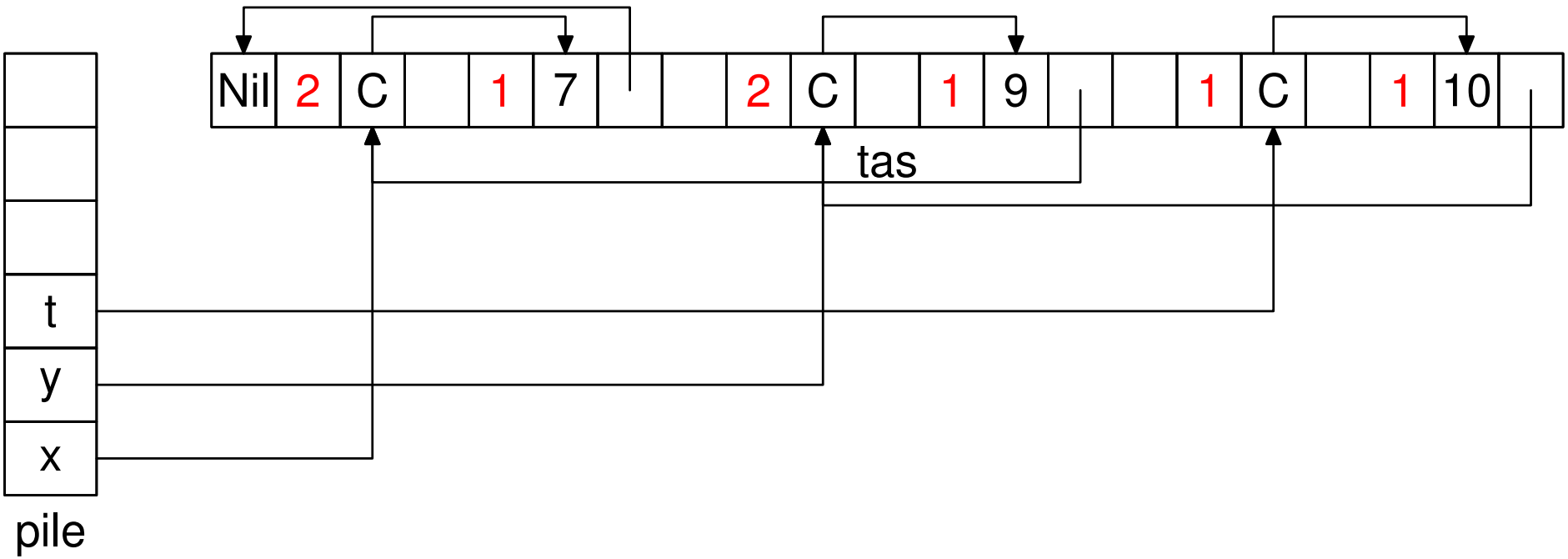
Etape 2

```
let y = Cons {value=9; next=x}
```



