Master Info - 2017-2018
MIF08 Compilation

Contrôle continu 1 - Durée 42 min (tiers-temps 56 min)

Attention ce sujet est recto-verso ! Tous documents papiers et électroniques interdits. Répondre sur la feuille. Vous pouvez garder l'annexe. Le barème est indicatif.
Nom : ..... **Prénom** .....

On rappelle en annexe les règles de génération de code pour les affectations, le test, et certaines règles de typage. On considère le programme suivant :

```
var x2: int;
x2 = 13;
while (x2 > 8) do
   x2 = x2 - 1;
done
```

Question 1 (8 points)

Montrer que ce programme est bien typé (déclarations, instructions).

Question 2 (12 points)

Générer le code 3 adresses LEIA pour ce programme, en suivant à la lettre les règles de génération de code données en annexe. Les calculs intermédiaires seront documentés. **Vous mettrez la feuille courante en format paysage, et vous utiliserez trois colonnes : une pour décrire les appels récursifs, une pour le code généré et une pour les temporaires.**

Question 3 (Bonus points)

Remplacez le saut conditionnel du code généré par les instructions `snif` adéquates.

MIF08 Feuille d'accompagnement

Mini-while (syntaxe abstraite)

Expressions Booléennes :

$$\begin{array}{rcll} b & ::= & true & constante \\ & | & false & constante \\ & | & b\ or\ b & ou \\ & | & b\ and\ b & et \\ & | & \ldots & \end{array}$$

Expressions numériques :

$$\begin{array}{rcll} e & ::= & c & constante \\ & | & x & variable \\ & | & e + e & addition \\ & | & e \times e & multiplication \\ & | & \ldots & \end{array}$$

Instructions :

$$\begin{array}{rcll} S(Smt) & ::= & x := e & \text{affectation} \\ & | & skip & \text{rien} \\ & | & S_1; S_2 & \text{séquence} \\ & | & \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 & \text{test} \\ & | & \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done} & \text{boucle} \end{array}$$

Typage

On ajoute les déclarations dans le langage :

$$D(decl) \quad ::= \quad var\ x : t \quad \text{type declaration}$$

Les informations de déclaration fournissent $\Gamma : Var \to Basetype$ construit à l'aide des règles :

$$\frac{}{var\ x : t \to_d [x \mapsto t]}$$

$$\frac{D_1 \to_d \Gamma_1 \quad D_2 \to_d \Gamma_2 \quad Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset}{D1; D_2 \to_d \Gamma_1 \cup \Gamma_2}$$

Ainsi la déclaration `var x,y:int;` fournit $\Gamma : \begin{cases} x & \to int \\ y & \to int \end{cases}$.

Ensuite un jugement de type est de la forme $\Gamma \vdash e : \tau \in Basetype$ pour Γ construit comme précédemment. Les programmes/instructions bien typé(e)s sont de type `void`. Un programme est bien typé si son code type `void` sous l'environnement Γ :

$$\frac{D \to \Gamma \quad \Gamma \vdash C : \texttt{void}}{\Gamma \vdash DC : \texttt{void}}$$

Règles pour les expressions (les autres sont similaires) :

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}}$$

Règles pour les instructions :

$$\frac{\Gamma \vdash e : \texttt{t} \qquad \Gamma \vdash x : \texttt{t}}{\Gamma \vdash x := e : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S1 : \texttt{void} \qquad \Gamma \vdash S2 : \texttt{void}}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ S1\ \texttt{else}\ S2 : \texttt{void}}$$

$$\frac{\Gamma \vdash S1 : \texttt{void} \qquad \Gamma \vdash S2 : \texttt{void}}{\Gamma \vdash S1; S2 : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done} : \texttt{void}}$$

LEIA ISA

Liste des instructions de la machine cible. Pour initialiser des registres à constante, vous pouvez utiliser la macro `.let r 585`). L'instruction `snif op1 <condition> op2` "skip next if" désactive l'instruction suivante si la condition est vraie. Les opérandes 1 et 2 sont des registres (temporaires dans le cas du 3-adresse, physiques sinon) ou des constantes immédiates. La condition peut-être : `eq`, `neq`, `sgt`, `slt`, `gt`, `ge`, `lt` et `le`.

15	14	13	12	mnemonic	class	description	ext(i)
0	0	0	0	`wmem`	wmem	write to memory	
0	0	0	1	`add`	ALU	addition	$z(i)$
0	0	1	0	`sub`	ALU	subtraction	$z(i)$
0	0	1	1	`snif`	snif	skip next if	$s(i)$
0	1	0	0	`and`	ALU	logical bitwise and	$s(i)$
0	1	0	1	`or`	ALU	logical bitwise or	$s(i)$
0	1	1	0	`xor`	ALU	logical bitwise xor	$s(i)$
0	1	1	1	`lsl`	ALU	logical shift left	$z(i)$
1	0	0	0	`lsr`	ALU	logical shift right	$z(i)$
1	0	0	1	`asr`	ALU	arithmetic shift right	$z(i)$
1	0	1	0	`call`	call	sub-routine call	
1	0	1	1	`jump` `return`	jump	relative jump if `offset` $\neq 1$ return from call if `offset` $= 1$	
1	1	0	0	`letl`	letl	8-bit constant to Rd, sign-extended	
1	1	0	1	`leth`	leth	8-bit constant to high half of Rd	
1	1	1	0	`print`	print	print or refresh	
1	1	1	1	`rmem` `copy`	rmem	read from memory if i=0 register-to-register copy if i=1	

Génération de code 3-adresses

On rappelle the le code 3 addresses LEIA a le même jeu d'instructions que le code LEIA standard, à part pour les conditions qui utilisent l'instruction `condJUMP(label,t1,condition,t2)` et que les registres sont remplacés par des temporaires. Vous pouvez utiliser `.let` si vous le désirez.

`newTemp` : $() \to \mathbb{N}$ crée un nouveau temporaire (temp0, temp1, ...) et `newLabel` : $() \to \mathbb{N}$ crée un nouveau label (donnez le nom que vous voulez).

c	`dr <-newTemp()`
	`code.add(InstructionLETL(dr, c))`
	`return dr`

x	`# récupère le temporaire associé à x`
	`regval<-getTemp(x)`
	`return regval`

e_1+e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionADD(dr, t1, t2))`
	`return dr`

e_1-e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionSUB(dr, t1, t2))`
	`return dr`

true	`dr <-newTemp()`
	`code.add(InstructionLETL(dr, 1))`
	`return dr`

$e_1 < e_2$	`t1 <- GenCodeExpr(e1)`
	`t2 <- GenCodeExpr(e2)`
	`dr <- newTemp()`
	`endrel <- newLabel()`
	`code.add(InstructionLET(dr, 0))`
	`#if t1>=t2 jump to endrel`
	`code.add(InstructionCondJUMP(endrel, t1, ">=" , t2)`
	`code.add(InstructionLET(dr, 1))`
	`code.addLabel(endrel)`
	`return dr`

x := e	`dr <- GenCodeExpr(e)`
	`# copie du résultat dans le tmp associé à x`
	`let loc = loc(x) in`
	`code.add(instructionCOPY(loc,dr))`

S1 ; S2	`#concatène les codes générés`
	`GenCodeSmt(S1)`
	`GenCodeSmt(S2)`

`if` b `then` $S1$ `else` $S2$	`lelse,lendif <-newLabels()`
	`t1 <- GenCodeExpr(b)`
	`#si la condition est fausse, saute à "else"`
	`code.add(`
	`InstructionCondJUMP(lelse, t1, "=", 0))`
	`GenCodeSmt(S1) #then`
	`code.add(InstructionJUMP(lendif))`
	`code.addLabel(lelse)`
	`GenCodeSmt(S2) #else`
	`code.addLabel(lendif)`

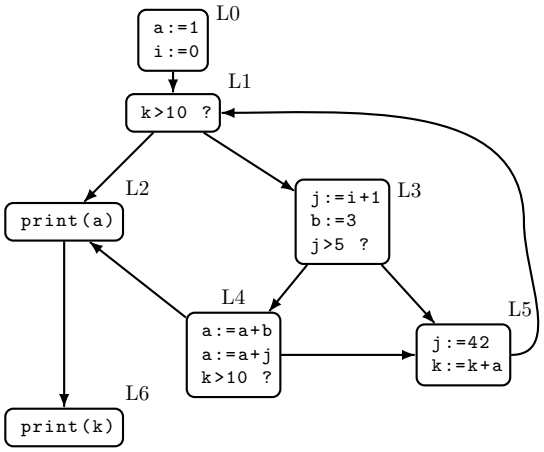
`while` b `do` S `done`	`ltest,lendwhile <-newLabels()`
	`code.addLabel(ltest)`
	`t1 <- GenCodeExpr(b)`
	`#si la condition est fausse saute à la fin`
	`code.add(`
	`InstructionCondJUMP(lendwhile, t1, "=", 0))`
	`GenCodeSmt(S) #exécute S`
	`code.add(InstructionJUMP(ltest))#et va au test`
	`code.addLabel(lendwhile) #fin while`

Master Info - 2016-2017
MIF08 Compilation

Contrôle continu 1 - Durée 20 min Éléments de correction

1 Dataflow

On donne un graphe de flot, et on rappelle qu'*une variable est vivante en sortie d'un bloc si il existe un chemin partant de ce bloc qui mène à une utilisation de cette variable sans redéfinition de cette variable.*



bloc	variables vivantes en sortie du bloc
L0	
L1	
L2	
L3	
L4	
L5	
L6	$\emptyset$

Question 1 (2 points)
Sans calcul, remplir le tableau avec les variables vivantes en sortie de chaque bloc.

Solution:
Voici le tableau rempli :

bloc	Out(bloc)
L0	a,i, k
L1	a,i,k
L2	k
L3	i,j,a,k,b
L4	a,i,k
L5	a,i, k
L6	$\emptyset$

Il y avait plusieurs petites difficultés :

- k est utilisé dans le programme, ce qui sous-entend qu'il a été défini précédemment.
- b est bien vivant en sortie du bloc L3 (sa valeur est utilisée juste après)
- lorsque j'ai une instruction d'impression, la variable utilisée doit être vivante en entrée du bloc...

J'ai compté des points pour des solutions partielles.

Question 2 (1 point)
Peut-on déplacer l'affectation `b:=3` de L3 à L0 ? Pourquoi ? Quel intérêt cela peut-il avoir ?

Solution:
On peut déplacer cette affectation avant la boucle, car b n'est définie qu'une seule fois (et L0 domine L3, c'est à dire que tout chemin passant par L3 est forcément passé par L0). Cela permet de ne pas recalculer b à chaque tour de boucle, donc accélère le programme.
j'ai mis des points pour "diminue la pression registre dans la boucle"

Question 3 (2 points)
Enlever le code mort, en justifiant.

Solution:
L'instruction j :=42 est morte dans le bloc L5, car j n'est pas vivante en sortie de L5.
Je n'attendais que cette réponse qui provient d'une utilisation directe du tableau. J'ai compté des points aux remarques sur le fait que l'on peut montrer que j est une constante, donc montrer que L4 n'est jamais accessible. Néanmoins, ce n'est pas une conséquence de l'analyse des variables vivantes, mais d'une analyse qui propagerait les constantes.
Attention, `a:=a+b` ne peut pas être supprimée car la valeur de a est modifiée ...

2 Génération de code

On rappelle en annexe les règles de génération de code pour les affectations, le test... Le visiteur de génération de code développé durant le TP4 a été lancé sur l'entrée suivante :

```
x=2;
if (x<3) y=7; else  y=8;
z=y+1;
```

Question 4 (5 points)
En vous aidant des règles en annexe, compléter le code généré suivant, en considérant l'allocation de temporaires suivante $x \mapsto tmp_1$, $y \mapsto tmp_6$, $z \mapsto tmp_9$:

Solution:
Voici le code généré complet :

;;Automatically generated code
;;(stat (assignment x = (expr (atom 2)) ;))
AND temp_1 , temp_1 , 0
ADD temp_1 , temp_1 , 2
;;(stat (if_stat if (condition_block (expr (atom ((expr (expr (atom x)) < (expr (atom 3)))))) (stat_block { (block (stat (assignment y = (expr (atom 7)) ;))) }))
else (stat_block { (block (stat (assignment y = (expr (atom 8)) ;))) }))
AND temp_2 , temp_2 , 0
ADD temp_2 , temp_2 , 3
NOT temp_4 , temp_2
ADD temp_4 , temp_4 , 1
ADD temp_3 , temp_1 , temp_4
BRzp l_cond_neg_2

AND temp_5 , temp_5 , 0
ADD temp_5 , temp_5 , 1
BR l_cond_end_2
l_cond_neg_2:
AND temp_5 , temp_5 , 0
l_cond_end_2:
BRz l_if_false_3
;;(stat (assignment y = (expr (atom 7)) ;))
AND temp_6 , temp_6 , 0
ADD temp_6 , temp_6 , 7
BR l_if_end_1
l_if_false_3 :
;;(stat (assignment y = (expr (atom 8)) ;))
AND temp_7 , temp_7 , 0
ADD temp_7 , temp_7 , 8
ADD temp_6 , temp_7 , 0
l_if_end_1 :
;;(stat (assignment z = (expr (expr (atom y)) + (expr (atom 1))) ;))
AND temp_8 , temp_8 , 0
ADD temp_8 , temp_8 , 1
ADD temp_9 , temp_6 , temp_8

Faites attention à générer le code en n'oubliant pas les temporaires générés, même s'ils sont inutiles ; en particulier, lors dans la seconde branche du test, même si y est affecté à $temp_6$ on ne peut pas directement mettre la constante dans $temp_6$. De même, lors de la génération de test pour l'addition il faut bien créer un nouveau temporaire pour la constante 1.

Il y a très peu de code correct pour les tests.

e ::= c	`dr <-newTemp()`
	`code.add(InstructionAND(dr, dr, 0))`
	`code.add(InstructionADD(dr, dr, c))`
	`return dr`

e ::= x	`regval=getTemp(x) #get the place associated to x.` `return regval`

e ::= e_1-e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionNOT(dr, t2))`
	`code.add(InstructionADD(dr, dr, 1))`
	`code.add(InstructionADD(dr, dr, t1))`
	`return dr`

e ::= $e_1 < e_2$	`dr <-newTemp()`
	`t1 <- GenCodeExpr (e1-e2)  #last write in register`
	`(lfalse,lend) <- newLabels()`
	`code.add(InstructionBRzp(lfalse))  #if =0 or >0 jump!`
	`code.add(InstructionAND(dr, dr, 0))`
	`code.add(InstructionADD(dr, dr, 1))  #dr <- true`
	`code.add(InstructionBR(lend))`
	`code.addLabel(lfalse)`
	`code.add(InstructionAND(dr, dr, 0))  #dr <- false`
	`code.addLabel(lend)`
	`return dr`

x := e	`dr <- GenCodeExpr(e)`
	`#a code to compute e has been generated`
	`if x has a location loc:`
	`  code.add(instructionADD(loc,dr,0))`
	`else:`
	`  storeLocation(x,dr)`

`if` b `then` $S1$ `else` $S2$	`dr <-GenCodeExpr(b)  #dr is the last written register`
	`lfalse,lendif=newLabels()`
	`code.add(InstructionBRz(lfalse) #if 0 jump to execute S2`
	`GenCodeSmt(S1)  #else (execute S1`
	`code.add(InstructionBR(lendif)) #and jump to end)`
	`code.addLabel(lfalse)`
	`GenCodeSmt(S2)`
	`code.addLabel(lendif)`

FIGURE 1 – Code generation rules

Examen Session 2
Traduction et Compilation de Programmes
février 2017
Durée 1H

Aucun document autorisé

Instructions **à lire attentivement !**

1. Les exercices sont indépendants.
2. On vous donne des indications de temps.
3. Vous répondrez dans les cadres et rendrez l'examen entier **agraphé**, et à **l'intérieur d'une copie d'examen anonyme**. Attention à bien reporter le numéro d'anonymat dans le cadre prévu ci-dessous (en gros).
4. Vous pouvez pour plus d'efficacité enlever la dernière feuille (pages 11 et 12) contenant les règles de génération de code et ne pas la remettre.

Numéro d'anonymat :

Exercice 1 – Grammaires et Attributions (20min)

Considérons la grammaire suivante pour les listes d'entiers : $L \rightarrow$ **INT** $L|\{L\}L|\varepsilon$ où :

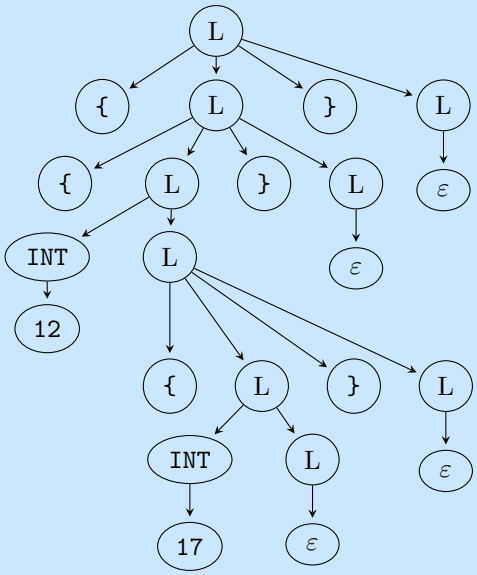
- L est le symbole de début (*start-symbol*)
- **INT** est un élément renvoyé par l'analyse lexicale (un *token*) quand elle détecte un entier.
- les accolades '{' et '}' sont aussi des *token*.

Question #1

Quel est l'arbre de dérivation pour la chaîne `{{12{17}}}` ?

Solution:

Chaque noeud interne de l'arbre représente l'application d'une règle de grammaire :



Cette grammaire est implémentée par la grammaire ANTLR suivante :

```
liste:  myint liste                    #cons
     |  '{' liste '}' liste            #concat
     |                                 #empty
     ;
myint: INT ;
INT: [0-9]+;
```

Question #2

Remplir le visiteur pour calculer la somme de tous les éléments d'une liste. On rappelle que :

- On a accès aux éléments de la règle parsée avec `ctx.name()` où `name` est le nom du non-terminal. Si il y a plusieurs non terminaux dans la règle on accède via la position : `ctx.name(0)` pour le premier, `ctx.name(1)` pour le deuxième ...
- On a accès à la chaîne d'un terminal `xx` avec `xx.getText()`.

Solution:

Le cas de base de la somme est 0 lorsque la liste est vide, dans le cas d'une liste construite avec plusieurs éléments il faut sommer ces éléments :

```
from ListeVisitor import ListeVisitor
from ListeParser import *

class MyListeVisitor(ListeVisitor):

    def visitCons(self, ctx):
        num=int(ctx.myint().getText())
        res2=self.visit(ctx.liste())
        return num+res2

    def visitConcat(self, ctx):
        return self.visit(ctx.liste(0))+self.visit(ctx.liste(1))

    def visitEmpty(self,ctx):
        return 0
```

Exercice 2 – Génération de code 3-adresses (40min)

On rappelle la syntaxe abstraite du mini-language while/Mu :

$$
\begin{array}{rcll}
S(Smt) & ::= & x := e & \textit{affectation} \\
 & | & \texttt{skip} & \textit{ne rien faire} \\
 & | & S_1; S_2 & \textit{sequence} \\
 & | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \textit{test} \\
 & | & \texttt{while } b \texttt{ do } S \texttt{ done} & \textit{boucle}
\end{array}
$$

avec e une expression numérique (entiers, $+$, ...) et b une expression booléenne (`true`, or, ...).

Les Figures 1 et 2 disponibles en annexe donnent les règles de génération de code 3 adresses que vous devez utiliser pour l'exercice. **Dans toute la suite on considère que les constantes numériques utilisées sont suffisamment petites pour être codées sur 5 bits en complément à 2.**

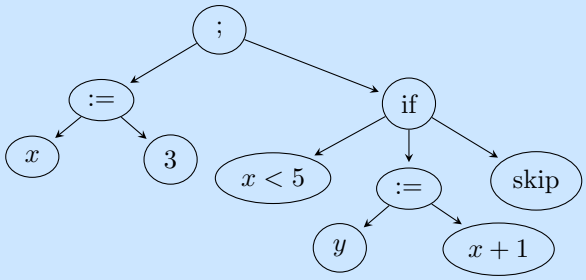
Question #1

Donner l'arbre de syntaxe abstrait (AST) pour le programme suivant.

```
x:=3; if (x<5) then y:=x+1 else skip;
```

On rappelle que chaque noeud de l'AST correspond à l'application d'une règle de grammaire de la grammaire abstraite. On pourra s'arrêter en mettant des expressions aux feuilles. Le noeud `ifthenelse` *aura 3 fils (la condition, S_1, et S_2).*

Solution:



Question #2

Pour le programme précédent, remplir (et commenter) le code 3 adresses généré à trous suivant (le temporaire associé à x est $temp_1$ et celui associé à y est $temp_7$) :

Solution:

;;Automatically generated code
;;(stat (assignment x = (expr (atom 3)) ;))
AND temp_1 , temp_1 , 0
ADD temp_1 , temp_1 , 3
;;(stat (if_stat if (condition_block (expr (atom ((expr (expr (atom x)) < (expr (atom 5))))))) (stat_block { (block (stat (assignment y = (expr (expr (atom x)) + (expr (atom 1))) ;))) }))))
AND temp_2 , temp_2 , 0
ADD temp_2 , temp_2 , 5

```
NOT temp_4 , temp_2
ADD temp_4 , temp_4 , 1
ADD temp_3 , temp_1 , temp_4
BRzp l_cond_neg_2
AND temp_5 , temp_5 , 0
ADD temp_5 , temp_5 , 1
BR l_cond_end_2
l_cond_neg_2:
AND temp_5 , temp_5 , 0
l_cond_end_2:
BRz l_if_false_3
;;( stat  (assignment y = (expr (expr (atom x)) + (expr (atom 1))) ;))
AND temp_6 , temp_6 , 0
ADD temp_6 , temp_6 , 1
ADD temp_7 , temp_1 , temp_6
BR l_if_end_1
l_if_false_3:
l_if_end_1:
```

On rajoute maintenant une instruction à notre langage : l'instruction `for`, avec cette syntaxe abstraite :

$$S(Smt) \quad ::= \ldots | \quad \texttt{for}\ i\ \texttt{from}\ num_1\ \texttt{to}\ num_2\ S$$

avec num_1, num_2 des entiers naturels vérifiant $num_1 < num_2$, et i une variable fraîche (non encore utilisée).

