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> ")"
```