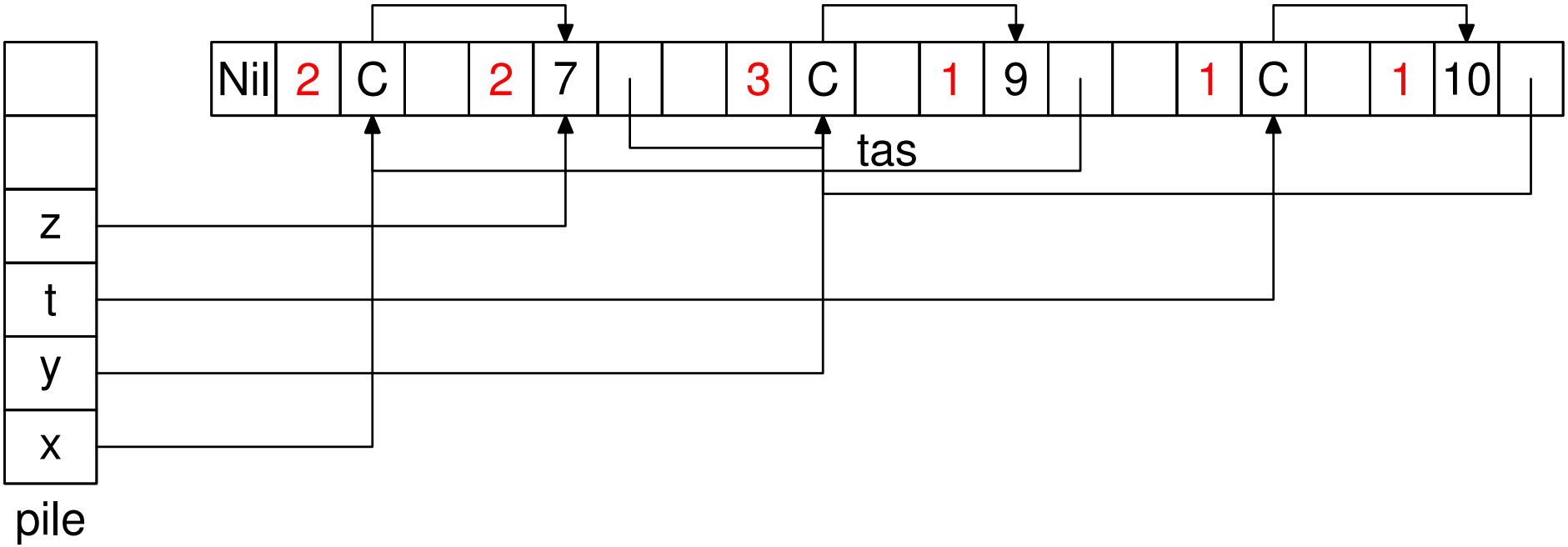
Etape 3

let t = Cons {value=10; next = y}



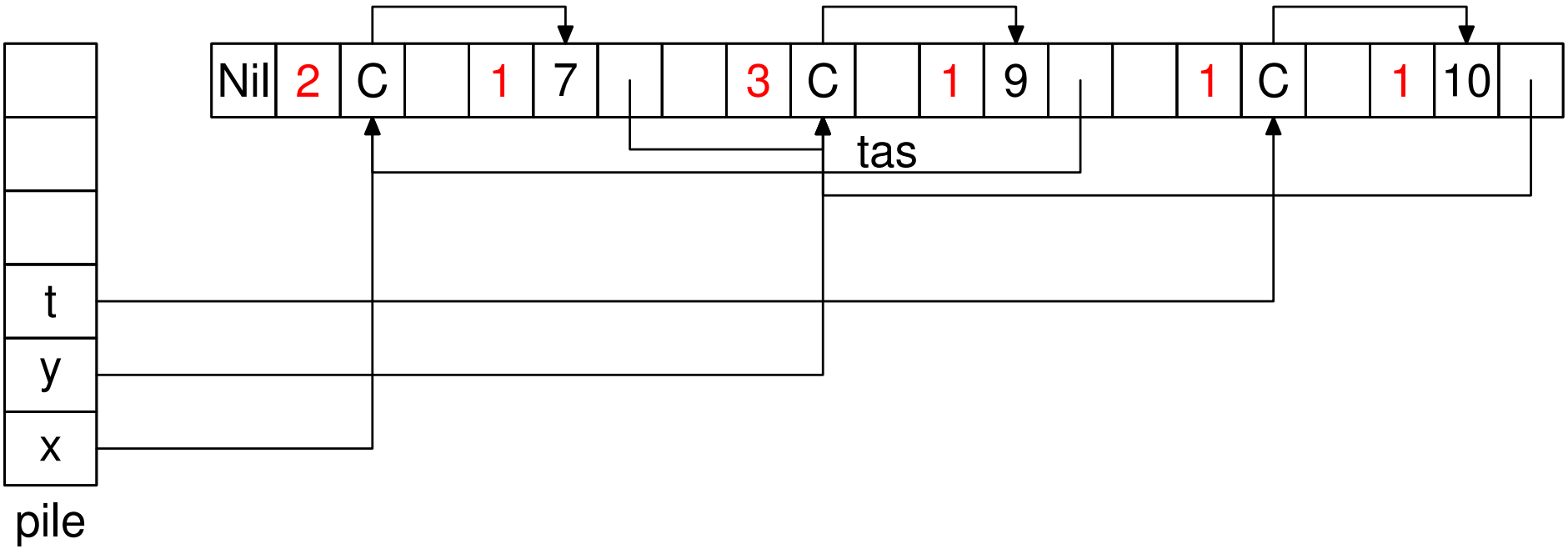
Etape 4

let Cons z = x **in** z.next<−y



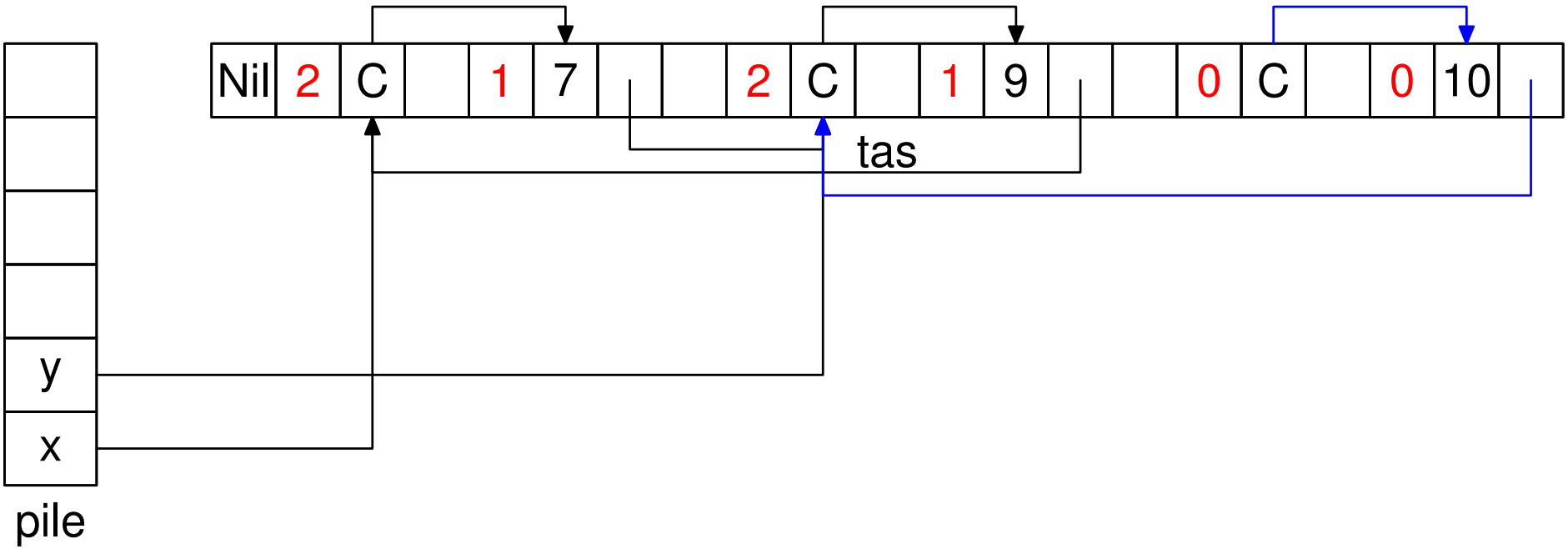
Etape 5

Libération de *z*



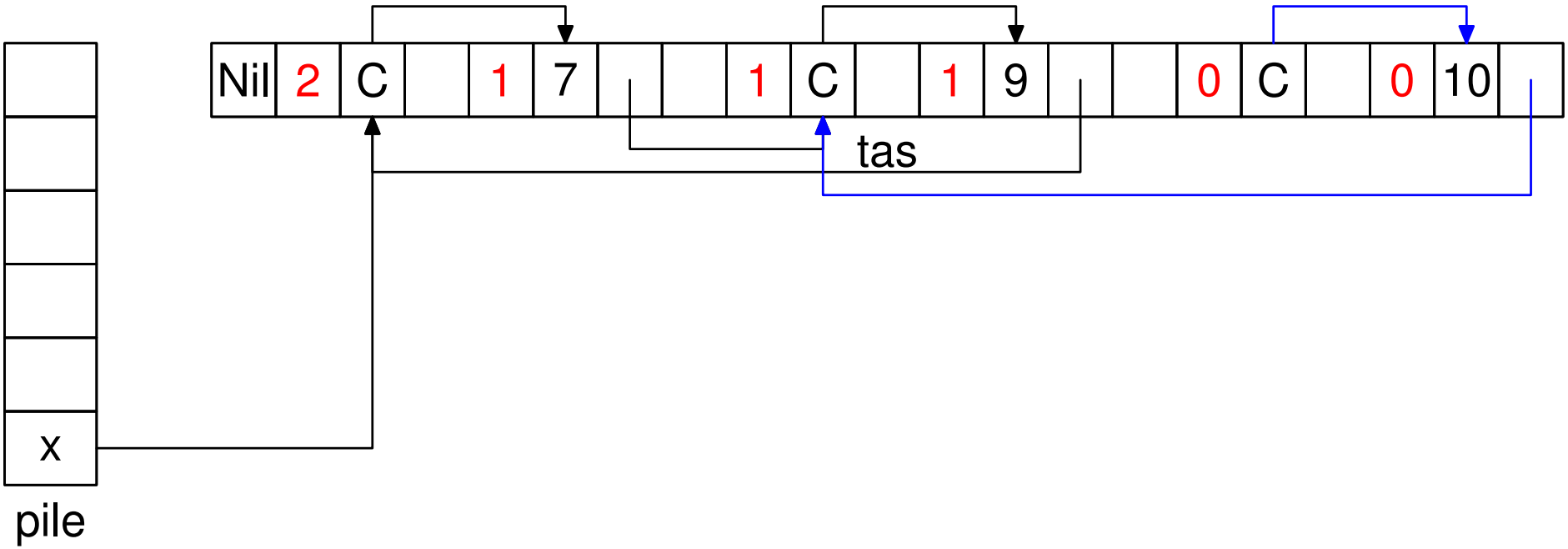
Etape 6

Libération de *t*



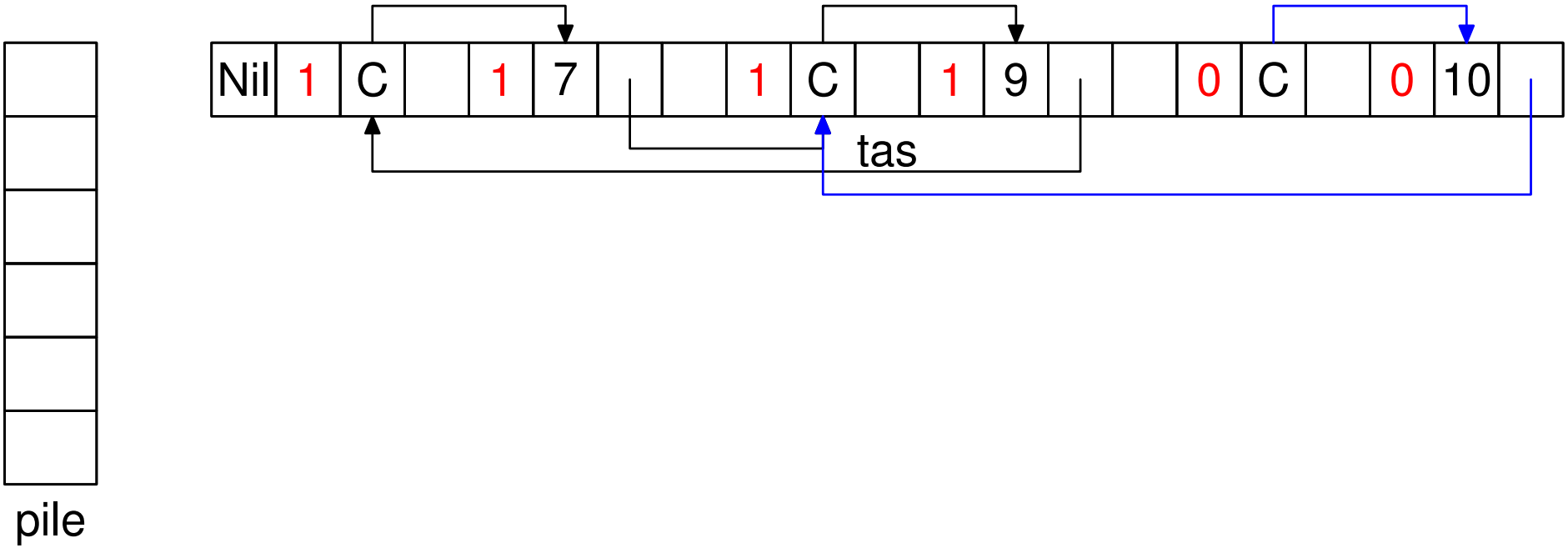
Etape 7

Libération de *y*



Etape 7

Libération de x



Il reste une structure circulaire non récupérable.

Marquer et Balayer

- À partir des variables de la pile (données accessibles au programme) on marque toutes les cellules du tas utilisables de manière récursive.
- Lorsque ce procédé est terminé, une deuxième passe libère les blocs inaccessibles.
- Algorithme (recherche d'une composante connexe)
  - On distingue les nœuds non visités, les nœuds visités dont les fils n'ont pas été visités, et les nœuds complètement examinés.
  - Initialement tous les nœuds sont non visités.
  - On marque les nœuds directement accessibles au programme comme partiellement visités et on les conserve dans une liste.
  - Tant qu'il reste un nœud partiellement visité on le marque comme visité et on le retire de la liste, s'il a des fils non encore visités, on les marque partiellement visités et on les ajoute à la liste.

Remarques

- Les données accessibles au programme peuvent aussi venir des registres.
- Cette récupération de mémoire est coûteuse en temps (problème pour les applications temps réel).
- Autre inconvénient : fragmentation de la mémoire.
- Le procédé récursif pour marquer la mémoire peut provoquer un dépassement de capacité mémoire.
  Algorithme de Shorr-Waite : inversion des liens mémoires pour mémoriser les zones à parcourir.

Stopper et Copier

- Pour éviter les problèmes de fragmentation on peut décider de toujours allouer linéairement des données.
- Lorsque la zone est pleine on s'arrête et on recopie toute la partie utile soit vers la base du tas soit vers une seconde zone de manière continue.
- On a toujours une zone libre linéaire (plutôt qu'un chainage de blocs libres)
- Cette méthode ajout un coût supplémentaire de recopie et modification des pointeurs.

Méthodes mixtes

- Les GC modernes utilisent un compromis entre ces différentes méthodes.
- Caml-light utilise un GC mis au point par Damien Doligez.
- L'espace mémoire est séparé en deux zones :
  - l'une dite vieille organisée à l'aide d'une liste de places accessibles. Cette zone pourra être étendue dynamiquement si nécessaire
  - l'autre dite jeune organisée comme un tableau linéaire de taille fixe.

Méthode

- Allocation Les cellules sont allouées linéairement dans le tableau.
- gc mineur Lorsque le tableau est plein on stoppe et on copie dans la vieille zone les objets utiles:
  (accessibles à partir des variables de la pile, des registres ou des objets de la vieille zone).
- gc majeur on procède à un marquage/balayage incrémental de la vieille mémoire.

GC dans CAML

- Pour marquer les objets on utilise un coloriage des entêtes de blocs.
  - Nœud blanc : pas encore visité
  - Nœud gris : visité mais pas leurs fils immédiats
  - Nœud noir : visité ainsi que leurs fils immédiats.
- On garde une pile des nœuds gris.
  Lorsqu'il ne reste plus de noeuds gris alors on a fini le marquage.
- Les nœuds qui sont blancs après le marquage peuvent être rendus pour la liste libre de la vieille génération.
- Afin de gérer le GC, la représentation de CAML réserve un bit sur chaque mot pour distinguer les adresses (que le GC doit suivre) des entiers.
- Le tas est organisé en zones correspondant aux différentes structures allouées (tableau, type de données structuré, clôture,. . .).
- Dans l'entête de chaque bloc on réserve deux bits pour le marquage (blanc,gris,noir) du GC.

Remarques sur la gestion de la mémoire

- La bonne gestion de la mémoire a une influence sur les performances du programmes.
- Problème difficile qui dépend à la fois du langage de programmation et des applications considérées.

Sommaire

Critères pour un bon code

- Rapidité d'exécution
- Sécurité
- Taille du code
- Consommation d'énergie

Remarques:

- Ces critères peuvent être contradictoires entre eux.
- Trouver un programme compilé optimal est un problème indécidable.
- On se contentera d'heuristiques.
- Il y a même des méthodes dynamiques d'optimisation qui permettent de choisir les options de compilation en fonction du comportement du programme sur des données bien choisies.

Sommaire

Registres

- Les registres permettent de stocker des informations et d'y accéder de manière rapide.
- On les utilise pour les calculs arithmétiques intermédiaires, le passage des arguments à une fonction . . .
- Les registres sont en nombre limité.
- Comment les exploiter ?

Graphe d'interférence

- On introduit pour chaque valeur intermédiaire une nouvelle variable.
- On construit un graphe d'interférence :
  - les nœuds sont les variables
  - une arête entre x et y si x et y ne peuvent être sauvées dans le même registre
    (ie x et y sont simultanément vivantes: valeurs utilisées dans la suite)
  - On peut également introduire les registres comme nœuds de ce graphe et exprimer par une arête le fait que certaines valeurs ne peuvent pas être stockées dans un registre particulier.

Allocation des registres

- Déterminer une répartition des variables sur k-registres revient à trouver un k-coloriage du graphe.
- Problème NP-complet en général, on utilise une approximation.

Algorithme

- Si le graphe est vide, alors le coloriage est évident.
- Si le graphe comporte un nœud x de degré strictement inférieur à k,
  - on construit un nouveau graphe G' en retirant ce nœud et les arêtes correspondantes,
  - on colorie recursivement G'
  - on peut donner une couleur à x différente de celle de ses voisins
- Si le graphe ne comporte que des nœuds de degré supérieur à k alors on choisit un nœud x que l'on retire et on colorie le graphe résultant. On regarde le nombre de couleurs utilisées par les voisins de x
  - S'il en reste une de libre alors il est possible de colorier x et on l'ajoute au graphe.
  - Sinon, le coloriage a échoué

Gestion des échecs

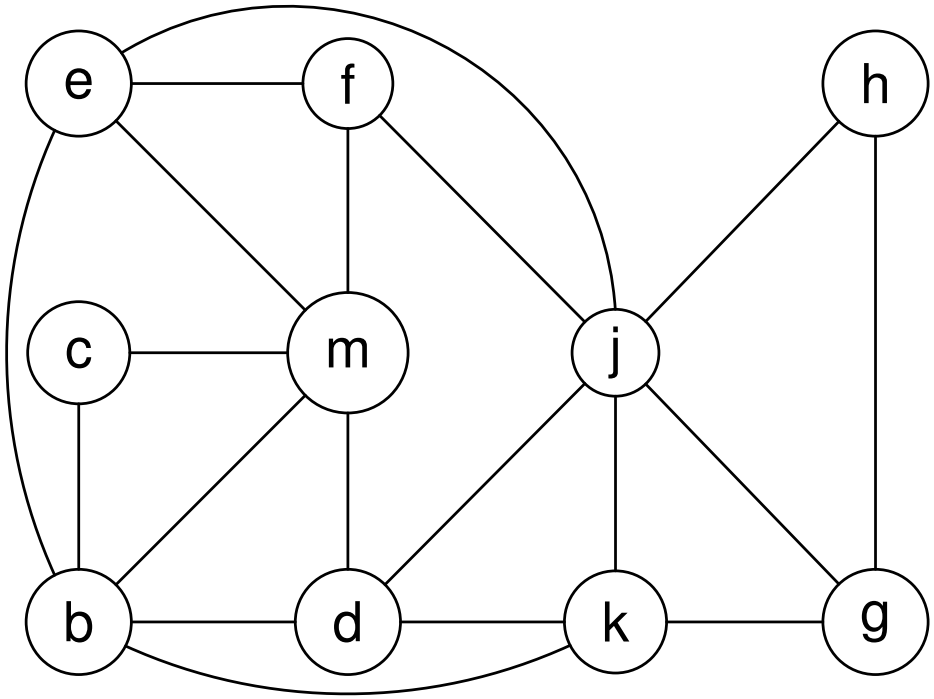
- Si le coloriage a échoué pour x, on décide de stocker x dans la mémoire.
- On choisit un emplacement mémoire m_x pour stocker x et on introduit pour chaque utilisation de x dans le programme une nouvelle variable x_i.
  - Si la valeur de x est utilisée dans une expression, on commencera par mettre dans x_i la valeur stockée à l'adresse m_x de la mémoire.
  - Si x est mise à jour alors on remplace x par x_i dans la mise à jour, et on fait suivre cette instruction d'une mise à jour de m_x dans la mémoire par la valeur x_i.
- La durée de vie des variables x_i est courte, elles n'interfèrent pas avec les autres variables.
- Il faut alors modifier le graphe d'interférence et recommencer le coloriage.

Exemple (d'après Appel)

Affectations simples et accès à la mémoire.
Les variables j, k doivent être définies avant le programme.
Les variables d, k, j sont utilisées après cette portion de programmes.

instruction	var. vivantes
g := mem [j+12]	j,k
h := k - 1	j,g,k
f := g * h	j,g,h
e := mem[j+8]	j,f
m := mem[j+16]	e,j,f
b := mem[f]	e,m,f
c := e+8	e,m,b
d := c	c,m,b
k := m+4	d,m,b
j := b	d,k,b
	d,k,j

Graphe d'interférence

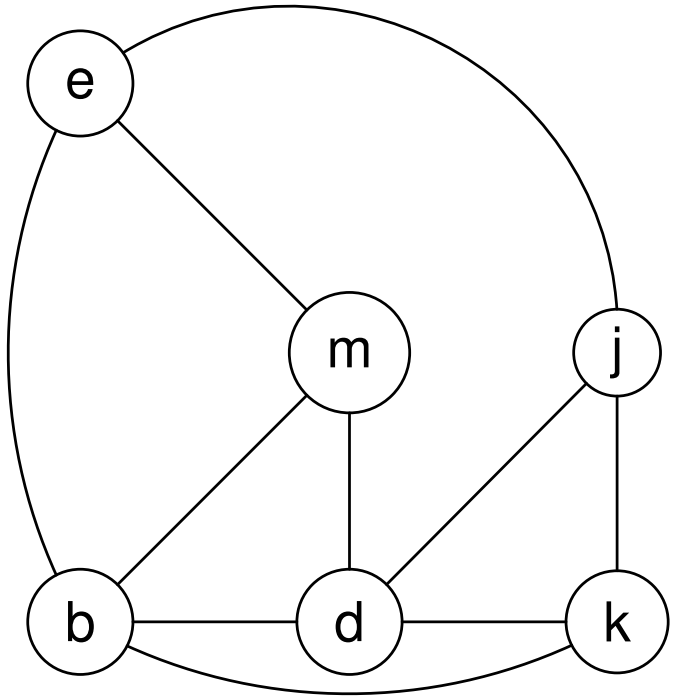


Degrés des noeuds:

b:	5	c:	2	d:	4	e:	4	f:	3
g:	3	h:	2	j:	6	k:	4	m:	5

Coloriage avec 4 registres

On enlève des nœuds de degré inférieur à 3: g, h, c, f.

