Examen Session 2
Traduction et Compilation de Programmes
Février 2018
Durée 2H

Aucun document autorisé

Instructions **à lire attentivement !**

1. Les exercices sont indépendants.
2. On vous donne des indications de temps.
3. Merci de justifier vos réponses.
4. Vous répondrez dans les cadres et rendrez l'examen entier **agraphé**.
5. Des étiquettes assureront l'anonymat, le même numéro sera collé dans le cadre ci dessous et sur la feuille d'émargement.
6. Vous pouvez pour plus d'efficacité enlever la dernière feuille "compagnon" et ne pas la remettre.

Numéro d'anonymat :

Exercice 1 – Grammaires et Attributions (20min)

On considère la grammaire (.g4) suivante :

```
grammar Tree;

tree:  INT                       #feuille
    |  NODE '(' INT tree+ ')'    #arbre
    ;

INT: [0-9]+;
NODE: 'node';
```

Cette grammaire permet de représenter des arbres n-aires, par exemple `node (42 12 1515 17)` représente l'arbre de racine 42 avec 3 fils $12, 1515, 17$.

Question #1

Écrire une attribution qui permet de décider si un arbre reconnu par la grammaire est binaire (une feuille est un arbre binaire, un noeud est un arbre binaire ssi il a deux 2 fils qui sont des arbres binaires). On vous fournit le code à remplir.

```
#MyTreeVisitor.py
from TreeVisitor import TreeVisitor
from TreeParser import *

class MyTreeVisitor(TreeVisitor):

    def visitFeuille(self, ctx):
#TODO!




    def visitArbre(self, ctx):
        children=ctx.tree()
#TODO! (children is the list of subtrees)









-----------------------------------
#Main.py
    tree=parser.tree();
    visitor = MyTreeVisitor();
#TODO : call the method and print the result!




-
```

Exercice 2 – Génération de code 3-adresses (30min)

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, ...).

L'annexe habituelle vous donne les règles de typage et de génération de code que vous devez utiliser pour l'exercice.

Question #1

Montrer que le programme suivant :

```
i:=42;
while(i<=77)
  i := i - 1;
```

est bien typé sous l'hypothèse $\Gamma : [i \mapsto int]$.

....

....

....

....

....

....

....

....

....

Question #2

Remplir le code **3 adresses généré par les règles** pour le programme.

```
1   ;;Automatically generated LEIA code, MIF08 2017 3 adresses. i -> temp0
            ;; (stat (assignment i = (expr (atom 42)) ;))




6
            ;; (stat (while_stat while (expr (atom ( (expr (expr (atom i)) <= (expr (atom 77))) ))) (
        stat_block (stat (assignment i = (expr (expr (atom i)) - (expr (atom 1))) ;)))))
    lbl_l_while_begin_0:
    ;;;   comparison i <= 77 (4/6 lignes)


11
```

```
16




21
lbl_end_relational_1:
        ;; Le resultat de la comparaison est dans temp_3!
        ;; todo : saut conditionnel et corps de la boucle (5/6lignes)

26




31




36
        JUMP lbl_l_while_begin_0
lbl_l_while_end_0: ;;; FIN
```