Question #3

En une phrase, sans détailler les calculs, dire ce que fait le programme suivant :

```
x := 0; for i from 1 to 10 { x := x + i; } ; log(x);
```

Solution:

Ce programme calcule la somme des 10 premiers entiers, puis l'imprime.

Question #4

On décide d'inventer une autre variable j qui décroît jusqu'à zéro pendant que la variable i croît. Remplir le code pour le programme précédent (sans l'impression). *Il manque un label et une instruction de branchement, ainsi que les incrémentations/décrémentations de i, j. Vous pouvez raccourcir un peu en utilisant* `ADD` *avec la constante* 1 *pour incrémenter une variable.*

Solution:

On prend

```
;;Automatically generated code
;;( stat  (assignment x = (expr (atom 3)) ;))
AND temp_1 , temp_1 , 0
ADD temp_1 , temp_1 , 3
;;( stat  (assignment i : = (expr (atom 1)) ;))
```

AND temp_2 , temp_2 , 0
ADD temp_2 , temp_2 , 1
;;(stat (assignment j : = (expr (atom 9)) ;))
AND temp_3 , temp_3 , 0
ADD temp_3 , temp_3 , 9
l_begin_for:
;;(stat (assignment x : = (expr (expr (atom x)) + (expr (atom i))) ;))
ADD temp_4 , temp_1 , temp_2
ADD temp_1 , temp_4 , 0
;; increment i:
ADD temp_2 , temp_2 , 1
;; decrement j:
ADD temp_3 , temp_3 , −1
BRp l_begin_for
label_end_for:

Question #5

En s'inspirant de l'exemple précédent, remplir la règle de génération de code pour le `for`. Expliquer.

Solution:

On peut par exemple : pour `for` i `from` $num1$ `to` $num2$ S :

```
i,j=nouvellesVariables()
nbsteps=num2-num1
lbeginfor,lendfor=newLabels()
GenCodeSmt("i:=num1")
GenCodeSmt("j:=nbsteps")
code.addLabel(lbeginfor)
GenCodeSmt(S)  ; avec i dans l'environnement
GenCodeSmt("i:=i+1")
GenCodeSmt("j:=j-1")
code.add(InstructionBRp(lbeginfor) ; si j>0 continue.
code.addLabel(lendfor)
```

Exercice 3 - Génération de code & Allocation de registres (30 min)

On considère (sur deux colonnes) le code 3 adresses LC3 suivant (on considère que `sub t1 t2 t3` est une instruction supplémentaire qui calcule la soustraction $t_1 \leftarrow t_2 - t_3$). Les t_i sont des temporaires à allouer (en registre, en mémoire).

```
LEA R6 spinit
[...]
ld  t1,label1
```

```
ld  t2,label2
sub t3,t1,t2
ld  t4,label3
```

```
ld  t5,label4
sub t6,t4,t5
add t7,t6,0
add t8,t3,t7
st t8,label5
RET
```

```
;;données/résultats
label1 : .FILL #2
label2 : .FILL #3
label3 : .FILL #-1
label4 : .FILL 7
label5 : .BLZW

;;gestion de la pile
spinit: .FILL
stackend: .BLKW #15 ; adresse du fond de la pile

.END
```

Question #1

Quelle expression calculera le code précédent quand on aura alloué les temporaires ? À quelle adresse sera-t-elle stockée ?

Solution:
Ce code calcule la valeur $(2-3)+(-1-7)=-9$ puis la range à l'adresse "label5".

Question #2

Remplir le tableau suivant en mettant une étoile à chaque fois que le temporaire est vivant en entrée de l'instruction. *Après le* `st`, *plus aucun temporaire n'est vivant.*

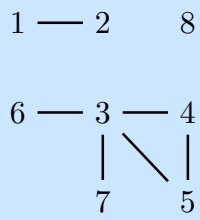
Solution:
A VERIFIER

code	t1	t2	t3	t4	t5	t6	t7	t8
`ld t1, label1`								
`ld t2, label2`	*							
`sub t3,t1,t2`	*	*						
`ld t4, label3`			*					
`ld t5, label4`			*	*				
`sub t6,t4,t5`			*	*	*			
`add t7,t6,0`			*			*		
`add t8,t3,t7`			*				*	
`st t8, label5`								*

Question #3

Dessiner le graphe d'interférence (à 8 sommets, 1 par temporaire) pour le code précédent. *On rappelle qu'en l'absence de code mort, deux temporaires sont en conflit (et donc reliés dans le graphe) si ils sont vivants en entrée d'un même bloc.*

Solution:
Le noeud i désigne le temporaire t_i :

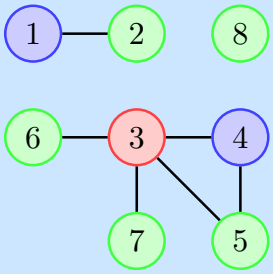


Question #4

Colorier le graphe d'interférence avec l'algorithme du cours avec 3 couleurs (vert, bleu, rouge, dans cet ordre). Quand il y a un choix pour empiler un temporaire, toujours considérer le temporaire de plus petit numéro. Préciser l'ordre d'empilement des sommets. *On rappelle que l'on empile en premier les sommets de plus petit degré.*

Solution:

On empile donc les numéros dans cet ordre : 8,1,2,6,7,3,4,5 (la pile est à lire dans l'ordre inverse, 5 est en haut de pile). Cela nous donne le coloriage suivant :

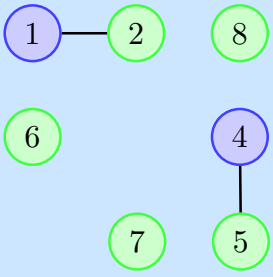


Question #5

Nous décidons de mettre en mémoire le temporaire t_3 sur la pile. Colorier le reste du graphe avec 2 couleurs (vert, bleu).

Solution:

La pile obtenue est 5,4,2,1,8,7,6 (5 en haut de pile) :



Question #6

Générer le code final avec 2 registres (R1,R2) pour les temporaires, R6 pointeur de pile, R5 pour la gestion de la variable spillée t_3.

Solution:

Pour les variables allouées en registre, on remplace "vert" par R1, "bleu" par R2. On donne le décalage 0 pour t_3 car c'est la seule variable spillée.

```
LEA R6 spinit
[...]
ld  R2,label1
ld  R1,label2
sub R5,R2,R1  #calcul ds R5
STR R5 R6 1   #stockage en mémoire
ld  R2,label3
ld  R1,label4
sub R1,R2,R1
add R2,R1,0
LDR R5 R6 0    #récupération
add R1,R5,R1   #calcul
st R1,label5
RET
```

A Règles de génération de code

(Stm) `x := e`	`dr <- GenCodeExpr(e)`
	`#a code to compute e has been generated`
	`if x has a location loc:`
	`  code.add(instructionADD(loc,dr,0))`
	`else:`
	`  storeLocation(x,dr)`

(Stm)`if` b `then` $S1$ `else` $S2$	`dr <-GenCodeExpr(b)  #dr is the last written register`
	`lfalse,lendif=newLabels()`
	`code.add(InstructionBRz(lfalse) #if 0 jump to execute S2`
	`GenCodeSmt(S1)  #else (execute S1`
	`code.add(InstructionBR(lendif)) #and jump to end)`
	`code.addLabel(lfalse)`
	`GenCodeSmt(S2)`
	`code.addLabel(lendif)`

`while` b `do` S `done`	`ltest,lendwhile=newLabels()`
	`code.addLabel(ltest)`
	`dr <-GenCodeExpr(b)  #dr is the last written register`
	`code.add(InstructionBRz(lendwhile) #if 0 jump to end`
	`GenCodeSmt(S)  #else (execute S`
	`code.add(InstructionBR(ltest))  #and jump to the test)`
	`code.addLabel(lendwhile)`

FIGURE 1 – Génération de code pour les instructions

(constant expression) c	`#not valid if c is too big`
	`dr <-newTemp()`
	`code.add(InstructionAND(dr, dr, 0))`
	`code.add(InstructionADD(dr, dr, c))`
	`return dr`

e ::= e_1+e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionADD(dr, t1, t2))`
	`return dr`

e ::= e_1-e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionNOT(dr, t2))`
	`code.add(InstructionADD(dr, dr, 1))`
	`code.add(InstructionADD(dr, dr, t1))`
	`return dr`

e ::= $e_1 < e_2$	`dr <-newTemp()`
	`t1 <- GenCodeExpr (e1-e2)  #last write in register`
	`(lfalse,lend) <- newLabels()`
	`code.add(InstructionBRzp(lfalse))  #if =0 or >0 jump!`
	`code.add(InstructionAND(dr, dr, 0))`
	`code.add(InstructionADD(dr, dr, 1))  #dr <- true`
	`code.add(InstructionBR(lend))`
	`code.addLabel(lfalse)`
	`code.add(InstructionAND(dr, dr, 0))  #dr <- false`
	`code.addLabel(lend)`
	`return dr`

FIGURE 2 – Génération de code pour les expressions

Master Info - 2017-2018
MIF08 Compilation

Contrôle continu 1 - Durée 42 min (tiers-temps 56 min)

Éléments de correction

On rappelle en annexe les règles de génération de code pour les affectations, le test, et certaines règles de typage. On considère le programme suivant :

```
var x2: int;
x2 = 13;
while (x2 > 8) do
   x2 = x2 - 1;
done
```

Question 1 (8 points)

Montrer que ce programme est bien typé (déclarations, instructions).

Solution:

La déclaration de variable fournit l'environnement de typage $\Gamma : x_2 \rightarrow int$ (en toute rigueur il faudrait utiliser les règles pour construire Γ je ne vous en ai pas tenu rigueur lors de la correction). Ensuite, il s'agit de construire un arbre de typage clos pour ce programme : En ascii art ici :

```
Gamma(x2)=int
---------------   ---------------
Gamma |- x2:int   Gamma |- 13:int
---------------------                        ---- ** ----
Gamma |- x2=13 : void         Gamma |- while (x2 >8) do ... done : void
------------------------------------------------------------------------ [;]
Gamma |- x2=13 ; while ... done : void
```

où les doubles étoiles sont le sous-arbre suivant :

```
Gamma(x2)=int                            Gamma(x2)=int          ***
---------------   -------------          -------------------------------[-]
Gamma |- x2:int   Gamma |- 8:int         Gamma |- x2:int   Gamma |- x2-1:int
-------------------------------            ----------------------------[:=]
Gamma |- x2>8 : bool                          Gamma |- x2=x2-1 : void
-------------------------------------------------------------------------
```

et les triples étoiles celui-là :

```
Gamma (x2)=int
--------------    --------------
Gamma |- x2:int   Gamma |- 1: int
```

Cet arbre de typage étant cohérent et clos, le programme est bien typé. Attention à bien utiliser Γ sur les feuilles qui concernent des variables du programme.

Question 2 (12 points)

Générer le code 3 adresses LEIA pour ce programme, en suivant à la lettre les règles de

génération de code données en annexe. Les calculs intermédiaires seront documentés. **Vous mettrez la feuille courante en format paysage, et vous utiliserez trois colonnes : une pour décrire les appels récursifs, une pour le code généré et une pour les temporaires.**

Solution:

Voici le début de la rédaction, le reste est laissé au lecteur (ainsi que la mise en page). Dans la colonne de droite, l'effet de la déclaration de la variable x_2 est la création d'un temporaire $temp_0$. Ensuite, dans la colonne de gauche :

```
genCodeSmt(x2=13;reste)=
   genCodeSmt(x2=13)=
     dr <- genCodeSmt(13)
           dr2 = temp_1 <- newtmp()
           code.add(InstructionLETL(temp_1,13)
     (donc dr = dr2 = temp1)
     loc (x2) = temp0
     code.add(InstructionCopy(temp_0,temp_1)

   genCodeSmt(reste) ...
```

en regard dans la colonne du milieu, on obtient donc :

```
        .let temp_1 13    #ou letl temp_1 13
        copy temp_0 temp_1
```

Le code généré est finalement le suivant (obtenu avec la version prof du TP4), dans la deuxième colonne).

```
        ;; (stat (assignment x2 = (expr (atom 13)) ;))
        .let temp_1 13
        copy temp_0 temp_1
        ;; (stat (while_stat while (expr (atom ( (expr (expr (atom x2)) > (expr (atom
    8))) ))) (stat_block { (block (stat (assignment x2 = (expr (expr (atom x2)) - (
    expr (atom 1))) ;))) }))))
lbl_l_while_begin_0 :
        .let temp_2 8
        .let temp_3 0
        CONDJUMP lbl_end_relational_1 temp_0 "<=" temp_2
        .let temp_3 1
lbl_end_relational_1 :
        CONDJUMP lbl_l_while_end_0 temp 3 "!=" 1
        ;; (stat (assignment x2 = (expr (expr (atom x2)) - (expr (atom 1))) ;))
        .let temp_4 1
        sub temp_5 temp_0 temp_4
        copy temp_0 temp_5
        jump lbl_l_while_begin_0
lbl_l_while_end_0 :
```

Question 3 (Bonus points)

Remplacez le saut conditionnel du code généré par les instructions `snif` adéquates.

Solution:

Le premier cond jump est remplacé par :

SNIF temp_0 **gt** temp_2
JUMP lbl_end_relational_1

---

et le deuxième par :

SNIF temp_3 **eq** 1
JUMP lbl_l_while_end_0

Examen Session 1
Traduction et Compilation de Programmes
Janvier 2017
Durée 2H

Éléments de correction (non contractuels...) Quelques typos et remarques de correction en rouge

Instructions **à lire attentivement !**

1. Les exercices sont indépendants.
2. On vous donne des indications de temps. **Regardez bien les différents exercices proposés et gérez votre temps en conséquence !**
3. Vous répondrez dans les cadres et rendrez l'examen entier **agraphé** tel quel.
4. Des étiquettes assureront l'anonymat, le même numéro sera collé dans le cadre ci dessous et sur la feuille d'émargement.
5. Vous pouvez pour plus d'efficacité enlever la dernière feuille contenant les règles de génération de code et ne pas la remettre.

Remarques générales : quand il est demandé de justifier une réponse, la réponse doit être justifiée. Le code sans justification a été lu, mais des points ont été retirés.

Exercice	Points (total)
1	5
2	7
3	5
4	7

Exercice 1 – Grammaires et Attributions (30min)

Source : TD de Compilation, UFR IEEA, Université de Lille, 2008

On s'intéresse à reconnaître un langage de description d'arbres dont chaque noeud possède au plus deux fils, implémenté par la grammaire ANTLR suivante :

```
grammar Tree;

tree: onetree EOF
    ;

onetree:  LEAF value                                #feuille
    | BINARY value '(' onetree ',' onetree ')'      #binaire
    | UNARY value '(' onetree ')'                   #unaire
    ;

value : '[' LETTER ']' #lettre
    |   # eps
    ;

LEAF : 'f' ;
BINARY : 'b';
UNARY : 'u' ;
LETTER : [A-Za-z] ;
```

Tous les noeuds de l'arbre sont éventuellement décorés d'une lettre. Ainsi, les chaînes `b[l](f[a],f[e])`, `u[d](f[o])` sont des éléments du langage et représentent les deux arbres de gauche de la figure 1. En l'absence de crochet, la feuille est décorée par la lettre vide ε.

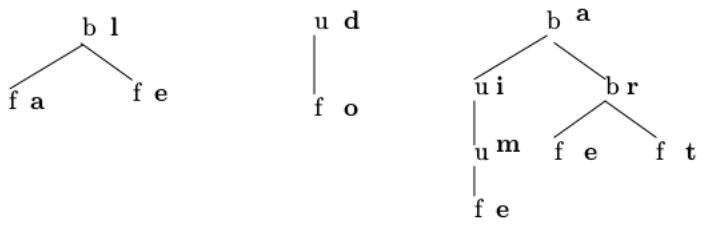


FIGURE 1 – Exemples d'arbres décorés

Question #1

Quelle chaîne d'entrée représente le sous-arbre de droite de la Figure 1 ?

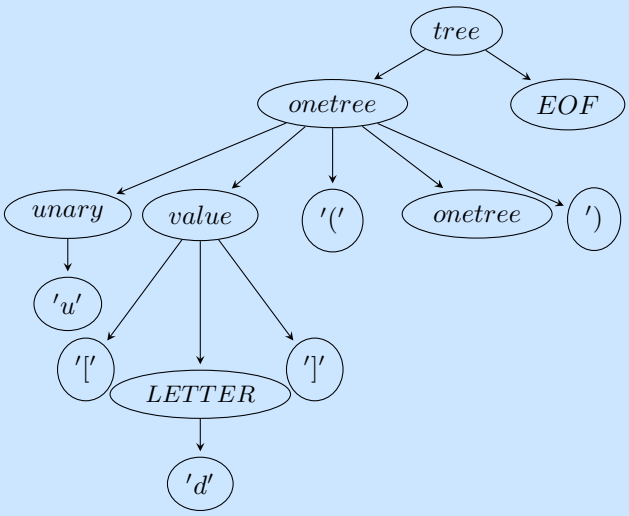
Solution:

`b[a](u[i](u[m](f[e])), b[r](f[e],f[t]))`

Question #2

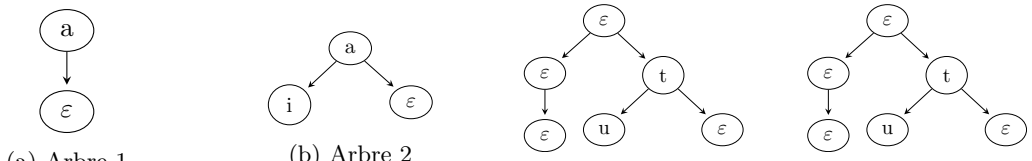
Quelle est l'arbre de dérivation pour la chaîne `u[d](f[o])` ? *On rappelle qu'un arbre de dérivation n'est pas un AST, chaque noeud interne dénote une application de règle. Les feuilles sont des terminaux.*

Solution:
Le début de l'arbre, car dessiner en LaTeX est long :



Un arbre décoré représente un ensemble de mots, qui sont les mots lus en parcourant les branches de l'arbre de la racine vers les feuilles. Ainsi les arbres de la Figure 1 représentent respectivement les ensembles $\{la, le\}$, $\{do\}$ et $\{aime, are, art\}$. Si un noeud interne ou une feuille contient ε, cette "lettre" est tout simplement ignorée dans le mot.

Dans la Figure 2, on omet les préfixes `u`,`f`,`b` pour ne laisser que les décorations.



(a) Arbre 1 (b) Arbre 2 (c) Arbre 3 (d) Arbre 4

FIGURE 2 – Exemples

Problème de copier coller, les arbres 3 et 4 étaient identiques dans l'énoncé.

Question #3

Pour chacun des exemples de la Figure 2, donner le nombre de mots représentés par l'arbre, ainsi que la taille maximale de cet ensemble de mots. *Un chemin qui ne passe que par des nœuds décorés par ε correspond au mot vide. Le mot vide est compté pour 1 dans le nombre de mots possibles, et sa taille est 0. Deux chemins étiquetés par le même mot comptent pour 2 mots.*

Ici je voulais bien évidemment dire "la taille maximale d'un mot dans l'ensemble de mots"

Solution:
Je dénote (nb, max) : Arbre 1 : (1,1), arbre 2 : (2,2), arbre 3 : (3, 2).

Question #4

Remplir le premier visiteur ci-dessous pour calculer la longueur du plus long mot représenté par l'arbre reconnu par la grammaire, ainsi que le nombre de mots représentés par l'arbre. *On s'aidera des exemples, et on* **JUSTIFIERA** *sur les lignes ci-dessous notamment en se reportant aux exemples. Des éléments de syntaxe sont présents dans la feuille d'accompagnement.* Les attributs calculés devront se conformer à l'appel "main" suivant :

```
    tree = parser.tree()  # Parse
    visitor = MyTreeVisitor()  # Visit
    maxlen, nbmots = visitor.visit(tree)
    print("maxlen,nbmots␣"+str(maxlen)+",␣"+str(nbmots))
```

La justification est importante, sans elle, lire le code est difficile.

Solution:

Les visiteurs pour les valeurs retournent 1 si il s'agit du nombre de mots, qu'il s'agisse d'epsilon ou pas, et 0 ou 1 pour la taille. Lorsque l'on visite un arbre unaire (resp binaire), le nombre de mots est égal au nombre de mots de son unique fils (ou la somme des deux mots de ses fils). Lorsqu'on visite un arbre unaire, la taille maximale du mot du sous arbre dont il est la racine est la taille maximale portée par son unique fils, plus 0 ou 1 suivant sa "value". Lorsqu'on visite un arbre binaire, il s'agit de faire le max des tailles maximales des deux sous arbres, auquel on ajoute 0 ou 1.

```
from TreeVisitor import TreeVisitor


class MyTreeVisitor(TreeVisitor):
    # Visit a parse tree produced by TreeParser#tree.
    def visitTree(self, ctx):
        return self.visit(ctx.onetree())

    # Visit a parse tree produced by TreeParser#feuille.
    def visitFeuille(self, ctx):
        return self.visit(ctx.value())

    # Visit a parse tree produced by TreeParser#binaire.
    def visitBinaire(self, ctx):
        valg, nbg = self.visit(ctx.onetree(0))
        vald, nbd = self.visit(ctx.onetree(1))
        myval, nb = self.visit(ctx.value())
        maxlen = myval + max(valg, vald)
        return (maxlen, nbg+nbd)

    # Visit a parse tree produced by TreeParser#unaire.
    def visitUnaire(self, ctx):
        valrec, nb = self.visit(ctx.onetree())
        myval, nb2 = self.visit(ctx.value())
        return ((myval + valrec), nb)

    def visitLettre(self, ctx):
        return 1, 1

    def visitEps(self, ctx):
        return 0, 1
```

Question #5

Pour les exemples de la Figure 2, donner la liste des mots reconnus par l'arbre. *On rappelle que deux chemins étiquetés par le même mot sont comptés pour 2.*

Solution:

Sans difficulté, il s'agit des ensembles $\{a\}$ pour le premier, $\{a, ai\}$ pour le deuxième, et $\{\varepsilon, tu, t\}$ pour le troisième.

