Université Mohamed Khider – Biskra. Module: **Compilation**
Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie Niveau : **3ème Licence**
Département d'Informatique Année universitaire: **2013/2014**
Documents non autorisés

Contrôle

Date : 22/01/2014 **Durée : 1h30m**

Exercice 01 (04 pts)

Donner les expressions régulières pour Lex permettant de lire les mots suivants donnés en entrée :

1. les mots non vides indiquant la longitude en degrés, minutes et secondes : `-180*00'00"`, `180*00'00"`, `179*5'59"`, … sont des mots acceptés. `180*`, `180*00'`, `180*00'10"`, `179*00'60"`, `00'00"`, … ne sont pas acceptés.
Avec `*` (resp. `'` et `"`) qui indique le degré (resp. minute et seconde)
2. les mots non vides :
– composés de lettres minuscules de l'alphabet,
– où `a`, `b` et `c` apparaissent au plus une fois (0 ou 1 fois),
– si `a` est présent, il n'y aura pas `b` ni `c` à sa gauche de près ou de loin,
– si `b` est présent, il n'y aura pas `c` à sa gauche de près ou de loin.
`a`, `b`, `c`, `abde`, `btgcf`, `xaxbefc`, ... sont des mots acceptés,
`aeaf`, `ebdag`, `acbde`, ... ne sont pas acceptés.

Exercice 02 (08 pts)

Soit G la grammaire suivante pour des blocs Java (simplifiés) :
bloc → '{' inst_l '}'
inst_l → inst_l inst | ε
inst → decl | exp | bloc

decl et exp sont ici des unités lexicales terminales pour les déclarations et les expressions.

1. Il est évident que cette grammaire n'est pas LL(1) : pourquoi (en détail)? Transformez-la en Grammaire G1de type LL(1).
2. Construire la table d'analyse LL(1) pour G1.
3. Analyser le mot « **{ d1 {} e1}**» par un analyseur LL(1).;
4. Construire l'automate LR pour G.
5. G est-elle LR(0) ? SLR(1) ? Justifier rigoureusement en énumérant tous les conflits et en précisant leur type (shift/reduce ou reduce/reduce).

Exercice 03 (08 pts)

On souhaite ajouter aux expressions arithmétiques traditionnelles, les opérateurs binaires :

a) `<`, `<=`, `>` et `>=` opérateurs de comparaison, binaire, associativité de gauche à droite, moins prioritaires que les `+` et `-`, mais plus que `?:`.
b) `==` et `!=` opérateurs de comparaison, binaire, associativité de gauche à droite, moins prioritaires que les précédents, mais plus que `?:`
c) `<<` et `>>` (décalage de bits vers la droite et vers la gauche) ayant associativité gauche à droite, moins prioritaire que les `+` et `-`, mais plus prioritaire que les comparateurs : `<` , `<=`, `>` et `>=`

1. Compléter le ou les fichiers d'entrée (Lex et/ou Yacc) pour accepter ces nouveaux opérateurs binaires et pour construire l'arbre correspondant.
2. Proposer un modèle de code (d'instructions) MIPS pour l'opérateur de décalage.
3. Construire l'arbre et donner le code MIPS complet correspondant au code MNL suivant (registre de départ c'est $8) :

```
var a,b,c,d,e,f ;
lire a ;
lire b ;
lire c ;
lire d ;
lire e ;
lire f ;
ecrire a+b*c << d+e >> f ;
.
```

