Corrigé du TD de L2 N°1

Patrick Poulingeas.

- **Exercice 1.**

1.a)

Un identificateur commence par une lettre (majuscule ou minuscule) suivi d'un nombre quelconque (éventuellement aucun) de symboles.

Ces symboles sont constitués des lettres, des chiffres, et du caractère '_'.

Exemples d'idenficateurs valides : i, i1, Une_variable, a____1

Exemples d'idenficateurs incorrects : 3a, _i, Une variable

N.B. Dans les réponses aux questions suivantes, on se contentera de donner les règles de production des grammaires. L'axiome de la grammaire sera le premier symbole non terminal apparaissant en partie gauche de la première règle. Le vocabulaire non terminal et le vocabulaire terminal se déduiront implicitement de l'écriture des règles.

<identificateur> ::= <lettre><suite de symboles>
<suite de symboles> ::= <symbole><suite de symboles> | ^
<symbole> ::= <lettre> | <chiffre> | _
<lettre> ::= a | b | ... | z | A | B | ... | Z
<chiffre> ::= 0 | 1 | ... | 9

1.b)

Une constante (ou plus rigoureusement un *littéral*) de type chaîne de caractères est délimitée par des apostrophes et constituée d'un nombre quelconque de caractères. Pour permettre à une apostrophe de faire partie d'une chaîne de caractères, on double celle-ci.

Ce sont les règles syntaxiques employées par Turbo Pascal.

<constante chaîne> ::= '<suite de caractères>'
<suite de caractères> ::= <caractère><suite de caractères> | ^
<caractère> ::= <lettre> | <chiffre> | " | @ | ...

1.c)

On se limitera à un sous-ensemble des déclarations de constantes possibles en Turbo Pascal.

Exemple de déclarations de constantes simples :

```
CONST c1 = 20000;          { constante entière }
      c2 = 0.141;          { constante réelle }
      c3 = 'a';            { constante caractère }
      c4 = 'chaine';       { constante chaîne de caractères }
```

Exemple de déclarations de constantes typées :

```
CONST c1 : Integer = 42;
      c2 : Real = 14.1E-2;
      c3 : Char = 'z';
      c4 : String = 'foo';
      c5 : String[30] = 'bar';    { 30 est le nombre maximum de caractères autorisés
                                    dans la chaîne }
```

```
<déclaration de constantes>  ::= CONST <suite de constantes>
<suite de constantes>        ::= <déclaration de constante>;
                                 | <déclaration de constante>;<suite de constantes>
<déclaration de constante>   ::= <identificateur> = <valeur numérique>
                                 | <identificateur> = '<caractère>'
                                 | <identificateur> = <constante chaîne>
                                 | <identificateur> : Integer = <constante entière>
                                 | <identificateur> : Real = <constante réelle>
                                 | <identificateur> : Char = '<caractère>'
                                 | <identificateur> : String = <constante chaîne>
                                 | <identificateur> : String[<nombre positif>] = <constante chaîne>
<valeur numérique>           ::= <constante entière> | <constante réelle>
<constante entière>          ::= <signe><nombre positif> | <nombre positif>
<signe>                      ::= + | -
<nombre positif>             ::= <chiffre> | <chiffre><nombre positif>
<constante réelle>           ::= <constante entière><exposant>
                                 | <constante entière>.<nombre positif><exposant>
<exposant>                   ::= ^ | E<constante entière>
```

1.d)

Exemple de déclarations de variables (même remarque qu'au 1.c) :

```
VAR i1 : Integer;
    i2 : Real;                      { Et les types Boolean, Char, String }
    i3, i4 : Integer;
    i5 : String[42];
    i6 : Array [1..5] of Real;      { tableau à une dimension }
    i7 : Array [3..7,1..8] of Char; { tableau à deux dimensions }
```

```
<déclaration de variables>   ::= VAR <suite de variables>
<suite de variables>         ::= <déclaration de variable>;
                                 | <déclaration de variable>;<suite de variables>
<déclaration de variable>    ::= <liste d'identificateurs> : <type>
<liste d'identificateurs>    ::= <identificateur> | <identificateur>,<liste d'identificateurs>
<type>                       ::= Integer | Real | Boolean | Char | String
                                 | String[<nombre positif>]
                                 | <tableau>
<tableau>                    ::= Array[<intervalles>] of <type>
<intervalles>                ::= <nombre positif>..<nombre positif>
                                 | <nombre positif>..<nombre positif>,<intervalles>
```

