

TD – Génération de code

Dans ce sujet, on suppose qu'on travaille sur un langage source « smallC » dont la grammaire type « C » est donnée en annexe. Le but est de traduire le programme suivant en langage d'assemblage pour une machine à registres simple décrite plus loin.

```
a = 12;
b = 13;

while (i > a * 2) {
    if (i == a + b * 5) {
        return a;
        d = 15;
    } else {
        c = 12;
        return b;
    }
}

return (2 + 5 * 3);
```

La machine cible est une machine à registres simple avec 8 registres nommés R0 à R7. Le langage d'assemblage de cette machine est le suivant :

Instruction	Description
LD Rn, x	Charger la valeur de la variable x dans Rn
ST Rn, x	Stocker la valeur de Rn dans la variable x
MOV Rn, [Rm imm]	Copier la valeur de Rm ou de imm dans Rn
ADD Rn, [Rm imm]	Ajouter la valeur de Rm ou de imm à Rn
SUB Rn, [Rm imm]	Soustraire la valeur de Rm ou de imm à Rn
MUL Rn, [Rm imm]	Multiplier la valeur de Rn par la valeur de Rm ou de imm
DIV Rn, [Rm imm]	Diviser la valeur de Rn par la valeur de Rm ou de imm
CMP Rn, [Rm imm]	Comparer la valeur de Rn avec la valeur de Rm ou de imm
B label	Sauter à l'étiquette label
BEQ label	Sauter à l'étiquette label si égal
BNE label	Sauter à l'étiquette label si différent
BGT label	Sauter à l'étiquette label si supérieur

Exercice 1 : arbre de syntaxe abstraite (*Abstract Syntax Tree*)

Question 1. En étudiant la grammaire du langage smallC, proposez les types de nœuds qui vous semblent nécessaires pour construire l'AST d'un programme et pertinents pour la génération de code.

Question 2. Donnez l'AST correspondant au programme.

Exercice 2 : code trois adresses (*Three Address Code*)

L'idée est de traduire l'AST en un code intermédiaire linéaire appelé *code trois adresses* (TAC) pour lequel chaque instruction est de la forme `address op arg1 arg2 r`. On associe à chaque instruction TAC une forme lisible.

op	a1	a2	r	Écriture
+	cvt	cvt	vt	$r = a1 + a2$
>	cvt	cvt	e	if $a1 > a2$ goto e
goto	e			goto e
return	a1			return a1

- c : constante
- v : variable
- t : temporaire
- e : étiquette

Question 1. Pour chaque type de nœud utile de l'AST, donnez ses règles de traduction vers le TAC. Il peut y avoir plusieurs règles pour un même nœud selon le contexte (type des opérandes, etc.)

Question 2. Donnez le code TAC correspondant au programme.

Exercice 3 : graphe de flot de contrôle (*control flow graph*)

À partir du TAC, on peut construire un graphe de flot de contrôle qui représente les chemins d'exécution possibles dans le code initial. À cette fin on définit la notion de *leader* et de bloc de base (*basic block*).

- Un *leader* est une instruction qui est le premier élément d'un bloc de base. Il est déterminé par les règles suivantes :
 - La première instruction du programme est un leader.
 - Une instruction qui est la cible d'un saut direct (instruction `goto` ou d'un saut conditionnel) est un leader.
 - Une instruction qui suit une instruction de saut est un leader.
- Un *bloc de base* est une séquence d'instructions qui s'exécute de manière linéaire. Il commence par un leader et se termine immédiatement avant le leader suivant ou la fin du programme.

Question 1. À partir du TAC du programme, construisez le graphe de flot de contrôle.

Question 2. Quelles optimisations peut-on appliquer sur le graphe de contrôle de flot? Donnez le nouveau code TAC après les optimisations.

Exercice 4 : analyse de vie des variables

L'analyse de vie est une technique qui permet de déterminer quelles variables sont nécessaires à chaque point du programme. Elle est utile pour optimiser l'utilisation des registres et minimiser les accès mémoire.

