

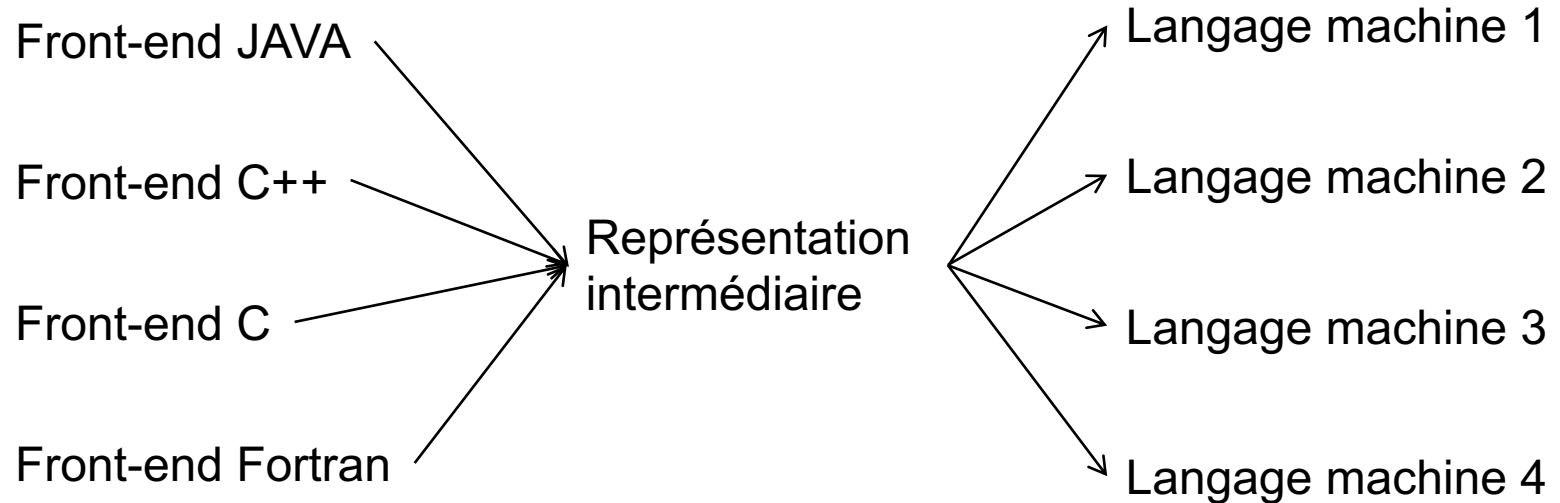
# LOG3210

## Cours 6

Génération de code intermédiaire

# Génération de code intermédiaire

- But de la représentation intermédiaire: interface entre le front-end (spécifique au langage) et le back-end (spécifique à la machine).