1.e)

Exemple de déclarations de types (même remarque qu'au 1.c) :

```
TYPE chaine = String[10];
     tableau = Array[1..100] of chaine;
     personne = Record
                  nom : String;
                  prenom : String;
                  age : Integer;
                end;
```

```
<déclaration de types>       ::= TYPE <suite de types>
<suite de types>             ::= <déclaration de type>; | <déclaration de type>;<suite de types>
<déclaration de type>        ::= <identificateur> = <type>
```

```
<type>          ::= Integer | Real | Boolean | Char | String
                  | String[<nombre positif>]
                  | <tableau2>
                  | <enregistrement>
<tableau2>      ::= Array[<intervalles>] of <nom>
<nom>           ::= <type> | <identificateur>
<enregistrement> ::= Record <suite de variables> end
```

Dans les règles de productions que l'on vient d'écrire, pour la définition d'un nouveau type reprenant un type déclaré précédemment, on s'est limité au cas d'un type tableau, c'est-à-dire à la forme : Array[min..max] of "type déclaré précédemment" (comme dans l'exemple). Cette grammaire ne permet donc pas d'engendrer des déclarations du style :

```
TYPE premier_type = Char;
     second_type  = premier_type;        { autorisé en Turbo Pascal }
```

Pour gérer des déclarations de ce genre, il faudrait mettre le non-terminal <identificateur> dans une alternative de la règle de production définissant <type>.

1.f)

Exemple de programme (c'est-à-dire un élément du langage considéré) :

```
Algo exemple
Variables
    a,b : entier;
    c   : réel;
Début
    a ← (3*5);
    b ← c;
    Ecrire(a);
    Ecrire(b+1);
    Ecrire(a,c);
    Lire(b);
    Si (a<b) et (b<2-3) alors
        c ← 42;
    Fin Si;
    Tant que non(a<b) faire
        Lire(b,c);
        a ← a+1;
    Fin Tant que;
    Pour i de 1 à 10 faire
        Ecrire(i);
    Fin Pour;
    Répéter
        a ← a+1;
    Jusqu'à (a=10) ou (7+b*6≥8);
Fin
```

Remarques :

- On interdira les programmes "vides" en rendant la partie contenant les instructions obligatoire (c'est-à-dire la partie commençant par le mot-clé Début et se terminant par le mot-clé Fin).
- Le titre (c'est-à-dire la ligne commençant par le mot-clé Algo) et la section de déclaration des variables ne sont pas obligatoires.
- Le ';' joue le rôle d'un terminateur d'instructions (comme en C, C++ ou Java) et non pas le rôle d'un séparateur d'instructions (comme avec Turbo Pascal).

<programme> ::= <titre><ensemble de déclarations><ensemble d'instructions>

<titre> ::= Algo <identificateur> | ^

<ensemble de déclarations> ::= Variables <suite de déclarations> | ^
<suite de déclarations> ::= <déclaration> | <déclaration><suite de déclarations>
<déclaration> ::= <suite d'identificateurs> : <type>;
<suite d'identificateurs> ::= <identificateur> | <identificateur>,<suite d'identificateurs>
<type> ::= entier | réel

<ensemble d'instructions> ::= Début <suite d'instructions> Fin
<suite d'instructions> ::= <instruction>; | <instruction>;<suite d'instructions>
<instruction> ::= <affectation> | <conditionnelle> | <itération>
| <lecture> | <écriture>
<affectation> ::= <identificateur> ← <EA>
<conditionnelle> ::= Si <condition> alors <suite d'instructions><suite conditionnelle>
<suite conditionnelle> ::= FinSi | Sinon <suite d'instructions> FinSi
<itération> ::= Répéter <suite d'instructions> Jusqu'à <condition>
| Tant que <condition> faire <suite d'instructions> Fin Tant que
| Pour <identificateur> de <EA> à <EA> faire
<suite d'instructions> Fin Pour
<lecture> ::= lire (<suite d'identificateurs>)
<écriture> ::= écrire (<suite d'expressions>)
<suite d'expressions> ::= <EA> | <EA>,<suite d'expressions>

<EA> ::= <EA1> | <EA>+<EA1> | <EA>-<EA1>
<EA1> ::= <EA2> | <EA1>*<EA2> | <EA1>/<EA2>
<EA2> ::= (<EA>) | <signe><opérande>
<opérande> ::= <identificateur> | <entier> | <réel>
<signe> ::= ^ | + | -

<condition> ::= <expression booléenne>
| (<expression booléenne>)
| <expression booléenne><connecteur><expression booléenne>
<expression booléenne> ::= <EA><comparaison><EA>
| (<EA><comparaison><EA>)
| non(<condition>)
<connecteur> ::= et | ou
<comparaison> ::= < | > | ≤ | ≥ | = | ≠

Avec la formulation d'une expression conditionnelle donnée ci-dessus, la grammaire est ambiguë.