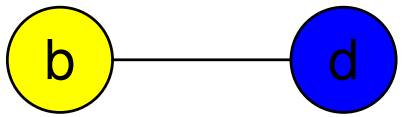


Degrés des noeuds:

b:	4	d:	4	e:	3
j:	3	k:	3	m:	3

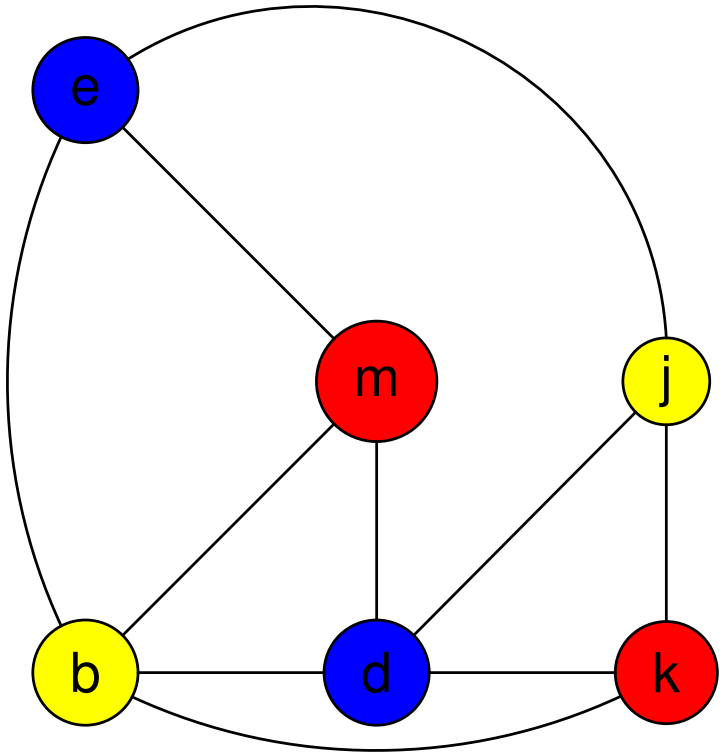
Coloriage - suite

On enlève des nœuds de degré inférieur à 3: j, k, e, m.
On peut colorier les nœuds restants.



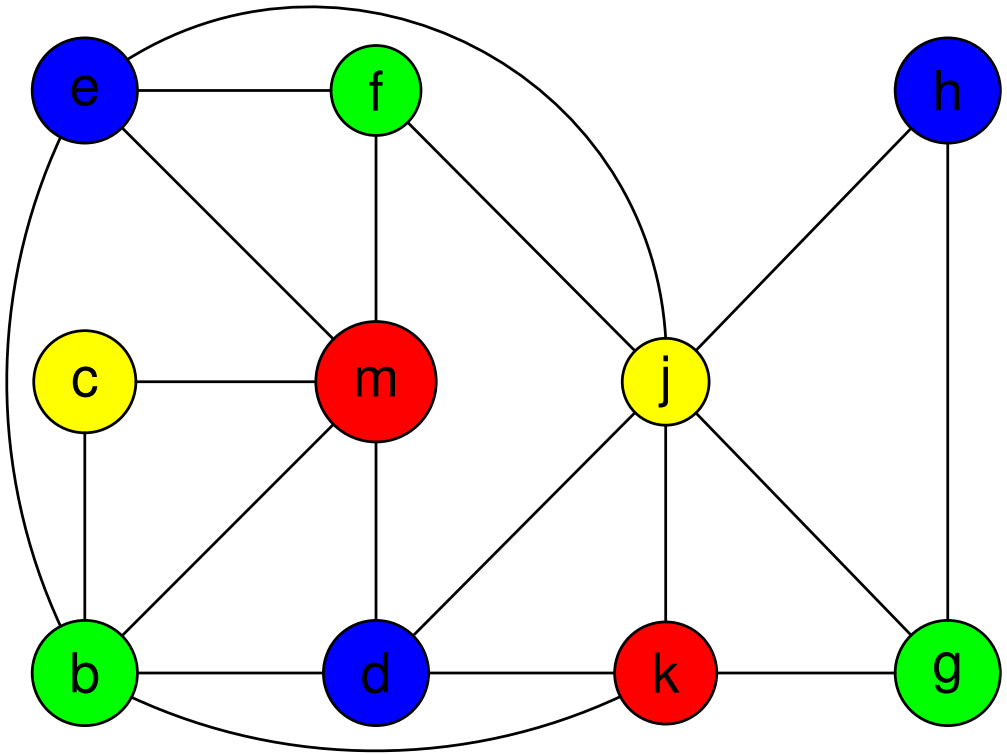
Coloriage - suite

On réintroduit les noeuds: *j*, *k*, *e*, *m*.



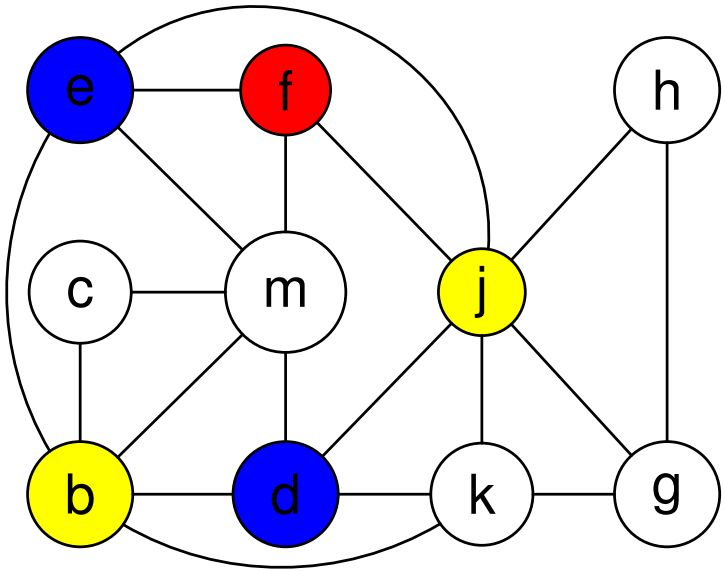
Coloriage - suite

On réintroduit les noeuds: *g*, *h*, *c*, *f*.



Coloriage avec 3 registres

- On peut refaire l'analyse en examinant les nœuds dans l'ordre suivant. $b, d, j, e, f, \mathbf{m}, \mathbf{k}, g, c, h$
- Les nœuds indiqués en gras sont ceux dont le degré est supérieur à 3 et donc qui peuvent poser un problème.
- Une tentative de coloriage b jaune, d bleu, j jaune, e bleu, f rouge ne permet pas de colorier m.



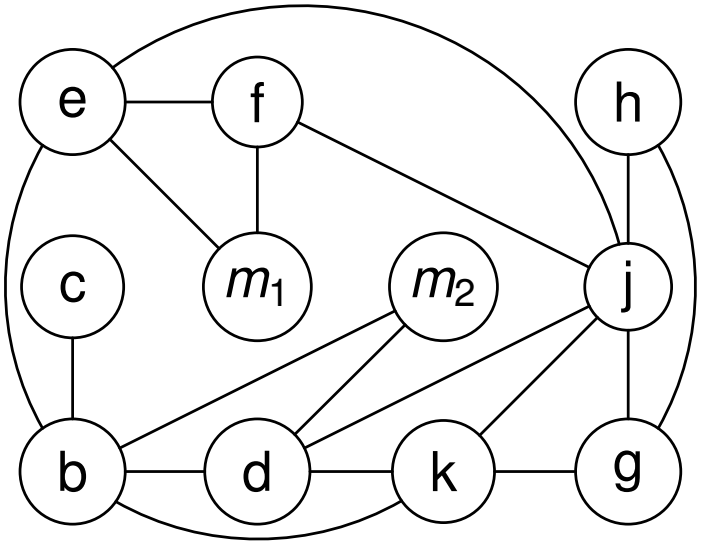
Transformation

- On stocke m en mémoire à l'adresse $\mathbb{M}$.
- Chaque accès à m se fait par l'intermédiaire d'une nouvelle variable m_i

g	:= mem [j+12]	k, j
h	:= k - 1	j, g, k
f	:= g * h	j, g, h
e	:= mem[j+8]	j, f
m_1	:= mem[j+16]	j, f, e
mem[$\mathbb{M}$]	:= m_1	m_1,f,e
b	:= mem[f]	f, e
c	:= e+8	b, e
d	:= c	b, c
m_2	:= mem[$\mathbb{M}$]	b, d
k	:= m_2+4	m_2, b, d
j	:= b	b, d, k
		d, k, j

Coloriage avec 3 registres

- Le nœud m de degré 5 s'est transformé en deux nœuds m_1 et m_2 de degré 2.

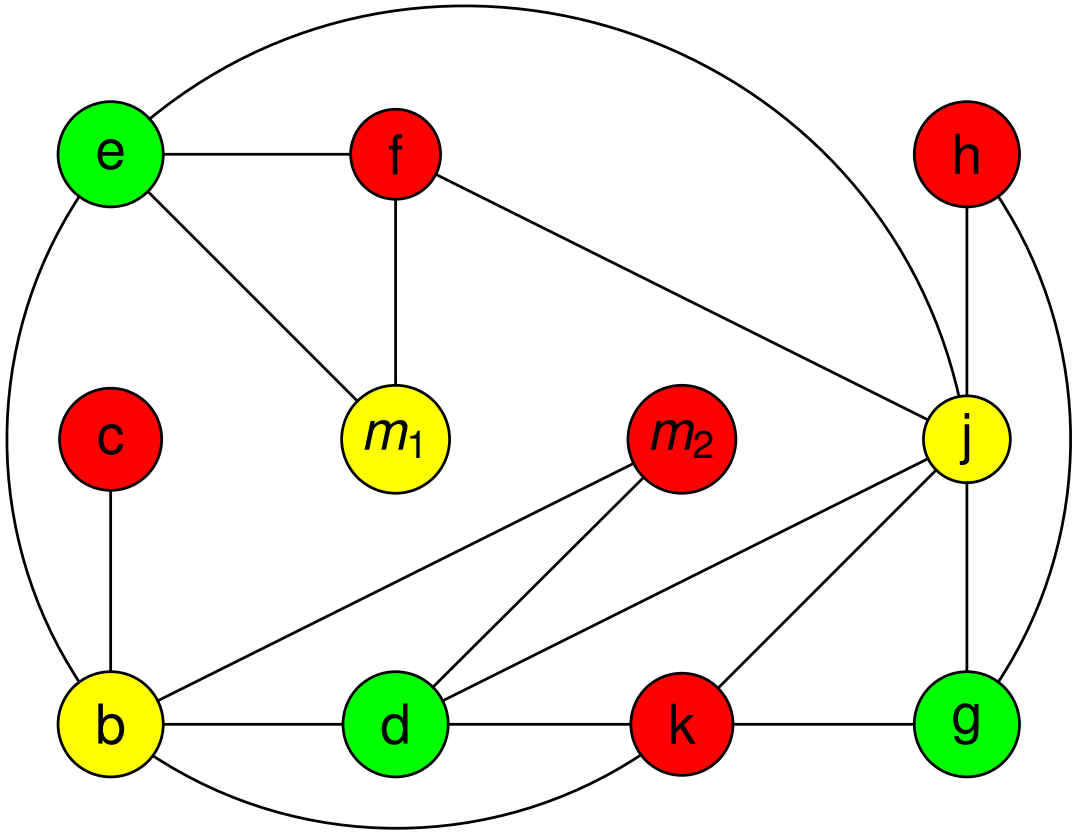


- On peut réexaminer les nœuds du nouveau graphe dans l'ordre suivant :

$$j, g, k, d, b, e, f, m_1, m_2, c, h$$

- Il n'y a plus de nœud à problème.

Résultat



Conclusion sur les registres

- Un principe de coloriage peut aussi être utilisé pour minimiser le nombre de places mémoires des variables stockées en mémoire.
- Un autre objectif est de minimiser le nombre de `move` (`x:=y`) même si cela augmente le nombre de cases mémoire utilisées.