Fichier Lex

```
……
%}
ER_ENTIER -?[0-9]+
ER_VARIABLE [a-zA-Z][0-9a-zA-Z]*
ER_SEPARATEUR [ \n\t]
%%
{ER_SEPARATEUR}+ {}
{ER_ENTIER} {yylval.entier = atoi(yytext);
return(TOKEN_ENTIER);}
{ER_VARIABLE} {
yylval.chaine = strcpy((char *)malloc(yyleng+1), yytext);
return(TOKEN_VARIABLE);}
[\+\-] {yylval.caractere = yytext[0];return(TOKEN_PLUS_MOINS);}
[\*\/\%] {yylval.caractere = yytext[0]; return(TOKEN_MULT_DIV);}
[\(\)] {return(yytext[0]);}
"++"|"--" {yylval.caractere = yytext[0];return(TOKEN_INC_DEC);}
[?:=] {return(yytext[0]);}
<<EOF>> {return(0);}
```

MiniLangage (MNL)	MIPS
lire x	li $2,val syscall sw $2, décalage_de_la_variable($30)
ECRIRE *expression*	*... code de l'expression* *résultat _$8* move $4, $8 *# avec la valeur de l'expression déjà mis dans $8* li $2, 1 syscall
ECRIRE *chaine*	la $4, adresse_de_la_chaine li $2, 4 syscall
SI *expression* **ALORS** *alternance_vraie* **SINON** *alternance_fausse*	*code de l'expression – résultat _$8* bne $8, $9, etiqv *code de l'alternance fausse* j etiqfin etiqv: *code de l'alternance vraie* ... etiqfin: nop
TANTQUE *comparaison* **FAIRE** *corps* **FINTQ**	*j etiqtest* *etiqcorps: code du corps* ... *etiqtest: ...code de l'expression* *résultat_$8* *bne $8, $0, etiqcorps*

Syntaxe	Effet	Syntaxe	Effet
move r1, r2	r1 ← r2	lw r1, o (r2)	r1 → tas (r2 + o)
add r1, r2, o	r1 ← o + r2	sw r1, o (r2)	tas.(r2 + o) ← r1
sub r1, r2, o	r1 ← r2 - o	slt r1, r2, o	r1 ← r2 < o
mul r1, r2, o	r1 ← r2 × o	sle r1, r2, o	r1 ← r2 <= o
div r1, r2, o	r1 ← r2 ÷ o	seq r1, r2, o	r1 ← r2 = o
and r1, r2, o	r1 ← r2 and o	sne r1, r2, o	r1 ← r2!= o
or r1, r2, o	r1 ← r2 or o	j o	pc ← o
xor r1, r2, o	r1 ← r2 xor o	jal o	ra ← pc+1 et pc ← o
sll r1, r2, o	r1 ← r2 sl o	beq r, o, a	pc← a si r = o
srl r1, r2, o	r1 ← r2 sr o	bne r, o, a	pc← a si r <> o
li r1, n	r1 ← n	syscall	appel système
la r1, a	r1 ← a	nop	ne fait rien

Nom	Nº	Effet
print_int	1	imprime l'entier contenu dans a0
print_string	4	imprime la chaîne en a0 jusqu'à '\000'
read_int	5	lit un entier et le place dans v0
sbrk	9	alloue a0 bytes dans le tas, retourne l'adresse du début dans v0.
exit	10	arrêt du programme en cours d'exécution

Fichier

yacc

```
%{
…
EXPR_ARBRE a;
%}
%union { EXPR_ARBRE arbre; char *chaine; int entier; char
caractere; }
%token <chaine> TOKEN_VARIABLE
%token <entier> TOKEN_ENTIER
%token <caractere> TOKEN_PLUS_MOINS TOKEN_MULT_DIV
TOKEN_INC_DEC
%token TOKEN_FIN
%type <arbre> EE E T F G V C
%start EE
%%
EE : C {a = $1;};
C : E '?' C ':' C{$$ = CreerTer($1, $3, $5);}
  | E {};
E : E TOKEN_PLUS_MOINS T {$$ = CreerBin($2, $1, $3);}
  | T {};
T : T TOKEN_MULT_DIV G {$$ = CreerBin($2, $1, $3);}
  | G {};
F : '(' C ')' {$$ = $2;}
  | TOKEN_VARIABLE {$$ = CreerVariable($1);}
  | TOKEN_ENTIER {$$ = CreerConstante($1);};
G : TOKEN_INC_DEC V {$$ = CreerUn($1, $2);}
  | F {};
V : '(' TOKEN_VARIABLE ')' {$$ = CreerVariable($2);}
  | TOKEN_VARIABLE {$$ = CreerVariable($1);};
%%
…
```