Considérons en effet l'expression : Si (a≤b) alors ...

On peut l'obtenir en effectuant les dérivations à gauche suivantes :

Si <condition> alors <suite d'instructions><suite conditionnelle>
→ Si <expression booléenne> alors <suite d'instructions><suite conditionnelle>
→ Si (<EA><comparaison><EA>) alors <suite d'instructions><suite conditionnelle>

ou en faisant les dérivations à gauche suivantes :

Si <condition> alors <suite d'instructions><suite conditionnelle>
→ Si (<expression booléenne>) alors <suite d'instructions><suite conditionnelle>
→ Si (<EA><comparaison><EA>) alors <suite d'instructions><suite conditionnelle>

Voici un autre ensemble de règles de productions pour une expression booléenne et dont l'avantage est d'introduire une priorité des opérateurs logiques :

<condition> ::= <condition> ou <expression booléenne> | <expression booléenne>
<expression booléenne> ::= <expression booléenne> et <terme booléen> | <terme booléen>
<terme booléen> ::= (<condition>) | non(<condition>) | <EA><comparaison><EA>

Avec cette grammaire, on a, pour les opérateurs logiques, l'ordre de priorité décroissant suivant : non, et, ou (C'est la convention utilisée par C, C++, Java et Turbo Pascal).

- **Exercice 2.**

Pour le stockage en mémoire centrale, on se préoccupe essentiellement des règles de productions de la grammaire.

Etudions sur un exemple la méthode de stockage que nous allons employer. On considère les règles de productions suivantes :

<A> ::= a<B> | b
<B> ::= ^ | <A>d

Remarque :
Des conventions que nous suivons depuis le 1.a, on déduit que :

- le vocabulaire terminal est l'ensemble V_T = {a,b,d}
- le vocabulaire non-terminal est l'ensemble V_N = {<A>,<B>}
- l'axiome est <A>

Pour représenter une règle, on introduit le type suivant :

```
Type Règle = Enregistrement
        nom : Chaîne
        premiere_alternative : Entier positif
Fin Enregistrement
```

Le champ 'nom' est une chaîne de caractère correspondant à l'élément non-terminal en partie gauche de la règle. Quant au champ 'premiere_alternative', il correspond à un indice dans un tableau contenant les alternatives (appellé tab_alter) et sert à indiquer le début de la partie droite de la règle de production (c'est-à-dire le premier symbole de la première alternative).

Les règles sont stockées dans un tableau d'éléments de type Règle appellé tab_règles.

Pour représenter une alternative, on se sert du type suivant :

```
Type Partie_d_alternative = Enregistrement
        symbole : Chaîne
        symbole_suivant : Entier positif
        alternative_suivante : Entier positif
Fin Enregistrement
```

Les alternatives sont stockées dans un tableau d'éléments du type Partie_d_alternative que l'on nomme tab_alter.

Sur l'exemple étudié, on obtient le stockage suivant :

nom	***premiere_alternative***
<A>	1
<B>	4

tab_règles

Indice de Tab_alter	***symbole***	***symbole_suivant***	***alternative_suivante***
1	a	2	3
2	<B>	0	3
3	b	0	0
4	^	0	5
5	<A>	6	0
6	d	0	0

tab_alter

On dispose par ailleurs de 2 variables nb_règles et nb_symboles indiquant respectivement le nombre de règles (c.à.d. la taille du tableau tab_règles) et le nombre de symboles intervenant dans les parties droites des règles (c.à.d. la taille du tableau tab_alter). Dans l'exemple, on a : nb_règles=2 et nb_symboles=6.