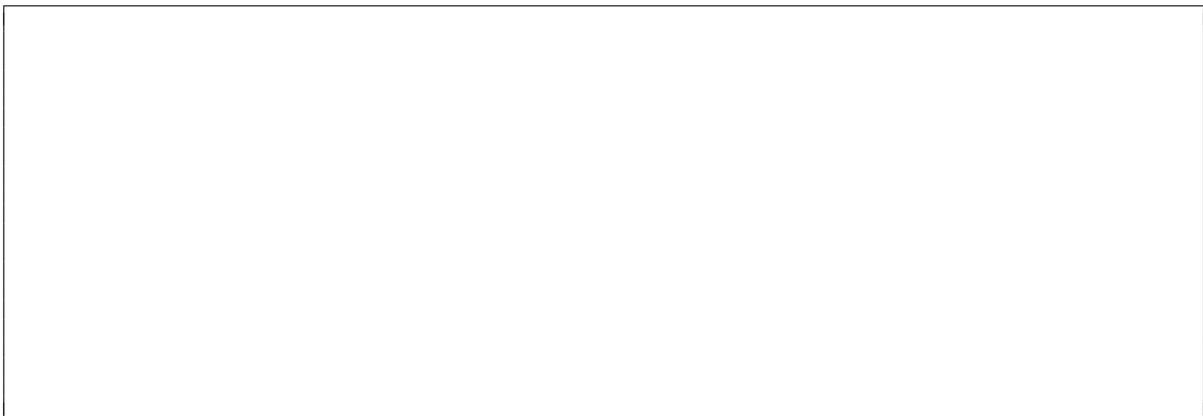
On rajoute maintenant la construction “repeat S until b” dont la sémantique informelle est :

1. Exécuter une première fois le corps S ;
2. Calculer la valeur de l'expression booléenne b ;
3. Si cette valeur est `faux`, terminer, sinon, recommencer à 1.

Question #3

Dessiner le graphe de flot du programme suivant (avec une instruction par bloc) :

```
i:=42;
repeat
  i := i - 1;
until(i< 77);
```



Question #4

Quel serait le code (3 adresses) généré pour le programme de la question précédente. *Pour aller plus*

vite, vous pouvez numéroter des blocs dans le programme précédent et uniquement reporter ces numéros.

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

Question #5

Écrire **et JUSTIFIER** la règle de génération de code pour le `repeat... until` :

. . . .

. . . .

. . . .

. . . .

(Stm)repeat S until b	

Exercice 3 – Dataflow (30 min)

Adapté d'un exercice dont j'ai perdu la référence

Considérons le programme suivant :

```
1  a:=0;
2  b:=1;
3  z:=0;
4  n:=12;
5  while ( n != z )
6    t:=a+b;
7    a:=b;
8    b:=t;
9    n:=n-1;
10   z:=0;
11 }
```

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 #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. Remplir les deux premières colonnes du tableau avec les numéros des blocs et les prédécesseurs dans le graphe.

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.*

				Step		Step		Step		Step	
$Pred(\ell)$	ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

	Step		Step		Step		Step		Step	
ℓ	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

Question #3

La ligne 10 est-elle vivante ? et la ligne 3 ? Quelles informations obtenues à la fin de l'analyse de la question précédente vous permettent de conclure ou non ?

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

Exercice 4 – Allocation de registre et génération de code machine (40 min)

D'après C. Paulin (LRI)

Soit le code 3 adresses suivant, où x et n sont supposées vivantes en entrée du code :

```
   y:=1
   m:=x
   e:=n
 4 loop:
   condjump (e , ``='',   0,    fin)   ; test e==0
   t:=y mod 2 ; premiere utilisation de y
   condjump (t, ``='',  0,   pair)
   y:= m*y    ; deuxieme utilisation de y
 9 pair:
   m:=m * m
   e:=e div 2
   jump loop
   fin:
14 log(y)       ; troisieme utilisation de y
```

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.*

....

....

....

....

Question #2

Remplir le tableau avec les durées de vie des variables *en entrée des lignes.*, puis tracer le graphe d'interférence.

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

Question #3

Colorier le graphe avec l'algorithme du cours et 4 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.*

Question #4

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

. . . .

. . . .

. . . .

. . . .

On se propose de stocker la variable y en mémoire, à l'adresse pointée par R_6, et d'utiliser R_1 et R_2 pour réaliser les calculs intermédiaires.

Question #5

Par quelles instructions est remplacée la ligne 1 dans le code finalement généré ?

. . . .

. . . .

. . . .

. . . .

Question #6

Même question pour la ligne 6. **EXPLIQUER**

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

Question #7

Même question pour la ligne 8.

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

Question #8

Même question pour la ligne 14.