Université Mohamed Khider – Biskra.
Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie
Département d'Informatique

Module: **Compilation**
Niveau : **3ème Licence**
Année universitaire: **2013/2014**
Documents non autorisés

Corrigé type

Fait le : 22/01/2014 **Durée : 1h30m**

Exercice 01 (04 pts)

`\-?(180\*00\'00\"|(0?[0-9]|1[0-7])?[0-9]\*[0-5]?[0-9]\'[0-5]?[0-9]\")` **(02 pts)**

`[d-z]+|([d-z]*a[d-z]*b?[d-z]*c?[d-z]*)|([d-z]*a?[d-z]*b[d-z]*c?[d-z]*)|([d-z]*a?[d-z]*b?[d-z]*c[d-z]*)` **(02 pts)**

Exercice 02 (08 pts) :

Soit G la grammaire suivante pour des blocs Java

B → { IL }
IL → IL inst | ε
inst → decl | exp | B
decl et exp sont ici des unités lexicales terminales pour les déclarations et les expressions.
Cette grammaire n'est pas LL(1) parce qu'il y a une récursivité à gauche dans la règle :
IL → IL inst | ε **(0,5 pts)**
Après la suppression de la récursivité à gauche, la grammaire G1 soit comme suit :
B → { IL }
IL→ inst IL| ε
inst → decl | exp | B **(0,5 pts)**
Construire la table d'analyse LL(1) pour G1**(1 pts)**.

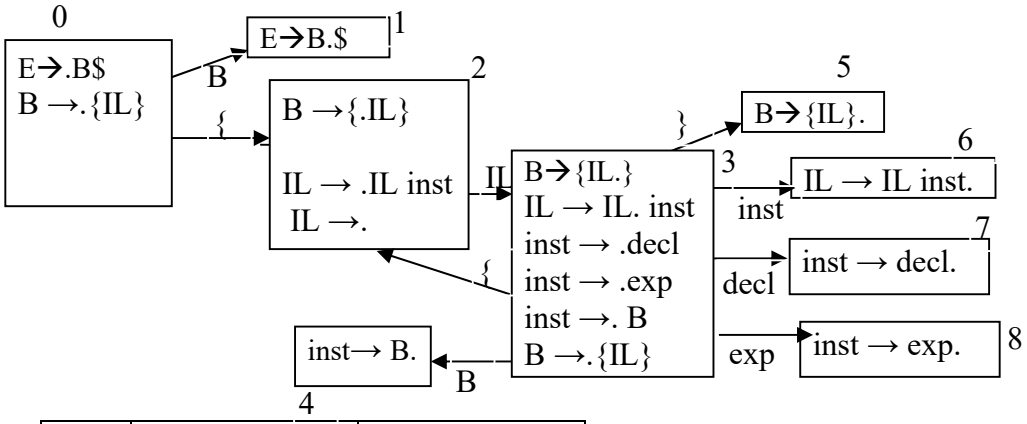
	Premiers	suivants
B	{	decl, exp, {,},$
IL	decl, exp, { , ε	}
inst	decl, exp, {	decl, exp, {,}

	{	}	decl	exp	$
B	{ IL }				
IL	inst IL	ε	inst IL	inst IL	
inst	bloc		decl	exp	

L'analyse du mot « { d1 {} e1}» par un analyseur LL(1) **(01 pts)**.