Sommaire

Selection d'instructions

- Il y a souvent plusieurs combinaisons d'instructions dans la machine cible pour effectuer une même opération.
- Comment sélectionner la meilleure.

Jeu d'instructions

On considère une machine à registres avec les opérations suivantes:

ADD r_i r_j r_k	$r_i \leftarrow r_j + r_k$
MUL r_i r_j r_k	$r_i \leftarrow r_j \times r_k$
SUB r_i r_j r_k	$r_i \leftarrow r_j - r_k$
DIV r_i r_j r_k	$r_i \leftarrow r_j / r_k$
ADDI r_i r_j c	$r_i \leftarrow r_j + c$
SUBI r_i r_j c	$r_i \leftarrow r_j - c$
LOAD r_i r_j c	$r_i \leftarrow M[r_j + c]$
STORE r_i r_j c	$M[r_j + c] \leftarrow r_i$
MOVEM r_i r_j	$M[r_j] \leftarrow M[r_i]$

Exemple

```
a[i]:=x
```

- a et x des variables locales dont on connait le décalage par rapport à fp;
- i dans un registre r_i;
- r_0 un registre qui contient 0.
- L'emplacement mémoire correspondant à $a[i]$ est l'adresse du tableau a trouvée au pointeur fp plus le décalage de a à laquelle on ajoute $4 \times i$.

Solution naive

ADDI	$r_1 \leftarrow fp + a$
LOAD	$r_1 \leftarrow M[r_1 + 0]$
ADDI	$r_2 \leftarrow r_0 + 4$
MUL	$r_2 \leftarrow r_i \times r_2$
ADD	$r_1 \leftarrow r_1 + r_2$
ADDI	$r_2 \leftarrow fp + x$
LOAD	$r_2 \leftarrow M[r_2 + 0]$
STORE	$M[r_1 + 0] \leftarrow r_2$

Optimisations:

Remplacer	ADDI	$r_1 \leftarrow fp + a$	par	LOAD	$r_1 \leftarrow M[fp + a]$
	LOAD	$r_1 \leftarrow M[r_1 + 0]$			
Remplacer	LOAD	$r_2 \leftarrow M[r_2 + 0]$	par	MOVEM	$M[r_1] \leftarrow M[r_2]$
	STORE	$M[r_1 + 0] \leftarrow r_2$			

Principes

- Représentation arborescente du programme à calculer (constantes, variables temporaires, opérations binaires, accès mémoires, opération de déplacement)
- Associer à chaque brique un coût.
- Recherche d'une solution de coût globalement minimum.
- Recherche d'une solution localement minimale : ie on ne peut pas combiner plusieurs instructions consécutives en une instruction de coût inférieur.
- Modèle RISC où les instructions ont des coûts équivalents.

Algorithme par absorption maximale

- On suppose que chaque instruction élémentaire peut être couverte.
- A partir de la racine de l'arbre, on cherche une brique couvrante de taille maximale.
- On se rappelle récursivement sur les sous-arbres et on génère le code.

Génération de code

Version naive:

```
let rec gen_expr r = function
  Op(Plus,e1,e2) -> gen_expr r e1;
                    let r'=newr() in gen_expr r' e2;
                    emit "ADD r <- r+r'"
| Const(c)       -> emit "ADDI r <- r0 + c"
| Tmp(i)         -> emit "ADDI r <- ri + 0"
| Mem(e)         -> gen_expr r e; emit "LOAD r <- M[r+0]"

let gen_instr = function
  Move(Tmp(i),e)    -> gen_expr i e
| Move(Mem(e1),e2) ->
          let r = newr() and r'=newr()
          in gen_expr r e1; gen_expr r' e2;
          emit "STORE M[r+0]<-r'"
```

Recherche de briques maximales

Cas des expressions

```
let rec gen_expr r = function
| Op(Plus,Tmp i,Const c) ->  emit "ADDI r <- ri+c"
| Op(Plus,e,Const c) -> gen_expr r e; emit "ADDI r <- r+c"
| Op(Plus,Tmp i,e) -> gen_expr r e; emit "ADD r <- ri+r"
| Op(Plus,Tmp i,Tmp j) -> emit "ADD r <- ri+rj"
| Op(Plus,e1,e2) -> gen_expr r e1;
                    let r'=newr() in gen_expr r' e2;
                    emit "ADD r <- r+r'"
| Mem(Op(Plus,Tmp i,Const c)) -> emit "LOAD r <- M[ri+c]"
| Mem(Op(Plus,e,Const c)) ->
                  gen_expr r e; emit "LOAD r <- M[r+c]"
| Mem(Tmp i) -> emit "LOAD r <- M[ri+0]"
| Mem(e)     -> gen_expr r e; emit "LOAD r <- M[r+0]"
```

+ les cas symétriques et les cas de base

Recherche de briques maximales

Cas des instructions

```
let gen_instr = function
  Move(Tmp(i),e)    -> gen_expr i e
| Move(Mem(Op(Plus,e1,Cte c)),e2) ->
              let r = newr() and r'=newr()
              in gen_expr r e1; gen_expr r' e2;
              emit "STORE_M[r+c]<-r'"
| Move(Mem(Cte c),e) -> let r = newr()
              in gen_expr r e; emit "STORE_M[r0+c]<-r"
| Move(Mem(e1),Mem(e2)) -> let r = newr() and r'=newr()
              in gen_expr r e1; gen_expr r' e2;
              emit "MOVEM_M[r]<-M[r']"
```

Algorithme de programmation dynamique

- On part des feuilles et on annote chaque sous-expression avec le coût minimal du code compilé pour ce sous-arbre.
- En chaque nœud, on regarde les briques possibles et on choisit la meilleure en fonction des coûts des sous-expressions de la brique.
- On conserve la brique utilisée et le coût minimal calculé.
- On reparcourt ensuite l'arbre à partir de la racine en choisissant la brique minimale puis en poursuivant avec les feuilles de la brique.

Exemple

$Mem(Op(Plus, Const(c_1), Const(c_2)))$

- $Const(c_i)$ a un coût 1 (ADDI $r \leftarrow r_0 + c_i$)
- $Op(Plus, Const(c_1), Const(c_2))$ est pavé par $Op(Plus, \_, \_)$ (coût global 3) $Op(Plus, Const\_, \_)$ et $Op(Plus, \_Const\_)$ (coût global 2) on garde une des solutions minimales de coût 2.
- $Mem(Op(Plus, Const(c_1), Const(c_2)))$ est pavé par $Mem(\_)$ (coût global 3) et par $Mem(Op(Plus, \_, Const(\_))$ (coût global 2)

Sommaire

Blocs de base

Pour effectuer des optimisations, il est utile d'organiser le graphe de flots de contrôle en blocs de base.
On suppose que l'on travaille sur des instructions à trois adresses (cf cours sur l'analyse sémantique).

- Le graphe de flots de contrôle a comme nœuds les instructions et une arête si une instruction peut en suivre une autre.
- Un bloc est une suite d'instructions consécutives maximale telle que
  - le flôt de contrôle ne peut rentrer que par la première instruction (et pas au milieu du bloc.
  - le contrôle ne peut quitter le bloc par un branchement qu'éventuellement à la dernière instruction.
- On considère maintenant que les blocs forment les nœuds du graphe de flots de contrôle.

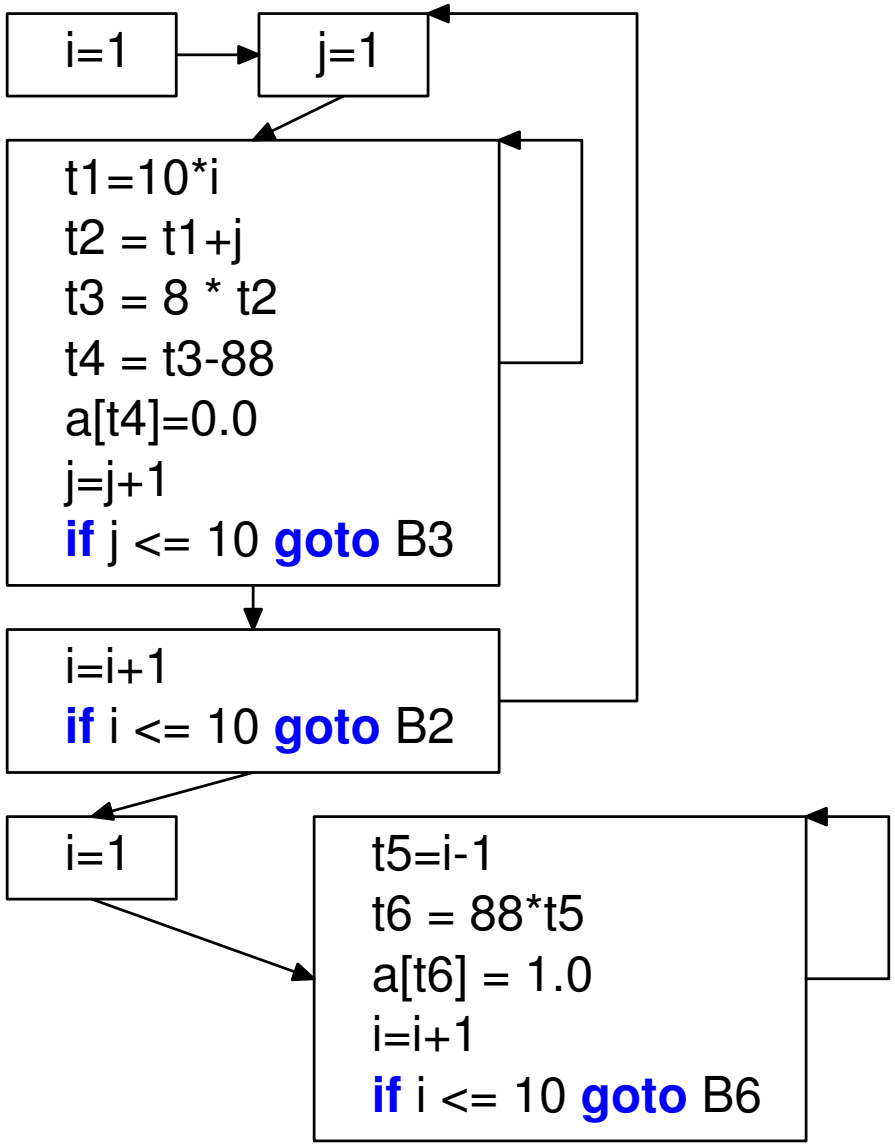
Algorithme pour trouver les blocs

- On détermine les instructions de tête des blocs (initialement le point d'entrée dans le programme.
- Toute instruction qui est la cible d'un branchement conditionnel ou inconditionnel est une tête de bloc
- Toute instruction qui suit un branchement conditionnel ou inconditionnel est une tête de bloc.
  Dans le cas d'un branchement inconditionnel, si l'instruction suivante n'est pas la cible d'un branchement alors c'est du code mort.
- Les blocs sont formés de la suite d'instructions entre deux têtes de bloc.

Exemple

```
    i=1
L2:j=1
L3:t1=10*i
    t2 = t1+j
    t3 = 8 * t2
    t4 = t3-88
    a[t4]=0.0
    j=j+1
    if j <= 10 goto L3
    i=i+1
    if i <= 10 goto L2
    i=1
L13:t5=i-1
    t6 = 88*t5
    a[t6] = 1.0
    i=i+1
    if i <= 10 goto L13
```

Graphe par blocs



Optimisations par bloc

- A l'intérieur d'un bloc il est possible de réordonner les opérations afin de procéder à des optimisations.
- Reconnaissance de sous-expressions locales
- Utilisation d'identité algébriques
- Identification de code mort
- Reorganisation des instructions

Sommaire

Reconnaissance de sous-expressions communes

- On peut reconnaître qu'une même expression a déjà été calculée et réutiliser la valeur.
- En présence d'alias (expressions différentes pouvant représenter la même place mémoire) comme les tableaux ou les pointeurs, il faut veiller que certaines affectations peuvent invalider des calculs faits précédemment.
- Les égalités algébriques doivent être utilisées avec précaution car certaines ne sont pas vérifiées par l'arithmétique des ordinateurs:

$$x < y \Leftrightarrow 0 < y - x$$

Représentation des expressions par des DAG

DAG : directed acyclic graph (comme un arbre mais avec partage de sous arbres)

- Une technique classique dite de “hash-consing” pour une représentation efficace (en mémoire) de grosses expressions.
- Entrée : un “terme” (par exemple une expression arithmétique) représenté sous forme arborescente.
- Sortie : Le même terme représenté sous forme de DAG (deux sous expressions égales auront la même représentation physique)
- Principe : une table garde une trace de tous les termes rencontrés.

Exemple

Type logique

```
type term =
    Var of ident | Cte of int | Op of op * term * term
```

Représentation concrète

```
type term_node =
    Var of ident | Cte of int | Op of op * term * term
and term = {node : term_node; tag : int}
```

- Le tag représente de manière unique le terme.
- La fonction de hash-consing prend en argument un objet de type `term_node` et renvoie un terme de type `term`
- Elle utilise une table de hachage qui associe à chaque `term_node` déjà rencontré, le terme “hash-consé” associé.