Question #6

Remplir en justifiant le deuxième visiteur ci-dessous, qui construit la liste des mots représentés par l'arbre. On vous donne les cas de base. *On rappelle que la concaténation des chaînes ainsi que des listes utilise l'opérateur '+', et la construction `[ xxx for word in yyy]` pourra être utilisée.* Les attributs calculés devront se conformer à l'appel "main" suivant :

```
tree = parser.tree()  # Parse
visitor = MyTreeVisitor2()  # Visit
mylist = visitor.visit(tree)
print(mylist)
```

Solution:

Justification partielle Pour récupérer l'ensemble (en fait la liste) des mots au niveau d'un arbre unaire, il faut pour chaque mot reconnu par son fils ajouter en tête la "value" du noeud (le cas vide est géré par la concaténation avec une chaîne vide). D'où l'itérateur de liste.

```
from TreeVisitor import TreeVisitor


class MyTreeVisitor2(TreeVisitor):
    # Visit a parse tree produced by TreeParser#tree.
    def visitTree(self, ctx):
        return self.visit(ctx.onetree())

    # Visit a parse tree produced by TreeParser#feuille.
    def visitFeuille(self, ctx):
        myval = self.visit(ctx.value())
        return [myval]

    # Visit a parse tree produced by TreeParser#binaire.
    def visitBinaire(self, ctx):
        listrec1 = self.visit(ctx.onetree(0))
        listrec2 = self.visit(ctx.onetree(1))
        concat = listrec1+listrec2
        myvalue = self.visit(ctx.value())
        return [myvalue+word for word in concat]

    # Visit a parse tree produced by TreeParser#unaire.
    def visitUnaire(self, ctx):
        listrec = self.visit(ctx.onetree())
        myvalue = self.visit(ctx.value())
        return [myvalue+word for word in listrec]  # chains

    def visitLettre(self, ctx):
        return ctx.LETTER().getText()

    def visitEps(self, ctx):
        return ""
```

Exercice 2 – Génération de code 3-adresses (40min)

On rappelle la syntaxe abstraite du mini-language while/Mu :

$$
\begin{array}{rcll}
S(Smt) & ::= & x := e & \textit{affectation} \\
 & | & \texttt{skip} & \textit{ne rien faire} \\
 & | & S_1 ; S_2 & \textit{sequence} \\
 & | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \textit{test} \\
 & | & \texttt{while } b \texttt{ do } S \texttt{ done} & \textit{boucle}
\end{array}
$$

avec e une expression numérique (entiers, $+$, ...) et b une expression booléenne (`true`, or, ...).

Les Figures disponibles en annexe donnent les règles de génération de code 3 adresses du cours.

On considère le programme P_0 suivant :

```
x := 2 ; while ( x > 0 ) do x = x + 1 done;
```

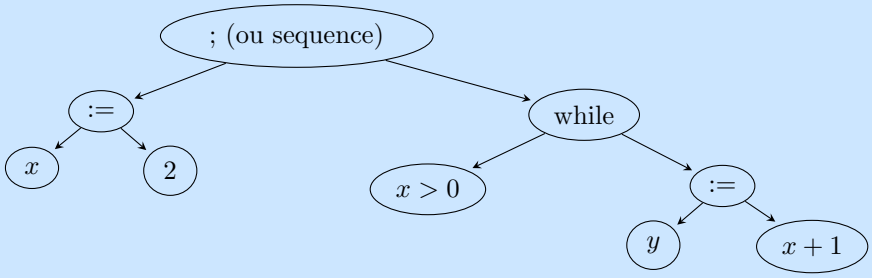
Oui, il y avait une typo ici, ' :=' et '=' dénotent tous deux l'affection !

Question #1

Donner l'arbre de syntaxe abstrait (AST) pour le programme P_0. *On rappelle que chaque noeud de l'AST correspond à l'application d'une règle de grammaire de la grammaire abstraite. On pourra s'arrêter en mettant des expressions aux feuilles.*

Solution:

Attention, il s'agit cette fois d'un AST.



Question #2

Remplir le code 3 adresses généré pour P_0 (le temporaire associé à x est $temp_0$) : il y avait une petite typo - au lieu de +

Solution:

Il suffit de suivre la feuille de génération.

```
;;Automatically generated LEIA code, MIF08 2017
;;non executable 3-Address instructions version
        ;; (stat (assignment x = (expr (atom 2)) ;))
        .let temp_1 2
        copy temp_0 temp_1
        ;; (stat (while_stat while (expr (atom ( (expr (expr (atom x)) > (expr (atom 0))) )) (
    stat_block { (block (stat (assignment x = (expr (expr (atom x)) + (expr (atom 1))) ;))) }))
lbl_1_while_begin_0:
        .let temp_2 0
        .let temp_3 0
        ;; cond_jump lbl_end_relational_1 temp_0 le temp_2
        SNIF temp_0 gt temp_2
        JUMP lbl_end_relational_1
```

```
        ;; end cond_jump lbl_end_relational_1 temp_0 le temp_2
        .let temp_3 1
lbl_end_relational_1:
        ;; cond_jump lbl_l_while_end_0 temp_3 neq 1
        SNIF temp_3 eq 1
        JUMP lbl_l_while_end_0
        ;; end cond_jump lbl_l_while_end_0 temp_3 neq 1
        ;; (stat (assignment x = (expr (expr (atom x)) + (expr (atom 1))) ;))
        .let temp_4 1
        add temp_5 temp_0 temp_4
        copy temp_0 temp_5
        jump lbl_l_while_begin_0
lbl_l_while_end_0:
```

On rajoute maintenant une instruction à notre langage : l'instruction `break`, avec cette syntaxe :

$$S(Smt) \quad ::= \ldots | \quad \texttt{break}\ (b)$$

Voici la signification de ce nouveau constructeur :

- Si b s'évalue à faux, `break` (b) se comporte comme `skip`.
- Si b s'évalue à vrai et l'on est en train d'exécuter une boucle tant que, alors `break` (b) termine la boucle et se place juste après.
- Si b s'évalue à vrai et le flot n'est dans dans une boucle, alors `break` (b) se comporte comme `skip`.

Question #3

Donner les étapes de calcul des programmes suivants :

- P_1 : `x:=2; while true do break(true)done; y:=x`
- P_2 : `x:=2; while (x>0)do x:=x+1;break(x>0)done ; y:=x`
- P_3 : `x:=2; while (x>0)do break(x>0);x:=x+1 done; y:=x`

Solution:
Rien de difficile, je ne fais que pour P_1 : x reçoit la valeur 2, puis la condition de boucle étant vraie on exécute le corps, la condition du *break* étant vraie, on sort immédiatement, et y est donc affectée à la valeur 2.

Question #4

Compléter le code 3 adresses généré pour P_2, dans lequel on a supprimé le traitement de l'instruction `break`. Expliquer ci-dessous. Il fallait comprendre remplir le code pour le break dans les lignes manquantes. Évidemment générer le code pour le break consiste aussi à évaluer le test de ce break.

Solution:
On évalue le test du break (et donc on génére quelque chose qui ressemble au test de la boucle), puis si ce test est vrai, on saute vers le label de fin de boucle. Attention à être le plus générique possible.

Solution:

```
        copy temp_0 temp_5
;;;  instruction break: calcule le test, puis saute apre's la boucle
        .let temp_6 0
        .let temp_7 0
```

```
        ;; cond_jump lbl_end_relational_2 temp_0 le temp_6
        SNIF temp_0 gt temp_2
        JUMP lbl_end_relational_2
        .let  temp7_1
        lbl_end_relational_2
        SNIF temp_7 eq 1
        ;; je sais que je suis dans une boucle, et je connais le label de fin
        JUMP lbl_l_while_end_0
;;;  end of loop
        jump lbl_l_while_begin_0
lbl_l_while_end_0:
```

Question #5

En s'inspirant de l'exemple précédent, remplir la règle de génération de code pour le `break`, et modifier la règle du `while`. Expliquer. *On pourra s'inspirer du traitement des tests imbriqués dans le TP.*

Solution:

Sans surprise, la génération de code pour le break est très similaire à celle d'un test, mais le saut est vers un label auquel il n'a pas directement accès, en faisant attention à verifier qu'on est effectivement dans une pile. Pour connaître cette information, il faut la stocker au moment où on la produit, c'est à dire dans la génération du while.

(Stm)`break` (*b*)	`t1 <- GenCodeExpr(b)`
	`#recupere le label de fin de la boucle tant que courante`
	`label <- stack_whileloop.gettop()`
	`if label != None : #on est dans une boucle.`
	`  #si la condition est vraie, saute à "label"`
	`  code.add(`
	`  InstructionCondJUMP(label, t1, "=", 1))`

La seule modification dans le while est l'empilement au début du label `lendwhile` au début de la génération dans la pile `stack_whileloop` considérée globale.

Exercice 3 – Dataflow (20 min)

source : F. Pereira. Considérons le programme suivant

```
1  x:=1;
2  y:=read(); // valeur random pour y
3  while (y>0)  {
4    x:=x+1;
5    y:=y-1;     }
6  x:=2
7  print(x);
```

Question #1

Construire le graphe de flot de contrôle avec une instruction par bloc. Les blocs seront numérotés avec le numéro de la ligne de code correspondante.

Solution:

Rien de difficile. J'attendais un graphe de flot dont les blocs sont constitués d'une unique ligne. Un programme comportant une boucle tant que génère un graphe avec un cycle...

On rappelle qu'une variable est vivante après un bloc si il existe un chemin de ce bloc vers une utilisation de cette variable, et les définitions :

— $gen_{LV}(\ell)$ (aussi noté $gen(\ell)$) est l'ensemble des variables qui apparaissent comme opérandes sources dans ℓ, mais qui ne sont pas affectées avant dans ℓ.
— $kill_{LV}(\ell)$ (aussi noté $kill(\ell)$) est l'ensemble des variables définies dans le bloc (affectées).

On rappelle aussi les relations entre les ensembles $kill$, gen, In et Out vues en TD :

$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{si } \ell = \text{final} \\ \bigcup\{LV_{entry}(\ell') | (\ell, \ell') \in flow(G)\} \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

Les ensembles In et Out, qui convergeront vers LV_{in} et LV_{out} sont initialisés à $\emptyset$, et grossissent jusqu'à point fixe.

Question #2

Effectuer l'analyse *dataflow* des variables vivantes sur ce programme, en utilisant le tableau ci dessous. *Si vous n'avez pas assez de temps les initialisations et le résultat final attendu vous rapporteront des points.*

Solution:

Les initialisations / le résultat (les étapes sont laissées au lecteur) : par commodité les ensembles sont dénotés comme des listes.

			Resultat	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$
1	x	$\emptyset$	$\emptyset$	x
2	y	$\emptyset$	x	x,y
3	$\emptyset$	y	x,y	x,y
4	x	x	x,y	x,y
5	y	y	x,y	x
6	x	$\emptyset$	$\emptyset$	x
7	$\emptyset$	x	x	$\emptyset$

Au passage remarquez que x est vivant à la sortie de la boucle, donc on ne peut tuer du code mort. L'analyse n'est pas toujours capable de trouver du code mort alors qu'il existe.

Question #3

Quel serait le résultat de l'analyse sur le programme suivant ? Le résultat de l'analyse de vivacité, bien sûr !

```
x:=1;
x:=x-1;
x:=2;
print(x);
```

Solution:

On trouverait, après analyse, que la variable x est morte à la sortie de la deuxième instruction, mais qu'elle est vivante à l'entrée de la seconde instruction, même si sa valeur est utilisée pour calculer la valeur d'une variable morte ... Cet exemple montre les limites de l'analyse de vivacité.

Exercice 4 – Allocation de registre et génération de code machine.

Soit le programme Mu suivant :

```
var x,y,z,t:int;
x=12; y=3+x; z=4+y; t=x-y+z;
```

Pour des raisons de lisibilité les temporaires s'appellent ti et non $temp_i$ dans le tableau et le graphe de conflits, mais $temp_i$ dans le code. La production de code 3 adresses du TP4 produit le code décrit dans le tableau ci-dessous $(t, z, y, x) \mapsto (t1, t2, t2, t3)$: Typo, t associé à t0, z à t1, y à t2, x à t3. Dans le tableau idéalement il faudrait rajouter une colonne

code	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
`.let t4 12`										
`copy t3 t4`										
`.let t5 3`										
`add t6 t5 t3`										
`copy t2 t6`										
`.let t7 4`										
`add t8 t7 t2`										
`copy t1 t8`										
`sub t9 t3 t2`										
`add t10 t9 t1`										
`copy t0 t10`										

Question #1

Générer le code final pour les 2 premières lignes avec la stratégie d'allocation "tout en mémoire" (cf TP4). *On utilisera R_0 pour calculer les adresses de pile, R_6 comme pointeur de pile, et R_1 et R_2 pour accéder aux éléments de pile.*

Solution:

```
;;Automatically generated LEIA code, MIF08 2017
;;all-in-memory allocation version
;; stack management
.set r6 stack
        ;; (stat (assignment x = (expr (atom 12)) ;))
        ;; .let temp_4 12
        .LET r1 12
        SUB r0 r6 3
        WMEM r1 [r0]
        ;; end .let temp_4 12
        ;; copy temp_3 temp_4
        SUB r0 r6 3
        RMEM r1 [r0]
        COPY r1 r1
        SUB r0 r6 2
        WMEM r1 [r0]
        ;; end copy temp_3 temp_4
```

Question #2

Proposer une amélioration pour l'allocation d'une copie dans le cas de l'allocation *all-in-mem. Expliquer sur l'exemple puis dans le cas général.*

Solution:
Une copie peut tenir dans une même place. Dans le cas all inmem, rmem copy wmem on peut supprimer la copie.

Question #3
Proposer une amélioration pour l'allocation d'une copie dans le cas de l'allocation *non naïve* de registres. *Mêmes consignes.*

Solution:
L'idée serait de considérer ensemble les variables copiées dans le graphe de conflit, dans le même noeud. Il faut faire attention aux durées de vie néanmoins.

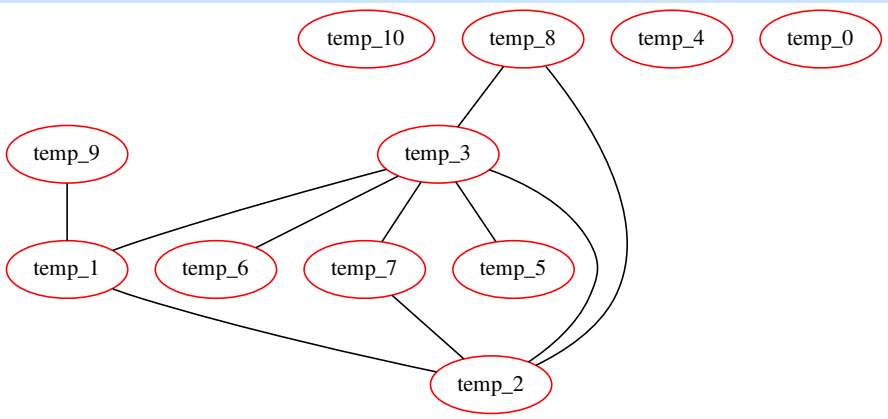
Question #4
Remplir le tableau ci-dessus avec le résultat de l'analyse de vivacité. Chaque étoile dans une ligne signifie "le temporaire est vivant en entrée de cette ligne de code").

Solution:

code	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
`.let t4 12`										
`copy t3 t4`				*						
`.let t5 3`			*							
`add t6 t5 t3`			*		*					
`copy t2 t6`			*			*				
`.let t7 4`		*	*							
`add t8 t7 t2`		*	*				*			
`copy t1 t8`		*	*					*		
`sub t9 t3 t2`	*	*	*							
`add t10 t9 t1`	*								*	
`copy t0 t10`										*

Question #5
Tracer le graphe d'interférence.

Solution:



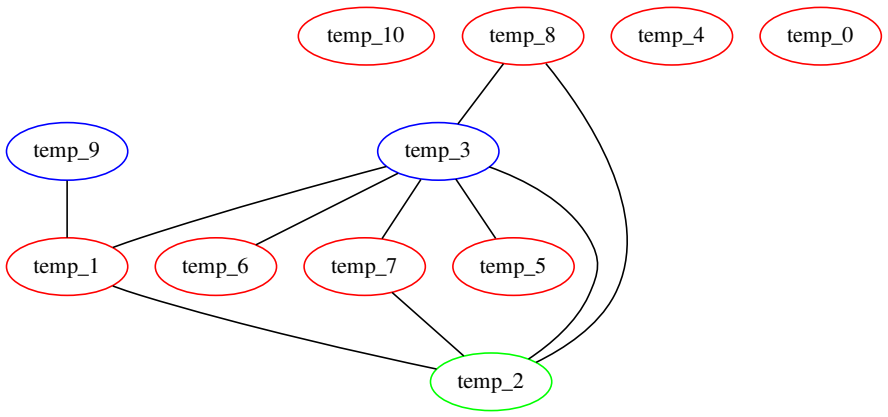
Dans la suite on utilisera les conventions suivantes : — Couleur 1 = rouge, associée au registre R3 ; — Couleur 2 = bleu, associée au registre R4.

Question #8

Colorier le graphe avec l'algorithme du cours et 2 couleurs. *On écrira aussi la pile de coloriage en montrant clairement le haut de pile. On rappelle qu'il faut empiler les sommets de plus bas degré en premier, que les degrés sont mis à jour à chaque fois que l'on empile, et que si il y a un choix c'est le temporaire de plus bas numéro qui est empilé en premier. Enfin le coloriage se passe dans le sens inverse.*

Solution:

La pile de coloriage est (fond de pile à gauche) $[(0), 4, 10, 5, 6, 9, 1, 7, 2, 3, 8]$. Avec 3 couleurs ça donnerait :



Avec deux couleurs on obtient le même coloriage et $temp_2$ ne peut être colorié.

Question #9

Quelles est/sont la/les variable(s) à "spiller" ? Expliquer le processus de *spill* et ce que l'on fait si l'on a plus d'une variable à "spiller". On gardera les mêmes conventions qu'à la question 1.

Solution:

La graphe précédent n'est pas deux coloriable, on s'en aperçoit en essayant de colorier la variable $temp_2$. Si on continue le processus le graphe est totalement colorié sauf $temp_2$. On va donc spiller (ie stocker sa valeur en mémoire) la variable $temp_2$. Le procédé est décrit dans le cours.

Question #10

Compléter le code finalement généré, gestion des variable(s) spillée(s) comprise.

Solution:

```
;;Automatically generated LEIA code, MIF08 2017
;;Smart Allocation version
;; stack management
.set r6 stack
        ;; (stat (assignment x = (expr (atom 12)) ;))
        ;; .let temp_4 12
        .LET r3 12
        ;; end .let temp_4 12
        ;; copy temp_3 temp_4
        COPY r4 r3
        ;; end copy temp_3 temp_4
        ;; (stat (assignment y = (expr (expr (atom 3)) + (expr (atom x))) ;))
```

```
        ;; .let temp_5 3
        .LET r3 3
        ;; end .let temp_5 3
        ;; add temp_6 temp_5 temp_3
        ADD r3 r3 r4
        ;; end add temp_6 temp_5 temp_3
        ;; copy temp_2 temp_6
        COPY r1 r3
        SUB r0 r6 0
        WMEM r1 [r0]
        ;; end copy temp_2 temp_6
        ;; (stat (assignment z = (expr (expr (atom 4)) + (expr (atom y))) ;))
        ;; .let temp_7 4
        .LET r3 4
        ;; end .let temp_7 4
        ;; add temp_8 temp_7 temp_2
        SUB r0 r6 0
        RMEM r1 [r0]
        ADD r3 r3 r1
        ;; end add temp_8 temp_7 temp_2
        ;; copy temp_1 temp_8
        COPY r3 r3
        ;; end copy temp_1 temp_8
        ;; (stat (assignment t = (expr (expr (expr (atom x)) - (expr (atom y))) + (expr (atom z))
    ) ;))
        ;; sub temp_9 temp_3 temp_2
        SUB r0 r6 0
        RMEM r1 [r0]
        SUB r4 r4 r1
        ;; end sub temp_9 temp_3 temp_2
        ;; add temp_10 temp_9 temp_1
        ADD r3 r4 r3
        ;; end add temp_10 temp_9 temp_1
        ;; copy temp_0 temp_10
        COPY r3 r3
        ;; end copy temp_0 temp_10


;;postlude
jump 0
.align16
stackend:
.reserve 42
stack:
```

MIF08 Feuille d'accompagnement

Mini-while (syntaxe abstraite)

Instructions :

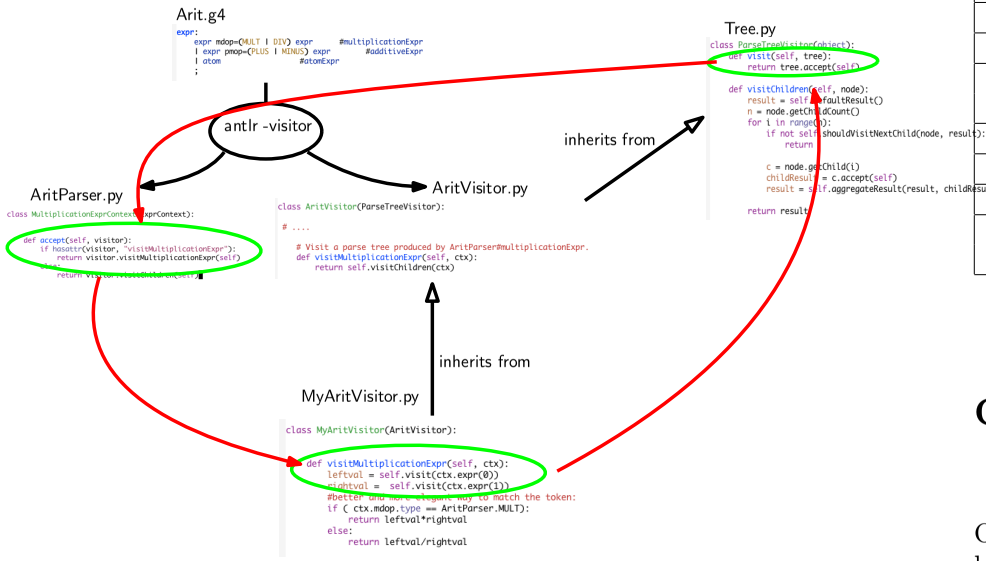
$S(Smt)$	::=	$x := e$	affectation
	|	*skip*	rien
	|	$S_1; S_2$	séquence
	|	`if` b `then` S_1 `else` S_2	test
	|	`while` b `do` S `done`	boucle

Grammaires ANTLR et Visiteurs

Syntaxe Python-ANTLR

- Accès à un non-terminal `name` : `ctx.name()`, si plus d'un : `ctx.name(0)`, `ctx.name(1)`...
- Appel récursif `self.visit(child)`
- Chaîne parsée d'un terminal : `xx.getText()`.