Pour déterminer la vie d'une variable, on va d'abord définir deux ensembles pour chaque bloc de base :

- **use** : variables utilisées,
- **def** : variables définies,

puis calculer **in** et **out** :

- **in** : ensemble des variables qui sont utilisées avant d'être définies dans le bloc de base.
- **out** : ensemble des variables qui sont utilisées après le bloc de base.

Question 1. À partir du graphe de contrôle de flot, calculez les ensembles **use**, **def**, **in** et **out** pour chaque bloc de base. **Attention** pour cette partie chaque bloc de base ne peut contenir qu'une seule instruction.

Le langage *smallC* ne supporte pas les fonctions, donc il n'y a pas de variables locales. Si on voulait en ajouter, on pourrait en calculer leur durée de vie de la même manière.

Question 2. En supposant maintenant que le programme est une fonction (donc qu'il n'y a que des variables locales), calculer la vie des variables locales.

Question 3. Dans ce cas, à partir de l'analyse de vie des variables locales, quelles optimisations peut-on appliquer sur le programme?

Exercice 5 : allocation des registres

L'allocation des registres est une étape importante de la génération de code qui consiste à assigner les variables aux registres disponibles. On va utiliser l'analyse de vie pour déterminer quelles variables peuvent être assignées aux registres. On considère que le programme est une fonction et qu'on veut que toutes les variables (temporaires ou non) soient dans des registres.

Question 1. À partir de l'analyse de vie des variables du programme, construisez un graphe d'interférence où chaque sommet représente une variable et les arêtes représentent les interférences entre ces variables. Deux variables sont en interférence si elles sont utilisées en même temps dans le programme.

Question 2. À partir du graphe d'interférence, appliquez un algorithme de coloration de graphe pour assigner les variables aux registres disponibles.

Exercice 6 : génération du code cible final

On veut maintenant générer le code d'assemblage final à partir du code TAC optimisé. Pour chaque instruction TAC, il faudra générer une ou plusieurs instructions d'assemblage. Cette étape s'appelle la sélection d'instructions. On remplacera les registres fictifs du code TAC par les registres physiques qui leur sont associés.

Question 1. Donnez le code d'assemblage pour le programme.

Annexe : grammaire du langage smallC

```

<program> ::= <stmt_list> | ""

<stmt_list> ::= <stmt> <stmt_list> | <stmt>

<stmt> ::= <assign_stmt> | <if_stmt> | <while_stmt> | <return_stmt>

<assign_stmt> ::= <ident> "=" <eq_expr> ";"

<if_stmt> ::= "if" "(" <eq_expr> ")" "{" <stmt_list> "}"
           | "if" "(" <eq_expr> ")" "{" <stmt_list> "}" "else" "{" <stmt_list> "}"

<while_stmt> ::= "while" "(" <eq_expr> ")" "{" <stmt_list> "}"

<return_stmt> ::= "return" <eq_expr> ";"

<eq_expr> ::= <rel_expr> | <eq_expr> "==" <rel_expr> | <eq_expr> "!=" <rel_expr>

<rel_expr> ::= <add_expr>
           | <rel_expr> "<" <add_expr>
           | <rel_expr> "<=" <add_expr>
           | <rel_expr> ">" <add_expr>
           | <rel_expr> ">=" <add_expr>

<add_expr> ::= <mul_expr>
           | <add_expr> "+" <mul_expr>
           | <add_expr> "-" <mul_expr>

<mul_expr> ::= <primary_expr>
           | <mul_expr> "*" <primary_expr>
           | <mul_expr> "/" <primary_expr>

<primary_expr> ::= <identifier> | <constant> | "(" <eq_expr> ")"