Pile	Chaine	Action
$B	{ d1 {} e1}$	dépiler, empiler }IL{
$}IL{	{ d1 {} e1}$	dépiler, Avancer
$}IL	d1 {} e1}$	dépiler, empiler IL inst
$}IL inst	d1 {} e1}$	dépiler, empiler decl
$}IL decl	d1 {} e1}$	dépiler, Avancer
$}IL	{} e1}$	dépiler, empiler IL inst
$}IL inst	{} e1}$	dépiler, empiler B
$}IL B	{} e1}$	dépiler, empiler }IL{
$}IL}IL{	{} e1}$	dépiler, Avancer
$}IL}IL	} e1}$	dépiler, empiler ε
$}IL}	} e1}$	dépiler, Avancer
$}IL	e1}$	dépiler, empiler IL inst
$}IL inst	e1}$	dépiler, empiler exp
$}IL exp	e1}$	dépiler, Avancer
$}IL	}$	dépiler, empiler ε
$}	}$	dépiler, Avancer
$	$	Accépter

Construction de l'automate LR pour G **(1,5 pts)**.



	Premiers	suivants
B	{	decl, exp, {,},$
IL	decl, exp, { , ε	}
inst	decl, exp, {	decl, exp, {,}

LR(0) ? oui il n'y a aucun cas de conflit **(0,5 pts)**

LR(0)	{	}	decl	exp	$	**(01 pts)**	B	IL	Inst
0	d2						1		
1	Accépter								
2	Réduire 3							3	
3	d2	d5	d7	d8			4		6
4	Réduire 6								
5	Réduire 1								
6	Réduire 2								
7	Réduire 4								
8	Réduire 5								

SLR(1) ? LR(0) ? oui il n'y a aucun cas de conflit **(0,5 pts)**

SLR(1)	{	}	decl	exp	$	**(01 pts)**	B	LI	Inst
0	d2						1		
1					Acc				
2		r3						3	
3	d2	d5	d7	d8			4		6
4	r6	r6	r6	r6					
5	r1	r1	r1	r1	r1				
6		r2							
7	r4	r4	r4	r4					
8	r5	r5	r5	r5					

Exercice 0 3 (08 pts)

On souhaite ajouter aux expressions arithmétiques traditionnelles, les opérateurs binaires :

a) <, <=, > et >= opérateurs de comparaison, binaire, associativité de gauche à droite, moins prioritaires que les + et −, mais plus que `?:`.

b) == et != opérateurs de comparaison, binaire, associativité de gauche à droite, moins prioritaires que les précédents, mais plus que `?:`

c) << et >> (décalage de bits vers la droite et vers la gauche) ayant associativité gauche à droite, moins prioritaire que les + et −, mais plus prioritaire que les comparateurs : < , <=, > et >=

1. Compléter le ou les fichiers d'entrée (Lex et/ou Yacc) pour accepter ces nouveaux opérateurs binaires et pour construire l'arbre correspondant.

Fichier Lex (2 pts)

```
……
 ">="|">"|"<="|">"{ yylval.chaine = strcpy((char *)
malloc(yyleng+1), yytext);return(TOKEN_EGDIFF);}

"=="|"!=" {yylval.chaine = strcpy((char *) malloc(yyleng+1),
yytext);return(TOKEN_SUPINF);}

">>"|"<<"{ yylval.chaine = strcpy((char *) malloc(yyleng+1),
yytext);return(TOKEN_DECL);}

"++"|"--" {yylval.caractere =
yytext[0];return(TOKEN_INC_DEC);}

[?:=] {return(yytext[0]);}

<<EOF>> {return(0);}
```