Python visitor mecanism



Visitor implementation Python/Antlr. Antlr generates `AritParser` *as well as* `AritVisitor`. *This* `AritVisitor` *inherits from the ParseTree visitor class (defined in* `Tree.py` *of the Antlr-Python library. When visiting a grammar object, a call to* `visit` *calls the highest level* `visit`*, which itself calls the accept method of the Parser object of the good type (in* `AritParser`*) which finally calls your implementation of* `MyAritVisitor` *that match this particuler type (here Multiplication).*

LEIA ISA

Liste des instructions de la machine cible. Pour initialiser des registres à constante, vous pouvez utiliser la macro `.let r 585`). L'instruction `snif op1 <condition> op2` "skip next if" désactive l'instruction suivante si la condition est vraie. Les opérandes 1 et 2 sont des registres (temporaires dans le cas du 3-adresse, physiques sinon) ou des constantes immédiates. La condition peut-être : `eq`, `neq`, `sgt`, `slt`, `gt`, `ge`, `lt` et `le`.

15	14	13	12	mnemonic	class	description	ext(i)
0	0	0	0	`wmem`	wmem	write to memory	
0	0	0	1	`add`	ALU	addition	$z(i)$
0	0	1	0	`sub`	ALU	subtraction	$z(i)$
0	0	1	1	`snif`	snif	skip next if	$s(i)$
0	1	0	0	`and`	ALU	logical bitwise and	$s(i)$
0	1	0	1	`or`	ALU	logical bitwise or	$s(i)$
0	1	1	0	`xor`	ALU	logical bitwise xor	$s(i)$
0	1	1	1	`lsl`	ALU	logical shift left	$z(i)$
1	0	0	0	`lsr`	ALU	logical shift right	$z(i)$
1	0	0	1	`asr`	ALU	arithmetic shift right	$z(i)$
1	0	1	0	`call`	call	sub-routine call	
1	0	1	1	`jump` `return`	jump	relative jump if `offset` $\neq 1$ return from call if `offset` $= 1$	
1	1	0	0	`letl`	letl	8-bit constant to Rd, sign-extended	
1	1	0	1	`leth`	leth	8-bit constant to high half of Rd	
1	1	1	0	`print`	print	print or refresh	
1	1	1	1	`rmem` `copy`	rmem	read from memory if i=0 register-to-register copy if i=1	

Génération de code 3-adresses

On rappelle the le code 3 addresses LEIA a le même jeu d'instructions que le code LEIA standard, à part pour les conditions qui utilisent l'instruction `condJUMP(label,t1,condition,t2)` et que les registres sont remplacés par des temporaires. Vous pouvez utiliser `.let` si vous le désirez.

`newTemp` : $() \to \mathbb{N}$ crée un nouveau temporaire (temp0, temp1, ...) et `newLabel` : $() \to \mathbb{N}$ crée un nouveau label (donnez le nom que vous voulez).

c	`dr <-newTemp()`
	`code.add(InstructionLETL(dr, c))`
	`return dr`

x	`# récupère le temporaire associé à x`
	`regval<-getTemp(x)`
	`return regval`

e_1+e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionADD(dr, t1, t2))`
	`return dr`

e_1-e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionSUB(dr, t1, t2))`
	`return dr`

true	`dr <-newTemp()`
	`code.add(InstructionLETL(dr, 1))`
	`return dr`

$e_1 < e_2$	`t1 <- GenCodeExpr(e1)`
	`t2 <- GenCodeExpr(e2)`
	`dr <- newTemp()`
	`endrel <- newLabel()`
	`code.add(InstructionLET(dr, 0))`
	`#if t1>=t2 jump to endrel`
	`code.add(InstructionCondJUMP(endrel, t1, ">=" , t2)`
	`code.add(InstructionLET(dr, 1))`
	`code.addLabel(endrel)`
	`return dr`

x := e	`dr <- GenCodeExpr(e)`
	`# copie du résultat dans le tmp associé à x`
	`let loc = loc(x) in`
	`code.add(instructionCOPY(loc,dr))`

S1 ; S2	`#concatène les codes générés`
	`GenCodeSmt(S1)`
	`GenCodeSmt(S2)`

if b then $S1$ else $S2$	`lelse,lendif <-newLabels()`
	`t1 <- GenCodeExpr(b)`
	`#si la condition est fausse, saute à "else"`
	`code.add(`
	`InstructionCondJUMP(lelse, t1, "=", 0))`
	`GenCodeSmt(S1) #then`
	`code.add(InstructionJUMP(lendif))`
	`code.addLabel(lelse)`
	`GenCodeSmt(S2) #else`
	`code.addLabel(lendif)`

while b do S done	`ltest,lendwhile <-newLabels()`
	`code.addLabel(ltest)`
	`t1 <- GenCodeExpr(b)`
	`#si la condition est fausse saute à la fin`
	`code.add(`
	`InstructionCondJUMP(lendwhile, t1, "=", 0))`
	`GenCodeSmt(S) #exécute S`
	`code.add(InstructionJUMP(ltest))#et va au test`
	`code.addLabel(lendwhile) #fin while`

Master Info - 2018-2019
MIF08 Compilation

Contrôle continu 1 (préparation) - Durée 15 min

Quelques exercices corrigés pour réviser

1 Un exercice d'assembleur

Écrire un code assembleur SARUMAN qui calcule la taille d'une chaîne de caractère stockée en mémoire.

On vous fournit le code à remplir, avec des commentaires.

Listing 1 – minmax.s

```
1         ;;
          lea r0 str                ; l'adresse du label str
strlen:                             ;
          leti r1 0                 ;
          setctr a0 r0              ; a0 contient l'adresse du premier char
6 loop:                             ; boucle qui calcule la taille du mot
          ;; TODO (5 lignes)




11
end:
          print string "and the result is: "
          print signed r1
          ;; mettre le résultat en mémoire au label res
16        ;; 3 lignes - considérer que le résultat tient sur 16bits




21 fin:                             ;boucle infinie pour le simul
          jump fin
str:
          .string "Hello, world!"
res:
26        .const 16 #100
```

Solution:

Listing 2 – minmax.s

```
          lea r0 str                ; the address of label str
strlen:                             ;
```

```
        leti r1 0
        setctr a0 r0            ; r0 has the parameter @str
loop:
        readse a0 8 r2          ; read 8 bits in memory (a char)
        add r2 r2 0
        jumpif z end            ; if the char is 0, end.
        add r1 1
        jump loop
end:
        print signed r1
        ;; now store in memory
        lea r3 res
        setctr a1 r3
        write a1 16 r1

fin :
        jump fin
str :
        .string "Hello, world!"
res :
        .const 16 #100
```

2 Deux corrections de TP1

2.1 Min et max en mémoire

Solution:

Listing 3 – minmax.s

```
leti r1 1232342342342342 ; first integer
leti r2 4523423423 ; second integer

lea r0 min                      ; adress of the begining of the stack
setctr a0 r0                    ; from this we can safely store values

cmp r1 r2
jumpif sgt swap ; If the integers are not in the right order, we swap then

write a0 64 r1 ; We write the min at mem[a0]
write a0 64 r2 ; We write the max at mem[a0] (a0 has been incremented by 64)

jump end

swap:
    write a0 64 r2
    write a0 64 r1
```

```
end:
    setctr  a0 r0
    readze a0 64 r3  ; We read the min from memory
    readze a0 64 r4  ; We read the max from memory
    print signed r3  ; We print the min
    print char '\n'
    print signed r4  ; We print the max
    print char '\n'

min:
```

2.2 Dessiner les carrés d'étoile

Solution:
Pour les carrés, il s'agit de traduire le pseudo-code : (on suppose que N est dans le registre R1)

```
R2 <- -R1
R3 <- 0
while( R3 < R1 ) {
  R4 <- 0
  while( R4 < R1 ) {
    print('*'); // and shift 15 pts right
    R4++
  }
  print('\n');
  R3++;
}
```

Listing 4 – carres.s

```
leti r0 0  ; zero
leti r1 4  ; n
leti r2 0
sub2 r2 r1
leti r3 0

loop:                       ; boucle externe sur r3
    cmp r3 r1
    jumpif ge halt
    leti r4 0

star:                       ; boucle interne qui dessine les *
    cmp r4 r1               ; compteur = r4
    jumpif ge endline
    print char '*'          ; et hop, dessine l'étoile
    add2i r4 1              ; incrémente r4
    jump star

endline:
```

```
    print char '\n'          ; à la fin de la ligne, on imprime...
    add2i r3 1               ; un saut de ligne, puis on incrémente r3
    jump loop

halt:
    jump -13
```

Master Info - 2018-2019
MIF08 Compilation

Contrôle continu 1 SUJET A - Durée 15 min

Barème sur 10 : 1 pt pour la comparaison à 0, 2 pour la deuxième comp + jump, 1 pt pour la troisième comparaison, 1 pt pour le add, 2 pts pour l'adresse dans r3, 2 points pour écrire mémoire, 1 pour un print (énoncé flou donc soit r2 dans la boucle, soit r1 à la fin).

L'objet de cet exercice est d'imprimer et compter le nombre de caractères *en minuscule* d'un mot stocké en mémoire, et de stocker ce nombre en mémoire. On utilise $r0, a0$ pour les adresses d'accès mémoire en lecture, $r2$ pour le code ascii de la lettre courante, $r3, a0$ pour l'accès mémoire pour écrire le résultat. On vous fournit le code à remplir et quelques indications.

Solution:

Listing 1 – nbminus.s

```
1       ;;
        lea r0 str              ; the address of label str
strlen:                         ;
        leti r1 0
        setctr a0 r0            ; r0 has the parameter @str
6   loop:
        readse a0 8 r2          ; read 8 bits in memory (a char)
        add r2 r2 0
        ;; a0 is incremented by 8 and r2 is the last written register ;
        jumpif z end            ; if the char is 0, end.
11      cmp r2 97               ; if not a regular char, do not inc
        jumpif slt endif
        cmp r2 122              ; same
        jumpif sgt endif
        print char r2
16      add r1 1
    endif:
        jump loop
    end:
        print signed r1
21      ;; now store in memory
        lea r3 res
        setctr a1 r3
        write a1 16 r1

26  fin:
        jump fin
    str:
        .string "He190LLo, 99wOrLd!01"
    res:
31      .const 16 #100
```

Master Info - 2018-2019
MIF08 Compilation

Contrôle continu 1 SUJET B - Durée 15 min

Barème sur 10 : 1 pt pour la comparaison à 0, 2 pour la deuxième comp + jump, 1 pt pour la troisième comparaison, 1 pt pour le add, 2 pts pour l'adresse dans r3, 2 points pour écrire mémoire, 1 pour un print (énoncé flou donc soit r2 dans la boucle, soit r1 à la fin).

L'objet de cet exercice est d'imprimer et de compter le nombre de caractères *chiffres* d'un mot stocké en mémoire, et de stocker ce nombre en mémoire. On utilise $r0, a0$ pour les adresses d'accès mémoire en lecture, $r2$ pour le code ascii de la lettre courante, $r3, a0$ pour l'accès mémoire pour écrire le résultat. On vous fournit le code à remplir et quelques indications.

Solution:

Listing 1 – nbchiffres.s

```
1       ;;
        lea r0 str              ; the address of label str
strlen:                         ;
        leti r1 0
        setctr a0 r0            ; r0 has the parameter @str
6   loop:
        readse a0 8 r2          ; read 8 bits in memory (a char)
        add r2 r2 0
        ;; a0 is incremented by 8 and r2 is the last written register ;
        jumpif z end            ; if the char is 0, end.
11      cmp r2 48               ; if not a digit, do not inc.
        jumpif slt endif
        cmp r2 57               ; same
        jumpif sgt endif
        print char r2
16      add r1 1
    endif:
        jump loop
    end:
        print signed r1
21      ;; now store in memory
        lea r3 res
        setctr a1 r3
        write a1 16 r1

26  fin:
        jump fin
    str:
        .string "He190llo, 99world!01"
    res:
31      .const 16 #100
```

Master Info - 2018-2019
MIF08 Compilation

Contrôle continu 2 (CC-TP) SUJET A - Durée 20 min

Éléments de correction

1 Manipulations préliminaires

— Sauvegardez vos modifications faites dans le git étudiant, par exemple :

```
git commit -a -m "mes modifs"
```

— Récupérer le sujet de contrôle :

```
git pull
```

— Vous travaillez dans le répertoire `CC-TPA`.

— Ouvrir le Makefile et changer `JohnDoe` en votre prénom suivi de votre nom (sans accent, sans espace, sans caractère spécial, tirets autorisés).

Solution:
attention le nom respect des consignes = points en moins. on attend un tgz, avec votre nom, et l'archive doit contenir uniquement ce qu'on a demandé (make clean avant de rendre)
5 points pour la grammaire et respect des consignes, 5 points pour les tests.

2 Exercice - grammaire avec ANTLR

L'objet de cet exercice est d'écrire un analyseur qui reconnaît les palindromes sur l'alphabet $\{a, b\}^*$ (de taille > 0 : ε n'est pas un palindrome), c'est-à-dire les mots qui sont égaux à leur "mot miroir". Votre analyseur complet devra ignorer les espaces, sauts de ligne, mais rejeter pendant la phase d'analyse lexicale les mots comportant d'autres caractères.

1. Écrire un premier fichier de test `tests/ex0.txt`.
2. Éditer le `.g4` pour coder l'analyseur (lexical/syntaxique). Tester avec :

   ```
   make
   python3 sujetA.py tests/ex0.txt
   ```

   On rappelle qu'un fichier accepté par la grammaire ne cause aucun affichage sur la sortie standard, seules d'éventuelles erreurs lexicales ou syntaxiques sont affichées.
3. Dans le répertoire `tests/` ajouter 5 tests pertinents pour cet analyseur (positifs, négatifs).
4. Pour déposer :

   ```
   make clean
   make tar
   ```

   vous fournit un tgz à déposer sur TOMUSS.

Solution:

```
grammar Palin;

//UNCOMMENT prog: EOF {print("Sujet A!");} ;

//START CUT
prog: pal  EOF;

pal:
     'a' pal 'a'
    |'a'
    | 'a' 'a'
    |'b' pal 'b'
    |'b'
    | 'b' 'b'
;

// END CUT

WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Master Info - 2018-2019
MIF08 Compilation

Contrôle continu 2 (CC-TP) SUJET B - Durée 20 min

Éléments de correction

1 Manipulations préliminaires

— Sauvegardez vos modifications faites dans le git étudiant, par exemple :

```
git commit -a -m "mes modifs"
```

— Récupérer le sujet de contrôle :

```
git pull
```

— Vous travaillez dans le répertoire `CC-TPB`.

— Ouvrir le Makefile et changer `JohnDoe` en votre prénom suivi de votre nom (sans accent, sans espace, sans caractère spécial, tirets autorisés).

Solution:
attention le nom respect des consignes = points en moins. on attend un tgz, avec votre nom, et l'archive doit contenir uniquement ce qu'on a demandé (make clean avant de rendre)
5 points pour la grammaire et respect des consignes, 5 points pour les tests.

2 Exercice - grammaire avec ANTLR

L'objet de cet exercice est d'écrire un analyseur qui reconnaît les mots sur l'alphabet $\{a, b\}^*$ (de taille > 0 : ε n'appartient pas au langage), tels que chaque a est immédiatement suivi d'au moins un b.

Votre analyseur complet devra ignorer les espaces, sauts de ligne, tabulations mais rejeter pendant la phase d'analyse lexicale les mots comportant d'autres caractères.

1. Écrire un premier fichier de test `tests/ex0.txt`.
2. Éditer le `.g4` pour coder l'analyseur (lexical/syntaxique). Tester avec :

   ```
   make
   python3 sujetB.py tests/ex0.txt
   ```

   On rappelle qu'un fichier accepté par la grammaire ne cause aucun affichage sur la sortie standard, seules d'éventuelles erreurs lexicales ou syntaxiques sont affichées.
3. Dans le répertoire `tests/` ajouter 5 tests pertinents pour cet analyseur (positifs, négatifs).
4. Pour déposer :

   ```
   make clean
   make tar
   ```

vous fournit un tgz à déposer sur TOMUSS.

Solution:

```
grammar Tutu;

//UNCOMMENT prog: EOF {print("Sujet B!");} ;

//START CUT

prog: tutu  EOF;

tutu:
      'a' 'b' tutu
    | 'b' tutu
    | 'a' 'b'
    | 'b'
;

// END CUT

WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Examen Session 2
Traduction et Compilation de Programmes
Mars 2019
Durée 2H

Éléments de correction (non contractuels...) Quelques typos et remarques de correction en rouge

Instructions **à lire attentivement !**

1. Les exercices sont indépendants.
2. On vous donne des indications de temps. **Regardez bien les différents exercices proposés et gérez votre temps en conséquence !**
3. L'examen est long, le barème en tiendra compte.
4. On demande de **justifier** les réponses.
5. Vous répondrez dans les cadres et rendrez l'examen entier **agraphé** tel quel.
6. L'anonymat est assuré par des copies classiques, sur lesquelles vous n'écrirez RIEN. Vous devez reporter le numéro d'anonymat sur la première page du sujet. MERCI DE COLLER SOIGNEUSEMENT LE RABAT.
7. Vous pouvez pour plus d'efficacité enlever la dernière feuille d'annexe et ne pas la remettre.

Exercice	Points (total)
1	5
2	2
3	6
4	3.5
5	6.5

Exercice 1 – Grammaires, Arbres, Attributions (30min)

Inspiration : Examen de compilation, Lille, M. Nebut

On s'intéresse à un langage de description d'assemblages de résistances du type de la figure 1. Pour décrire

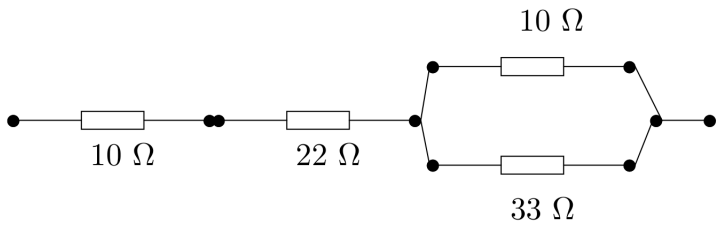


FIGURE 1 – Un assemblage de résistances

un tel assemblage, nous proposons un langage décrit par la grammaire ANTLR suivante :

```
grammar Resistance;

prog : decl CIRC circexpr EOF
     ;

decl : onedecl+ #listDecl ;

onedecl : ID INT #oneDecl ;

circexpr:
         ID       #idExpr
     | circexpr SER circexpr #serialExpr
     | circexpr PAR circexpr #parallelExpr
     | '(' circexpr ')'      #parentExpr
     ;
CIRC : 'circuit' ;
SER : 'ser';
PAR : 'par' ;
INT : [0-9]+ ;
ID : [A-Za-z0-9]+ ;
WS  :   (' '|'\t'|'\n')+  -> skip;
```

Ainsi un programme possède deux parties : tout d'abord une liste de déclaration de résistances avec un nom et une valeur. Ensuite une seconde section, introduite par le mot-clef "circuit", décrit l'assemblage (série - mot clef `ser`, et/ou parallèle - mot clef `par`)).

Le circuit de la figure est donc décrit par le programme :

```
R1 10
R2 22
R3 33
circuit
R1 ser R2 ser (R1 par R3)
```

La mise en série de deux composants est prioritaire sur leur mise en parallèle. Le calcul de résistance se fait de la gauche vers la droite : on considère que les opérateurs de mise en série et mise en parallèle sont associatifs à gauche.

Question #1

Pourquoi la règle lexicale `INT` précède-t-elle la règle lexicale `ID` ?

Solution:

Lorsque l'on lit une suite de chiffres, les deux règles peuvent être "matchées", et le nombre de lettres alors lues serait identiques. IL y a donc conflit de règles, et ANTLR choisit d'utiliser la règle mise en premier dans le texte. Dans le cas qui nous intéresse il faut donc que la règle INT soit en premier.

Question #2

Dessiner le sous-arbre **de dérivation** correspondant à la chaîne `R1 ser R2 ser (R1 par R3)`.

Solution:

RAS attention un arbre de dérivation n'est pas un AST, et on ne demandait pas le programme entier

On se propose tout d'abord d'écrire un visiteur pour vérifier que les résistances utilisées dans le circuit sont bien déclarées auparavant. On fournit le code pour vérifier qu'il n'y a pas deux fois la même déclaration.

Question #3

Compléter le visiteur pour réaliser la vérification dans la partie circuit *uniquement à l'endroit* `A COMPLETER`.

Solution:

Le comportement attendu est de laisser passer la chaîne si toutes les résistances sont bien déclarées (avec une valeur, par construction), il n'y a qu'à lancer une exception si on ne trouve pas dans le dictionnaire).

```
from ResistanceVisitor import ResistanceVisitor

class ResistanceTypeError(Exception):
    pass


class MyResistanceVisitor(ResistanceVisitor):
    def __init__(self):
        self._memoryRes = dict()  # pour stocker les variables

    # Visiteur pour *une declaration* de resistance
    def visitOneDecl(self, ctx):
        thevar = ctx.ID().getText()
        if thevar in self._memoryRes:
            raise ResistanceTypeError("Variable␣{}␣already␣declared".format(
                thevar))
        else:
            self._memoryRes[thevar] = None
        return

    # Visiteur pour *idExpr*
    def visitIdExpr(self, ctx):
        thevar = ctx.ID().getText()
        if thevar not in self._memoryRes:
            raise ResistanceTypeError("Variable␣{}␣undeclared".format(thevar))
        return
```

!

On se propose maintenant d'écrire une attribution sous forme d'un visiteur pour calculer la valeur de la résistance du circuit. On rappelle que la résistance r d'un assemblage de résistances est :

— la valeur de la résistance si le circuit est réduit à une résistance.

— pour une combinaison de deux sous-circuits C_1, supposé de résistance r_1, et C_2, de résistance r_2 :

  — si l'assemblage est série : $r = r_1 + r_2$

  — si l'assemblage est parallèle : $r = r_1 r_2/(r_1 + r_2)$.

La valeur de la résistance de l'assemblage de la Figure 1 est donc : $10 + 22 + 10 \times 33/(10 + 33)$.