```

Solution de l'exercice 1

Question 1. L'idée est de représenter chaque nœud de l'AST par une structure qui permet de stocker les informations nécessaires pour la génération de code. La grammaire ambiguë de l'AST doit ressembler à ceci :

$$P \rightarrow SL \quad (1)$$

$$P \rightarrow \varepsilon \quad (2)$$

$$SL \rightarrow S SL \quad (3)$$

$$SL \rightarrow \varepsilon \quad (4)$$

$$S \rightarrow ID = E \quad (5)$$

$$S \rightarrow \text{if } E \text{ } SL \quad (6)$$

$$S \rightarrow \text{while } E \text{ } SL \quad (7)$$

$$S \rightarrow \text{return } E \quad (8)$$

$$E \rightarrow E \text{ op } E \quad (9)$$

$$E \rightarrow ID \quad (10)$$

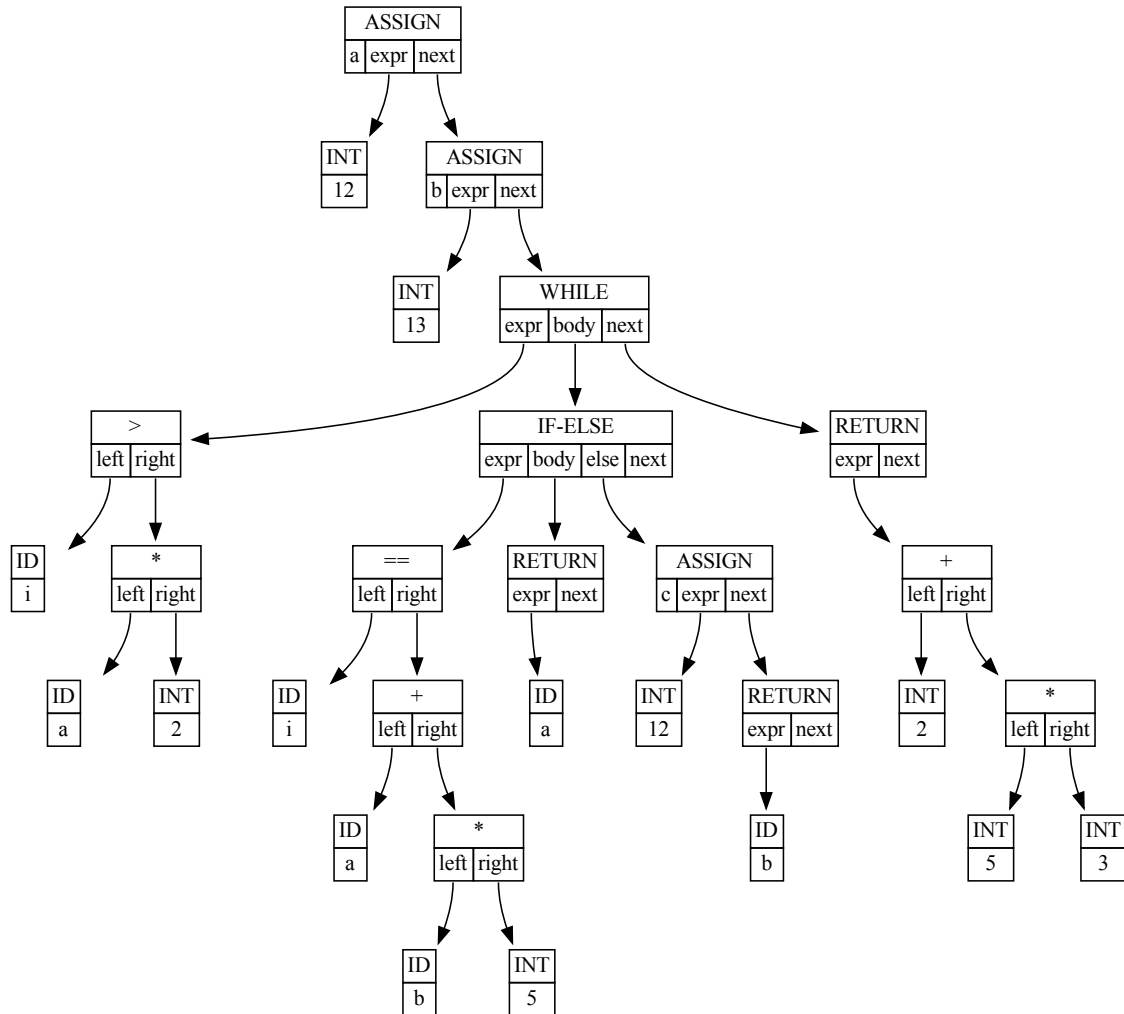
$$E \rightarrow n \quad (11)$$

$$(12)$$

On peut représenter chaque nœud de l'AST par une structure qui reprend les informations données dans les règles de la grammaire. Pour la production $E \rightarrow E \text{ op } E$, on peut par exemple avoir une structure de nœud qui contient les deux opérandes et l'opérateur. Pour la production $E \rightarrow ID$, on peut avoir une structure de nœud qui contient le nom de la variable. Pour la production $\text{if } E \text{ } SL \text{ } SL$, on peut avoir une structure