# Graphe de flux de contrôle

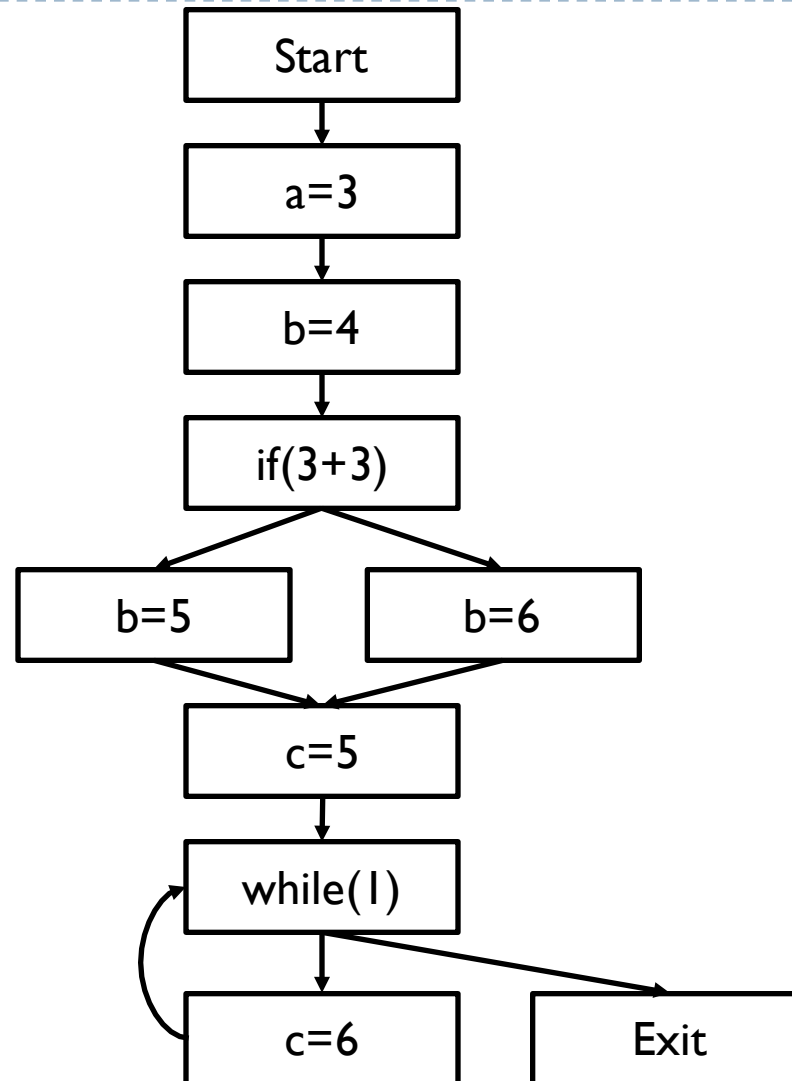
---

- ▶ Les définitions et les traductions dirigées par la syntaxe permettent de générer le graphe de flux de contrôle d'un programme.
- ▶ Le graphe de flux de contrôle (*Control Flow Graph*, CFG) est une représentation, sous forme de graphe, d'un programme.
- ▶ Les chemins dans un graphe de flux de contrôle représentent une sur-approximation de tous les chemins d'exécution d'un programme.

# CFG

## Exemple de CFG

```
a=3;  
b=4;  
if(3+3) {  
    b=5;  
} else {  
    b=6;  
}  
c=5;  
while(1) {  
    c=6;  
}
```



# Construction du CFG

---

- ▶ Typiquement, chaque instruction devient un nœud dans le CFG.
- ▶ Il y a un arc entre le nœud A et le nœud B s'il est possible que le contenu du nœud B soit exécuté immédiatement après le contenu du nœud A.
- ▶ Les instructions de contrôle, tels les *if*, *for*, *while*, etc., génèrent plusieurs arcs dans le graphe de flux de contrôle.

# Construction du CFG

---

## ► Exemple

```
int test() {  
    int a = 2;  
    int b = 3;  
    a *= 3;  
    if (a > 3)  
        b = b+1;  
    return b - a;  
}
```

# Construction du CFG

## Boucles *for*

- ▶ Certaines instructions, notamment les boucles *for*, doivent parfois être séparées en plusieurs nœuds.
- ▶ Exemple

```
int test() {  
    int a = 2;  
    for (int i = 0; i < 3; i++)  
        a *= a;  
    return a;  
}
```

# Construction du CFG

## Blocs d'instructions

- ▶ Il est possible de réunir certaines instructions au sein d'un même bloc
  - ▶ Cette approche est surtout applicable pour du code à trois adresses
  
- ▶ Algorithme de base:
  1. Identifier des instructions *leader*:
    1. La première instruction d'un programme est un *leader*
    2. Toute instruction qui est la cible d'un saut conditionnel ou inconditionnel est un *leader*
    3. Toute instruction qui suit un saut conditionnel ou inconditionnel est un *leader*
  2. Pour chaque *leader*, on crée un bloc qui contient le *leader* et toutes les instructions qui le suivent, jusqu'au prochain *leader* ou la fin du programme.



# Exercice 1: wordcount

---

```
#define YES 1
#define NO 0

main() {
    int c, nl, nw, nc, inword;
    inword = NO;
    nl = 0;
    nw = 0;
    nc = 0;
    c = getchar();
    while(c != EOF) {
        nc = nc + 1;
        if(c == '\n')
            nl = nl + 1;
        if(c == ' ' || c == '\n' || c == '\t')
            inword = NO;
```

# Exercice 1: wordcount (partie 2)

---

```
        else if (inword == NO) {  
            inword = YES;  
            nw = nw + 1;  
        }  
        c = getchar();  
    }  
    printf("%d \n", nl);  
    printf("%d \n", nw);  
    printf("%d \n", nc);  
}
```

**Exercice: construisez le CFG de ce programme où chaque bloc contiendra une seule instruction.**

# Applications du CFG

---

- ▶ Le CFG nous permet de faire des approximations statiques (sans exécution) du comportement dynamique (à l'exécution) des programmes.
- ▶ Le CFG trouve des applications dans différents domaines:
  - ▶ Optimisation de programmes par analyses de flux
  - ▶ Débogage de logiciels
  - ▶ Analyse automatique de programmes
  - ▶ Tests à boîte blanche

# Code à trois adresses

---

- ▶ Le code à trois adresses est une représentation d'un programme.
- ▶ Au plus un opérateur du côté droit d'une instruction.
- ▶ Exemple:

$x + y * z$  devient

$$t_1 = y * z$$

$$t_2 = x + t_1$$

où  $t_1$  et  $t_2$  sont des variables temporaires générées par le compilateur.