Question #4

Remplir le visiteur pour réaliser cette attribution *aux trois endroits* `A COMPLETER`. L'attribut calculé doit se conformer à l'appel "main" suivant :

```
circ = parser.prog()  # Parse
visitor = MyResistanceVisitor2()  # Visit
valint = visitor.visit(circ)
print(''La resistance du circuit est ''+str(valint))
```

Solution:

Rien n'a dire non plus :

```
from ResistanceVisitor import ResistanceVisitor

#calculer la resistance

class ResistanceRuntimeError(Exception):
    pass
class ResistanceTypeError(Exception):
    pass

class MyResistanceVisitor2(ResistanceVisitor):
    def __init__(self):
        self._memoryRes = dict()  # pour stocker les variables

    def visitProg(self, ctx):
        self.visit(ctx.decl())
        return self.visit(ctx.circexpr())

    # Visit a parse tree produced by ResistanceParser#oneDecl.
    def visitOneDecl(self, ctx):
        thevar = ctx.ID().getText()
        thevalue = int(ctx.INT().getText())
        if thevar in self._memoryRes:
            raise ResistanceTypeError("Variable␣{}␣already␣declared".format(
                thevar))
        else:
            self._memoryRes[thevar] = thevalue
        return

    def visitParentExpr(self, ctx):
        return self.visit(ctx.circexpr())

    # Visit a parse tree produced by ResistanceParser#idExpr.
    def visitIdExpr(self, ctx):
        thevar = ctx.ID().getText()
        if thevar not in self._memoryRes:
```

```
            raise ResistanceTypeError("Variable␣{}␣undeclared".format(thevar))
        else:
            return self._memoryRes[thevar]

    # Visit a parse tree produced by ResistanceParser#serialExpr.
    def visitSerialExpr(self, ctx):
        val1 = self.visit(ctx.circexpr(0))
        val2 = self.visit(ctx.circexpr(1))
        return val1+val2

    # Visit a parse tree produced by ResistanceParser#parallelExpr.
    def visitParallelExpr(self, ctx):
        val1 = self.visit(ctx.circexpr(0))
        val2 = self.visit(ctx.circexpr(1))
        return val1*val2/(val1+val2)
```

!

Exercice 2 – Typage (10min)

On rappelle la syntaxe abstraite du mini-language while/Mu :

$S(Smt)$	::=	$x := e$	*affectation*
	|	`skip`	*ne rien faire*
	|	$S_1; S_2$	*sequence*
	|	`if` b `then` S_1 `else` S_2	*test*
	|	`while` b `do` S `done`	*boucle*

avec e une expression numérique (entiers, $+$, ...) et b une expression booléenne (`true`, or, ...).

Les règles de typage du langage Mu sont données en annexe.

```
var x,y: int; var b: bool;
[...]
I : if ((y < x) or b) then  x := x - y;
```

L'analyse des déclarations fournit l'environnement Γ suivant pour le reste du typage :

$$\Gamma = \{x \mapsto int, y \mapsto int, b \mapsto bool\}$$

Question #1

Montrer que la ligne de label I est bien typée. *On fera un arbre de preuve en appliquant les règles de typage de l'annexe.*

Solution:
On demande ici un arbre de preuve complet. Attention à bien fermer les feuilles en utilisant la règle "$Gamma(x) = t$".

Exercice 3 – Génération de code 3-adresses (40min)

On rappelle la syntaxe abstraite du mini-language while/Mu :

$S(Smt)$	::=	$x := e$	*affectation*
	|	`skip`	*ne rien faire*
	|	$S_1; S_2$	*sequence*
	|	`if` b `then` S_1 `else` S_2	*test*
	|	`while` b `do` S `done`	*boucle*

avec e une expression numérique (entiers, +, ...) et b une expression booléenne (`true`, or, ...).

Les Figures disponibles en annexe donnent les règles de génération de code 3 adresses du cours.

On considère le programme P_0 suivant :

```
var x,y: int;
x = 7;
y = 1;
while (y < 10) {x = x + 1 ; y = y + 1 ;}
```

Question #1

Remplir le code 3 adresses généré (*aux trois endroits* `TODO`) pour P_0 (le temporaire associé à x est $temp_1$, pour y c'est $temp_0$)

Solution:

Il suffit de suivre la feuille de génération.

```
;;Automatically generated TARGET code, MIF08 & CAP 2018
;;non executable 3-Address instructions version
        ;; (stat (assignment x = (expr (atom 7)) ;))
        leti temp_2 7
        let temp_1 temp_2
        ;; (stat (assignment y = (expr (atom 1)) ;))
        leti temp_3 1
        let temp_0 temp_3
        ;; (stat (while_stat while (expr (atom ( (expr (expr (atom y)) < (expr (atom 10))) )) ) (
    stat_block { (block (stat (assignment x = (expr (expr (atom x)) + (expr (atom 1))) ;)) (stat (
    assignment y = (expr (expr (atom y)) + (expr (atom 2))) ;))) }))))
lbl_l_while_begin_0:
        leti temp_4 10
        leti temp_5 0
        ;; cond_jump lbl_end_relational_1 temp_0 sge temp_4
        cmp temp_0 temp_4
        jumpif sge lbl_end_relational_1
        ;; end cond_jump lbl_end_relational_1 temp_0 sge temp_4
        leti temp_5 1
lbl_end_relational_1:
        ;; cond_jump lbl_l_while_end_0 temp_5 neq 1
        cmp temp_5 1
        jumpif neq lbl_l_while_end_0
        ;; end cond_jump lbl_l_while_end_0 temp_5 neq 1
        ;; (stat (assignment x = (expr (expr (atom x)) + (expr (atom 1))) ;))
        leti temp_6 1
        add3 temp_7 temp_1 temp_6
        let temp_1 temp_7
        ;; (stat (assignment y = (expr (expr (atom y)) + (expr (atom 2))) ;))
        leti temp_8 2
        add3 temp_9 temp_0 temp_8
        let temp_0 temp_9
        jump lbl_l_while_begin_0
lbl_l_while_end_0:


;;postlude
end:
```

jump end

On rajoute maintenant une instruction à notre langage, l'instruction `for`, avec cette syntaxe :

$$S(Smt) \quad ::= \ldots | \quad for(y = e_1 \ to \ e_2)S$$

avec e_i des expressions arithmétiques. Voici la signification de ce nouveau constructeur :

- y est une nouvelle variable **qui n'apparaît pas dans les déclarations**. Cette variable est déclarée et initialisée au moment du début de l'évaluation du for, elle est initialisée à la valeur de e_1 à ce moment du programme.
- Si e_1 est inférieur à e_2, alors l'instruction S est exécutée, en considérant que y vaut la valeur e_1, sinon, c'est fini.
- Une fois S exécuté, on incrémente y de 1, et on réévalue e_1 et e_2, et on retourne à l'étape précédente.

Question #2

Donner en justifiant un programme P_1 équivalent à P_0 en utilisant cette nouvelle construction.

Solution:
on attendait un programme mu

Question #3

(Question pas simple) Donner (justifier !) une règle de typage pour cette construction. Attention, il faut ajouter y à l'environnement de typage.

Solution:
Il faut donc évaluer le statement du for avec $\Gamma \cup \{y \mapsto t\}$ avec t le type de e_1 (qui probablement doit être int)

Question #4

Quel doit être le code généré **automatiquement** pour P_1 quand on aura écrit la règle de génération de la question suivante ? (évidemment ne montrez que les différences avec P_0 !)

Solution:
Il n'y a pas grand chose de différent, à part que l'incrément de y est à la fin et qu'il y a génération d'un nouveau temporaire pour lui.

Question #5

En justifiant, écrire la règle de génération de code pour le for. *Il est interdit d'utiliser la règle du while...*

Solution:
todo

Exercice 4 – Dataflow (15 min)

Considérons le programme suivant

```
1 var a,b,c:int
2 a = 0;
3 while (a < 9) {
4 b = a + 1 ;
5 c = c +b ;
6 a = b * 2; }
7 log(c);
```

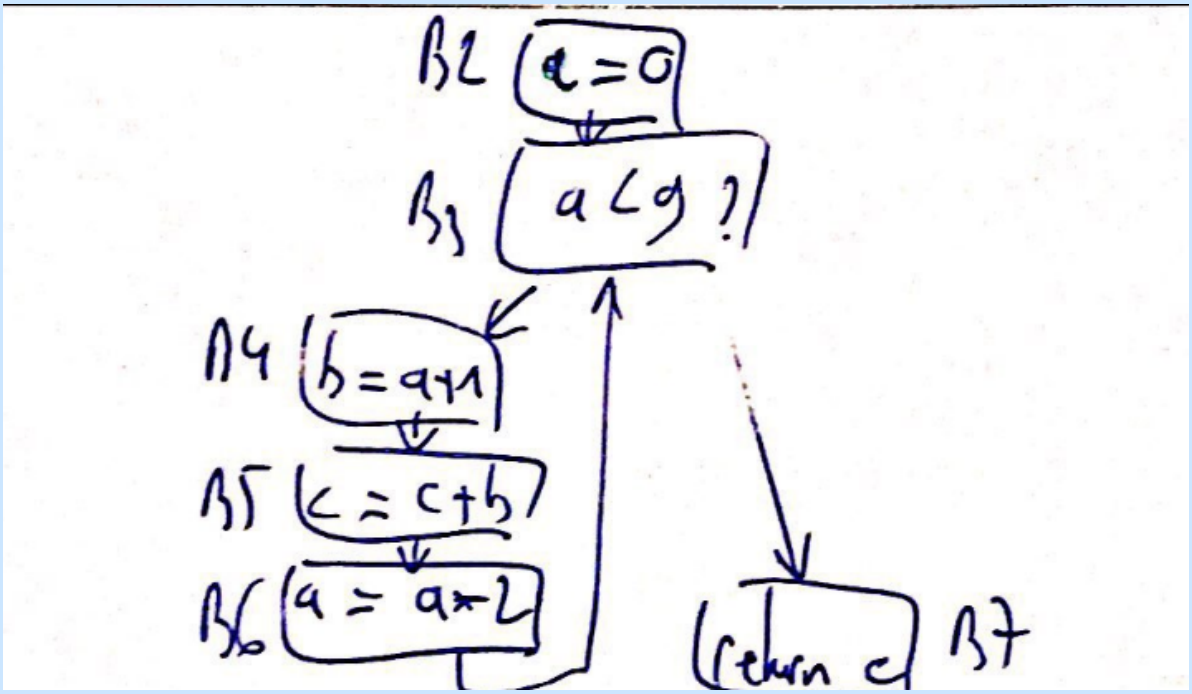
L'analyse de vivacité que nous allons réaliser se fait normalement sur le code 3 adresses, mais ici nous le faisons sur le graphe de flot contenant les instructions réelles du programme.

Question #1

Construire le graphe de flot de contrôle avec une **UNIQUE instruction par bloc**. Les blocs seront numérotés avec le numéro de la ligne de code correspondante.

Solution:

Rien de difficile. J'attendais un graphe de flot dont les blocs sont constitués d'une unique ligne. Un programme comportant une boucle tant que génère un graphe avec un cycle..., et ce qu'il y a après la boucle est après....



Attention un graphe de flot ne comporte pas de déclaration.

On rappelle qu'une variable est vivante après un bloc si il existe un chemin de ce bloc vers une utilisation de cette variable, et les définitions :

- $gen_{LV}(\ell)$ (aussi noté $gen(\ell)$) est l'ensemble des variables qui apparaissent comme opérandes sources dans ℓ, mais qui ne sont pas affectées avant dans ℓ.
- $kill_{LV}(\ell)$ (aussi noté $kill(\ell)$) est l'ensemble des variables définies dans le bloc (affectées).

On rappelle aussi les relations entre les ensembles $kill$, gen, LV_{entry} et LV_{exit} vues en TD :

$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{si } \ell = \text{final} \\ \bigcup\{LV_{entry}(\ell') | (\ell, \ell') \in flow(G)\} \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell),$$

avec $(\ell, \ell') \in flow(G)$ signifiant ℓ' successeur de ℓ dans le graphe de flot (aussi noté $\ell' \in Succ(\ell)$).

Les ensembles In et Out, qui convergeront vers LV_{entry} et LV_{exit} sont initialisés à $\emptyset$, et grossissent jusqu'à point fixe.

Question #2

Effectuer l'analyse *dataflow* des variables vivantes sur ce programme, en utilisant le tableau ci dessous. *Si vous n'avez pas assez de temps les initialisations et le résultat final attendu vous rapporteront des points.*

Solution:
Les initialisations / le résultat (les étapes sont laissées au lecteur) : par commodité les ensembles sont dénotés comme des listes.

				Step		Step		Step		Step	
ℓ	$Succ(\ell)$	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$
2	3	a	∅	∅	a	∅	a, c	c	a, c	c	a, c
3	4, 7	∅	a	a	a, c	a, c	a, c	a, c	a, c	a, c	a, c
4	5	b	a	a	b, c	a, c	b, c	a, c	b, c	a, c	b, c
5	6	c	b, c	b, c	b	b, c	b	b, c	b, c	b, c	b, c
6	3	a	b	b	∅	b	a, c	b, c	a, c	b, c	a, c
7	end	∅	c	c	∅	c	∅	c	∅	c	∅

Question #3

Y-a-t-il du code mort ? Si oui, quels blocs sont concernés ? Justifier.

Solution:
Il n'y a aucun code mort : toutes les variables définies à un endroit sont utilisées dans la suite du flot. On voit ceci dans le résultat du calcul, pour chaque bloc, les variables définies dans le bloc (kill) sont toutes vivantes en sortie.

Exercice 5 – Allocation de registre et génération de code machine.

D'après C. Paulin (LRI)

Soit le code 3 adresses suivant, où les lettres désignent des temporaires et x et n sont supposées vivantes en entrée du code :

```
1   leti y 1
2   let m x
3   let e n
4  loop:
5   condjump (e , "=", 0,    fin)   ; test e==0
6   add3i t y 2 ; premiere utilisation de y
7   condjump (t, "=", 0,   pair)
8   add3 y m y    ; deuxieme utilisation de y
9  pair:
10  add3 m m m
11  sub3 e e 2
12  jump loop
13 fin:
14  print signed y    ; troisieme utilisation de y
```

Question #1

Générer (et expliquer) le code final pour la **deuxième** ligne avec la stratégie d'allocation "tout en mémoire" (cf TP4), en supposant un offset 2 (fois 16) pour x et 3 (fois 16) pour m. *On utilisera $R_0/sp/a_0$ pour calculer et accéder aux adresses de pile, sp comme pointeur de pile, et R_0 et R_1 pour accéder aux éléments de pile, comme dans le TP.*

Solution:

Si ce n'est pas justifié, je ne donne pas tous les points

```
; lecture de x, qui se trouve à l'adresse sp+32
getctr sp r0
add r0 32
setctr a0 r0  ; positionnement du pointeur de lecture
read a0 16 r1 ; ou readse
getctr sp r0  ; et le symétrique
add r0 48
setctr a0 r0
write a0 16 r1
```

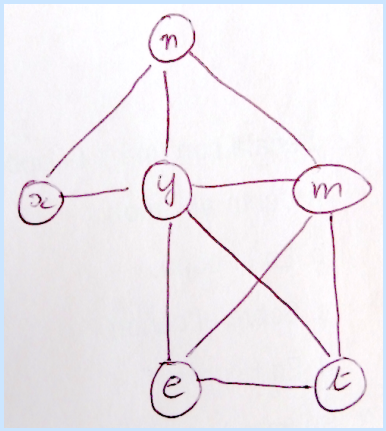
pas de lea là dedans

Question #2

Remplir les cases du tableau en mettant une étoile si la variable considérée est vivante *en entrée* de la ligne considérée, puis tracer le graphe d'interférence.

Solution:

ℓ	e	m	n	t	x	y
1			x		x	
2			x		x	x
3		x	x			x
4	x	x				x
5	x	x				x
6	x	x				x
7	x	x		x		x
8	x	x				x
9	x	x				x
10	x	x				x
11	x	x				x
12	x	x				x
13						x
14						x
15						x



Des intervalles de vivacité à trous ne sont pas des intervalles de vivacité...

Question #3

Colorier le graphe avec l'algorithme du cours et 4 couleurs (0 = rouge, 1 = vert, 2 = bleu, 3 = noir). *On écrira aussi la pile de coloriage en montrant clairement le haut de pile. On rappelle qu'il faut empiler les sommets de plus bas degré en premier, que les degrés sont mis à jour à chaque fois que l'on empile, et que si il y a un choix c'est le temporaire le plus proche du début de l'alphabet qui est empilé en premier ($e < m < n < t < x < y$). Enfin le coloriage se passe dans le sens inverse.*

Solution:

L'algorithme empile les sommets dans cet ordre : x, n, e, m, t, y. Les couleurs obtenues sont donc :

- variable y : couleur 0
- variable t : couleur 1
- variable m : couleur 2
- variable e : couleur 3
- variable n : couleur 1
- variable x : couleur 2

J'ai donné des points à des piles/ coloriages conformes avec le graphe de la question précédente, même faux. Merci de préciser le sens de la pile (ou le bas de pile)

Question #4

Montrer qu'il n'est pas possible d'utiliser uniquement 3 registres.

Solution:

La "pression registre" est trop forte : en particulier à l'entrée de la ligne 7 du programme, 4 variables sont vivantes simultanément. Attention l'argument "je n'arrive pas à colorier mon graphe avec 4 couleurs donc .." est archi faux. De même le fait qu'un graphe est 4 coloriable ne signifie pas qu'il n'est pas 3 coloriable. J'ai donné des points avec l'argument de la clique à 4 sommets, mais j'attendais un argument sur les variables en conflit ?

On se propose maintenant de stocker la variable y en mémoire à l'adresse pointée par sp, et d'allouer aux autres variable les registres suivants :

- variable e : registre R_2
- variable m : registre R_3
- variable t : registre R_4
- variable x : registre R_5

Question #5

Par quelles instructions est remplacée la ligne 1 du code 3 adresses dans le code finalement généré ? On utilisera R_0 et R_1 pour réaliser les calculs intermédiaires.

Solution:

NB : on n'a pas ajouté le `add r0 0` inutile vu que l'offset de y est 0. Un compilateur non optimisé pourrait la rajouter (idem questions 6 et 7).

```
leti r1 1
getctr sp r0
setctr a0 r0
write a0 16 r1
```

Question #6

Même question pour la ligne 6 (expliquer).

Solution:

On lit y dans R_0, puis on ajoute 2 en stockant le résultat dans t (registre R_4) :

```
getctr sp r0
setctr a0 r0
read a0 16 r0
add3i r4 r0 2
```

Pas de justification, pas tous les points

Question #7

Même question pour la ligne 8 (expliquer).

Solution:

On commence par lire la variable y dans le registre R_0 (par exemple), on ajoute m (registre R_3), puis on réécrit y en mémoire. On peut utiliser le même registre pour la deuxième occurence de y, ou bien un registre différent (comme ici).

```
getctr sp r0
setctr a0 r0
read a0 16 r0
add3 r1 r0 r3
getctr sp r0
setctr a0 r0
write a0 16 r1
```

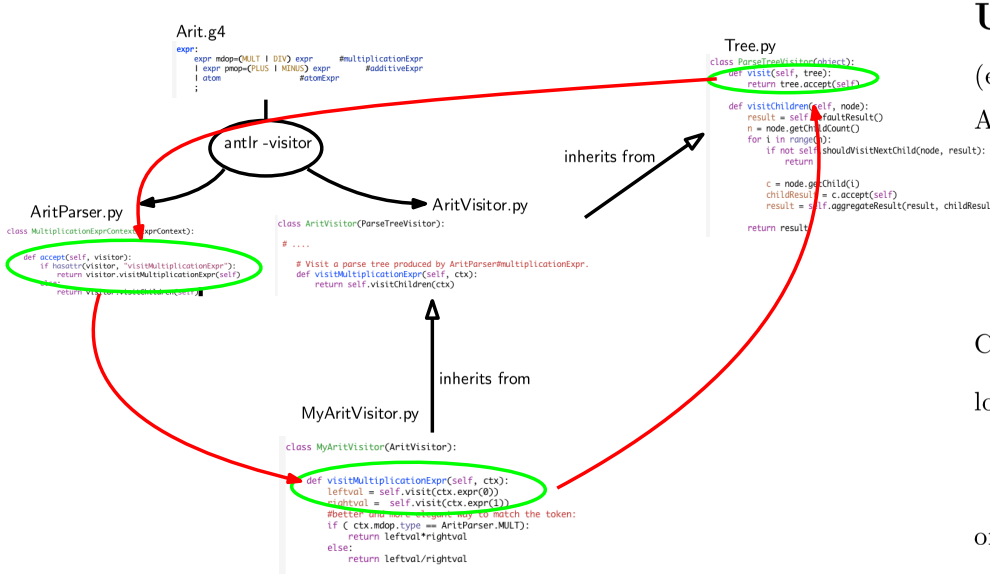
MIF08 Feuille d'accompagnement - version 2018/19

Grammaires ANTLR et Visiteurs

Syntaxe Python-ANTLR

- Accès à un non-terminal `name` : `ctx.name()`, si plus d'un : `ctx.name(0)`, `ctx.name(1)`...
- Appel récursif `self.visit(child)`
- Chaîne parsée d'un terminal : `xx.getText()`.

