Faculté des Sciences
Aix*Marseille Université

Année universitaire 2018/2019

Site : ☒ Luminy ☐ St-Charles ☐ St-Jérôme ☐ Cht-Gombert ☒ Aix-Montperrin ☐ Aubagne-SATIS
Sujet de : ☐ 1er semestre ☒ 2ème semestre ☐ Session 2 Durée de l'épreuve : 2h
Examen de : L3 Nom du diplôme : Licence d'Informatique
Code du module : ENSIN6U1 Libellé du module : Compilation
Calculatrices autorisées : NON Documents autorisés : NON

---

1 Grammaires attribuées (10pt)

Soit la grammaire G, dont les arbres de dérivation représentent des arbres abstraits de L :

1. $P \rightarrow LD_1\ LD_2$

2. $LD \rightarrow D\ LD_1$
3. $LD \rightarrow$ `null`

4. $D \rightarrow$ `fct id` $LD_1\ LD_2\ LI$
5. $D \rightarrow$ `var id`
6. $D \rightarrow$ `var id taille`

7. $V \rightarrow$ `simple id`
8. $V \rightarrow$ `indicee id` E

9. $LI \rightarrow I\ LI_1$
10. $LI \rightarrow$ `null`

11. $I \rightarrow$ `aff` $V\ E$
12. $I \rightarrow$ `si` $E\ LI_1\ LI_2$
13. $I \rightarrow$ `tq` $E\ LI$
14. $I \rightarrow$ `app` APP
15. $I \rightarrow$ `ret` E
16. $I \rightarrow$ `ecr` E

17. $APP \rightarrow$ `id` LE
18. $LE \rightarrow E\ LE_1$
19. $LE \rightarrow$ `null`

20. $E \rightarrow$ `op2` $E_1\ E_2$
21. $E \rightarrow$ `op1` E_1
22. $E \rightarrow V$
23. $E \rightarrow$ `entier`
24. $E \rightarrow APP$
25. $E \rightarrow$ `lire`

Dans cette grammaire, les terminaux sont représentés par des symboles composés de lettres `minuscules` et les non terminaux par des symboles composés de lettres $MAJUSCULES$. Quand des non terminaux apparaissent plusieurs fois dans une règle, ils sont indicés. Par exemple, $LD_1\ LD_2$ contient deux instances du même non terminal LD.
Les terminaux `id`, `entier`, `op1` et `op2` possèdent des valeurs auxquelles on peut avoir accès par l'intermédiaire de l'attribut `val`. Ainsi `id.val` est la valeur d'un identifiant particulier.
Dans chacune des questions suivantes on cherche à répondre à une question concernant un arbre abstrait, qui est aussi un arbre de dérivation généré par G. Pour chacune, il vous est demandé de définir un ou plusieurs attributs ainsi que les actions sémantiques associées aux règles de G permettant de calculer la valeur des attributs.
Pour chaque attribut défini, indiquer : sont type, ce qu'il représente et s'il est hérité ou synthétisé. Pour chaque question, vous pourrez utiliser les attributs définis dans les questions précédentes.
Il est inutile d'écrire les règles qui ne sont pas concernées par les attributs de la question.

Q1.1 Combien d'instructions comporte un programme ?
(2pt)

1. $P \rightarrow LD_1\ LD_2$	$P.n = LD_2.n$
2. $LD \rightarrow D\ LD_1$	$LD.n = D.n + LD_1.n$
3. $LD \rightarrow$ `null`	$LD.n = 0$
4. $D \rightarrow$ `fct id` $LD_1\ LD_2\ LI$	$D.n = LI.n$
9. $LI \rightarrow I\ LI_1$	$LI.n = I.n + LI_1.n$
10. $LI \rightarrow$ `null`	$LI.n = 0$
11. $I \rightarrow$ `aff` $V\ E$	$I.n = 1$
12. $I \rightarrow$ `si` $E\ LI_1\ LI_2$	$I.n = LI_1.n + LI_2.n + 1$
13. $I \rightarrow$ `tq` $E\ LI$	$I.n = LI.n + 1$
14. $I \rightarrow$ `app` APP	$I.n = 1$
15. $I \rightarrow$ `ret` E	$I.n = 1$
16. $I \rightarrow$ `ecr` E	$I.n = 1$

Q1.2 Quelle est la longueur de la fonction la plus longue [1] d'un programme ?