# Adresses et instructions

---

- ▶ Conceptuellement, le code à trois adresses est composé d'adresses et d'instructions.
  
- ▶ Les adresses peuvent être:
  1. **Un nom** : Un nom de variable dans le code source par exemple (où un pointeur vers la table de symboles).
  2. **Une constante**: Un type par exemple ou une valeur numérique.
  3. **Un temporaire généré automatiquement**: Voir l'exemple précédent.

# Instructions

---

- ▶ Dans le code 3 adresses, nous considérerons les instructions suivantes:
  1. Assignment:  $x = y \text{ op } z$  où  $op$  est un opérateur arithmétique binaire ou un opérateur logique et  $x, y, z$  sont des adresses.
  2. Assignment unaire:  $x = op \ y$  où  $op$  est un opérateur unaire comme la négation, le moins unaire ou l'opérateur de *cast*.
  3. Instruction de copie:  $x = y$  où  $x$  et  $y$  sont des adresses.
  4. Saut non conditionnel: `goto L`. L'instruction à l'index  $L$  est exécuté.

# Instructions (suite)

---

## 5. Sauts conditionnels de la forme:

- ▶ `if x goto L`
- ▶ `ifFalse x goto L.`

## 6. Sauts conditionnels de la forme:

- ▶ `If x relop y goto L` où *relop* est un opérateur relationnel (<, >, <=, >=, ==, etc.)

## 7. Copies indexées:

- ▶ `x = y[i]`
- ▶ `x[i] = y`

# Instructions (suite)

8. Appels à procédure et retours de fonction. Supposons l'appel suivant:  $p(x_1, x_2, \dots, x_n)$ . Le code trois adresses correspondant serait:

- ▶ param  $x_1$
- ▶ param  $x_2$
- ▶ ...
- ▶ param  $x_n$
- ▶ call  $p, n$

9. Assignations de pointeurs et d'adresses

- ▶  $x = \&y$
- ▶  $x = *y$
- ▶  $*x = y$



# Étiquettes des instructions

---

## Étiquettes symboliques

L:       $t_1 = i + 1$   
          $i = t_1$   
          $t_2 = i * 8$   
          $t_3 = a[t_2]$   
          $\text{if } t_3 < v \text{ goto L}$

## Numéros de position

100:     $t_1 = i + 1$   
101:     $i = t_1$   
102:     $t_2 = i * 8$   
103:     $t_3 = a[t_2]$   
104:     $\text{if } t_3 < v \text{ goto 100}$

# Implémentation code 3 adresses

---

- ▶ Deux représentations possibles: quadruplets (*quads*) et triplets (*triples*).
- ▶ Champs d'un quadruplet:
  1. Opérateur *op*
  2. Premier argument *arg1*
  3. Second argument *arg2*
  4. Résultat *result*
- ▶ Exemple avec l'expression  $a = b * -c + b * -c ;$