Python visitor mecanism



Visitor implementation Python/Antlr. Antlr generates `AritParser` *as well as* `AritVisitor`. *This* `AritVisitor` *inherits from the ParseTree visitor class (defined in* `Tree.py` *of the Antlr-Python library. When visiting a grammar object, a call to* `visit` *calls the highest level* `visit`*, which itself calls the accept method of the Parser object of the good type (in* `AritParser`*) which finally calls your implementation of* `MyAritVisitor` *that match this particuler type (here Multiplication).*

Typage

Les informations de déclaration fournissent $\Gamma : Var \to \texttt{t}$ avec $\texttt{t} \in \{int, bool\}$. Ensuite un jugement de type est de la forme $\Gamma \vdash e : \tau \in Basetype$ pour Γ construit comme précédemment. Les programmes/instructions bien typé(e)s sont de type `void`. Un programme est bien typé si son code type `void` sous l'environnement Γ

Règles pour les expressions (les autres sont similaires) :

$$\frac{\Gamma(x) = \texttt{t}}{\Gamma \vdash x : \texttt{t}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{ or } e_2 : \texttt{bool}}$$

Règles pour les instructions :

$$\frac{\Gamma \vdash e : \texttt{t} \qquad \Gamma \vdash x : \texttt{t}}{\Gamma \vdash x := e : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S1 : \texttt{void} \qquad \Gamma \vdash S2 : \texttt{void}}{\Gamma \vdash \texttt{if } b \texttt{ then } S1 \texttt{ else } S2 : \texttt{void}}$$

$$\frac{\Gamma \vdash S1 : \texttt{void} \qquad \Gamma \vdash S2 : \texttt{void}}{\Gamma \vdash S1; S2 : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while } b \texttt{ do } S \texttt{ done} : \texttt{void}}$$

Un peu de SARUMAN

(extraits de la documentation)

Arithmetical and logical instructions have 2 or 3 operands : (also sub, xor, ...)

```
add3i r1 r0 3      ; r1 <- r0+3
add2i r1 15        ; r1 <- r1+15
add3 r1 r2 r3      ; r1 <- r2+r3
add2 r1 r2         ; r1 <- r1+r2
```

Conditional jumps can be made with :

```
loop:
  sub2i r0 1       ; last written register r0
  jumpif nz loop   ; if r0 different from 0, jump
```

or

```
loop:
  cmpi r0 0        ; compare r0 to 0
  jumpif neq loop  ; if r0 different from 0, jump
```

(*nz* is non zero, *neq* not equal, you also have *sle* $=\leq$ and *lt* $=<$)

Read from memory ($r_2 \leftarrow Mem[r_1]$) :

```
setctr a0 r1       ; r1 is supposed to contain an address
readse a0 xxx r2   ; xxx the number of bits to read
```

Write to memory : same with `writese`.

Other useful instruction : `lea` (load effective address from a label).

Génération de code 3-adresses

On rappelle the le code 3 addresses SARUMAN a le même jeu d'instructions que le code SARUMAN standard, à part pour les conditions qui utilisent l'instruction `condJUMP(label,t1,condition,t2)` et qu'il n'y a pas de registres physiques mais des temporaires.

`newTemp` : $() \to \mathbb{N}$ crée un nouveau temporaire (temp0, temp1, ...) et `newLabel` : $() \to \mathbb{N}$ crée un nouveau label (donnez le nom que vous voulez).

x := e	`dr <- GenCodeExpr(e)`
	`# copie du résultat dans le tmp associé à x`
	`let loc = loc(x) in`
	`code.add(instructionLET(loc,dr))`

S1 ; S2	`#concatène les codes générés`
	`GenCodeSmt(S1)`
	`GenCodeSmt(S2)`

if b then $S1$ else $S2$	`lelse,lendif <-newLabels()`
	`t1 <- GenCodeExpr(b)`
	`#si la condition est fausse, saute à "else"`
	`code.add(`
	`InstructionCondJUMP(lelse, t1, "=", 0))`
	`GenCodeSmt(S1) #then`
	`code.add(InstructionJUMP(lendif))`
	`code.addLabel(lelse)`
	`GenCodeSmt(S2) #else`
	`code.addLabel(lendif)`

while b do S done	`ltest,lendwhile <-newLabels()`
	`code.addLabel(ltest)`
	`t1 <- GenCodeExpr(b)`
	`#si la condition est fausse saute à la fin`
	`code.add(`
	`InstructionCondJUMP(lendwhile, t1, "=", 0))`
	`GenCodeSmt(S) #exécute S`
	`code.add(InstructionJUMP(ltest))#et va au test`
	`code.addLabel(lendwhile) #fin while`

c	`dr <-newTemp()`
	`code.add(InstructionLETI(dr, c))`
	`return dr`

x	`# récupère le temporaire associé à x`
	`regval<-getTemp(x)`
	`return regval`

e_1+e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionADD(dr, t1, t2))`
	`return dr`

e_1-e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionSUB(dr, t1, t2))`
	`return dr`

true	`dr <-newTemp()`
	`code.add(InstructionLETI(dr, 1))`
	`return dr`

$e_1 < e_2$	`t1 <- GenCodeExpr(e1)`
	`t2 <- GenCodeExpr(e2)`
	`dr <- newTemp()`
	`endrel <- newLabel()`
	`code.add(InstructionLETI(dr, 0))`
	`#if t1>=t2 jump to endrel`
	`code.add(InstructionCondJUMP(endrel, t1, ">=" , t2)`
	`code.add(InstructionLETI(dr, 1))`
	`code.addLabel(endrel)`
	`return dr`

Master Info - 2019-2020
MIF08 Compilation

Contrôle continu 1 SUJET A - Durée 15 min

Aucun document autorisé. Répondre sur la feuille.
Nom : ..... **Prénom** .....

Un message `.LC1` est stocké en mémoire, ainsi qu'un entier `dec`. Remplir le code suivant pour réaliser le codage de César, c'est-à-dire "modifier chacune des lettres du message pour les remplacer en les décalant de `dec` caractères (cad +dec sur l'ascii)". Suivre les conventions et les instructions.

Listing 1 – 'cesar'

```
1  .section .text #Cesar Code (student version)
   .globl main
   main:
        addi    sp,sp,-16
        sd      ra,8(sp)
6  ## Your assembly code there
        la t3, .LC1
        mv a0, t3
        # now a0 contains the address of the chain
        call     print_string
11      # now: encode with cesar code.
        # use t2 and load byte instruction (lb) to load the chars from the msg
        # use sb sr, offet(reg) instruction to store a byte
        # the next byte is at address + 1

16

21

26

   end:    # now prints the transformed word.
        la a0, .LC1
        call print_string
        call newline
31 ## /end of user assembly code
        ld      ra,8(sp)
        addi    sp,sp,16
```

```
        jr      ra
ret

# Data comes here
        .section        .data
        .align  3
.LC1:
        .string "Hello world!\0"
.dec :
        .word 4
```

Mini-doc RISCV

Inst	Name	Description (C)
`li rd, imm`	Load immediate value	`rd = imm`
`mv rd, rs`	Copy register	`rd = rs`
`add rd, rs1, rs2`	ADD	`rd = rs1 + rs2`
`sub`	SUB	`rd = rs1 - rs2`
`xor`	XOR	`rd = rs1 ^ rs2`
`or`	OR	`rd = rs1 | rs2`
`and`	AND	`rd = rs1 & rs2`
`sll`	Shift Left Logical	`rd = rs1 << rs2`
`srl`	Shift Right Logical	`rd = rs1 >> rs2`
`addi rd, rs1, imm`	ADD Immediate	`rd = rs1 + imm`
`xori`	XOR Immediate	`rd = rs1 ^ imm`
`ori`	OR Immediate	`rd = rs1 | imm`
`andi`	AND Immediate	`rd = rs1 & imm`
`la rd, lbl`	Load address of label “lbl’ in rd ’	
`lb rd, imm(rs1)`	Load Byte at address rs1+imm	`rd = M[rs1+imm][0:7]`
`lw`	Load Word (32bits)	`rd = M[rs1+imm][0:31]`
`sb rs2, imm1(rs1)`	Store Byte (8bits)	`M[rs1+imm][0:7] = rs2[0:7]`
`sw`	Store Word	`M[rs1+imm][0:31] = rs2[0:31]`
`beq rs1, rs2, lbl`	Branch ==	`if(rs1 == rs2) jump to lbl`
`bne rs1, rs2, lbl`	Branch !=	`if(rs1 != rs2) jump to lbl`
`blt rs1, rs2, lbl`	Branch <	`if(rs1 <  rs2) jump to lbl`
`bge rs1, rs2, lbl`	Branch ≥	`if(rs1 >= rs2) jump to lbl`
`bgt rs1, rs2, lbl`	Branch >	`if(rs1 >  rs2) jump to lbl`
`ble rs1, rs2, lbl`	Branch ≤	`if(rs1 <= rs2) jump to lbl`
`j lbl`	Unconditional jump	`Jump to lbl`
`nop`	no operation	
`call lbl`	call the function at lbl	
`ret`	return from subroutine	

Master Info - 2019-2020
MIF08 Compilation

Contrôle continu 1 SUJET B - Durée 15 min

Aucun document autorisé. Répondre sur la feuille.
Nom : ..... **Prénom** .....

Un message `msg` est stocké en mémoire, ainsi qu'une clé `key`. Remplir le code suivant pour réaliser un codage de nom inconnu : "modifier chacune des lettres du message pour les remplacer par le résultat du xor avec la lettre correspondante de la clef". Suivre les conventions et les instructions.

Listing 1 – 'cesar'

```
1  .section .text #A noname dummy encoding
   .globl main
   main:
       addi    sp,sp,-16
       sd      ra,8(sp)
6  ## Your assembly code there
       la t3, .msg # address of the message
       mv a0, t3
       call print_string

11     # use t2 and load byte instruction (lb) to load the chars from the msg
       # now: encode with xor : msg[i] <- msg[i] xor key[i]
       # use sb sr, offet(reg) instruction to store a byte
       # the next byte is at address + 1

16

21

26

31
   end:  # now prints the transformed word.
       la a0, .msg
```

```
        call print_string
        call newline
## /end of user assembly code
        ld      ra,8(sp)
        addi    sp,sp,16
        jr      ra
ret

# Data comes here
        .section        .data
        .align  3
.msg:
        .string "Hello world!\0"
.key:
        .string "keykeykeykey\0"
```

Mini-doc RISCV

Inst	Name	Description (C)
`li rd, imm`	Load immediate value	`rd = imm`
`mv rd, rs`	Copy register	`rd = rs`
`add rd, rs1, rs2`	ADD	`rd = rs1 + rs2`
`sub`	SUB	`rd = rs1 - rs2`
`xor`	XOR	`rd = rs1 ^ rs2`
`or`	OR	`rd = rs1 | rs2`
`and`	AND	`rd = rs1 & rs2`
`sll`	Shift Left Logical	`rd = rs1 << rs2`
`srl`	Shift Right Logical	`rd = rs1 >> rs2`
`addi rd, rs1, imm`	ADD Immediate	`rd = rs1 + imm`
`xori`	XOR Immediate	`rd = rs1 ^ imm`
`ori`	OR Immediate	`rd = rs1 | imm`
`andi`	AND Immediate	`rd = rs1 & imm`
`la rd, lbl`	Load address of label “lbl’ in rd ’	
`lb rd, imm(rs1)`	Load Byte at address rs1+imm	`rd = M[rs1+imm][0:7]`
`lw`	Load Word (32bits)	`rd = M[rs1+imm][0:31]`
`sb rs2, imm1(rs1)`	Store Byte (8bits)	`M[rs1+imm][0:7] = rs2[0:7]`
`sw`	Store Word	`M[rs1+imm][0:31] = rs2[0:31]`
`beq rs1, rs2, lbl`	Branch ==	`if(rs1 == rs2) jump to lbl`
`bne rs1, rs2, lbl`	Branch !=	`if(rs1 != rs2) jump to lbl`
`blt rs1, rs2, lbl`	Branch <	`if(rs1 <  rs2) jump to lbl`
`bge rs1, rs2, lbl`	Branch ≥	`if(rs1 >= rs2) jump to lbl`
`bgt rs1, rs2, lbl`	Branch >	`if(rs1 >  rs2) jump to lbl`
`ble rs1, rs2, lbl`	Branch ≤	`if(rs1 <= rs2) jump to lbl`
`j lbl`	Unconditional jump	`Jump to lbl`
`nop`	no operation	
`call lbl`	call the function at lbl	
`ret`	return from subroutine	

Éléments de correction. CC1 2019-20 MIF08 Lyon1.

Caesar code (sujetA)

```
        call     print_string
        # now: encode with cesar code.
        li t0, 0 #init
        #load .dec address
        la a1, .dec
        lb t1, 0(a1)
loop:   lb t2, 0(t3)        # t2 : the char at the t3
        beq t2, zero, end
        add t2, t2, t1        # code in t2 += dec
        sb t2,0(t3)
        addi t3, t3, 1  # compute the next char address (+1, yes!)
        j loop
end:
# now prints the transformed word.
        la a0, .LC1
        call print_string
        call newline
```

---

xorcode (sujetB)

```
loop:   lb t2, 0(t3)        # t2 : the char at @stored in t3
        lb t1, 0(t4)
        beq t2, zero, end
        beq t1, zero, end
        xor t2, t2, t1        # code in t2 = t2 xor t1
        addi t2, t2, 65     # not mandatory in the paper version.
        sb t2, 0(t3)
        addi t3, t3, 1  # compute the next char address (+1, yes!)
        addi t4, t4, 1
        j loop
end :
```

Master Info - 2019-2020
MIF08 Compilation

Contrôle continu 1 SUJET A - Durée 15 min

Aucun document autorisé. Répondre sur la feuille.
Nom : ..... **Prénom** .....

Un message `.LC1` est stocké en mémoire, ainsi qu'un entier `dec`. Remplir le code suivant pour réaliser le codage de César, c'est-à-dire "modifier chacune des lettres du message pour les remplacer en les décalant de `dec` caractères (cad +dec sur l'ascii)". Suivre les conventions et les instructions.

Listing 1 – 'cesar'

```
1  .section .text #Cesar Code (student version)
   .globl main
   main:
        addi    sp,sp,-16
        sd      ra,8(sp)
6  ## Your assembly code there
        la t3, .LC1
        mv a0, t3
        # now a0 contains the address of the chain
        call     print_string
11      # now: encode with cesar code.
        # use t2 and load byte instruction (lb) to load the chars from the msg
        # use sb sr, offet(reg) instruction to store a byte
        # the next byte is at address + 1

16

21

26

   end:    # now prints the transformed word.
        la a0, .LC1
        call print_string
        call newline
31 ## /end of user assembly code
        ld      ra,8(sp)
        addi    sp,sp,16
```

```
        jr      ra
ret

# Data comes here
        .section        .data
        .align  3
.LC1:
        .string "Hello world!\0"
.dec :
        .word 4
```

Mini-doc RISCV

Inst	Name	Description (C)
`li rd, imm`	Load immediate value	`rd = imm`
`mv rd, rs`	Copy register	`rd = rs`
`add rd, rs1, rs2`	ADD	`rd = rs1 + rs2`
`sub`	SUB	`rd = rs1 - rs2`
`xor`	XOR	`rd = rs1 ^ rs2`
`or`	OR	`rd = rs1 | rs2`
`and`	AND	`rd = rs1 & rs2`
`sll`	Shift Left Logical	`rd = rs1 << rs2`
`srl`	Shift Right Logical	`rd = rs1 >> rs2`
`addi rd, rs1, imm`	ADD Immediate	`rd = rs1 + imm`
`xori`	XOR Immediate	`rd = rs1 ^ imm`
`ori`	OR Immediate	`rd = rs1 | imm`
`andi`	AND Immediate	`rd = rs1 & imm`
`la rd, lbl`	Load address of label “lbl’ in rd ’	
`lb rd, imm(rs1)`	Load Byte at address rs1+imm	`rd = M[rs1+imm][0:7]`
`lw`	Load Word (32bits)	`rd = M[rs1+imm][0:31]`
`sb rs2, imm1(rs1)`	Store Byte (8bits)	`M[rs1+imm][0:7] = rs2[0:7]`
`sw`	Store Word	`M[rs1+imm][0:31] = rs2[0:31]`
`beq rs1, rs2, lbl`	Branch ==	`if(rs1 == rs2) jump to lbl`
`bne rs1, rs2, lbl`	Branch !=	`if(rs1 != rs2) jump to lbl`
`blt rs1, rs2, lbl`	Branch <	`if(rs1 <  rs2) jump to lbl`
`bge rs1, rs2, lbl`	Branch ≥	`if(rs1 >= rs2) jump to lbl`
`bgt rs1, rs2, lbl`	Branch >	`if(rs1 >  rs2) jump to lbl`
`ble rs1, rs2, lbl`	Branch ≤	`if(rs1 <= rs2) jump to lbl`
`j lbl`	Unconditional jump	`Jump to lbl`
`nop`	no operation	
`call lbl`	call the function at lbl	
`ret`	return from subroutine	

Master Info - 2019-2020
MIF08 Compilation

Contrôle continu 1 SUJET B - Durée 15 min

Aucun document autorisé. Répondre sur la feuille.
Nom : ..... **Prénom** .....

Un message `msg` est stocké en mémoire, ainsi qu'une clé `key`. Remplir le code suivant pour réaliser un codage de nom inconnu : "modifier chacune des lettres du message pour les remplacer par le résultat du xor avec la lettre correspondante de la clef". Suivre les conventions et les instructions.

Listing 1 – 'cesar'

```
1  .section .text #A noname dummy encoding
   .globl main
   main:
       addi    sp,sp,-16
       sd      ra,8(sp)
6  ## Your assembly code there
       la t3, .msg # address of the message
       mv a0, t3
       call print_string

11     # use t2 and load byte instruction (lb) to load the chars from the msg
       # now: encode with xor : msg[i] <- msg[i] xor key[i]
       # use sb sr, offet(reg) instruction to store a byte
       # the next byte is at address + 1

16

21

26

31
   end:  # now prints the transformed word.
       la a0, .msg
```

```
        call print_string
        call newline
## /end of user assembly code
        ld      ra,8(sp)
        addi    sp,sp,16
        jr      ra
ret

# Data comes here
        .section        .data
        .align  3
.msg:
        .string "Hello world!\0"
.key:
        .string "keykeykeykey\0"
```

Mini-doc RISCV

Inst	Name	Description (C)
`li rd, imm`	Load immediate value	`rd = imm`
`mv rd, rs`	Copy register	`rd = rs`
`add rd, rs1, rs2`	ADD	`rd = rs1 + rs2`
`sub`	SUB	`rd = rs1 - rs2`
`xor`	XOR	`rd = rs1 ^ rs2`
`or`	OR	`rd = rs1 | rs2`
`and`	AND	`rd = rs1 & rs2`
`sll`	Shift Left Logical	`rd = rs1 << rs2`
`srl`	Shift Right Logical	`rd = rs1 >> rs2`
`addi rd, rs1, imm`	ADD Immediate	`rd = rs1 + imm`
`xori`	XOR Immediate	`rd = rs1 ^ imm`
`ori`	OR Immediate	`rd = rs1 | imm`
`andi`	AND Immediate	`rd = rs1 & imm`
`la rd, lbl`	Load address of label “lbl’ in rd ’	
`lb rd, imm(rs1)`	Load Byte at address rs1+imm	`rd = M[rs1+imm][0:7]`
`lw`	Load Word (32bits)	`rd = M[rs1+imm][0:31]`
`sb rs2, imm1(rs1)`	Store Byte (8bits)	`M[rs1+imm][0:7] = rs2[0:7]`
`sw`	Store Word	`M[rs1+imm][0:31] = rs2[0:31]`
`beq rs1, rs2, lbl`	Branch ==	`if(rs1 == rs2) jump to lbl`
`bne rs1, rs2, lbl`	Branch !=	`if(rs1 != rs2) jump to lbl`
`blt rs1, rs2, lbl`	Branch <	`if(rs1 <  rs2) jump to lbl`
`bge rs1, rs2, lbl`	Branch ≥	`if(rs1 >= rs2) jump to lbl`
`bgt rs1, rs2, lbl`	Branch >	`if(rs1 >  rs2) jump to lbl`
`ble rs1, rs2, lbl`	Branch ≤	`if(rs1 <= rs2) jump to lbl`
`j lbl`	Unconditional jump	`Jump to lbl`
`nop`	no operation	
`call lbl`	call the function at lbl	
`ret`	return from subroutine	

Éléments de correction. CC1 2019-20 MIF08 Lyon1.

Caesar code (sujetA)

```
        call     print_string
        # now: encode with cesar code.
        li t0, 0 #init
        #load .dec address
        la a1, .dec
        lb t1, 0(a1)
loop:   lb t2, 0(t3)        # t2 : the char at the t3
        beq t2, zero, end
        add t2, t2, t1        # code in t2 += dec
        sb t2,0(t3)
        addi t3, t3, 1  # compute the next char address (+1, yes!)
        j loop
end:
# now prints the transformed word.
        la a0, .LC1
        call print_string
        call newline
```

---

xorcode (sujetB)

```
loop:   lb t2, 0(t3)        # t2 : the char at @stored in t3
        lb t1, 0(t4)
        beq t2, zero, end
        beq t1, zero, end
        xor t2, t2, t1        # code in t2 = t2 xor t1
        addi t2, t2, 65     # not mandatory in the paper version.
        sb t2, 0(t3)
        addi t3, t3, 1  # compute the next char address (+1, yes!)
        addi t4, t4, 1
        j loop
end :
```

Master Info - 2019-2020
MIF08 Compilation

Contrôle continu 2 (CC-TP) SUJET Exemple - Durée 20 min

Éléments de correction

1 Manipulations préliminaires

— Sauvegardez vos modifications faites dans le git étudiant, par exemple :

```
git commit -a -m "mes modifs"
```

— Récupérer le sujet de contrôle :

```
git pull
```

— Vous travaillez dans le répertoire `CC2/CC2-ex`. Un Makefile, un main et une grammaire commentée est fournie :

```
grammar AnB2n;

start : EOF ;

COMMENT : '//' ~[\r\n]* -> skip ; // for unit tests in comments
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

— Ouvrir le Makefile et changer `JohnDoe` en votre prénom suivi de votre nom (sans accent, sans espace, sans caractère spécial, tirets autorisés).

— Vérifier que `make; make tests` fonctionnent. Les tests doivent retourner des erreurs, bien sûr.

2 Exercice - grammaire avec ANTLR

L'objet de cet exercice est d'écrire un analyseur qui reconnaît le langage $a^n b^{2n}$, $(n > 0)$. Les autres caractères alphabétiques (c,d, ... z, A, ... Z) ainsi que les blancs et tabulations seront ignorés, les autres caractères feront une erreur de syntaxe.

1. On vous fournit deux fichiers de tests `tests/ex0.txt` et . `tests/ex1.txt`
2. Éditer le `.g4` pour coder l'analyseur (lexical/syntaxique). Tester avec :

```
make
python3 sujetEx.py testsfiles/ex0.txt
```

On a écrit pour vous dans le main un affichage de "ok" (avec `EXITCODE 0`) si le fichier est accepté par la grammaire, par exemple ici. Dans le cas d'un fichier non accepté par le lexer, le programme affiche "syntax error" (avec `EXITCODE 2`), et dans le cas d'une erreur de lexicographie, "lexical error" (avec `EXITCODE 1`). En cas d'erreur, on peut obtenir le diagnostique d'ANTLR avec `python3 sujetEx.py --verbose`

3. Dans le répertoire `testfiles/` ajouter 5 tests pertinents pour cet analyseur (positifs, négatifs) et faire en sorte que cela fonctionne avec `make tests`.
4. Pour déposer :

   `make clean; make tar`

   vous fournit le tgz à déposer sur TOMUSS.

Solution:

```
grammar AnB2n;


//UNCOMMENT start: EOF;
// BEGIN CUT

start: akbk EOF;

akbk:
       A akbk B B
    |
;


A:'a' ;
B:'b' ;
CHARS: [c-z,A-Z] -> skip ; //skip chars


// END CUT



COMMENT
 : '//' ~[\r\n]* -> skip
 ;

WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Master Info - 2019-2020
MIF08 Compilation

Contrôle continu 2 (CC-TP) SUJET A - Durée 20 min (tiers-temps 30 min)

Éléments de correction sujet A

1 Manipulations préliminaires

— Sauvegardez vos modifications faites dans le git étudiant, par exemple :

```
git commit -a -m "mes modifs"
```

— Récupérer le sujet de contrôle :

```
git pull
```

— Vous travaillez dans le répertoire `CC2/CC2-TPA`. Un Makefile, un main et une grammaire à compléter sont fournis.