de nœud qui contient la condition et les deux branches du if. Enfin, pour chaque nœud d'instruction, on peut ajouter un champ qui indique l'instruction suivante pour capturer la production $SL \rightarrow S$.

Question 2. L'AST correspondant au programme est le suivant :



Solution de l'exercice 2

Question 1. Pour chaque type de nœud utile de l'AST, on peut définir les règles de traduction suivantes vers le TAC :

- $E \rightarrow E_1 + E_2$
 - $E.t = temp$
 - $gen(E.t = E_1.t + E_2.t)$
- $E \rightarrow ID$
 - $E.t = ID$
- $E \rightarrow n$
 - $E.t = n$
- $S \rightarrow ifESL_2$
 - $e1 = label$

```

—  $e2 = label$ 
—  $gen(if\ E.l.t\ E.op\ E.r.t\ goto\ e1)$ 
—  $gen(SL_1)$ 
—  $gen(gotoe2)$ 
—  $gen(e1 : SL_2)$ 
—  $gen(goto\ e2)$ 
—  $gen(e2 :)$ 
—  $S \rightarrow whileESL$ 
—  $e1 = label$ 
—  $e2 = label$ 
—  $gen(e1 :)$ 
—  $gen(if\ E.l.t\ E.op\ E.r.t\ gotoe2)$ 
—  $gen(SL)$ 
—  $gen(goto\ e1)$ 
—  $gen(e2 :)$ 

```

Question 2. Le code TAC correspondant au programme est le suivant :

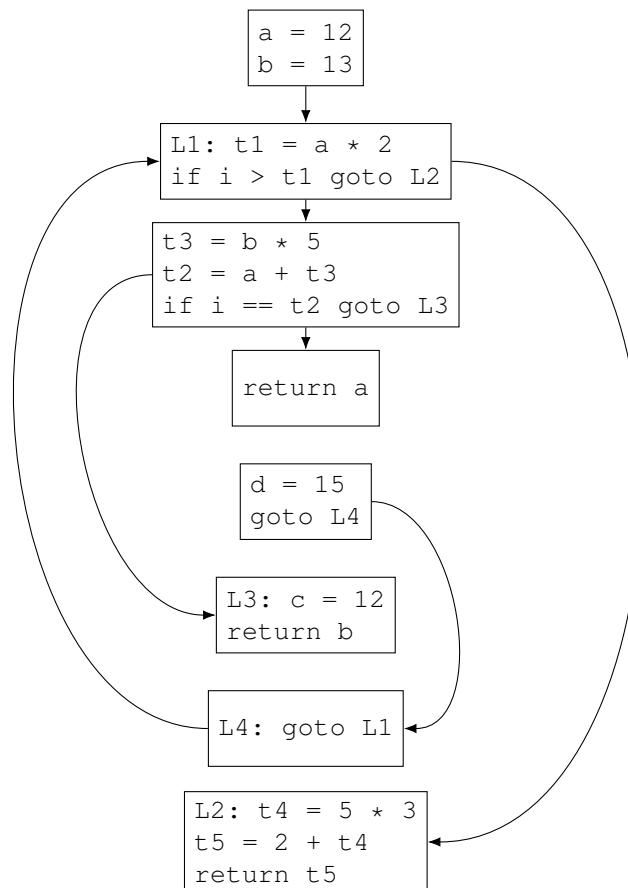
```

00:      a = 12
01:      b = 13
02:      i = 10
02:      L1: t1 = a * 2
03:      if i > t1 goto L2
04:      t3 = b * 5
05:      t2 = a + t3
06:      if i == t2 goto L3
07:      return a
08:      d = 15
09:      goto L4
10:      L3: c = 12
11:      return b
12:      L4: goto L1
13:      L2: t4 = 5 * 3
14:      t5 = 2 + t4
15:      return t5

