Documents : interdits **Durée :** 1h **Barème :** /**20** indicatif

Préambule

On rappelle la syntaxe de la représentation intermédiaire du cours[1], sachant que les labels sont de la forme `L1,L2` et les temporaires de la forme `t1,t2...` :

```
E := Int|Const|Temp|Read(Temp)|Unary(op,E)|Binary(op,E,E)
I := Jump(Label)|CJump(E,Label,Label)|Write(Temp,E)|Call(Frame,List<E>)
```

Exercice 1 : Linéarisation

Question 1 (/3) À quoi servent les frames ? Peut-on compiler un langage avec des fonctions sans ce mécanisme ? Si oui, à quelles conditions, si non, pourquoi ?

Question 2 (/6) Donnez la traduction en code intermédiaire du programme suivant. Vous devez représenter les frames en plus des commandes générées (indication : on attend entre 15 et 20 commandes en tout).

Pour rappel, Les frames doivent contenir les informations suivantes :

- paramètres
- temporaires locaux
- résultat
- point d'entrée et point de sortie
- (inutile d'indiquer la taille ici)

```
int f(int x, bool b){
   if(b){return -(x+3);}
   else{
      int y=0;
      while(true||false){
         y=f(y,not(b));
         x=y;
      }
      return y;
   }
}
```

```
int main(){
   int x = abs(25+2-42*8);
   x= x + f(x,x<10);
   return x;
}
```

Question 3 (/2) À quoi sert une représentation intermédiaire (en général) ?

[1] Les opérations sont les mêmes que celles de l'AST.

Exercice 2 : Assembleur

On considère le langage assembleur MIPS32, dont certaines instructions sont rappelées dans le tableau ci-dessous. On rappelle également que $sp est le pointeur de sommet de pile, que les adresses de la pile vont décroissant, que les mots sont de taille 4 (4 octets=32bits).

mv r1 r2	r1 ← r2 (registre à registre)
li r1 42	r1 ← 42 (entier à registre)
sw r1 adr	adr ← r1 (registre à adresse)
lw r1 adr	r1 ← adr (adresse à registre)
j L	saute au label d'instruction L
jal L	saute au label L, et enregistre l'adresse de la prochaine instruction dans $ra.
op r1 r2 r3	r1 ← r2 op r3 (op= add, sub, mul,)
beq r1 r2 L	← saute à L si r1=r2 (sinon continue à l'instruction suivante)
syscall	← appel système lisant $v0. (10=>exit, 1=>impression de l'entier stocké en $a0.

On considère également le code suivant, généré par un compilateur[2]depuis le langage `sdm` vu en TP :

```
main:
        jal L0
        li $v0, 10
        syscall
L0:
        li $t0, 12
        sub $sp, 4
        sw $t0, 4($sp)
L2:
        li $t0, 1
        sub $sp, 4
        sw $t0, 4($sp)
        lw $t0, 4($sp)
        add $sp, 4
        beq $t0, $zero, L4
        j L3
L3:
        li $t0, 1
        sub $sp, 4
        sw $t0, 4($sp)
        lw $t2, 4($sp)
        add $sp, 4
        lw $t1, 4($sp)
        add $sp, 4
        add $t0, $t1, $t2
        sub $sp, 4
        sw $t0, 4($sp)
        lw $t0, 4($sp)
        add $sp, 4

        sub $sp, 4
        sw $t0, 4($sp)
        lw $a0, 4($sp)
        jal PrintInt
        j L2
L4:
        li $t0, 0
        sub $sp, 4
        sw $t0, 4($sp)
        lw $t0, 4($sp)
        add $sp, 4
        j $ra

PrintInt:
        li $v0, 1
        syscall
        j $ra
```

Question 1 (/6) Que fait ce code ?

1. Décrivez le résultat produit à l'exécution de ce code.
2. Décompilation : Proposez un programme sdm qui pourrait être la source du code assembleur produit.

Question 2 : optimisations(/3) Ce code utilise beaucoup la pile, mais il peut être simplifié.

1. Peut-on le coder sans utiliser la pile ? Si oui, indiquez les modifications à apporter. Si non, expliquez pourquoi.

[2]Et légèrement simplifié.