— Ouvrir le Makefile et changer `JohnDoe` en votre prénom suivi de votre nom (sans accent, sans espace, sans caractère spécial, tirets autorisés).

— Vérifier que `make; make tests` fonctionnent. Les tests doivent retourner des erreurs, bien sûr.

— Ouvrir un navigateur avec un onglet TOMUSS pour le rendu.

2 Exercice - grammaire avec ANTLR

L'objet de cet exercice est d'écrire un analyseur qui reconnaît le langage $a^n b^m$, $(n \geq 0, m > n)$. Les autres caractères alphabétiques (c,d, ... z, A, ... Z) ainsi que les blancs et tabulations seront ignorés, les autres caractères feront une erreur de syntaxe.

1. On vous fournit deux fichiers de tests `tests/ex0.txt` et . `tests/ex1.txt`
2. Éditer le `.g4` pour coder l'analyseur (lexical/syntaxique). Tester avec :

   ```
   make
   python3 sujetA.py testsfiles/ex0.txt
   ```

   On a écrit pour vous dans le main un affichage de "ok" (avec `EXITCODE 0`) si le fichier est accepté par la grammaire, par exemple ici. Dans le cas d'un fichier non accepté par le lexer, le programme affiche "syntax error" (avec `EXITCODE 2`), et dans le cas d'une erreur de lexicographie, "lexical error" (avec `EXITCODE 1`). En cas d'erreur, on peut obtenir le diagnostique d'ANTLR avec `python3 sujetEx.py --verbose`
3. Dans le répertoire `testfiles/` ajouter 5 tests pertinents pour cet analyseur (positifs, négatifs) et faire en sorte que cela fonctionne avec `make tests`.
4. Pour déposer :

   ```
   make clean; make tar
   ```

vous fournit le tgz à déposer sur TOMUSS.

Solution:

```
grammar SujetA;

//UNCOMMENT start: EOF;
//BEGIN CUT
start: akbk moreb EOF;

akbk: A akbk B
    | // epsilon
    ;

moreb: B moreb
     | B
     ;

A: 'a';
B: 'b';
CHARS: [c-zA-Z] -> skip ; //skip chars

//END CUT

COMMENT
 : '//' ~[\r\n]* -> skip
 ;

WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Master Info - 2019-2020
MIF08 Compilation

Contrôle continu 2 (CC-TP) SUJET B - Durée 20 min

Éléments de correction sujet B

1 Manipulations préliminaires

— Sauvegardez vos modifications faites dans le git étudiant, par exemple :

```
git commit -a -m "mes modifs"
```

— Récupérer le sujet de contrôle :

```
git pull
```

— Vous travaillez dans le répertoire `CC2/CC2-TPB`. Un Makefile, un main et une grammaire à compléter sont fournis.

— Ouvrir le Makefile et changer `JohnDoe` en votre prénom suivi de votre nom (sans accent, sans espace, sans caractère spécial, tirets autorisés).

— Vérifier que `make; make tests` fonctionnent. Les tests doivent retourner des erreurs, bien sûr.

— Ouvrir un navigateur avec un onglet TOMUSS pour le rendu.

2 Exercice - grammaire avec ANTLR

L'objet de cet exercice est d'écrire un analyseur qui reconnaît le langage $a^n b^m$, $(n > 0, m < n)$. Les autres caractères alphabétiques (c,d, ... z, A, ... Z) ainsi que les blancs et tabulations seront ignorés, les autres caractères feront une erreur de syntaxe.

1. On vous fournit deux fichiers de tests `tests/ex0.txt` et . `tests/ex1.txt`
2. Éditer le `.g4` pour coder l'analyseur (lexical/syntaxique). Tester avec :

   ```
   make
   python3 sujetB.py testsfiles/ex0.txt
   ```

   On a écrit pour vous dans le main un affichage de "ok" (avec `EXITCODE 0`) si le fichier est accepté par la grammaire, par exemple ici. Dans le cas d'un fichier non accepté par le lexer, le programme affiche "syntax error" (avec `EXITCODE 2`), et dans le cas d'une erreur de lexicographie, "lexical error" (avec `EXITCODE 1`). En cas d'erreur, on peut obtenir le diagnostique d'ANTLR avec `python3 sujetEx.py --verbose`
3. Dans le répertoire `testfiles/` ajouter 5 tests pertinents pour cet analyseur (positifs, négatifs) et faire en sorte que cela fonctionne avec `make tests`.
4. Pour déposer :

   ```
   make clean; make tar
   ```

   vous fournit le tgz à déposer sur TOMUSS.

Solution:

```
grammar SujetB;


//UNCOMMENT start: EOF;
//BEGIN CUT
start: morea akbk EOF;

akbk: A akbk B
    | // epsilon
    ;

morea: A
     | A morea
     ;

A: 'a';
B: 'b';
CHARS: [c-zA-Z] -> skip ; //skip chars

//END CUT

COMMENT
 : '//' ~[\r\n]* -> skip
 ;

WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Compilation de Programmes

CCF Session 1 - **NE PAS ENLEVER LES AGRAPHES**

Durée totale : 1 heure 30

Toute communication (orale, téléphonique, par messagerie, etc.) avec les autres étudiants est interdite. Aucun document autorisé.

— Pour la partie QCM, plusieurs réponses peuvent être valides à chaque question, on souhaite avoir **toutes** les réponses valides. Chaque question admet au moins une réponse valide et au moins une réponse incorrecte. Il n'y aura pas de point négatif.

— Pour les parties rédigées, vous répondrez obligatoirement dans les parties prévues pour. Il ne sera pas donné de nouvelle copie.

Consignes :

- Utiliser un **stylo à bille noir ou bleu**.
- **Noircir ou bleuir** la/les cases, sans dépasser !
- Pour corriger (dernier recours) : effacez proprement la case.
- Dans les parties rédigées, les carrés gris (prof) sont pour la correction, merci de ne rien écrire dedans.

Numéro étudiant à coder (sans le p, 8 chiffres, il est sur votre carte étudiant !)

- Notez-le ici : .....
- Encodez-le ci-contre (chiffre des unités tout à droite, en remplaçant p par 1) : par exemple, pour un numéro $p1234567$, ie 11234567 vous grisez le 1 de la première colonne, le 1 de la deuxième, le 2 de la troisième...).

0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9

1 Questions de cours/TP

Question 1 ♣ Les étapes d'analyse et transformation réalisées par un compilateur sont :

- ☐ L'analyse syntaxique.
- ☐ L'analyse p-adique.
- ☐ La coloration syntaxique.
- ☐ La transformation de Fourier.
- ☐ L'évaluation des opérations arithmétiques par un interprète.
- ☐ La vérification de type.
- ☐ Le chargement du fichier binaire exécutable en mémoire.
- ☐ L'allocation de registres.
- ☐ La génération de code 3 adresses.
- ☐ L'analyse lexicale.

Question 2 ♣

Pendant les TPs, du code généré par ANTLR est utilisé pour :

- ☐ La génération de code 3 adresses.
- ☐ L'analyse syntaxique de code MiniC.
- ☐ Le typage.
- ☐ L'analyse syntaxique de code RiscV.

For your examination, preferably print documents compiled from auto-multiple-choice.

Question 3 ♣ Cochez les affirmations correctes à propos des visiteurs générés par ANTLR4 :

- ☐ Les visiteurs permettent de faire simplement un parcours en largeur d'abord de l'arbre de dérivation.
- ☐ Les visiteurs permettent de faire simplement un parcours en profondeur d'abord de l'arbre de dérivation.
- ☐ Il est pertinent d'utiliser un visiteur pour réaliser un interprète pour le langage source.
- ☐ Il est pertinent d'utiliser un visiteur pour réaliser le coloriage de graphe utilisé pour l'allocation de registres.
- ☐ Il est pertinent d'utiliser un visiteur pour réaliser l'analyse de vivacité.
- ☐ Il est pertinent d'utiliser un visiteur pour réaliser un générateur de code à partir du langage source.
- ☐ Il est pertinent d'utiliser un visiteur pour réaliser l'analyse de typage.

Question 4 ♣ Dans les TPs, le visiteur de génération de code permet de :

- ☐ S'assurer que le code MiniC en entrée est correctement typé.
- ☐ Calculer la valeur des expressions du programme d'entrée.
- ☐ Générer du code RiscV prêt à être assemblé et exécuté.
- ☐ Générer du code pas tout à fait exécutable.

Question 5 ♣
Dans la phase d'allocation de registres, l'analyse de vivacité des variables ("*liveness*") :

- ☐ Calcule exactement les variables vivantes en chaque point du programme.
- ☐ Se fait sur l'arbre (de dérivation/syntaxique) du programme.
- ☐ Sert à calculer des informations de conflit.
- ☐ Calcule un sous-ensemble des variables vivantes en chaque point.
- ☐ Calcule un sur-ensemble des variables vivantes en chaque point.

Question 6 ♣
Dans notre série de TP, les tests unitaires (infrastructure `pytest`) :

- ☐ Permettent d'avoir confiance en notre compilateur.
- ☐ Servent à tester le code final généré.
- ☐ Servent à tester l'absence d'erreur à l'exécution.

Question 7 ♣
Dans un compilateur C, le typage :

- ☐ Peut être statique ou dynamique.
- ☐ S'effectue après la phase de génération de code.
- ☐ Permet de s'assurer de l'absence d'erreur à l'exécution.
- ☐ Permet d'implémenter le domaine d'application de certaines opérations de calcul, définies dans la spécification du langage.
- ☐ Permet d'éliminer les programmes que l'on ne désire pas compiler.

Question 8 ♣
Dans un compilateur, sont NP-complets les problèmes suivants :

- ☐ L'ordonnancement d'instructions à l'intérieur d'un bloc de base.
- ☐ La génération de code.
- ☐ L'analyse syntaxique.
- ☐ Le coloriage du graphe de conflit avec un nombre minimal de couleurs.
- ☐ Le calcul des intervalles de durée de vie des variables temporaires.

2 MiniC : Génération de code 3 adresses

Voici une partie d'un programme MiniC.

```
while (x > 3) {
   x = x - y; }
```

Question 9 En utilisant les règles de génération de code fournies, et en considérant la mémoire $x \mapsto temp_0, y \mapsto temp_1$, générer le code 3 adresses RISCV correspondant à ce code [a]. En commentaire du code généré, mettez les appels aux fonctions de génération de code utilisées. Il n'est pas nécessaire de dérouler tous les appels récursifs comme en TD.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Prof*

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

a. On rappelle que le code trois adresses a le même jeu d'instruction que la machine RISCV du cours, à part pour les conditions qui utilisent le raccourci `condJUMP(label,t1,condition,t2)`. De plus il utilise les registres temporaires au lieu des registres physiques.

3 MiniC : une nouvelle instruction

On considère des expressions numériques ternaires à la C, comme ceci : $b?e_1 : e_2$ qui informellement signifie que l'expression vaut la valeur de e_1 si l'expression b est évaluée à vrai, et la valeur de e_2 sinon.

On augmente donc la syntaxe des **expressions numériques** :

e	$::=$	c	*constant*
	$\mid$	x	*variable*
	$\mid$	$e + e$	*addition*
	$\mid$	$e \times e$	*multiplication*
	$\mid$	$b?e : e$	*if ternaire (NEW)*

Question 10 En s'inspirant des règles de typage fournies dans la feuille d'accompagnement, écrire une règle de typage pour ce nouveau constructeur d'expression.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Prof*

.....

.....

.....

.....

.....

.....

Question 11 En utilisant votre règle de typage précédente et celles de la feuille d'accompagnement, montrer que l'affectation $y = (x == 8)?18 : 42 + x$ est bien typée sous l'environnement $\Gamma : x \mapsto int, y \mapsto int$. *On fera attention à bien faire un arbre de preuve correct.*

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Prof*

Question 12 Sur le modèle de la règle de génération de code des expressions, écrire une règle de génération de code pour ce nouveau constructeur d'expression. *On fera attention à ne pas évaluer la deuxième expression trop tôt, et on justifiera la réponse !* ☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Prof*

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

4 Variables vivantes (liveness) et génération de code

On considère le programme suivant :

```
int x,y,z,t;
x=12; y=3+x; z=4+y; t=x-y+z;
```

Pour lequel notre compilateur des TP4 et TP5 produit le code suivant, avec :

$$(t, z, y, x) \mapsto (temp0, temp1, temp2, temp3)$$

```
1: li  temp4 12
2:mv temp3 temp4
3: li  temp5 3
4:add temp6 temp5 temp3
5:mv temp2 temp6
6: li  temp7 4
7:add temp8 temp7 temp2
8:mv temp1 temp8
9:sub temp9 temp3 temp2
10:add temp10 temp9 temp1
11:mv temp0 temp10
```

Question 13 Remplir le tableau suivant avec les résultats de l'analyse de variables vivantes. Une étoile dans une ligne veut dire "ce temporaire est vivant à **l'entrée** de cette ligne"

. . . . . ☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Prof*

code	temp0	temp1	temp2	temp3	temp4	temp5	temp6	temp7	temp8	temp9	temp10
`li temp4 12`											
`mv temp3 temp4`											
`li temp5 3`											
`add temp6 temp5 temp3`											
`mv temp2 temp6`											
`li temp7 4`											
`add temp8 temp7 temp2`											
`8:mv temp1 temp8`											
`9:sub temp9 temp3 temp2`											
`add temp10 temp9 temp1`											
`mv temp0 temp10`											
`...`											

Question 14 Dessiner le graphe d'interférence. *Laisser de la place pour pouvoir faire la question d'après dans le cadre.*

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Prof*

Dans le reste de l'exercice on utilise les notations suivantes :

- La couleur 1 est ROUGE et est associée au registre physique t2.
- La couleur 2 est BLEU et est associée au registre physique t3.
- La couleur 3 est VERT et est associée à un *spill* en mémoire avec un offset -1 (ie un décalage de -8 octets par rapport au registre `fp` qui contient l'adresse de la pile).
- Les couleurs suivantes sont NOIR, JAUNE, VIOLET, et les offset respectifs -2,-3,-4. . ..

On rappelle que l'algorithme de coloration du cours empile les sommets de plus bas degré en premier, que les degrés sont mis à jour à chaque fois que l'on empile, et que si il y a un choix c'est le temporaire de numéro le plus faible qui est empilé en premier. Enfin le coloriage se passe dans le sens inverse de la création de la pile de coloriage.

Question 15 Colorier dans le cadre précédent le graphe avec l'algorithme du cours et un nombre infini de couleurs (3 devraient suffire !). *On écrira aussi la pile de coloriage en montrant clairement le haut de pile.* Donner l'allocation déduite dans le cadre ci dessous.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Prof*

.....

.....

.....

.....

Question 16 Générer les (vraies) instructions RISCV pour les instructions des lignes 8 et 9. *On utilisera s_1, s_2, s_3, fp pour la gestion de la pile.*

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Prof*

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Feuille supplémentaire de rédaction à utiliser seulement en cas de drame

$0Prof$

CORRECTED

Compilation de Programmes

CCF Session 1 - **NE PAS ENLEVER LES AGRAPHES**

Durée totale : 1 heure 30

Toute communication (orale, téléphonique, par messagerie, etc.) avec les autres étudiants est interdite. Aucun document autorisé.

— Pour la partie QCM, plusieurs réponses peuvent être valides à chaque question, on souhaite avoir **toutes** les réponses valides. Chaque question admet au moins une réponse valide et au moins une réponse incorrecte. Il n'y aura pas de point négatif.

— Pour les parties rédigées, vous répondrez obligatoirement dans les parties prévues pour. Il ne sera pas donné de nouvelle copie.

Consignes :

- Utiliser un **stylo à bille noir ou bleu**.
- **Noircir ou bleuir** la/les cases, sans dépasser !
- Pour corriger (dernier recours) : effacez proprement la case.
- Dans les parties rédigées, les carrés gris (prof) sont pour la correction, merci de ne rien écrire dedans.

Numéro étudiant à coder (sans le p, 8 chiffres, il est sur votre carte étudiant !)

- Notez-le ici : .....
- Encodez-le ci-contre (chiffre des unités tout à droite, en remplaçant p par 1) : par exemple, pour un numéro $p1234567$, ie 11234567 vous grisez le 1 de la première colonne, le 1 de la deuxième, le 2 de la troisième...).

☐0	☐0	☐0	☐0	☐0	☐0	☐0	☐0
☐1	☐1	☐1	☐1	☐1	☐1	☐1	☐1
☐2	☐2	☐2	☐2	☐2	☐2	☐2	☐2
☐3	☐3	☐3	☐3	☐3	☐3	☐3	☐3
☐4	☐4	☐4	☐4	☐4	☐4	☐4	☐4
☐5	☐5	☐5	☐5	☐5	☐5	☐5	☐5
☐6	☐6	☐6	☐6	☐6	☐6	☐6	☐6
☐7	☐7	☐7	☐7	☐7	☐7	☐7	☐7
☐8	☐8	☐8	☐8	☐8	☐8	☐8	☐8
☐9	☐9	☐9	☐9	☐9	☐9	☐9	☐9

1 Questions de cours/TP

Question 1 ♣ Les étapes d'analyse et transformation réalisées par un compilateur sont :

- ■ L'analyse syntaxique.
- ☐ L'analyse p-adique.
- ☐ La coloration syntaxique.
- ☐ La transformation de Fourier.
- ☐ L'évaluation des opérations arithmétiques par un interprète.
- ■ La vérification de type.
- ☐ Le chargement du fichier binaire exécutable en mémoire.
- ■ L'allocation de registres.
- ■ La génération de code 3 adresses.
- ■ L'analyse lexicale.

Question 2 ♣

Pendant les TPs, du code généré par ANTLR est utilisé pour :

- ■ La génération de code 3 adresses.
- ■ L'analyse syntaxique de code MiniC.
- ■ Le typage.
- ☐ L'analyse syntaxique de code RiscV.

Question 3 ♣ Cochez les affirmations correctes à propos des visiteurs générés par ANTLR4 :

- ☐ Les visiteurs permettent de faire simplement un parcours en largeur d'abord de l'arbre de dérivation.
- ■ Les visiteurs permettent de faire simplement un parcours en profondeur d'abord de l'arbre de dérivation.
- ■ Il est pertinent d'utiliser un visiteur pour réaliser un interprète pour le langage source.
- ☐ Il est pertinent d'utiliser un visiteur pour réaliser le coloriage de graphe utilisé pour l'allocation de registres.
- ☐ Il est pertinent d'utiliser un visiteur pour réaliser l'analyse de vivacité.
- ■ Il est pertinent d'utiliser un visiteur pour réaliser un générateur de code à partir du langage source.
- ■ Il est pertinent d'utiliser un visiteur pour réaliser l'analyse de typage.

Question 4 ♣ Dans les TPs, le visiteur de génération de code permet de :

- ☐ S'assurer que le code MiniC en entrée est correctement typé.
- ☐ Calculer la valeur des expressions du programme d'entrée.
- ☐ Générer du code RiscV prêt à être assemblé et exécuté.
- ■ Générer du code pas tout à fait exécutable.

Question 5 ♣
Dans la phase d'allocation de registres, l'analyse de vivacité des variables ("*liveness*") :

- ☐ Calcule exactement les variables vivantes en chaque point du programme.
- ☐ Se fait sur l'arbre (de dérivation/syntaxique) du programme.
- ■ Sert à calculer des informations de conflit.
- ■ Calcule un sous-ensemble des variables vivantes en chaque point.
- ■ Calcule un sur-ensemble des variables vivantes en chaque point.

Question 6 ♣
Dans notre série de TP, les tests unitaires (infrastructure `pytest`) :

- ■ Permettent d'avoir confiance en notre compilateur.
- ■ Servent à tester le code final généré.
- ☐ Servent à tester l'absence d'erreur à l'exécution.

Question 7 ♣
Dans un compilateur C, le typage :

- ☐ Peut être statique ou dynamique.
- ☐ S'effectue après la phase de génération de code.
- ☐ Permet de s'assurer de l'absence d'erreur à l'exécution.
- ■ Permet d'implémenter le domaine d'application de certaines opérations de calcul, définies dans la spécification du langage.
- ■ Permet d'éliminer les programmes que l'on ne désire pas compiler.

Question 8 ♣
Dans un compilateur, sont NP-complets les problèmes suivants :

- ■ L'ordonnancement d'instructions à l'intérieur d'un bloc de base.
- ☐ La génération de code.
- ☐ L'analyse syntaxique.
- ■ Le coloriage du graphe de conflit avec un nombre minimal de couleurs.
- ☐ Le calcul des intervalles de durée de vie des variables temporaires.

Attention ; il y a une erreur dans le pdf QCM corrigé généré automatiquement : l'analyse des variables vivantes calcule un sur-ensemble. La correction automatique a en revanche, tenu compte de la bonne réponse.

attention dans vos copies la note calculée n'est pas la note finale. La bonne note est celle sur tomuss.

Ce sont des éléments de correction, tout n'est pas complètement rédigé ! LG Janvier 2020

Notes de barème

Pour le QCM $0,5$ ou 1 point par question. Exercice 2 deux points, Exercice 3 2 pts pour la règle, 1 point pour l'arbre, 2 points pour la règle de génération de code. Exercice 4 2 points pour la liveness, 1.5 pour le graphe d'interférence, 3 points pour le coloriage et l'allocation, 1 point pour le code final.