# Implémentation code 3 adresses

## ► Champs d'un triplet:

1. Opérateur *op*
2. Premier argument *arg1*
3. Second argument *arg2*

## ► Exemple $a = b * -c + b * -c ;$

## ► Les chiffres entre parenthèses représentent l'index d'un triplet.

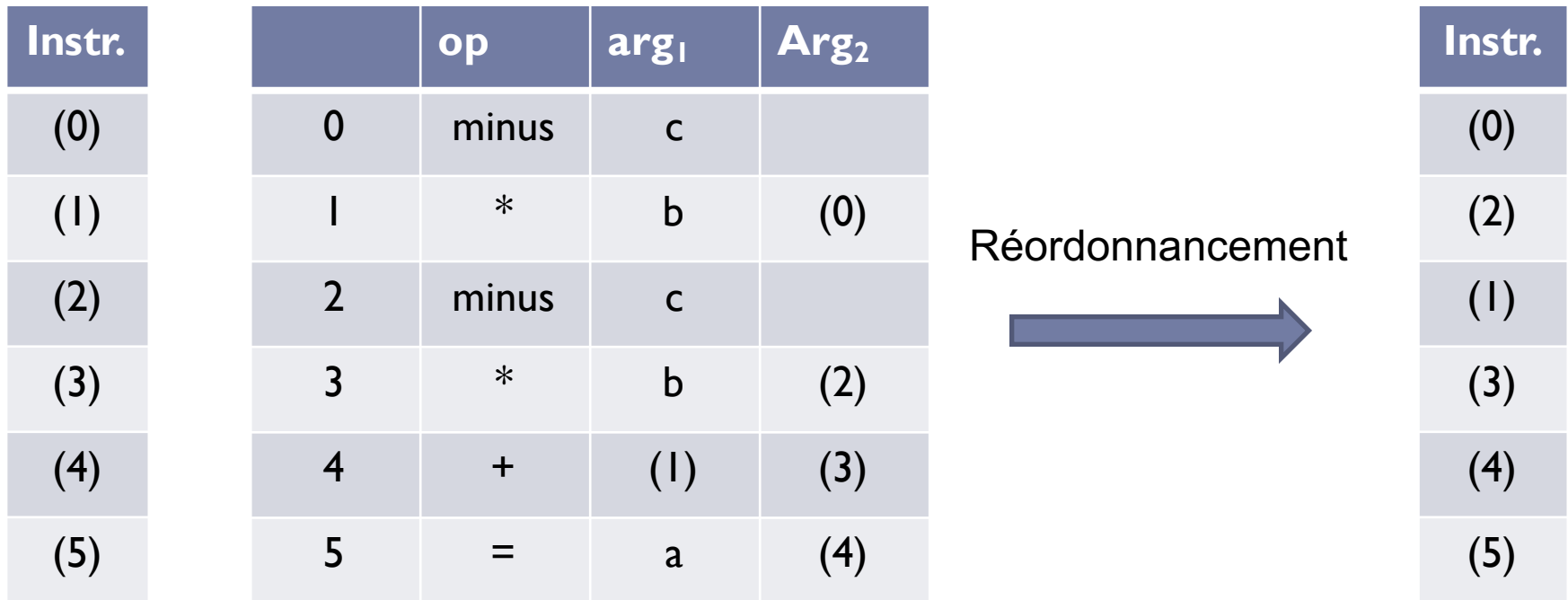
	op	arg <sub>1</sub>	Arg <sub>2</sub>
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

# Quads vs. triples

---

- ▶ S'il est nécessaire de déplacer des instructions, pour des opérations d'optimisation par exemple, la représentation par quadruplets peut être avantageuse.
- ▶ Dans une représentation par triplets, il faut recalculer les index si les instructions sont déplacées.
- ▶ Une représentation par triplets indirects peut régler ce problème.

# Triplets indirects



# Static single assignment

- ▶ La représentation static single assignment (SSA) est largement utilisée dans les compilateurs avec optimisations car elle facilite grandement certains calculs.
- ▶ Dans cette représentation, chaque variable est assignée **une seule** fois.

$$p = a + b$$

$$q = p - c$$

$$p = q * d$$

$$p = e - p$$

$$q = p + q$$

$$p_1 = a + b$$

$$q_1 = p_1 - c$$

$$p_2 = q_1 * d$$

$$p_3 = e - p_2$$


$$q_2 = p_3 + q_1$$

# Static single assignment (suite)

- ▶ Dans la forme SSA, les structures de contrôle peuvent poser problème. On utilise alors la fonction  $\phi$ .

- ▶ Exemple:     if (flag)  $x = -1$ ; else  $x = 1$ ;  $y = x * a$ ;

- ▶ En SSA:
 

<pre> if flag goto L<sub>1</sub> goto L<sub>2</sub> L<sub>1</sub>:  x<sub>1</sub> = -1       goto L<sub>3</sub> L<sub>2</sub>:  x<sub>2</sub> = 1 L<sub>3</sub>:  y = ??? * a           </pre>		<pre> if flag goto L<sub>1</sub> goto L<sub>2</sub> L<sub>1</sub>:  x<sub>1</sub> = -1       goto L<sub>3</sub> L<sub>2</sub>:  x<sub>2</sub> = 1 L<sub>3</sub>:  x<sub>3</sub> = <math>\phi(x_1, x_2)</math>;       y = x<sub>3</sub> * a           </pre>
--	--	---