(1pt)

1. $P \rightarrow LD_1\ LD_2$	$P.m = LD_2.m$
2. $LD \rightarrow D\ LD_1$	$LD.m = \max(D.n, LD_1.m)$
3. $LD \rightarrow$ `null`	$LD.m = 0$

Q1.3 Quelle est la portée de chaque variable (globale, locale, argument) d'un programme ?

(2pt)

1. $P \rightarrow LD_1\ LD_2$	$LD_1.portee = global$
2. $LD \rightarrow D\ LD_1$	$D.portee = LD_1.portee = LD.portee$
4. $D \rightarrow$ `fct id` $LD_1\ LD_2\ LI$	$LD_1.portee = locale \qquad LD_2.portee = arg$

Q1.4 Pour chacune des variables locales et des paramètres de fonction, dire si elle masque une variable globale.

1. $P \rightarrow LD_1\ LD_2$	$LD_2.vg = LD_1.vg$
2. $LD \rightarrow D\ LD_1$	$si(LD.portee = global)alorsLD.vg = D.vg \cup LD_1.vg$
3. $LD \rightarrow$ `null`	$LD.vg = \{\}$
(2pt) 4. $D \rightarrow$ `fct id` $LD_1\ LD_2\ LI$	$LD_1.vg = D.vg\ LD_2.vg = D.vg$
5. $D \rightarrow$ `var id`	$si(D.portee == global)alorsD.vg = \{id.val\}$
. $\rightarrow$	$sinonD.masque = id.val \in D.vg$
6. $D \rightarrow$ `var id taille`	$D.vg = \{id.val\}\ D.masque = faux$

Q1.5 Une fonction est dite fermée, si toute exécution possible de la fonction se termine par une instruction `retour`. Pour chacune de trois fonctions suivantes, dire si elle est fermée ou pas.

```
f() {si a alors {retour 1;}}
g() {si a alors {retour 1;} retour 0;}
h() {si a alors {retour 1;} sinon {retour 0;}}
```

(1pt)

`f` non fermée, `g` fermée, `h` fermée

Q1.6 Définir un attribut et les actions sémantiques associées pour déterminer si chacune des fonctions du programme est fermée ?

4. $D \rightarrow$ `fct id` $LD_1\ LD_2\ LI$	$D.ferme = LI.ferme$
9. $LI \rightarrow I\ LI_1$	$LI.ferme = I.ferme \vee LI_1.ferme$
10. $LI \rightarrow$ `null`	$LI.ferme = faux$
11. $I \rightarrow$ `aff` $V\ E$	$I.ferme = faux$
(2pt) 12. $I \rightarrow$ `si` $E\ LI_1\ LI_2$	$I.ferme = LI_1.ferme \wedge LI_2.ferme$
13. $I \rightarrow$ `tq` $E\ LI$	$I.ferme = faux$
14. $I \rightarrow$ `app` APP	$I.ferme = faux$
15. $I \rightarrow$ `ret` E	$I.ferme = vrai$
16. $I \rightarrow$ `ecr` E	$I.ferme = faux$