2. Génération de code

Voici le code généré par notre compilo : (il manque quelques commentaires).

```
python3 ../../../TP2019-20/TP04/MiniC-codegen/Main.py exam19.c --reg-alloc=none
```

```
##Automatically generated RISCV code, MIF08 & CAP 2019
##non executable 3-Address instructions version
# coupé pour la correction
        # (stat (while_stat while ( (expr (expr (atom x)) > (expr (atom 3))) ) (stat_block { (block (stat
    (assignment x = (expr (expr (atom x)) - (expr (atom y)))) ;)) }))))
lbl_l_while_begin_0:
        li temp_2, 3
        li temp_3, 0
        ble temp_0, temp_2, lbl_end_relational_1
        li temp_3, 1
lbl_end_relational_1:
        beq temp_3, zero, lbl_l_while_end_0
        # (stat (assignment x = (expr (expr (atom x)) - (expr (atom y)))) ;)
        sub temp_4, temp_0, temp_1
        mv temp_0, temp_4
        j lbl_l_while_begin_0
lbl_l_while_end_0:
        # Return at end of function: #CUTCUT
```

3. Expression ifthenelse

Règle de typage Sans difficulté

$$\frac{\Gamma \vdash b : \texttt{bool} \qquad \Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash b ? e_1 : e_2 : \texttt{int}}$$

Attention, on type une expression, donc le type obtenu ne peut pas être `void`.

Arbre de typage Le début

```
\Gamma(y)=int     \Gamma |- x==8:bool  \Gamma |-18: int \Gamma |- 42+x : int
----------------            -------------------------------
```

```
\Gamma  |- y:int              \Gamma |- (x==8)?18: 42+x : int
-------------------------------------------------
\Gamma |- y = (x==8)?18: 42+x : void
```

Il manquait une règle dans la feuille d'accompagnement, il fallait appliquer la règle "naturelle".

Règle de génération de code On fait attention à ne pas évaluer la deuxième expression trop tôt, en particulier calculer la valeur pour tous les cas de b est une mauvaise idée. Attention on génère un code pour les expressions, donc on calcule dans un nouveau temporaire que l'on retourne à la fin.

```
CodeGenExpr(b?e1:e2) ==
dr <- newtemp()  # pour stocker le résultat final
lmilieu,lfin <- newlabels()
tb <- CodeGenExpr(b)
code.addinstructionCONDJUMP(tb , "=" , 0, lmilieu) # b faux
t1 <- CodeGenExpr(e1)
code.addinstructionMV(dr, t1)  # ne pas oublier ceci !
code.addInstructionJUMP(lfin)
code.addLabel(lmilieu)
t2 <- CodeGenExpr(e1)
code.addinstructionMV(dr, t2)  # ne pas oublier ceci !
code.addLabel(lfin)
return dr
```

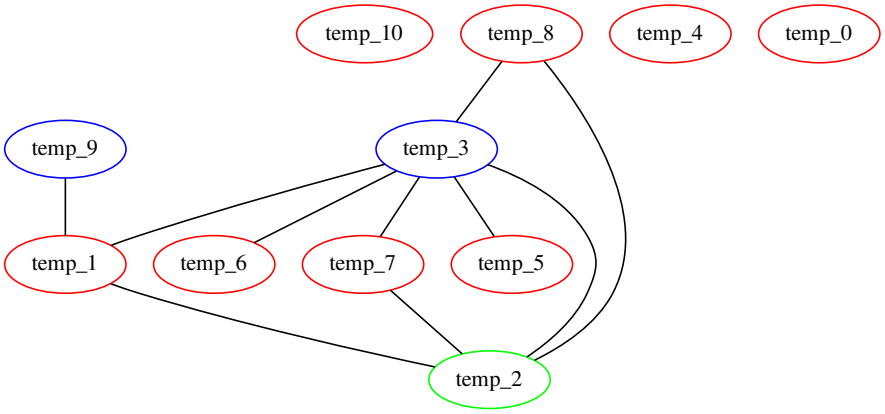
3. Liveness et génération de code

Liveness

code	temp0	temp1	temp2	temp3	temp4	temp5	temp6	temp7	temp8	temp9	temp10
`li temp4 12`											
`mv temp3 temp4`					*						
`li temp5 3`				*							
`add temp6 temp5 temp3`				*		*					
`mv temp2 temp6`				*			*				
`li temp7 4`			*	*							
`add temp8 temp7 temp2`			*	*				*			
`8:mv temp1 temp8`			*	*					*		
`9:sub temp9 temp3 temp2`		*	*	*							
`add temp10 temp9 temp1`		*								*	
`mv temp0 temp10`											*
`...`	*										

Graphe de conflits colorié La pile (fond de pile en premier) :

$$0, 4, 10, 5, 6, 9, 1, 7, 2, 3, 8$$



Allocation On attend une map dans le bon sens :

```
temp0 -> RegistrePhysique(t2), .... temp2 -> OffsetMemoire(-1)
```

Réécriture La ligne 8 devient (et un compilateur un peu malin éliminerait purement et simplement cette ligne ...) :

```
mv t2, t2
```

La ligne 9 devient, par exemple :

```
ld s2, -8(fp)  # chargement du deuxième opérande
sub t3, t3, s2
```

CORRECTED

Compilation de Programmes

CCF Session 1 2020-21 - **NE PAS ENLEVER LES AGRAFES**

Durée totale : 1 heure 30 / Tiers Temps = 2 heures

Toute communication (orale, téléphonique, par messagerie, etc.) avec les autres étudiants est interdite. Aucun document autorisé.

— Pour la partie QCM, plusieurs réponses peuvent être valides à chaque question, on souhaite avoir **toutes** les réponses valides. Chaque question admet au moins une réponse valide et au moins une réponse incorrecte. Il n'y aura pas de point négatif (une mauvaise réponse vaut au pire 0 points).

— Pour les parties rédigées, vous répondrez obligatoirement dans les parties prévues pour. Il ne sera pas donné de nouvelle copie.

— Le barème (sur 20) est indicatif.

Consignes :

- Utiliser un **stylo à bille noir ou bleu**.
- **Noircir ou bleuir** la/les cases, sans dépasser !
- Pour corriger (dernier recours) : effacez proprement la case (ne pas la redessiner).
- Dans les parties rédigées, les carrés gris (prof) sont pour la correction, merci de ne rien écrire dedans.

Numéro étudiant à coder (sans le p, 8 chiffres, il est sur votre carte étudiant !)

- Notez-le ici : Numéro d'étudiant : .....
- Encodez-le ci-contre (chiffre des unités tout à droite, en remplaçant p par 1) : par exemple, pour un numéro $p1234567$, ie 11234567 vous grisez le 1 de la première colonne, le 1 de la deuxième, le 2 de la troisième...).

☐0	☐0	☐0	☐0	☐0	☐0	☐0	☐0
☐1	☐1	☐1	☐1	☐1	☐1	☐1	☐1
☐2	☐2	☐2	☐2	☐2	☐2	☐2	☐2
☐3	☐3	☐3	☐3	☐3	☐3	☐3	☐3
☐4	☐4	☐4	☐4	☐4	☐4	☐4	☐4
☐5	☐5	☐5	☐5	☐5	☐5	☐5	☐5
☐6	☐6	☐6	☐6	☐6	☐6	☐6	☐6
☐7	☐7	☐7	☐7	☐7	☐7	☐7	☐7
☐8	☐8	☐8	☐8	☐8	☐8	☐8	☐8
☐9	☐9	☐9	☐9	☐9	☐9	☐9	☐9

1 Questions de cours/TP

Question 1 ♣ (1 point) Parmi les étapes suivantes du cycle de vie du logiciel, quelles sont celles qui font partie du *front-end* d'un compilateur ?

- ☐ Le coloriage du graphe de conflits.
- ■ Le typage
- ■ L'analyse lexicale
- ■ L'analyse syntaxique
- ☐ La génération de code

Question 2 ♣ (1 point) Dans notre compilateur, quelles parties faudrait-il modifier si l'on voulait changer de machine cible ?

- ☐ Le typeur
- ■ L'allocation de registre
- ☐ L'algorithme de coloriage du graphe de conflits
- ■ La génération de code
- ☐ L'analyse syntaxique

CORRECTED

Question 3 ♣ (1 point) Dans notre compilateur, quelles parties utilisent le patron visiteur ?

- ☐ Le coloriage du graphe de conflits.
- ☐ L'analyse syntaxique
- ■ La génération de code
- ■ Le typage
- ☐ L'allocation de registre

Question 4 ♣ (1 point) Dans un compilateur, sont NP-complets les problèmes suivants :

- ■ Le coloriage du graphe de conflits avec un nombre minimal de couleurs.
- ☐ Le calcul des intervalles de durée de vie des variables temporaires.
- ■ L'ordonnancement d'instructions à l'intérieur d'un bloc de base.
- ☐ La génération de code.
- ☐ L'analyse lexicale

4

Toutes les solutions données ne sont que des éléments de correction qu'il faudrait rédiger mieux.

2 MiniC : Génération de code 3 adresses

Voici une partie d'un programme MiniC.

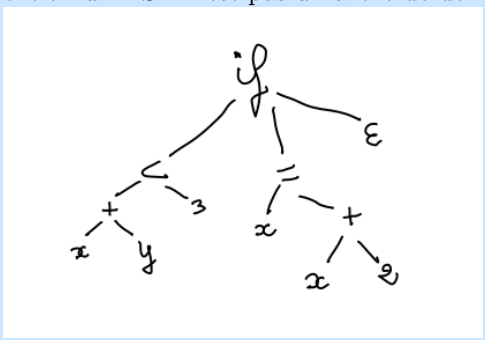
```
if (x + y < 3) {
   x = x + 2; }
```

Question 5 (1.5 points)
Donner l'arbre de syntaxe abstrait (AST) pour cette portion de programme. *On rappelle que chaque noeud de l'AST correspond à l'application d'une règle de grammaire de la grammaire abstraite.*

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Prof*

CORRECTED

Attention un AST n'est pas un arbre de dérivation ni un arbre de typage



Question 6 **(2 points)** En utilisant les règles de génération de code fournies, et en considérant la mémoire $y \mapsto temp_0, x \mapsto temp_1$, générer le code 3 adresses RISCV correspondant à ce code [a]. En commentaire du code généré, mettez les appels aux fonctions de génération de code utilisées. Il n'est pas nécessaire de dérouler tous les appels récursifs des fonctions de génération de code comme en TD.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Prof*

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

a. On rappelle que le code trois adresses a le même jeu d'instruction que la machine RISCV du cours, à part pour les conditions qui utilisent le raccourci `condJUMP(label,t1,condition,t2)`. De plus il utilise les registres temporaires au lieu des registres physiques.

CORRECTED

Notre compilateur prof fourni le code suivant, auquel il manque quelques commentaires pour obtenir tous les points de la question

```
        # (stat (if_stat if ( (expr (expr (expr (atom x)) + (expr (atom y))) < (expr (atom 3)))
(stat_block { (block (stat (assignment x = (expr (expr (atom x)) + (expr (atom 2)))) ;)) }))
        add temp_2, temp_1, temp_0
        li temp_3, 3
        li temp_4, 0
        bge temp_2, temp_3, lbl_end_relational_3_main
        li temp_4, 1
lbl_end_relational_3_main:
        beq temp_4, zero, lbl_else_2_main
        # (stat (assignment x = (expr (expr (atom x)) + (expr (atom 2)))) ;)
        li temp_5, 2
        add temp_6, temp_1, temp_5
        mv temp_1, temp_6
lbl_else_2_main:
lbl_end_if_1_main:
```

3 MiniC : une petite extension

On considère une nouvelle construction d'expressions : `ifzero(`e`) then` e_1 `else` e_2
dont l'évaluation vaut la valeur de l'expression e_1 si l'expression e s'évalue à 0, et e_2 sinon.

trop de confusion expression instruction.

On augmente donc la syntaxe abstraite des **expressions numériques et booléennes** :

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
 & | & x & \textit{variable} \\
 & | & e + e & \textit{addition} \\
 & | & e \times e & \textit{multiplication} \\
 & | & ifz(e,e,e) & \textit{ifzero (NEW)}
\end{array}
$$

Question 7 **(1.5 points)** En s'inspirant des règles de typage fournies dans la feuille d'accompagnement, écrire une règle de typage pour ce nouveau constructeur d'expression.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Prof*

.....

.....

.....

.....

.....

.....

CORRECTED

L'expression testée à doit être de type int, e1,e2 doivent être du même type t et l'expression est alors bien typée et aussi du type t (int/bool) (regle laissée au lecteur)

Question 8 (2 points) En utilisant votre règle de typage précédente et celles de la feuille d'accompagnement, montrer que l'affectation $y = \texttt{ifzero}(x - 8)$ `then` 18 `else` $42 + x$ est bien typée sous l'environnement $\Gamma : x \mapsto int, y \mapsto int$. *On fera attention à bien faire un arbre de preuve correct.*

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Prof*

Attention à bien typer l'affectation complète. Pour $x - 8$ utiliser la règle de typage de la soustraction. Attention les constantes ne sont pas dans l'environnement de typage Γ.

CORRECTED

Question 9 **(2 points)** Sur le modèle de la règle de génération de code des expressions, écrire une règle de génération de code pour ce nouveau constructeur d'expression. *On justifiera soigneusement la réponse !*

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Prof*

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Comme il s'agit d'une expression, la règle doit retourner un temporaire dans lequel l'expression est calculée. Attention à ne pas retourner ce temporaire trop tot sinon le code restant n'est pas généré.

```
   GenCodeExpr (ifz(e0,e1,e2) =
dr <- nouveau_reg
    dr1 <- genCodeExpr(e_0)
        label_else, label_fin = genlabels()
addInstructioncondjump(dr1,"!=", "0",label_else)
        dr2 <- genCodeExpr(e_1)
addInstructionMV(dr1,dr2)
addInstructionJUMP(label_fin)
addLabel(label_else)
dr3 <- genCodeExpr(e_2)
addInstructionMV(dr1,dr3) # attention dr3 doit être diférent de dr2.
 addLabel(label_fin)
 return dr1
```


4 Variables vivantes (liveness) et génération de code

On considère le programme suivant :

CORRECTED

```
int x, y, z, t;
x=60;
y=x+5;
z=42*y;
t=x+y+z;

println_int(x);
println_int(t);
// tout le monde est mort ici.
```

Pour lequel notre compilateur des TP4 et TP5 produit le code suivant, avec :

$$(t, z, y, x) \mapsto (temp0, temp1, temp2, temp3)$$

```
1   li temp_0, 0
    li temp_1, 0
    li temp_2, 0
    li temp_3, 0
    li temp_4, 60
6   6:  mv temp_3, temp_4
    7:  li temp_5, 5
    8:  add temp_6, temp_3, temp_5
    9:  mv temp_2, temp_6
    10: li temp_7, 42
11  11: mul temp_8, temp_7, temp_2
    12  mv temp_1, temp_8
    13: add temp_9, temp_3, temp_2
    14: add temp_10, temp_9, temp_1
    15: mv temp_0, temp_10
16  # Ensuite le code pour les impressions
    16:
```

Question 10 **(1 point)** Expliquer rapidement à quoi servent les quatre premières lignes du programme généré. ☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Prof*

.....

.....

.....

.....

.....

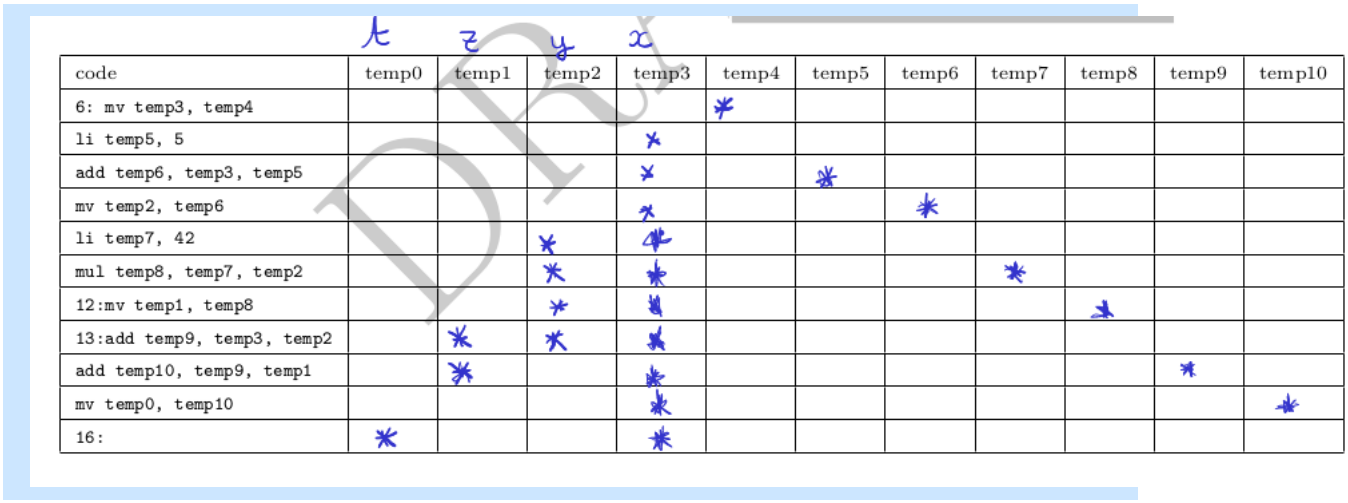
Les 4 premières lignes, générées par le visiteur de la déclaration des variables du programme, servent à initialiser les variables déclarées selon la valeur 0. Cela permet de "déterminiser" le programme lorsque le programmer a oublié d'initialiser ses variables et les utilise quand même (toutes les exécutions se passeront alors de la même façon).

CORRECTED

Question 11 **(2 points)** Remplir le tableau suivant avec les résultats de l'analyse de variables vivantes en commençant à la ligne 6. Une étoile dans une ligne veut dire "ce temporaire est vivant à **l'entrée** de cette ligne". Attention au traitement des `print` pour remplir la ligne 16.

..... ☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Prof*

code	temp0	temp1	temp2	temp3	temp4	temp5	temp6	temp7	temp8	temp9	temp10
`6: mv temp3, temp4`											
`li temp5, 5`											
`add temp6, temp3, temp5`											
`mv temp2, temp6`											
`li temp7, 42`											
`mul temp8, temp7, temp2`											
`12:mv temp1, temp8`											
`13:add temp9, temp3, temp2`											
`add temp10, temp9, temp1`											
`mv temp0, temp10`											
`16:`											



code	temp0	temp1	temp2	temp3	temp4	temp5	temp6	temp7	temp8	temp9	temp10
`6: mv temp3, temp4`					*						
`li temp5, 5`				*							
`add temp6, temp3, temp5`				*		*					
`mv temp2, temp6`				*			*				
`li temp7, 42`			*	*							
`mul temp8, temp7, temp2`			*	*				*			
`12:mv temp1, temp8`			*	*					*		
`13:add temp9, temp3, temp2`		*	*	*							
`add temp10, temp9, temp1`		*		*						*	
`mv temp0, temp10`				*							*
`16:`	*			*							

CORRECTED

Question 12 **(2 points)** Dessiner le graphe d'interférence. *Laisser de la place pour pouvoir faire la question d'après dans le cadre.*

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Prof*

Voir le graphe à la question suivante.

Dans le reste de l'exercice on utilise les notations suivantes :
— La couleur 1 est ROUGE et est associée au registre physique s4.
— La couleur 2 est BLEU et est associée au registre physique s5.
— La couleur 3 est VERT et est associée à un *spill* en mémoire avec un offset -1 (ie un décalage de -8 octets par rapport au registre `fp` qui contient l'adresse de la pile).
— Les couleurs suivantes sont NOIR, JAUNE, VIOLET, et les offset respectifs -2,-3,-4...

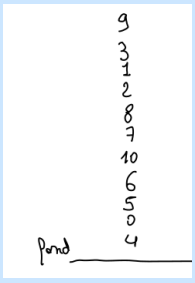
On rappelle que l'algorithme de coloration du cours empile les sommets de plus bas degré en premier, que les degrés sont mis à jour à chaque fois que l'on empile, et que si il y a un choix c'est le temporaire de numéro le plus faible qui est empilé en premier. Enfin le coloriage se passe dans le sens inverse de la création de la pile de coloriage.

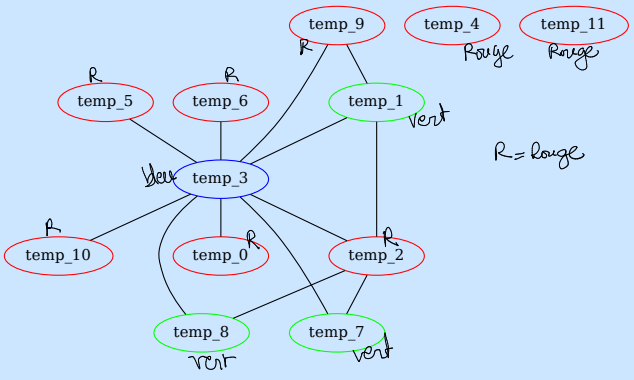
CORRECTED

Question 13 **(2 points)** Colorier dans le cadre précédent le graphe avec l'algorithme du cours et un nombre infini de couleurs (3 devraient suffire !). *On écrira aussi la pile de coloriage en montrant clairement le haut de pile.* Donner l'allocation déduite dans le cadre ci dessous.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Prof*

On trouve la pile suivante et le coloriage suivant :

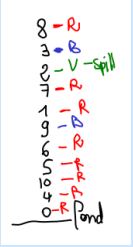




L'allocation est alors du style *temporaire* → *lieu* (registre physique + nom ou alors place mémoire avec offset fixé).

La pile associé au remplissage incorrect de temp3 est celle-ci, avec le coloriage associé (compté juste même si la liveness était fausse) :

CORRECTED



Question 14 (1.5 points) Générer les (vraies) instructions RISCV pour les instructions des lignes 11 et 12 *On utilisera* s_1, s_2, s_3, fp *pour la gestion de la pile, comme en TP.*

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Prof*

Donc, les registres temporaires $temp_1, temp_7, temp_8$ sont spillés et $temp_2$ est alloué dans le registre s_4.

L'instruction 3 adresses `mul temp8 temp7 temp2` devient donc

```
ld s1 -8(fp) #chargement de l'opérande temp7
mul s2 s1 s4
sd s2 -8(fp) #stockage mémoire de la valeur nouvellement calculés.
```

CORRECTED

Feuille supplémentaire de rédaction à utiliser seulement en cas de drame

Prof

8.5 Total : 21.5