Examen
Traduction et Compilation de Programmes
4 janvier 2017
Durée 1H30

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 :

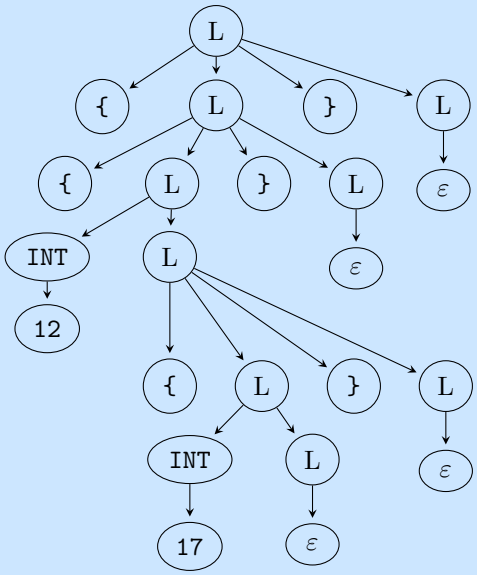
- L est le symbole de début (*start-symbol*)
- **INT** est un élément renvoyé par l'analyse lexicale (un *token*) quand elle détecte un entier.
- les accolades '{' et '}' sont aussi des *token*.

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.**

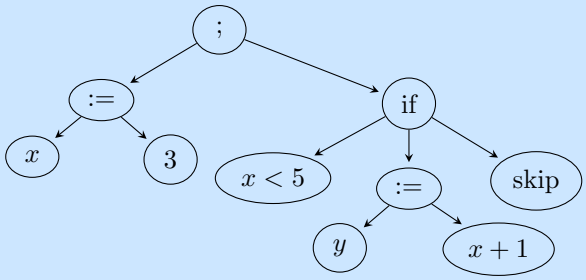
Question #1

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.*

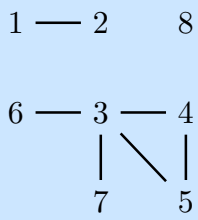
Solution:
A VERIFIER

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 :

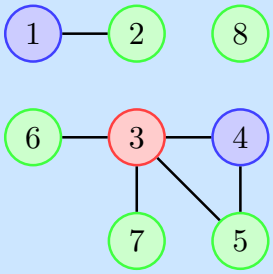


Question #4

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 :

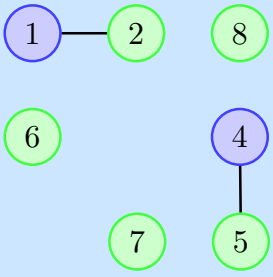


Question #5

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