Table de hachage

Module `Hashtbl`

```
module type HashedType = sig
    type t val equal : t -> t -> bool val hash : t -> int
                                end

module Make : functor (H : HashedType) ->
    sig
    type key = H.t
    type 'a t = 'a Hashtbl.Make(H).t
    val create : int -> 'a t
    val clear : 'a t -> unit
    val add : 'a t -> key -> 'a -> unit
    val find : 'a t -> key -> 'a
    val mem : 'a t -> key -> bool
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
    ....
    end
 val hash : 'a -> int
```

Mise en œuvre sur les termes

- Les clés sont les objets de type `term_node`
- Les objets stockés dans la table sont de type `term`
- La fonction d'égalité sur les clés ne doit pas être trop coûteuse
- Elle utilise l'égalité des tags pour les sous-termes.

```
let eq_term t1 t2 = match (t1,t2) with
  Var x1, Var x2 -> x1=x2
| Cte c1, Cte c2 -> c1=c2
| Op (op1,e1,f1), Op(op2,e2,f2)
            -> op1=op2 & e1.tag=e2.tag & f1.tag=f2.tag
| _ -> false
module H : HashedType = struct
    type t = term_node
    let equal = eq_term let hash = Hashtbl.hash
                                end
module HashTable = Hashtbl.Make(H)
```

Fonction de hash-consing

```
let (hashcons : term_node -> term) =
 let table = HashTable.create 251
 and tagref = ref 0
 in fun x -> try Hashtbl.find table x
             with Not_found ->
             let hc = { node = x; tag = !tagref} in
             incr tagref; HashTable.add table x hx; hx
```

On peut ensuite définir des fonctions de construction :

```
let (var:ident->term) x = hashcons (Var x)
let (cte:int->term) x = hashcons (Cte x)
let (op:op->term->term->term) o e1 e2
                        = hashcons (Op(o,e1,e2))
```

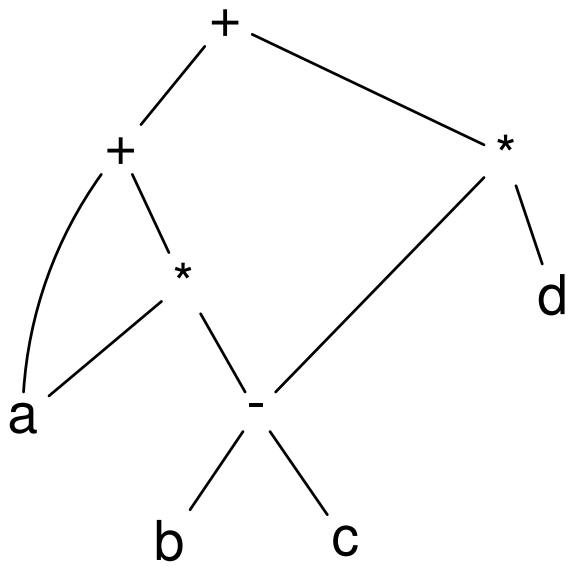
Invariant: $x = y \Leftrightarrow x == y \Leftrightarrow x.tag = y.tag$

Génération de code

- Pour engendrer le code intermédiaire pour une expression représentée sous forme de DAG, on peut associer une variable à chaque tag utilisé dans l'expression (ou la liste d'expressions) et engendrer le code à trois valeurs correspondant.
- On peut aussi exploiter des propriétés arithmétiques: lorsqu'on traite le terme $a + b$ ou $a \times b$ on peut chercher si $b + a$ ou $b \times a$ ont déjà été entrés et réutiliser les valeurs.

Exemple

- Expression $a + a * (b - c) + (b - c) * d$
- DAG



Génération de code

- Terme

```
{node=Var"a";tag=0}
{node=Var"b";tag=1}
{node=Var"c";tag=2}
{node=Op("-",{tag=1;..},{tag=2;..});tag=3}  (*b-c*)
{node=Op("*",{tag=0;..},{tag=3;..});tag=4}  (*a*(b-c)*)
{node=Op("+",{tag=0;..},{tag=4;..});tag=5}  (*a+a*(b-c)*)
{node=Var"d";tag=6}
{node=Op("*",{tag=3;..},{tag=6;..});tag=7}  (*(b-c)*d*)
{node=Op("+",{tag=5;..},{tag=7;..});tag=8}
```

- Code

```
r0 = a          r5 = r0+r4
r1 = b          r6 = d
r2 = c          r7 = r3*r6
r3 = r1-r2      r8 = r5+r7
r4 = r0*r3
```

Cas des instructions

- On part d'une suite d'instructions à 3 registres
- On cherche à partager des sous-expressions
- Exemple
  a = b+c
  b = a-d
  c = b+c
  d = a-d
- Les variables sont modifiables : les deux sous-expressions $b + c$ n'ont pas la même valeur.
- On construit le DAG correspondant aux expressions calculées et un environnement qui dit pour chaque variable quel est le terme correspondant.
- On suppose données les valeurs initiales des variables (contexte).
- On calcule librement les expressions puis on met à jour l'environnement.

Algorithme

- Pour chaque variable a utilisée, hash-conser le terme $\texttt{Var}\ "a"$ et associer dans l'environnement le terme obtenu à a.
- Pour une instruction $a = b\ \texttt{op}\ c$:
  - chercher les termes e_b et e_c associés à b et c;
  - hash-conser le terme $\texttt{Op}(op, e_b, e_c)$ et associer dans l'environnement ce terme à la variable a.
- On calcule librement les expressions dans le dag à l'aide de registres.
- On met à jour à la fin les variables réutilisées dans la suite.

Traitement de tableaux

- Deux instructions $x = a[i]$ et $a[i] = y$.
- a est une variable représentant une adresse mémoire.
- Problème: x=a[i]
  a[j]=y
  z=a[i]
- Plusieurs expressions $a[i]$ et $a[j]$ peuvent représenter le même emplacement mémoire (alias).
- On associe à la variable a un ensemble de termes calculés valides.
- Lorsqu'une affectation $a[i] = y$ est traitée, l'ensemble est vidé.
- Lorsqu'une affectation $x = a[i]$ est traitée, on hash-conse le terme $a[i]$.
  - si ce terme existe, on vérifie qu'il est valide
  - si le terme n'existe pas ou est invalide, on le crée et on ajoute le terme crée à la liste des valides de a, on associe également le terme obtenu à la variable x dans l'environnement.

Choix d'opérations simplificatrices

- Certaines opérations arithmétiques peuvent être implantées par des opérations machines différentes:
  - x^2 est équivalent à $x \times x$
  - $2 \times x$ est équivalent à $x + x$ ou à un shift de x
  - $\frac{x}{2}$ est équivalent à $x \times 0.5$
- On choisit les opérations les moins coûteuses.

Sommaire

Choix des registres

- On garde pour chaque variable, une liste d'emplacements mémoire (registres, accès mémoire) où on peut la trouver.
- Certaines variables intermédiaires ne seront jamais stockées en mémoire.
- On garde pour chaque registre, une liste de variables dont il contient la valeur.
- Une fonction de choix de registres, associe à chaque instruction les registres qui devront être utilisés.

Traduction des expressions

- $x = y + z$
  - Le choix de registres nous propose rx, ry, rz.
  - Si ry ne contient pas la valeur y, on choisit y' emplacement mémoire qui contient cette valeur et on engendre $ry \leftarrow M[y']$.
    De même pour z
  - On produit *ADD rx ry rz*.
- $x = y$
  - Le choix de registres nous propose le même registre r pour x et y.
  - Si r ne contient pas la valeur y, on choisit y' emplacement mémoire qui contient cette valeur et on engendre $r \leftarrow M[y']$.
  - On ajoute au descripteur du registre r le fait qu'il représente y et aussi x.
- A la fin du bloc, on stocke les variables actives en mémoire.

Mise à jour des descripteurs

- $r \leftarrow M[x]$: r ne contient plus que x, le descripteur de x contient maintenant r
- $M[x] \leftarrow r$: le descripteur de x contient maintenant sa propre adresse mémoire.
- $x \leftarrow y + z$
  - le descripteur du registre rx ne contient plus que x
  - le descripteur de x ne contient plus que rx
  - le registre rx est retiré des descripteurs des autres variables
- $x = y$ avec y dans le descripteur de ry. On ajoute x au descripteur de ry le descripteur de x ne contient plus que ry.

Remarque: à certains moments, les variables peuvent n'être stockées que dans des registres.

Principe de choix des registres

Instruction $x = y + z$

- Choisir si possible un registre qui contient y
- Sinon choisir un registre libre
- Sinon choisir un registre qui contient une valeur “inutile” (variable non vivante, variable stockée à un autre endroit)
- Sinon choisir un registre r et engendrer une instruction de sauvegarde dans la mémoire des variables associées uniquement à r.
- On choisit le registre qui provoque le moins de copie.

Exemple

```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

- seuls *a*, *b*, *c* et *d* sont actifs après ce bloc, *t*, *u* et *v* sont des variables temporaires.

Exemple

	r1	r2	r3	a	b	c	d	t	u	v
				a	b	c	d			
t=a-b										
r1←M[a]	a			a,r1	b	c	d			
r2←M[b]	a	b		a,r1	b,r2	c	d			
r2←r1-r2	a	t		a,r1	b	c	d	r2		
u=a-c										
r3←M[c]	a	t	c	a,r1	b	c,r3	d	r2		
r1←r1-r3	u	t	c	a	b	c,r3	d	r2	r1	
v=t+u										
r3←r2-r1	u	t	v	a	b	c	d	r2	r1	r3
a=d										
r2←M[d]	u	a,d	v	r2	b	c	d,r2		r1	r3
d=v+u										
r1←r3+r1	d	a	v	r2	b	c	r1			r3
sortie										
M[a]←r2	d	a	v	a,r2	b	c	r1			r3
M[d]←r1	d	a	v	a,r2	b	c	d,r1			r3

Sommaire

Simplification de suites d'instruction

- On peut regarder linéairement le code engendré et repérer des suites d'instructions qui peuvent être simplifiées.
- **PUSHL** n
  STOREL n
  $\rightsquigarrow$ []
- **STOREL** n
  PUSHL n
  $\rightsquigarrow$ **DUP**
  STOREL n
- Attention, on doit s'assurer que les instructions sont toujours exécutées séquentiellement (pas d'étiquette arrivant sur la seconde instruction).

Simplification du flot de contrôle

- Une instruction non étiquetée après un branchement inconditionnel est du code mort
- Cette situation peut arriver après propagation de constante:

```
if debug then code1
code2
```

est compilé en :

```
          PUSHI debug
          JZ endif
          code1
endif:    code2
```

Si *debug*=`false` alors on peut simplifier

```
PUSHI 0
JZ endif
```

en

```
JUMP endif
```

la partie *code1* est alors du code mort.

Autres optimisations de flot

- Les branchements chainés peuvent être factorisés:

```
        JUMP/JZ L1
        :
L1:     JUMP L2
```

se simplifie en

```
        JUMP/JZ L2
        :
L1:     JUMP L2
```

- Schéma plus complexe

```
        JUMP L1
        :
L1:     PUSHL n
        JZ L2
L3:
```

se simplifie en