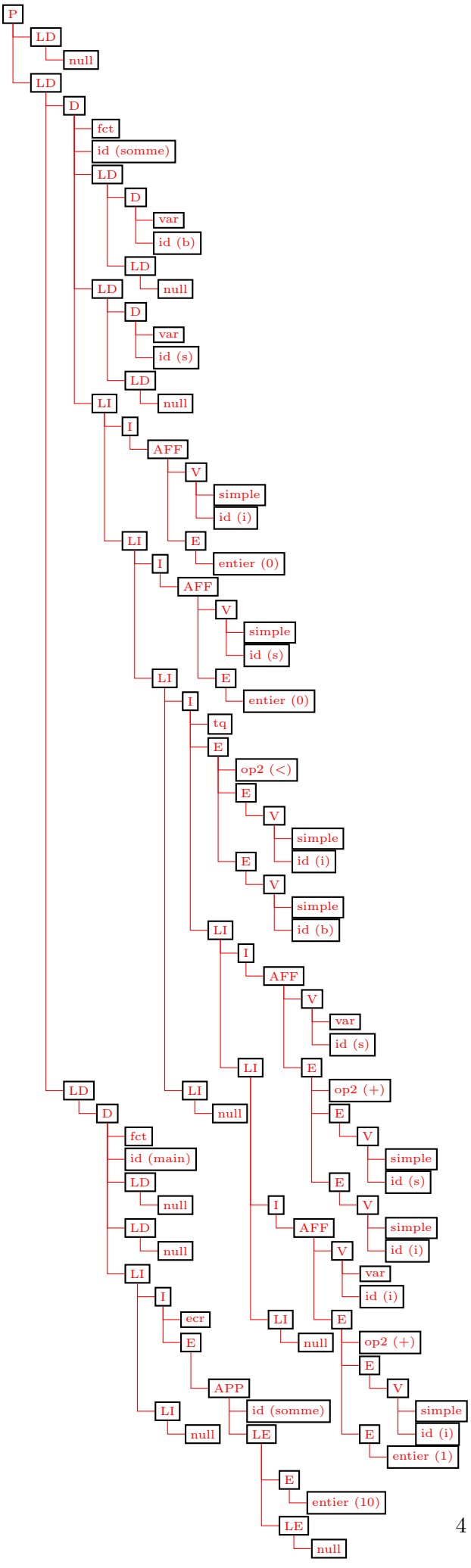
2 Arbres abstraits (2pt)

Dessiner l'arbre abstrait du programme suivant en utilisant la grammaire de la question précédente.

Q2.1
```
somme(entier b) entier s;{i = 0; s = 0; tantque i < b  faire{s = s + i; i = i + 1;}}
main(){ecrire(somme(10));}
```

1. La longueur d'une fonction est le nombre d'instruction qu'elle comporte.

(2pt)


P
LD
null
LD
D
fct
id (somme)
LD
D
var
id (b)
LD
null
LD
null
D
var
id (s)
LD
null
LI
I
AFF
V
simple
id (i)
E
entier (0)
LI
I
AFF
V
simple
id (s)
E
entier (0)
LI
I
tq
E
op2 (<)
E
V
simple
id (i)
E
V
simple
id (b)
LI
I
AFF
V
var
id (s)
E
op2 (+)
E
V
simple
id (s)
E
V
simple
id (i)
LI
null
LD
D
fct
id (main)
LD
null
LD
null
LI
I
ecr
E
APP
id (somme)
LE
E
entier (10)
LE
null
LI
I
AFF
V
var
id (i)
E
op2 (+)
E
V
simple
id (i)
E
entier (1)
LI
null
LI
null
LI
null

3 Code trois adresses (4pt)

Pour chacun des fragments de programmes suivants, donner le code trois adresses lui correspondant.

