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.

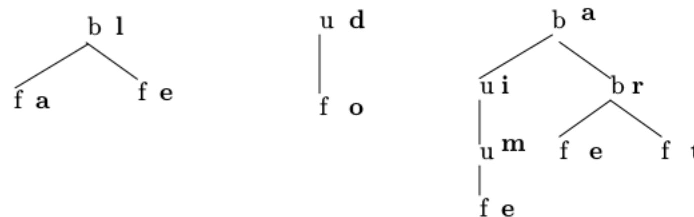


Figure 1 – Exemples d'arbres décorés

Question #1

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

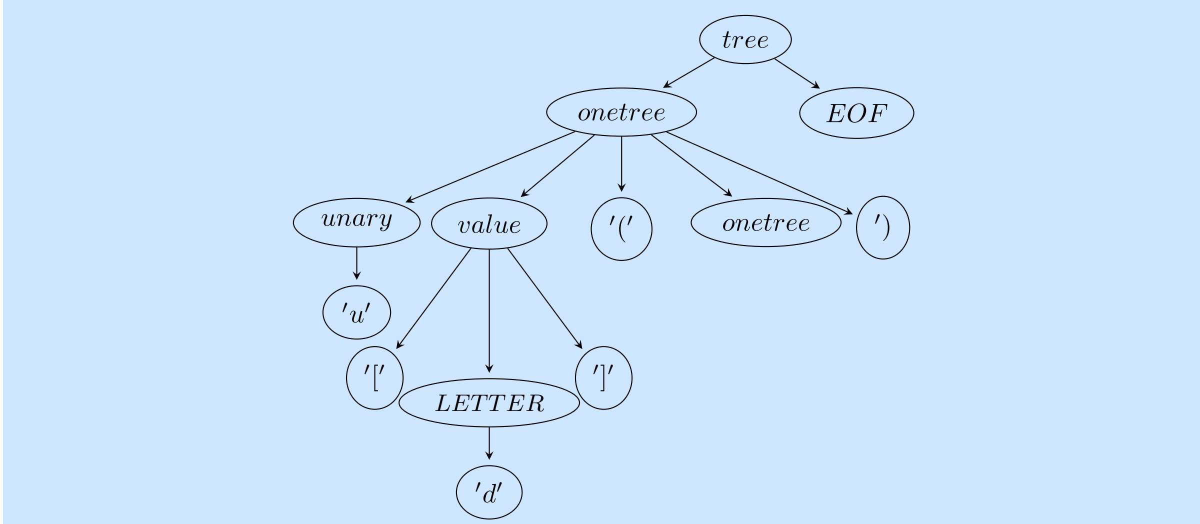
Solution:

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

Question #2

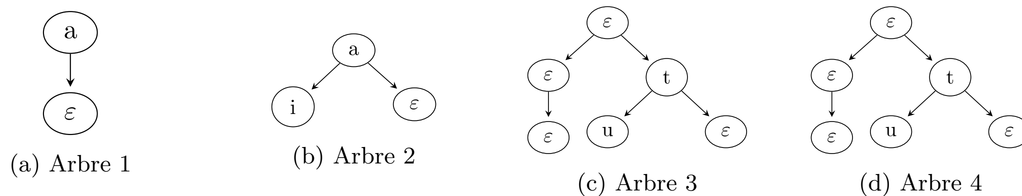
Quelle est l'arbre de dérivation pour la chaîne `u[d](f[o])` ? *On rappelle qu'un arbre de dérivation n'est pas un AST, chaque noeud interne dénote une application de règle. Les feuilles sont des terminaux.*

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 :

$S(Smt)$	::=	$x := e$	*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 :

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

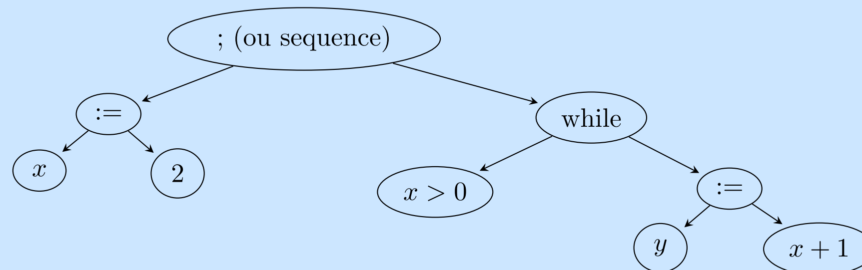
Oui, il y avait une typo ici, ' :=' et '=' dénotent tous deux l'affection !