```
        PUSHL n
        JZ L2
        JUMP L3
        :
L1:     PUSHL n
        JZ L2
L3:
```

- Si aucune instruction ne mène plus à une étiquette *L1*, le bloc pourra être supprimé.

Sommaire

Analyse de flots de données

- Schéma étudié pour l'analyse sémantique.
- Travail sur le graphe de flot de contrôle.
- Permet de calculer des informations approchées pour chaque point du programme.
- Construction des informations par itérations successives:
  (calcul de point fixe)

Principes du calcul

- On cherche à calculer une information D fonction du point de programme.
- Il s'agit d'une information approchée, en général dans un ensemble fini. Approche prudente d'estimation des situations possibles.
- Pour chaque instruction i on introduit deux fonctions in et out qui représentent l'information D, avant l'exécution de i et juste après l'exécution de i.
- Approche en avant :
  - la sémantique de chaque instruction permet de définir $out(i)$ en fonction de $in(i)$ (fonction de transfert)
  - le flot de contrôle permet de définir $in(i)$ en fonction de $out(j)$ pour toutes les instructions j telles qu'il y a un arc de j à i.
- Approche en arrière :
  - la sémantique de chaque instruction permet de définir $in(i)$ en fonction de $out(i)$ (fonction de transfert)
  - le flot de contrôle permet de définir $out(i)$ en fonction de $in(j)$ pour toutes les instructions j telles qu'il y a un arc de i à j.

Exemple d'analyse en avant

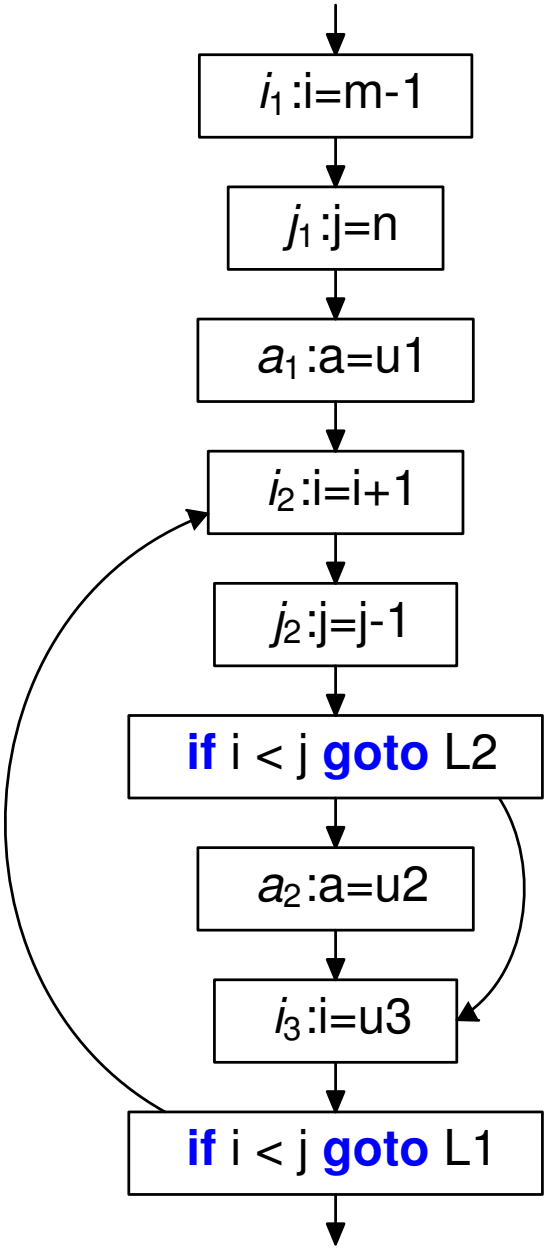
Recherche des définitions actives

- Instructions $x = \ldots$ qui peuvent être utilisées pour une instruction.
- Nombreuses applications: propagation de constantes, détection de variables non initialisées ...
- $out(i) = (x, i) \cup (in(i) - \{(x, k)\}_k)$ si $i \equiv x = d$
  $out(i) = in(i)$ si i ne définit aucune variable.
- $in(i) = \bigcup_{(j|j \to i)} out(j)$
  les définitions utiles avant i sont possiblement toutes les définitions utiles après les instructions qui mènent à i.
- On résoud ces équations par itération jusqu'à l'obtention d'un point fixe.

Exemple

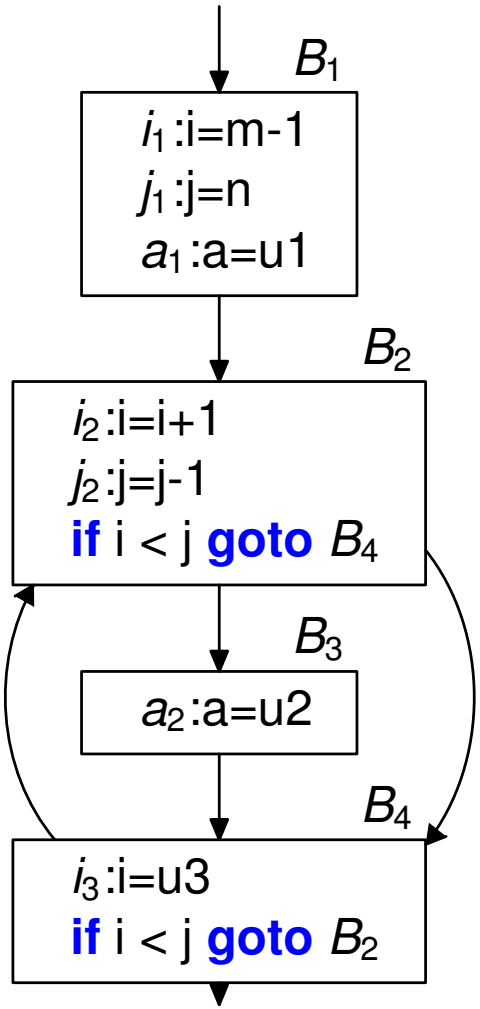
```
        i = m - 1
        j = n
        a = u1
L1:     i = i + 1
        j = j - 1
        if i < j goto L2
        a = u2
L2:     i = u3
        if i < j goto L1
```

Graphe de flots



$in(i_1) = \emptyset$	$\emptyset$	$\emptyset$
$out(i_1) = in(i_1) \setminus i \cup \{i_1\}$	i_1	i_1
$in(j_1) = out(i_1)$	i_1	i_1
$out(j_1) = in(j_1) \setminus j \cup \{j_1\}$	i_1, j_1	i_1, j_1
$in(a_1) = out(j_1)$	i_1, j_1	i_1, j_1
$out(a_1) = in(a_1) \setminus a \cup \{a_1\}$	i_1, j_1, a_1	i_1, j_1, a_1
$in(i_2) = out(a_1) \cup out(if_2)$	i_1, j_1, a_1	$i_1, i_3, j_1, j_2, a_1, a_2$
$out(i_2) = in(i_2) \setminus i \cup \{i_2\}$	i_2, j_1, a_1	i_2, j_1, j_2, a_1, a_2
$in(j_2) = out(i_2)$	i_2, j_1, a_1	i_2, j_1, j_2, a_1, a_2
$out(j_2) = in(j_2) \setminus j \cup \{j_2\}$	i_2, j_2, a_1	i_2, j_2, a_1, a_2
$in(if_1) = out(j_2)$	i_2, j_2, a_1	i_2, j_2, a_1, a_2
$out(if_1) = in(if_1)$	i_2, j_2, a_1	i_2, j_2, a_1, a_2
$in(a_2) = out(if_1)$	i_2, j_2, a_1	i_2, j_2, a_1, a_2
$out(a_2) = in(a_2) \setminus a \cup \{a_2\}$	i_2, j_2, a_2	i_2, j_2, a_2
$in(i_3) = out(a_2) \cup out(if_1)$	i_2, j_2, a_1, a_2	i_2, j_2, a_1, a_2
$out(i_3) = in(i_3) \setminus i \cup \{i_3\}$	i_3, j_2, a_1, a_2	i_3, j_2, a_1, a_2
$in(if_2) = out(i_3)$	i_3, j_2, a_1, a_2	i_3, j_2, a_1, a_2
$out(if2) = in(if2)$	i_3, j_2, a_1, a_2	i_3, j_2, a_1, a_2

Calcul par bloc



$in(B_1) = \emptyset$	$\emptyset$	$\emptyset$
$out(B_1) = in(B_1) \setminus \{i,j,a\} \cup \{i_1,j_1,a_1\}$	i_1, j_1, a_1	i_1, j_1, a_1
$in(B_2) = out(B_1) \cup out(B_4)$	i_1, j_1, a_1	$i_1, i_3, j_1, j_2, a_1, a_2$
$out(B_2) = in(B_2) \setminus \{i,j\} \cup \{i_2,j_2\}$	i_2, j_2, a_1	i_2, j_2, a_1, a_2
$in(B_3) = out(B_2)$	i_2, j_2, a_1	i_2, j_2, a_1, a_2
$out(B_3) = in(B_3) \setminus \{a\} \cup \{a_2\}$	i_2, j_2, a_2	i_2, j_2, a_2
$in(B_4) = out(B_3) \cup out(B_2)$	i_2, j_2, a_1, a_2	i_2, j_2, a_1, a_2
$out(B_4) = in(B_4) \setminus \{i\} \cup \{i_3\}$	i_3, j_2, a_1, a_2	i_3, j_2, a_1, a_2

Applications

- Recherche d'utilisations de variables non initialisées:
  - On ajoute une définition x_0 fictive après chaque déclaration de variable x.
  - Si lors d'une utilisation de x, la définition fictive x_0 est active, alors il y a un risque que x soit utilisée sans être initialisée.
- Propagation de constante
  - On cherche à repérer des définitions $x = d$ dans lesquelles d est une constante qui peut être calculée statiquement à la compilation.
  - Lors d'une définition $x = d$ on regarde les variables utilisées dans d.
  - La variable y est constante s'il y a une seule déclaration active et que cette déclaration est constante (s'il y a plusieurs déclarations actives, il faut qu'elles calculent la même valeur).
  - Si toutes les variables de d sont constantes alors la nouvelle déclaration $x = d$ est aussi constante.
  - Le compilateur doit calculer d. Le résultat peut dépendre de l'architecture.
  - On peut compiler le calcul de la constante et l'exécuter au vol.

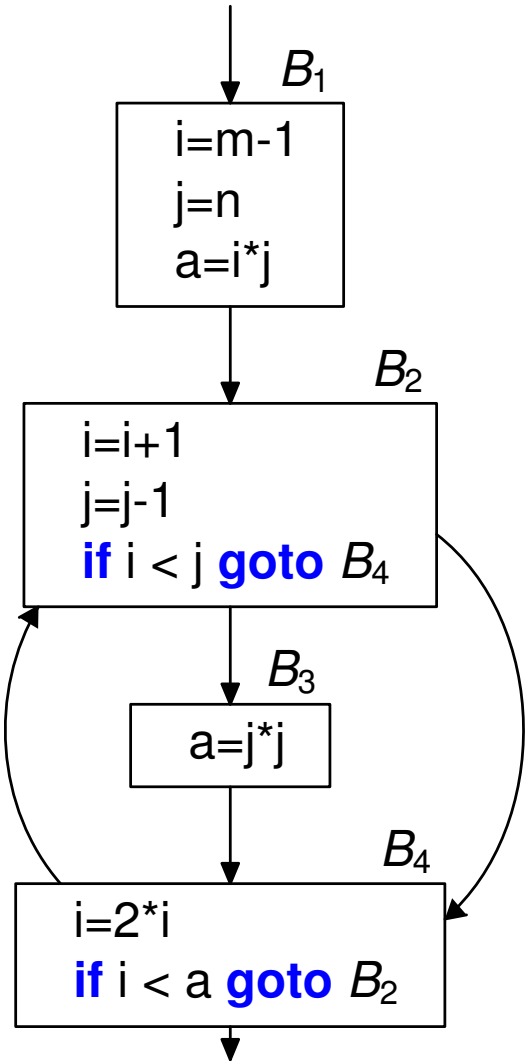
Exemple d'analyse en arrière

Recherche des variables actives

- Variables utilisées dans la suite sans être redéfinies.
- Utilisée pour l'allocation de registres.
- $in(i) = \texttt{used}(i) \cup (out(i) - \texttt{def}(i))$
- $out(i) = \bigcup_{(j|i \to j)} in(j)$
  les variables utiles après i sont les variables utiles avant toutes les instructions j qui peuvent suivre i.
- On résoud ces équations par itération jusqu'à l'obtention d'un point fixe.

Calcul des variables actives

On suppose que a est utilisée dans le reste du programme ($in(fin)$={a}).



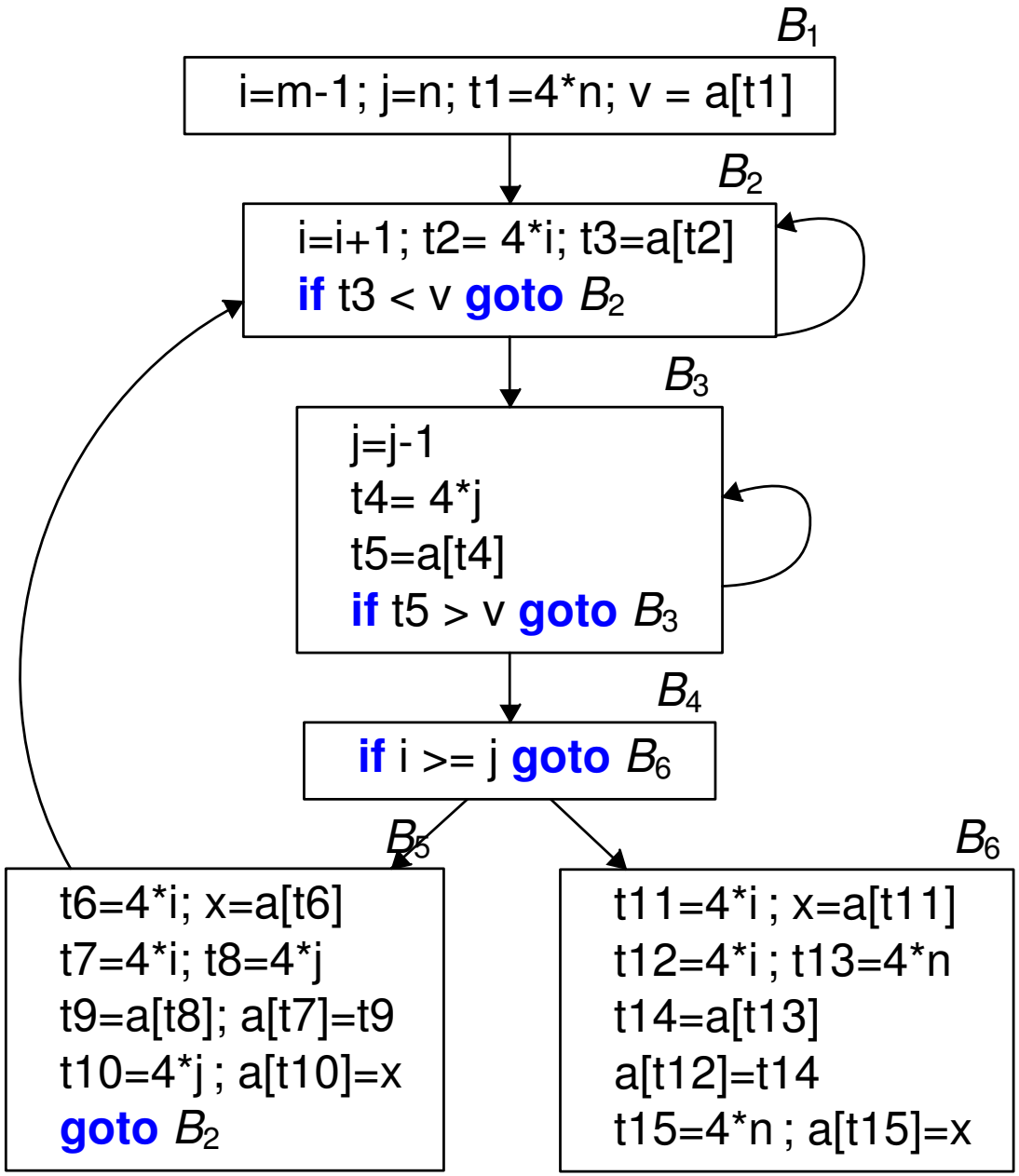
$in(B_1) = out(B_1)\setminus\{i,j,a\}\cup\{m,n\}$	m,n	m,n
$out(B_1) = in(B_2)$	a,i,j	a,i,j
$in(B_2) = out(B_2)\cup\{i,j\}$	a,i,j	a,i,j
$out(B_2) = in(B_3)\cup in(B_4)$	a,i,j	a,i,j
$in(B_3) = out(B_3)\setminus\{a\}\cup\{j\}$	i,j	i,j
$out(B_3) = in(B_4)$	a,i	a,i,j
$in(B_4) = out(B_4)\setminus\{i\}\cup\{i\}$	a,i	a,i,j
$out(B_4) = in(B_2)\cup in(fin)$	a	a,i,j

Sommaire

Optimisation de boucles

- Beaucoup de programmes passent une grande partie de leur temps dans des boucles internes.
- Intérêt d'optimiser ces boucles.
  - Faire sortir des instructions constantes
  - Eviter des calculs redondants

Exemple


B_1
i=m-1; j=n; t1=4*n; v = a[t1]
B_2
i=i+1; t2= 4*i; t3=a[t2]
if t3 < v goto B_2
B_3
j=j-1
t4= 4*j
t5=a[t4]
if t5 > v goto B_3
B_4
if i >= j goto B_6
B_5
t6=4*i; x=a[t6]
t7=4*i; t8=4*j
t9=a[t8]; a[t7]=t9
t10=4*j ; a[t10]=x
goto B_2
B_6
t11=4*i ; x=a[t11]
t12=4*i ; t13=4*n
t14=a[t13]
a[t12]=t14
t15=4*n ; a[t15]=x

Elimination de sous-expressions communes

On peut simplifier les blocs B_5 et B_6
($t6 = t7, t11 = t12, t11 = t12, t13 = t15$)

B_5 devient:

```
t6=4*i
x=a[t6]
t8=4*j
t9=a[t8]
a[t6]=t9
a[t8]=x
```

B_6 devient:

```
t11=4*i
x=a[t11]
t13=4*n
t14=a[t13]
a[t11]=t14
a[t13]=x
```

On peut aussi faire des optimisations globales pour reprendre les calculs de $t1 = 4 * n$, $t2 = 4 * i$ et $t4 = 4 * j$ effectués dans B_1, B_2 et B_3:

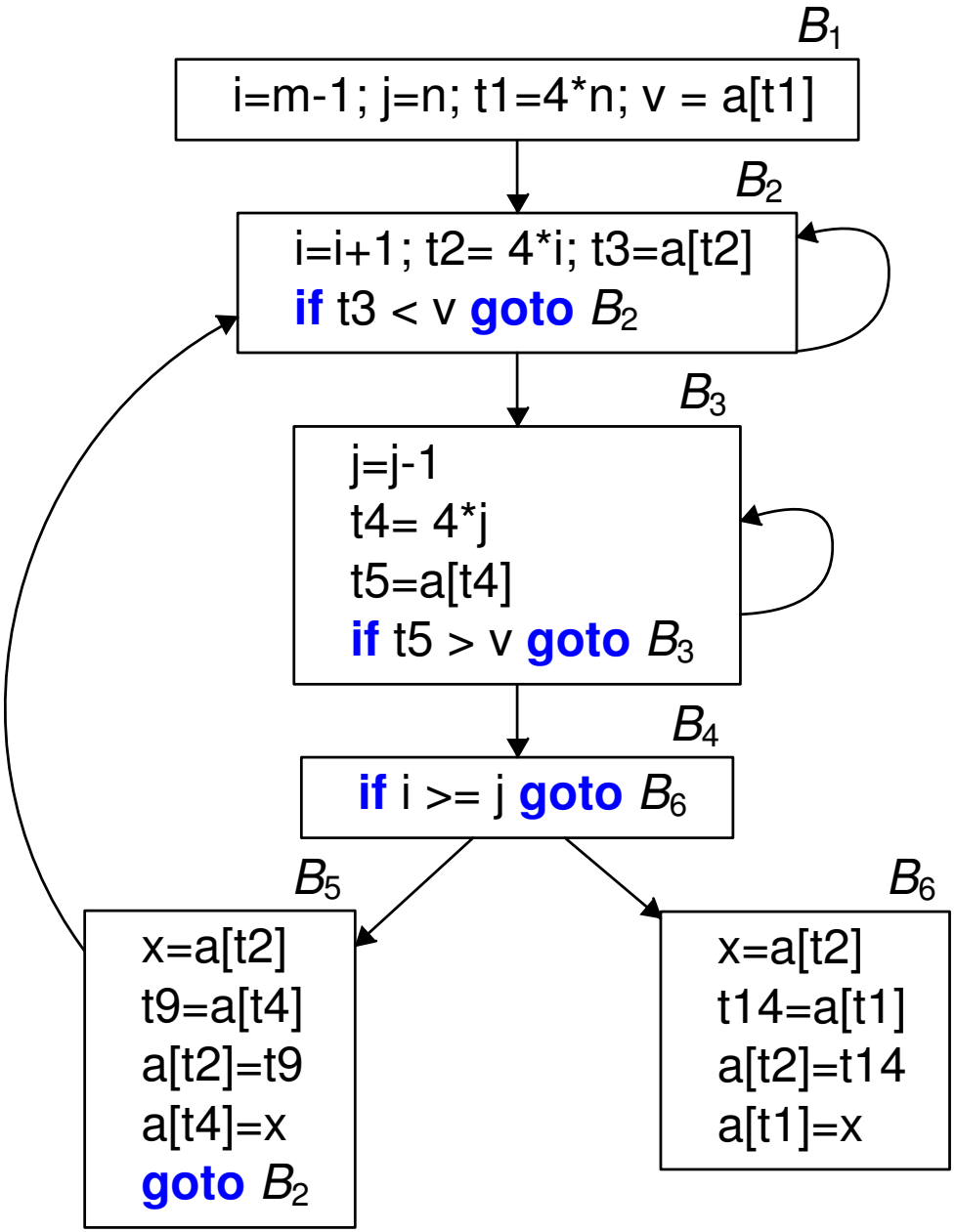
B_5 devient:

```
x=a[t2]
t9=a[t4]
a[t2]=t9
a[t4]=x
```

B_6 devient:

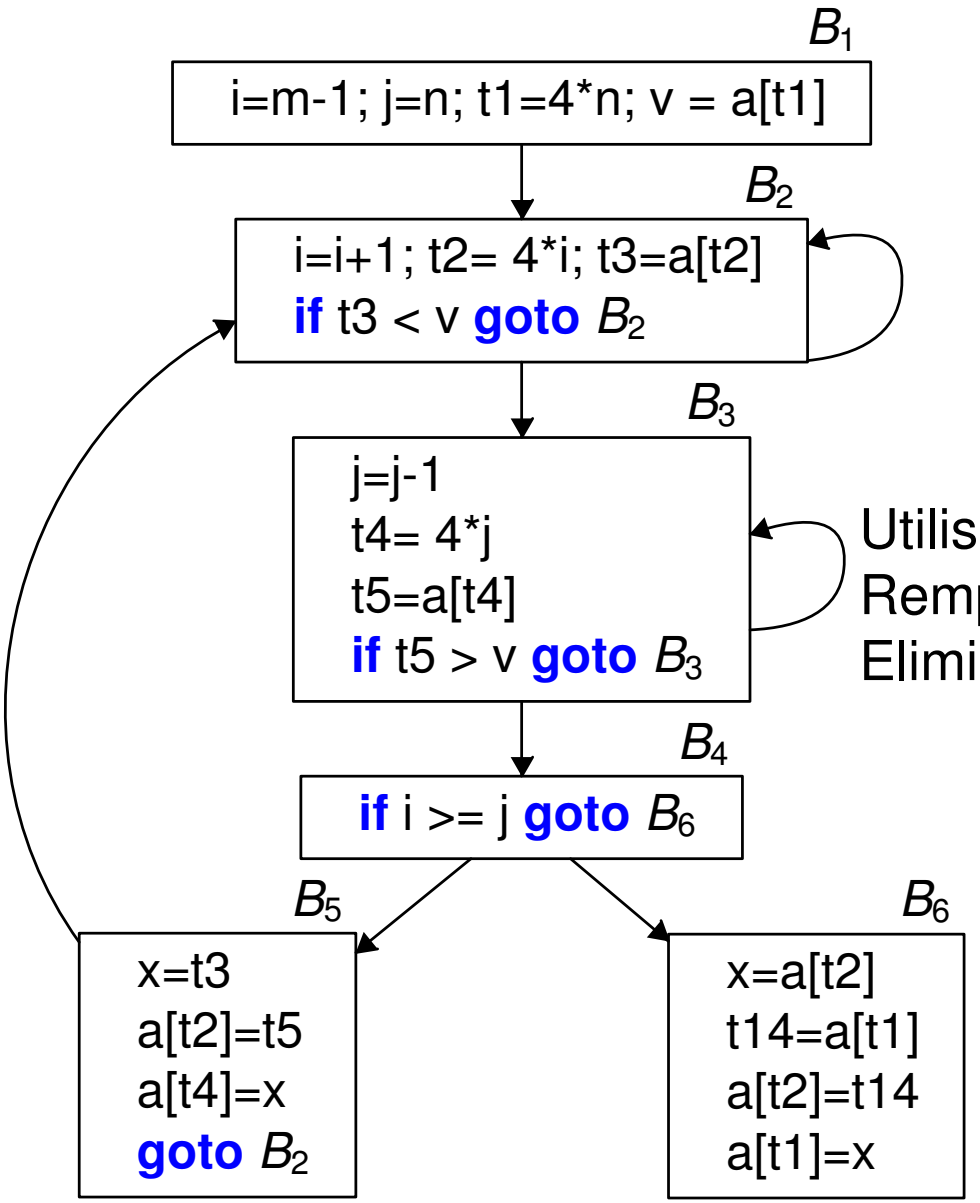
```
x=a[t2]
t14=a[t1]
a[t2]=t14
a[t1]=x
```

Résultat



$a[t2]$ calculé dans $t3$
pas modifié ensuite
$a[t4]$ calculé dans $t5$
pas modifié ensuite
$a[t1]$ calculé dans v
mais peut être modifié

Résultat



Utilisation de la copie $x = t3$
Remplacer x par $t3$ si possible
Eliminer x si non active

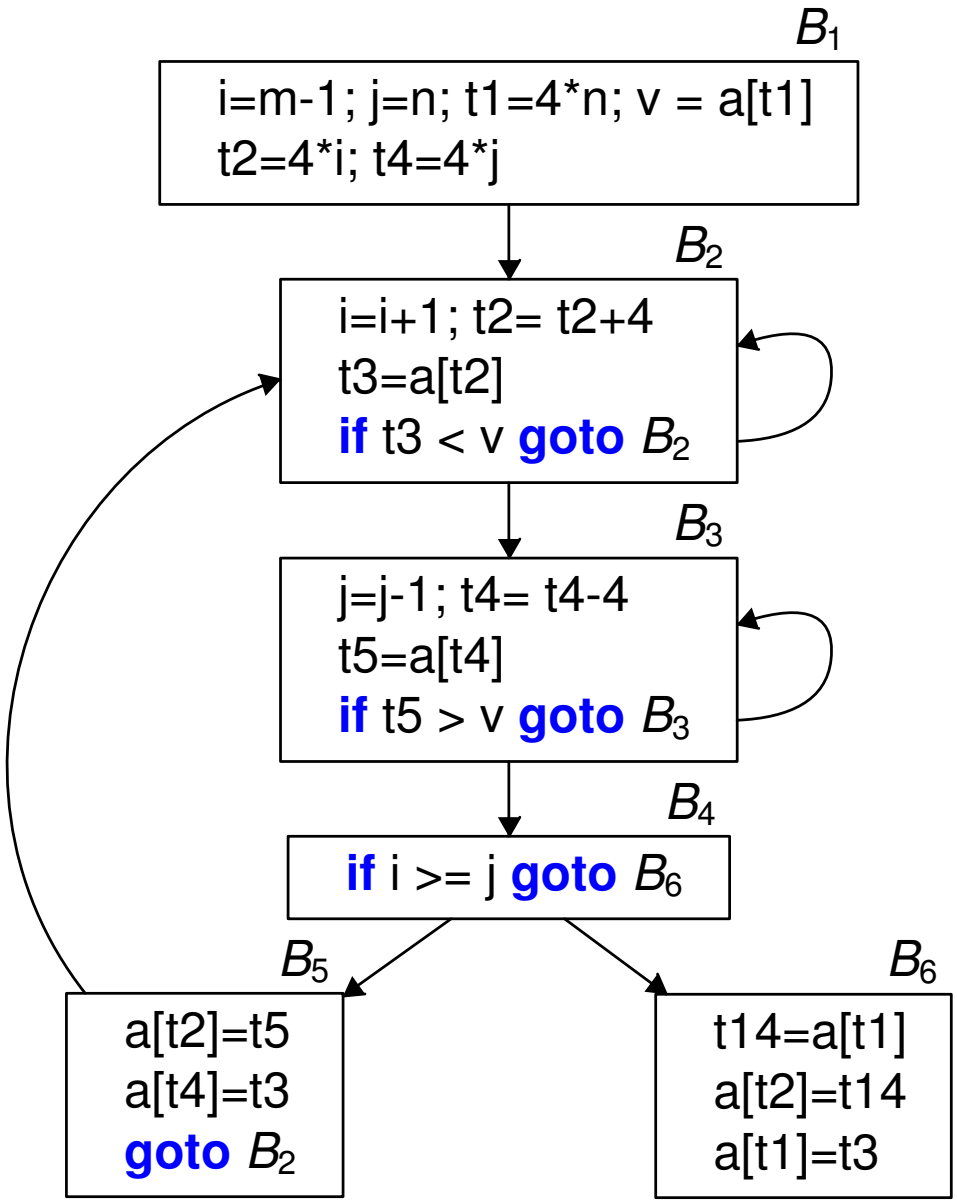
Notion de boucle

- Une boucle est un ensemble L de blocs dans le graphe de contrôle tel que
  - Il y a un seul bloc d'entrée B_0 dans L qui a un prédécesseur hors de L
  - Pour tout bloc B de L, il existe un chemin de B à B_0.
    (Il y a également un chemin de B_0 à B sinon B ne serait pas accessible.
- Exemple : $L = \{B_2\}$, $L = \{B_3\}$ sont des boucles ainsi que $L = \{B_2, B_3, B_4, B_5\}$.
- Données intéressantes sur les boucles:
  - invariant de boucle: expression qui prend la même valeur à toutes les itérations de boucle et qui pourra être calculée avant le bloc de base.
  - variable d'induction: variable qui à chaque passage dans la boucle augmente d'une valeur constante c.
    On utilise des opérations d'incrémentation moins coûteuses que les multiplications.
    Si plusieurs variables d'induction sont corrélées, on en garde une seule.

Exemple

- Dans la boucle B_2 on effectue les opérations $i = i + 1; t2 = 4 * i$.
- Les variables i et $t2$ ne sont pas modifiées par ailleurs.
- Il suffit d'initialiser $t2$ une fois, puis d'incrémenter $t2$ de 4 à chaque itération.
- On peut faire de même pour j et $t4$ dans le bloc B_3.

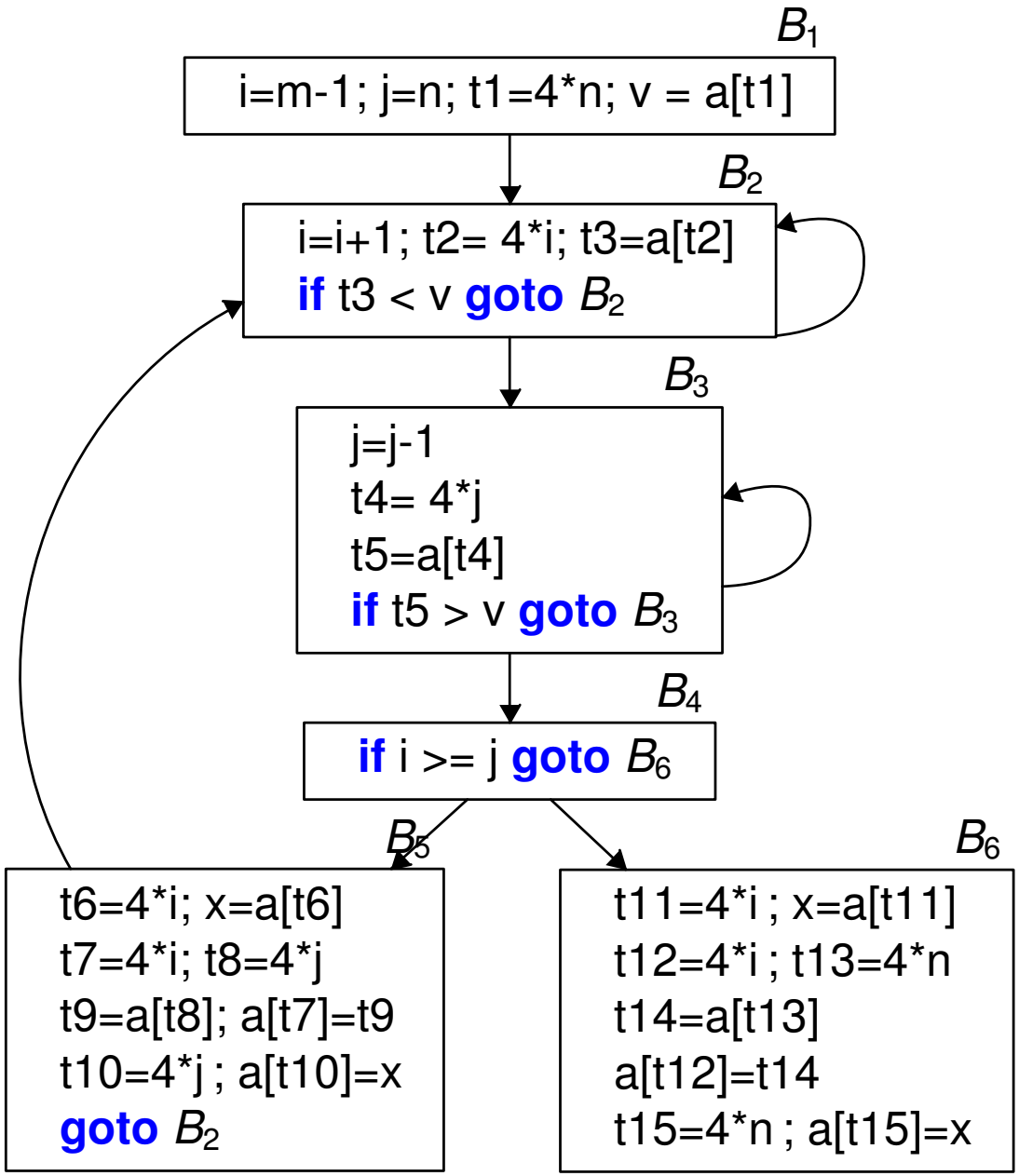
Résultat

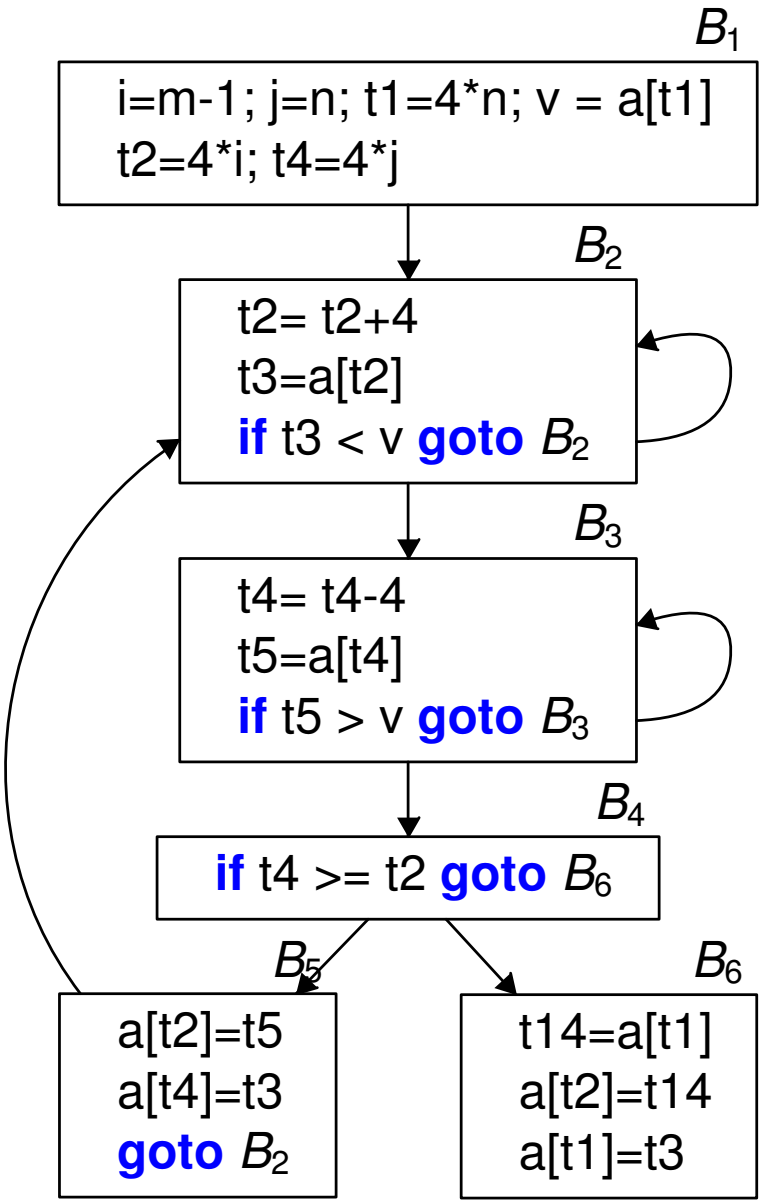

B_1
i=m-1; j=n; t1=4*n; v = a[t1]
t2=4*i; t4=4*j
B_2
i=i+1; t2= t2+4
t3=a[t2]
if t3 < v goto B_2
B_3
j=j-1; t4= t4-4
t5=a[t4]
if t5 > v goto B_3
B_4
if i >= j goto B_6
B_5
a[t2]=t5
a[t4]=t3
goto B_2
B_6
t14=a[t1]
a[t2]=t14
a[t1]=t3

Simplification supplémentaire

- On analyse ensuite la boucle B_2, B_3, B_4, B_5
- Les variables i et j ne sont utilisées que pour le test $i \geq j$
- La variable i est corrélée à $t2 = 4 * i$ et la variable j est corrélée à $t4 = 4 * i$.
- Le test $i \geq j$ est équivalent à $t2 \geq t4$.
- On peut alors éliminer les variables i et j dans les autres blocs que B_1.

Résultat transformation du code





Conclusion sur les optimisations

- Les optimisations sont des opérations complexes qui peuvent ralentir sensiblement la compilation.
- Les optimisations doivent préserver la sémantique des programmes.
- Les optimisations peuvent avoir des effets divers sur la taille du code et sa vitesse d'exécution.
- Les compilateurs proposent différentes options pour utiliser ou non certaines classes d'optimisations.
- Les optimisations en accroissant la distance entre le code source et le code exécuté peuvent rendre difficile voir impossible l'utilisation de debugueurs.

Sommaire

Résumé

Ce que l'on a vu dans ce cours

- La compilation est au cœur de la programmation et par conséquent de l'informatique.
- La compilation met en œuvre des notions avancées (théorie, structures de données et algorithmes) :
  - théorie des langages formels: automates finis, automates à pile
  - sémantique des langages de programmation: typage, inférence, évaluation
  - optimisation de code: manipulation de graphes, architecture des ordinateurs

Ce qu'il faut retenir

- La distinction entre ce qui peut être traité par l'analyse lexicale, l'analyse syntaxique et l'analyse sémantique.
- La correspondance entre les langages reconnus et les expressions régulières et les grammaires.
- Le fonctionnement de l'analyse descendante et de l'analyse ascendante (en particulier la notion de lecture, réduction et les conflits qui en découlent).
- Le rôle des règles de précédence dans la résolution des conflits.
- La mise en œuvre d'un analyseur lexical pour les transformations simples de texte.

Analyse sémantique

Ce qu'il faut retenir

- La description d'un arbre de syntaxe abstraite.
- L'analyse de portée (la construction et la représentation de la table des symboles).
- Le rôle du typage, les règles de typage simple.
- L'analyse de flots de données.

Génération de code

Ce qu'il faut retenir

- Les constructions de base d'une machine à pile.
- La compilation des procédures à l'aide de tableaux d'activation.
- La représentation en machine de données complexes (structures, objets, fonctions).
- Des langages intermédiaires: code à trois adresses ou bien SSA (Static Single-Assignement form) où chaque variable n'est définie qu'une seule fois.
- La problématique de l'allocation de registres.