```

Solution de l'exercice 3

Question 1. Le graphe de contrôle de flot du programme est le suivant :



Question 2. Les optimisations possibles sur le graphe de contrôle de flot incluent :

- Suppression des instructions inutiles (par exemple, des instructions qui ne sont jamais atteintes).
- Fusion de blocs de base adjacents si possible.
- Élimination des sauts inutiles.

Après les optimisations, le code TAC pourrait ressembler à ceci :

```

00:      a = 12
01:      b = 13
02:      L1: t1 = a * 2
03:      if i > t1 goto L2
04:      t3 = b * 5
05:      t2 = a + t3
06:      if i == t2 goto L3
07:      return a
08:      L3: c = 12
09:      return b
10:      L2: t4 = 5 * 3
11:      t5 = 2 + t4
12:      return t5

```

Notez que les instructions de saut vers L4 ont été supprimées car elles ne sont pas nécessaires. De plus, le bloc de base L4 a été fusionné avec L1 car il n'y a pas de saut vers L4 dans le programme optimisé. L'étiquette L1 n'est plus nécessaire

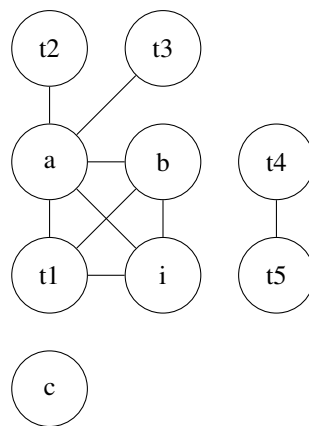
Solution de l'exercice 4

Question 1. La durée de vie des variables temporaires dans le programme est la suivante :

Noeud	Succ	def	use	Iteration 1		Iteration 2		Iteration 3		Iteration 4		Iteration 5	
				in	out	in	out	in	out	in	out	in	out
0	1	a							a		a,i	i	a,i
1	2	b					a	a	a,i,b	a,i	a,i,b	a,i	a,i,b
2	3	t1	a	a	i,t1	a	i,t1,b	a,i,b	i,t1,a,b	a,i,b	i,t1,a,b	a,i,b	i,t1,a,b
3	4,10		i, t1	i,t1	b	i,t1,b	a,b	i,t1,a,b	a,b	i,t1,a,b	a,b	i,t1,a,b	a,b
4	5	t3	b	b	a,t3	a,b	a,t3	a,b	a,t3	a,b	a,t3	a,b	a,t3
5	6	t2	a,t3	a,t3	t2	a,t3	a,t2	a,t3	a,t2	a,t3	a,t2	a,t3	a,t2
6	7,8		t2	t2	a	a,t2	a	a,t2	a	a,t2	a	a,t2	a
7			a	a		a		a		a		a	
8	9	c											
9			b	b		b		b		b		b	
10	11	t4			t4		t4		t4		t4		t4
11	12	t5	t4	t4	t5	t4	t5	t4	t5	t4	t5	t4	t5
12			t5	t5		t5		t5		t5		t5	

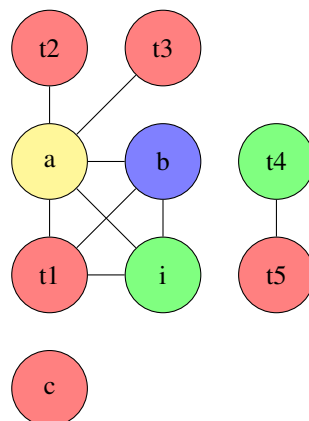
Question 3. On peut observer que la variable c n'existe que dans l'ensemble **def**. On pourrait donc la supprimer du programme ainsi les lignes de TAC associées.