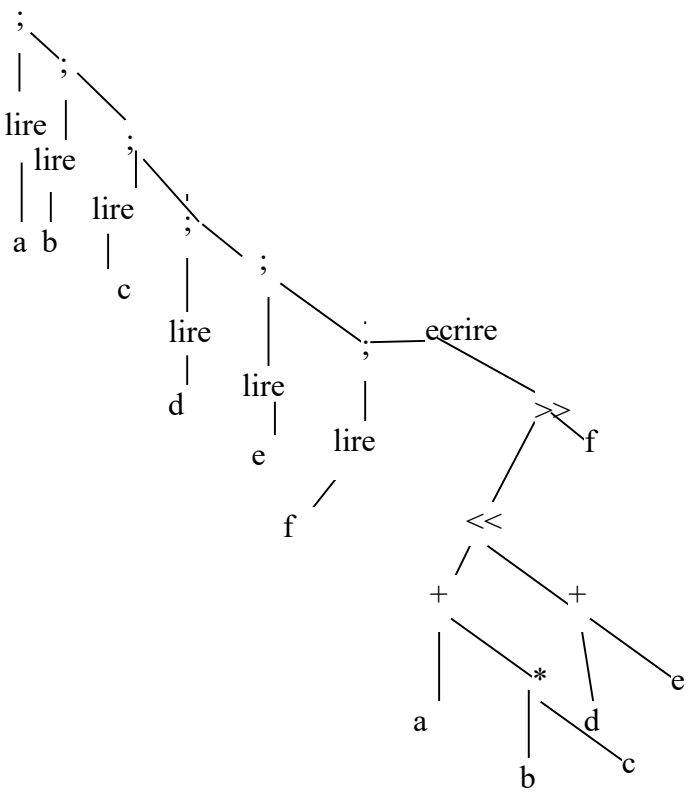
Fichier yacc (2 pts)

```
…
%token <caractere> TOKEN_PLUS_MOINS TOKEN_MULT_DIV
TOKEN_INC_DEC TOKEN_EGDIFF TOKEN_SUPINF TOKEN_DECL
%token TOKEN_FIN
%type <arbre> EE E T F G V C M N D
%start EE
%%
EE : C {a = $1;};
C : M'?' C ':' C{$$ = CreerTer($1, $3, $5);}
  | M {};
M : M TOKEN_EGDIFF N {$$ = CreerBin2($2, $1, $3);}
  | N { };
N : N TOKEN_SUPINF D {$$ = CreerBin2($2, $1, $3);}
  | D { };
D : D TOKEN_DECL E {$$ = CreerBin2($2, $1, $3);}
  | E { };
E : E TOKEN_PLUS_MOINS T {$$ = CreerBin($2, $1, $3);}
  | T {};
……
%%
……
```

2. un modèle de code (d'instructions) MIPS pour l'opérateur de décalage.

```
Code calculant l'expression G →$n
Code calculant l'expression D →$n+1     (1 pts)
if( !strcmp(aux->forme.bin2.op, « >> »)
  fprintf(sortie,"srl $%d, $%d, $%d",n,n,n+1);
else
  fprintf(sortie,"sll $%d, $%d, $%d",n,n,n+1);
```

Construire l'arbre **(1,5 pts)**



3. le code MIPS complet correspondant au code MNL **(1,5 pts)**:

```
.data
MEM: .space 24
.text
main: la $30,
MEM
li $2,5
syscall
sw $2, 0($30)
li $2,5
syscall
sw $2, 4($30)
li $2,5
syscall
sw $2, 8($30)
li $2,5
syscall
sw $2, 12($30)
li $2,5
syscall
sw $2, 16($30)
li $2,5
syscall
sw $2, 20($30)
lw $8, 0($30)
lw $9, 4($30)
lw $10, 8($30)
mult $9,$9,$10
add $8,$8,$9
lw $9, 12($30)
lw $10, 16($30)
add  $9,$9,$10
sll $8, $8, $9
lw $9, 20($30)
srl $8, $8, $9
move $8,$4
li $2,1
syscall
li $2,10
syscall
```

```
enum EXPR_TYPE { CONSTANTE, VARIABLE, BINAIRE, UNAIRE, TERNAIRE};

typedef struct _EXPR_ARBRE {
enum EXPR_TYPE type;
union {
…
struct {
char *op;
struct _EXPR_ARBRE *opg;
struct _EXPR_ARBRE *opd;
} bin2;
…
} forme;
} *EXPR_ARBRE;
EXPR_ARBRE CreerBin2 (char *op, EXPR_ARBRE g, EXPR_ARBRE d)
{
EXPR_ARBRE aux = (EXPR_ARBRE)malloc(sizeof (struct _EXPR_ARBRE));
aux->type = BINAIRE;
aux->forme.bin2.op = strcpy(malloc(sizeof (strlen(op)+1)),op);
aux->forme.bin2.opg = g;
aux->forme.bin2.opd = d;
return(aux);
}
```