Q3.1 `( ( a + b ) < ( 3 * c ) ) & ! ( c < 10 )`
(2,5pt)

```
    : t0 = va + vb
    : t1 = 3 * vc
    : t2 = -1
    : if t0 < t1 goto e0
    : t2 = 0
e0  : t3 = -1
    : if vc < 10 goto e1
    : t3 = 0
e1  : t4 = -1
    : if t3 == 0 goto e2
    : t4 = 0
e2  : if t2 == 0 goto e3
    : if t4 == 0 goto e3
    : t5 = -1
    : goto e4
e3  : t5 = 0
e4  : ...
```

Le résultat est dans `t5`

Q3.2 `tantque i < 10  faire { i = i + 1;}`
(1,5pt)

```
e0  : t0 = -1
    : if vi < 10 goto e2
    : t0 = 0
e2  : if t0 == 0 goto e1
    : t1 = vi + 1
    : vi = t1
    : goto e0
e1  : ...
```

4 Assembleur (4pt)

Dans chacun des deux exercices suivants, un court programme en code trois adresses est donné. L'objet de ces exercices est d'écrire le code en assembleur x86 issu de la traduction du code trois adresses [2], ainsi que, l'information contenue dans chacun des quatre registres `eax`, `ebx`, `ecx` et `edx` à tout moment de la génération. Pour cela, on présentera la solution sous la forme d'un tableau contenant pour chaque instruction en code trois adresses l'instruction ou les instructions en assembleur lui correspondant. Ce tableau contient aussi quatre colonnes représentant le descripteur de chacun des quatre registres, comme dans l'exemple suivant :

2. Il est inutile d'écrire le code assembleur qui précède l'étiquette `fmain`, à l'exception de la déclaration des variables globales.

C3A	a	b	c	d	X86
t7 = 45 + 12	t7 t7				mov eax, 45 add eax, 12
t8 = 3 * t7	t7 t7	t8 t8			mov ebx, 3 imul ebx, eax
...					...

Q 4.1

```
fmain:  fbegin
        alloc 1 va
        alloc 1 vb
        alloc 1 vc
        alloc 1 vdisc
        t0 = va * va
        t1 = 2 * va
        t2 = t1 * vb
        t3 = t0 + t2
        t4 = vb * vb
        t5 = t3 + t4
        vdisc = t5
        fend
```

Q 4.2

```
        alloc 1 va
ff:     fbegin
        alloc 1 vd
        t0 = vb + vc
        vd = t0
        ret vd
        fend
fmain:  fbegin
        alloc 1 vb
        alloc 1
        param vb
        param 3
        t1 = call ff
        va = t1
        fend
```

Q4.1 (2pt)

	C3A	a	b	c	d		X86
fmain:	fbegin					fmain:	push ebp mov ebp, esp
	alloc 1 va						sub esp, 4
	alloc 1 vb						sub esp, 4
	alloc 1 vc						sub esp, 4
	alloc 1 vdisc						sub esp, 4
	t0 = va * va	t0 t0					mov eax, [ebp - 4] imul eax, [ebp - 4]
	t1 = 2 * va	t0 t0	t1 t1				mov ebx, 2 imul ebx, [ebp - 4]
	t2 = t1 * vb	t0	t2				imul ebx, [ebp - 8]
	t3 = t0 + t2	t3					add eax, ebx
	t4 = vb * vb	t3 t3	t4 t4				mov ebx, [ebp - 8] imul ebx, [ebp - 8]
	t5 = t3 + t4	t5					add eax, ebx
	vdisc = t5						mov [ebp - 16], eax
	fend						add esp, 16 pop ebp ret

Q4.2 (2pt)

	C3A	a	b	c	d		X86
	alloc 1 va					va:	resd 1
	...						...
ff:	fbegin					ff:	push ebp mov ebp, esp
	alloc 1 vd						sub esp, 4
	t0 = vb + vc	t0 t0					mov eax, [ebp + 12] add eax, [ebp + 8]
	vd = t0						mov [ebp - 4], eax
	ret vd						mov eax, [ebp - 4] mov [ebp + 16], eax
	fend						add esp, 4 pop ebp ret
fmain:	fbegin					fmain:	push ebp mov ebp, esp
	alloc 1 vb						sub esp, 4
	alloc 1						sub esp, 4
	param vb						push dword[ebp - 4]
	param 3						push 3
	call ff	t1 t1 t1					call ff add esp, 8 pop eax
	va = t1	t1					mov [va], eax
	fend						add esp, 4 pop ebp ret