Solution de l'exercice 5 Question 1. Le graphe d'interférence du programme est le suivant :



Question 2. Pour colorer le graphe d'interférence, on peut utiliser un algorithme de coloration de graphe glouton. L'idée est de choisir un nœud dont le degré est plus petit que K et de le retirer du graphe et d'ajouter le nœud à une pile. On recommence jusqu'à ce que tous les nœuds soient retirés. Ensuite, on peut colorier les nœuds en partant de la pile et en assignant la première couleur disponible à chaque nœud.

Le graphe coloré final est :



En affectant les couleurs aux registres physiques (dans l'ordre RVBJ), on obtient :

Variable	Registre physique
a	R3
b	R2
c	R0
i	R1
t1	R0
t2	R0
t3	R0
t4	R1
t5	R0

Si le nombre de variables locales et de temporaires est supérieur au nombre de registres disponibles, on peut utiliser une technique appelée débordement (« spilling ») qui consiste à stocker certaines variables dans la mémoire plutôt que dans les registres. En général, elles seront stockées sur la pile.

Solution de l'exercice 6

Question 1. Pour générer le code d'assemblage final à partir du code TAC optimisé, il faut définir des règles de sélection d'instructions cible pour chaque instruction TAC. Il peut y avoir plusieurs règles pour une même instruction TAC selon le contexte (type des opérandes, etc.).

Par exemple, pour l'instruction TAC $t1 = t2 * 2$ le code d'assemblage pourrait être :

```
MOV t1, t2      ; Charger t2 dans t1
MUL t1, 2       ; Multiplier t2 par la constante
```

ou pour l'instruction TAC $t1 = t2 * t3$ le code d'assemblage pourrait être :

```
MOV t1, t2
MUL t1, t3
```

Pour l'instruction TAC `if i > t1 goto L2`, le code d'assemblage pourrait être :

```
CMP i, t1      ; Comparer i et t1
BGT L2         ; Sauter à L2 si i > t1
```

mais pour `if 3 > t1 goto L2`, le code d'assemblage pourrait être :

```
CMP t1, 3      ; Comparer t1 et 3
BLE L2         ; Sauter à L2 si R0 <= 3
```

Le code d'assemblage final pour le programme donné est avant l'allocation des registres :

```
MOV A, 12
MOV B, 13
MOV T1, a
MUL T1, 2
CMP I, T1
BGT L2
MOV T3, B
MUL T3, 5
MOV T2, T3
ADD T2, A
CMP I, T2
BEQ L3
MOV R0, A
RET

L3: MOV C, 12
    MOV R0, B
    RET

L2: MOV T4, 5
    MUL T4, 3
    MOV T5, 2
    ADD T5, T4
    MOV R0, T5
    RET
```

Il ne reste plus qu'à remplacer les registres fictifs par les registres physiques qui leur sont associés et on obtient :

```
MOV R3, 12
MOV R2, 13
MOV R0, R3
MUL R0, 2
CMP R1, R0
BGT L2
MOV R0, R2
MUL R0, 5
MOV R0, R0
ADD R0, R3
CMP R1, R0
BEQ L3
MOV R0, R3
RET

L3: MOV R0, 12
    MOV R0, R2
    RET

L2: MOV R1, 5
    MUL R1, 3
    MOV R0, 2
    ADD R0, R1
    MOV R0, R0
    RET
```

Une dernière optimisation que ferait un compilateur serait de parcourir le code d'assemblage pour optimiser les instructions de chargement et de stockage (suppression des MOV redondants par exemple). On appelle ce genre d'optimisation *peephole optimization*.