Université Mohamed Khider – Biskra. Module: **Compilation**
Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie Niveau : **3ème Licence**
Année universitaire: **2013/2014**
Département d'Informatique **Documents non autorisés**

Rattrapage

Date : 24/02/2014 **Durée : 1h30m**

Exercice 01 (04 pts)

1. Donner une expression régulière étendue en Lex permettant de reconnaître les chaînes composées de chiffres en ordre lexicographique croissant. Une chaîne vide est acceptée.
   Exemples de chaînes acceptées : 12345 23468 135 0127 122345
   Exemples de chaînes non acceptées : 2410
2. Donner une expression rationnelle étendue permettant de reconnaître des durées (strictement inférieures à 24h) en heures, minutes et secondes respectant les contraintes suivantes :
   - Une durée s'exprime avec le format `--h--m--s`.
   - Les secondes et minutes sont comprises entre 0 et 59 et composées d'un ou deux chiffres.
   - Les heures sont comprises entre 0 et 23 et composées d'un ou deux chiffres
   - Les heures ne peuvent être présentes que si les minutes y sont et les minutes ne peuvent être présentes que si les secondes y sont.

Exemples de durées acceptées : 05s 05m00s 02h3m04s 2h03m4s

Exemples de durées non acceptées: 05m : sans la présence des secondes

Exercice 02 (08 pts) : Soit la grammaire G :

$S \rightarrow R\ a|\ b\ T\ c$
$R \rightarrow b\ T\ c$
$T \rightarrow T\ c|d$
S est l'axiome.

1. Donner la forme générale des mots générés par cette grammaire.
2. G est-elle LL(1) ? justifier votre réponse. Sinon la transformer pour qu'elle le soit (grammaire G').
3. Construire la table d'analyse LL(1) .
4. Construire la table SLR de G. G est-elle SLR(1) ? Sinon Justifier rigoureusement en énumérant tous les conflits et en précisant leur type (shift/reduce ou reduce/reduce).
5. Construire la table LR(0) de G. G est-elle LR(0) ? Sinon Justifier rigoureusement en énumérant tous les conflits et en précisant leur type (shift/reduce ou reduce/reduce).

Exercice 03 (08 pts)

Dans certain langage, Impaire(A) est introduit comme une fonction mathématique et rend **la valeur 1 si A est impaire et 0 sinon**
Par exemple : Impaire(3*2+1)=1
On veut introduire cette expression Impaire () dans le MiniLangage comme un opérateur unaire :`Impaire(exp_opérande)`
1. Ecrire la fonction Créer `Impaire` qui permet de créer l'arbre abstrait de cette expression.
2. Modifier les fichiers d'entrée pour Lex et Yacc pour accepter ce nouvel opérateur binaire et pour construire l'arbre correspondant.
3. Proposer un modèle de code (d'instructions) MIPS pour ce nouvel opérateur.
4. Construire l'arbre et donner le code MIPS complet correspondant au code MNL suivant :

```
Var a,b ;
Lire a;
Lire b;
If(a<0)  a=-1*a ;
if(b<0)  b=-1*b ;
ecrire Impaire(a>b?a:b);
.
```