Code trois adresses

Le tableau ci-dessous décrit les instructions autorisées dans le code trois adresses. En plus de ces instructions, vous pouvez ajouter des étiquettes (`e:`) à certaines lignes. Les adresses utilisées peuvent être des constantes entières (`c`), des valeurs temporaires créés pour évaluer des expressions complexes (`t`), des variables du programme source en L (`v`) ou des étiquettes (`e`). Le tableau indique, pour chaque type d'opération, le nombre et les types des trois adresses `a1`, `a2` et `r`.

type	opérations	a1	a2	r	syntaxe	exemple
arithmétique	`+ - * /`	ctv	ctv	tv	`r = a1 op a2`	`t0 = var + 3`
affectation	`=`	ctv		tv	`r = a1`	`t0 = var`
saut test	`== < <= > >=`	ctv	ctv	e	`if a1 op a2 goto r`	`if t0 < 0 goto e1`
saut direct	`goto`	e			`op e`	`goto e7`
appel de fonction	`call`	e		tv	`op e` ou `r = op e`	`t7 = call fmax`
lecture	`read`			tv	`r = read`	`t0 = read`
écriture	`write`	ctv			`write a1`	`write t0`
allouer mém.	`alloc`	c	v		`alloc a1 a2`	`alloc 1 var`
e/s fonction	`param ret`	ctv			`op a1`	`param 12`
début/fin fonc.	`fbegin fend`				`fbegin`	`fbegin`

Intel x86

Le processeur possède 4 registres de 32 bits d'usage général : `eax`, `ebx`, `ecx` et `edx`. En général, les instructions ont la forme `opcode dest, source` avec le code de l'opération suivi de la destination et de la source. Plusieurs modes d'adressagage sont possibles pour la destination et la source, dont les noms de registres (`r`), les constantes (`imm`) et les adresses mémoire (`m`). La plupart des instructions n'accepte pas deux arguments de type `m` en même temps.

`mov`	`r1|m1, r2|m2|imm`	Charge le 2ème argument dans le registre `r1` ou dans la position mémoire `m1`
`push`	`r|m|imm`	Charge l'argument sur le sommet de la pile
`pop`	`r|m`	Charge le sommet de la pile dans le registre `r` ou dans la position mémoire `m`
`add/sub/imul`	`r1|m1, r2|m2|imm`	Somme/soustrait/multiplie le 2ème argument au 1er : `r1|m1 = r1|m1 +/-/* r2|m2|imm`
`idiv`	`r|m|imm`	Divise `edx:eax` par l'argument, reste dans `edx` : `eax = (edx:eax) / r|m, edx = (edx:eax) % r|m`
`cmp`	`r1|m1, r2|m2|imm`	Soustrait le 2ème argument au 1er sans stocker le résultat : `r1|m1 - r2|m2|imm`
`jl`	`e`	saut à l'adresse `e` si la flag SF≠OF (`r1|m1<r2|m2|imm`)
`je`	`e`	saut à l'adresse `e` si la flag ZF=1 (`r1|m1=r2|m2|imm`)
`jne`	`e`	saut à l'adresse `e` si la flag ZF=0 (`r1|m1! =r2|m2|imm`)
`jmp`	`e`	saut inconditionnel à l'adresse `e`
`call`	`e`	saut inconditionnel à la procédure `e` avec sauvegarde de `eip`
`ret`		retour de procédure, reviens à la valeur sauvegardée de `eip`

Nous utiliserons également les fonctions pré-définies suivantes, à l'aide de l'instruction `call` :

— `iprintLF` : affiche l'entier contenu dans `eax` à l'écran

— `readline` : lit une ligne dans la région mémoire pointée par `eax`

— `atoi` : met dans `eax` l'entier correspondant à la chaîne de caractères pointée par `eax`