. . . .

. . . .

. . . .

. . . .

MIF08 Feuille d'accompagnement

Mini-while (syntaxe abstraite)

Instructions :

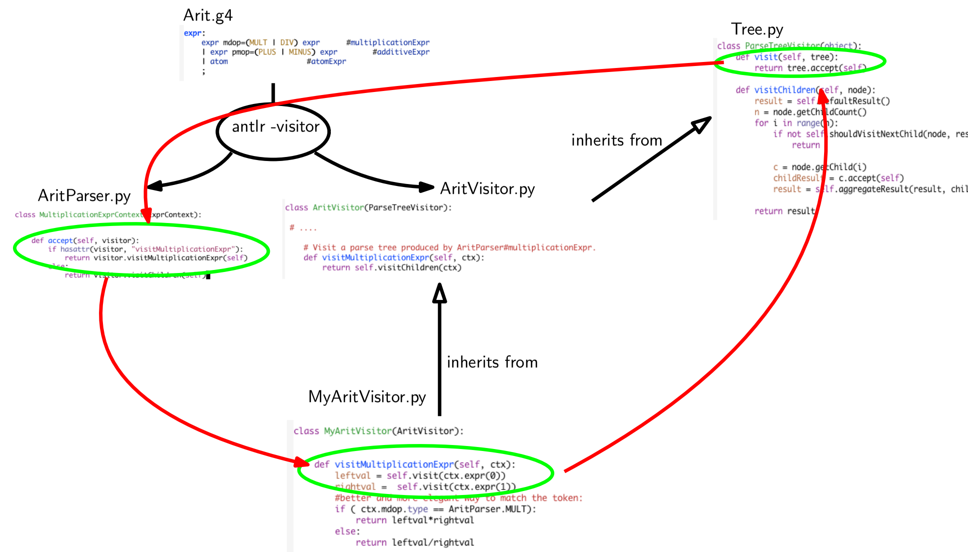
$$
\begin{array}{lcll}
S(Smt) & ::= & x := e & \text{affectation} \\
 & | & skip & \text{rien} \\
 & | & S_1 ; S_2 & \text{séquence} \\
 & | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \text{test} \\
 & | & \texttt{while } b \texttt{ do } S \texttt{ done} & \text{boucle}
\end{array}
$$

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

<table>
<tr><td>c</td><td><pre>dr <-newTemp()
code.add(InstructionLETL(dr, c))
return dr</pre></td></tr>
<tr><td>x</td><td><pre># récupère le temporaire associé à x
regval<-getTemp(x)
return regval</pre></td></tr>
<tr><td>e_1+e_2</td><td><pre> t1 <- GenCodeExpr(e_1)
  t2 <- GenCodeExpr(e_2)
  dr <- newTemp()
  code.add(InstructionADD(dr, t1, t2))
  return dr</pre></td></tr>
<tr><td>e_1-e_2</td><td><pre> t1 <- GenCodeExpr(e_1)
  t2 <- GenCodeExpr(e_2)
  dr <- newTemp()
  code.add(InstructionSUB(dr, t1, t2))
  return dr</pre></td></tr>
<tr><td>true</td><td><pre>dr <-newTemp()
code.add(InstructionLETL(dr, 1))
return dr</pre></td></tr>
<tr><td>$e_1 < e_2$</td><td><pre>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</pre></td></tr>
</table>

<table>
<tr><td>x := e</td><td><pre> dr <- GenCodeExpr(e)
copie du résultat dans le tmp associé à x
  let loc = loc(x) in
     code.add(instructionCOPY(loc,dr))</pre></td></tr>
<tr><td>S1 ; S2</td><td><pre>#concatène les codes générés
   GenCodeSmt(S1)
   GenCodeSmt(S2)</pre></td></tr>
<tr><td>if b then $S1$ else $S2$</td><td><pre>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)</pre></td></tr>
<tr><td>while b do S done</td><td><pre>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</pre></td></tr>
</table>