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)) ;))
4       .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_l_while_begin_0:
        .let temp_2 0
9       .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
```

```
14          ;; 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
19          ;; 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
24          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 :

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

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
```

```
5         ;; 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
10        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
15  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 :

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

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

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:

```
1  ;;Automatically generated LEIA code, MIF08 2017
   ;;all−in−memory allocation version
   ;; stack management
   .set r6 stack
           ;; (stat (assignment x = (expr (atom 12)) ;))
6          ;; .let temp_4 12
           .LET r1 12
           SUB r0 r6 3
           WMEM r1 [r0]
           ;; end .let temp_4 12
11         ;; copy temp_3 temp_4
           SUB r0 r6 3
           RMEM r1 [r0]
           COPY r1 r1
           SUB r0 r6 2
16         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").

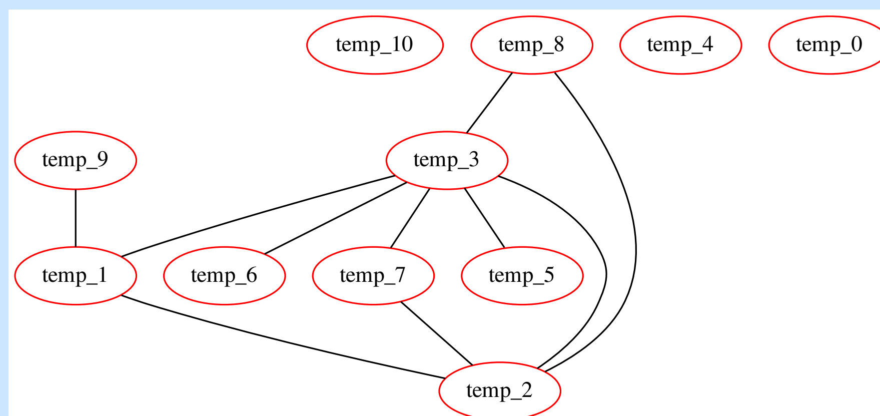
Solution:

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:



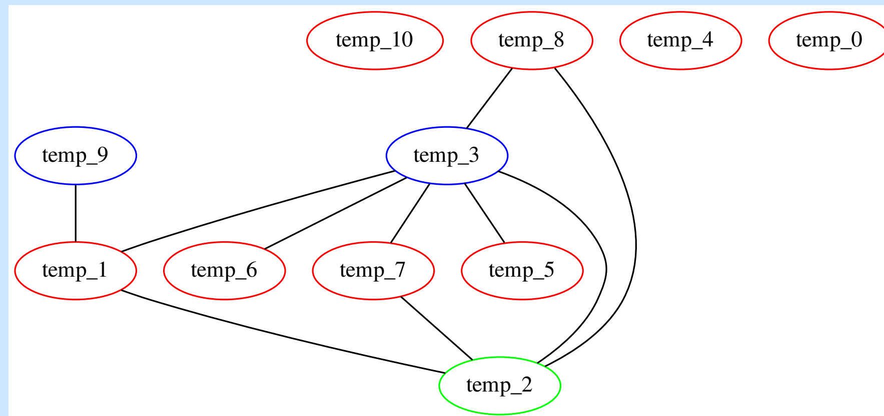
Dans la suite on utilisera les conventions suivantes : — Couleur 1 = rouge, associée au registre R3 ; — Couleur 2 = bleu, associée au registre R4.

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))) ;))
```

```
13          ;;  .let  temp_5 3
            .LET r3 3
            ;;  end  .let  temp_5 3
            ;;  add temp_6 temp_5 temp_3
            ADD r3 r3 r4
18          ;;  end add temp_6 temp_5 temp_3
            ;;  copy temp_2 temp_6
            COPY r1 r3
            SUB r0 r6 0
            WMEM r1 [r0]
23          ;;  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
28          ;;  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
33          ;;  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
38          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
43          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
48


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

MIF08 Feuille d'accompagnement

Mini-while (syntaxe abstraite)

Instructions :

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

Grammaires ANTLR et Visiteurs

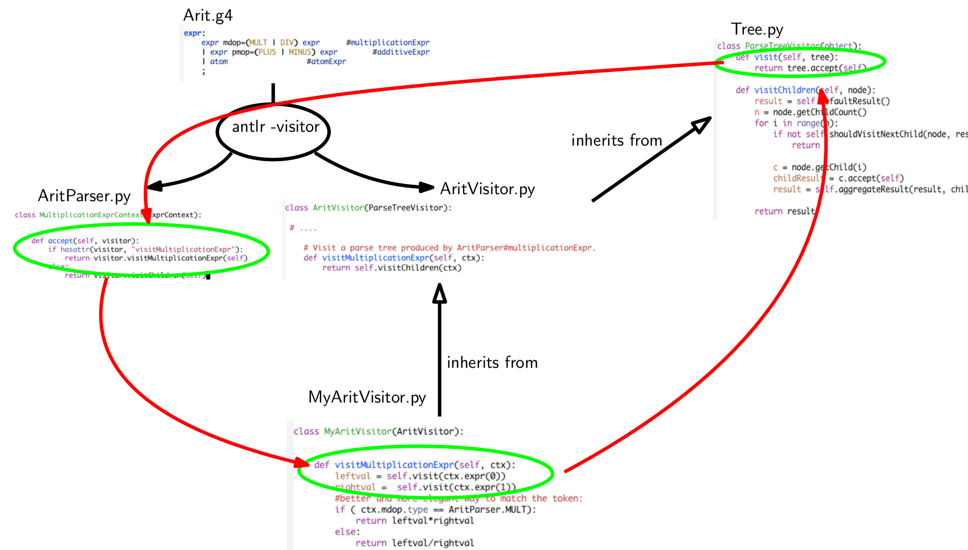
Syntaxe Python-ANTLR

— 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