Fichier Lex

```
……
%}
ER_ENTIER -?[0-9]+
ER_VARIABLE [a-zA-Z][0-9a-zA-Z]*
ER_SEPARATEUR [ \n\t]
%%
{ER_SEPARATEUR}+ {}
{ER_ENTIER} {yylval.entier = atoi(yytext);
return(TOKEN_ENTIER);}
{ER_VARIABLE} {
yylval.chaine = strcpy((char *)malloc(yyleng+1), yytext);
return(TOKEN_VARIABLE);}
[\+\-] {yylval.caractere = yytext[0];return(TOKEN_PLUS_MOINS);}
[\*\/\%] {yylval.caractere = yytext[0];
return(TOKEN_MULT_DIV);}
[\(\)] {return(yytext[0]);}
"++"|"--" {yylval.caractere = yytext[0];return(TOKEN_INC_DEC);}
[?:=] {return(yytext[0]);}
<<EOF>> {return(0);}
```

Université Mohamed Khider – Biskra. Module: **Compilation**
Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie Niveau : **3ème Licence**
Année universitaire: **2013/2014**
Département d'Informatique **Documents non autorisés**

Corrigé type de rattrapage

Date : 11/03/2013 **Durée : 1h30m**

Exercice 01 (04 pts)

1. 1*2*3*4*5*6*7*8*9*
2. **(([0-1] ?[0-9]|2[0-3]h) ?([0-5] ? [0-9]m)) ? ([0-5] ? [0-9])s**

Exercice 02 (08 pts)

1. Donner la forme générale des mots générés par cette grammaire.

$$\mathbf{bdc^n(a}|\ \varepsilon)\ /n>0 \text{ (01 pts)}$$

Exercice 03 (08 pts)

Dans certain langage, Carre(A) est introduit comme une fonction mathématique permettant de rendre la valeur **A à la puissance de 2**
On veut introduire ce **carre(exp)** dans le MiniLangage

1. Modification des fichiers Lex et Yacc **(1.5 pts)**.

Fichier Lex

```
%%
.........
carre {yylval.caractere = yytext[0]; return(TOKEN_Carre);}
{ER_VARIABLE} {
yylval.chaine = strcpy((char *)malloc(yyleng+1), yytext);
return(TOKEN_VARIABLE);}
.........
```

Fichier yacc

```
%%
.....
F : '(' E ')' {$$ = $2;}
| TOKEN_VARIABLE {$$ = CreerVariable($1);}
| TOKEN_ENTIER {$$ = CreerConstante($1);};
| TOKEN_Carre(C);{$$ = CreerUn($1,$3) ;}
%%
.....
```

2. une déclaration de la structure de cette **(0.5 pts)**.
   Puisque l'opérateur carre est une opération unaire on propose la structure suivante :

```
typedef struct _EXPR_ARBRE {
  enum EXPR_TYPE type;
  union {
    int   val;
    char *nom;
    struct {
      char             op;
      struct _EXPR_ARBRE *u;
    } un;
   …..
  } forme;
} *EXPR_ARBRE;
```

3. Proposer une règle de génération du code MIPS de cette expession **(01 pts)**.

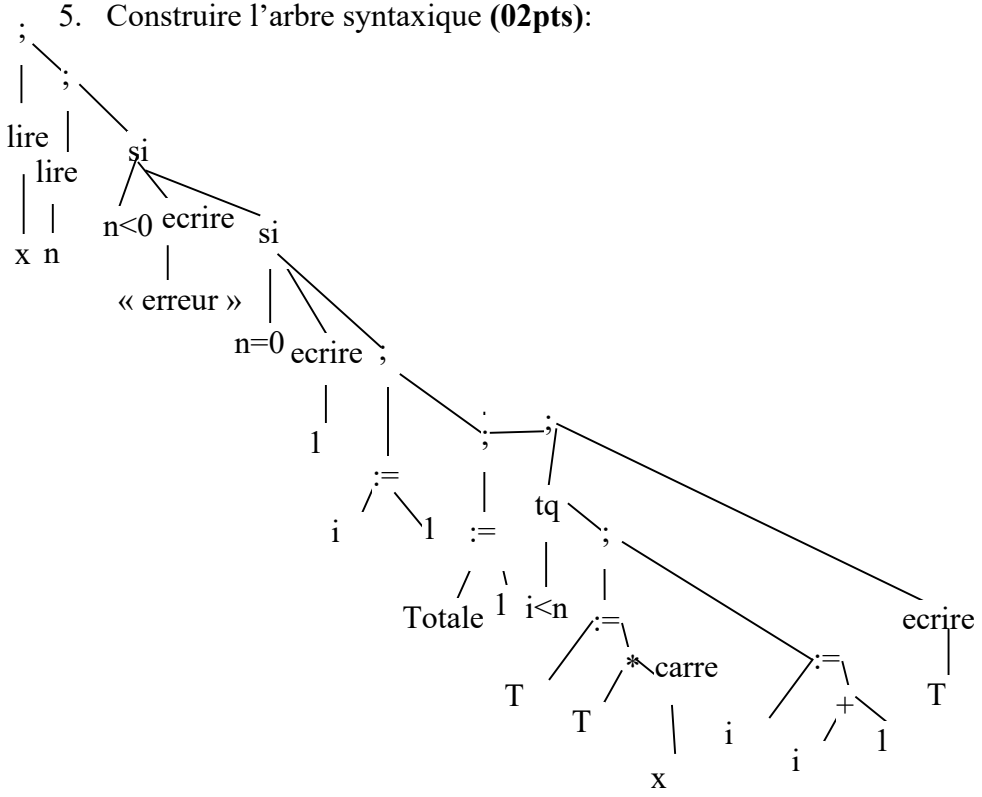
```
Code pour calculer son  paramètre (qui est
                une expression) résultat $8
Move $8,$9
Mult $8,$8,$9
```

4. Un programme en minilangage permettant de calculer x^{2*n}**(01 pts)**; :

```
var x,n,i,T

Ecrire"Donner la valeur de x, n:\n"
lire x ;
lire n ;
si n<0 alors
      ecrire « erreur » ;
sinon
      si n=0 alors ecrire 1 ;
            sinon
               i :=1 ;
               Totale :=1 ;
               tantque (i<n) faire
                       T=T*carre(x) ;
                       i :=i+1 ;
               Fin Tq ;
               ecrire T ;
            Fin si ;
        Finsi ;
   .
```

5. Construire l'arbre syntaxique **(02pts)**:



1. le code MIPS correspond au programme de la question 5 **(02 pts)**

```
.data
MEM: .space 16
CHAINE0: .asciiz "Donner la valeur de x, n:\n"
CHAINE1: .asciiz "Erreur \n"
      .text
main:  la $30, MEM
      la $4, CHAINE0
      li $2, 4
      syscall
      li $2, 5
      syscall
      sw $2, 0($30)
      li $2, 5
      syscall
      sw $2, 4($30)
       lw $8,4($30)
      blt $8,$0,et0
       lw $8,4($30)
      be $8,$0,et3
       li $8,1
       sw $8,8($30)
       li $8,1
       sw $8,12($30)
       j et5
et6 : lw $8,12($30)
       lw $9,0($30)
       move $9,$10
       mult $9,$9,$10
       mult $8,$8,$9
       sw $8,12($30)
et5 : lw $8,8($30)
       lw $9,4($30)
       blt $8,$9,et6
       lw $8,12($30)
       move $4, $8
       li $2, 4
       syscall
       j et4
et3 : li $8,1
       move $4, $8
       li $2, 4
       syscall
       j et4
et4 : nop
       j et1
et0 : la $4, CHAINE1
       li $2, 4
       syscall
et1 : nop
       li $2, 10      syscall
```