Remarques :

- Pour indiquer la fin d'une alternative dans tab_alter, on met le champ symbole_suivant à 0.
- Pour indiquer que l'on examine la dernière alternative d'une règle, dans tab_alter on fixe à 0 le champ alternative_suivante.

```
Action Stocker_grammaire
        Initialisation
        Tant que non(Fin(fichier)) faire
                Traiter_règle
        Fin Tant que
Fin Action

Action Initialisation
        nb_règles ← 0
        nb_symboles ← 0
Fin Action

Action Traiter_règle
        règle ← Lire(fichier)
        temp ← Lire_séparateur(fichier)      {Sert à lire le '::='}
        tab_règles[nb_règles+1].nom ← règle
        tab_règles[nb_règles+1].première_alternative ← nb_symboles+1
         nb_règles ← nb_règles+1
```

```
        {On insère à présent dans tab_alter les symboles composant les alternatives de la règle que
          l'on est en train de traiter}
        Tant que non(Fin_de_ligne(fichier)) faire
                Traiter_alternative
        Fin Tant que
Fin Action

Action Traiter_alternative
        { Traitement de l'alternative courante }
        symbole ← Lire(fichier)
        début_alternative ← nb_symboles+1
        compteur ←  début_alternative

        Tant que (symbole ≠ '|') et non(Fin_de_ligne(fichier)) faire
                symbole_suivant ← Lire(fichier)
                tab_alter[compteur].symbole ← symbole
                nb_symboles ← nb_symboles+1

                { Mise à jour du champ symbole_suivant dans tab_alter}
                Si (symbole_suivant ≠ '|') et non(Fin_de_ligne(fichier)) alors
                        { On n'a pas encore mis en mémoire tous les symboles de
                          l'alternative }
                        tab_alter[compteur]. symbole_suivant ← compteur+1
                sinon
                        tab_alter[compteur]. symbole_suivant ← 0
                Fin Si

                symbole ← symbole_suivant
                compteur ← compteur+1
        Fin Tant que

        { On termine le traitement de l'alternative courante par la mise à jour du champ
          alternative_suivante dans tab_alter }
        Si non(Fin_de_ligne(fichier)) alors
                { Il reste au moins une alternative de la règle à traiter }
                Pour indice de début_alternative à (compteur-1) faire
                        tab_alter[indice]. alternative_suivante ← nb_symboles+1
                Fin Pour
        sinon
                Pour indice de début_alternative à (compteur-1) faire
                        tab_alter[indice]. alternative_suivante ← 0
                Fin Pour
        Fin Si

Fin Action
```

• Exercice 3.

Rappels de cours :

Traitement des règles de production récursives à gauche :
La règle : $<A> ::= \alpha \mid <A>\beta$ où $\alpha,\beta \in V^*$ (avec $V=V_N \cup V_T$)
est transformée en : $<A> ::= \alpha<X>$ et
$<X> ::= \hat{} \mid \beta<X>$

Traitement des règles comprenant des alternatives commençant par le même symbole :
La règle : $<A> ::= u\alpha \mid u\beta$ où $u \in V$ et $\alpha,\beta \in V^*$
est transformée en : $<A> ::= u<X>$ et
$<X> ::= \alpha \mid \beta$

Le fait d'effectuer les transformations précédentes est nécessaire pour obtenir une grammaire LL(1) mais non suffisant (cf. cours pour la définition d'une grammaire LL(1)).

Pour déterminer si une grammaire est LL(1), on peut être amené à calculer les premiers d'un élément de V^* et les suivants d'un non-terminal.

Définition de l'ensemble des premiers d'un élément $\alpha \in V^*$:
$Premier(\alpha) = \{ a \in V_T \mid \alpha \rightarrow^* a\beta, \beta \in V^* \}$

Détermination des premiers de X ($X \in V_N$) :
On considère la règle suivante : $X ::= B_1...B_m \mid C_1...C_n$
On a alors :
$Premier(X)=Premier(B_1...B_m) \cup Premier(C_1...C_n)$
avec :
$Premier(B_1...B_m) = \cup_{(i=1..k)} Premier(B_i)$
où $k = max(\{p \mid 2 \le p \le m \text{ et } \forall j \le p, B_j \rightarrow^* \hat{} \} \cup \{1\})$
⇑
Cet ensemble peut être vide

Définition de l'ensemble des suivants d'un symbole $A \in V_N$:
$Suivant(A)=\{a \in V_T \mid S \rightarrow^* xA\beta, a \in Premier(\beta), x \in V_T^*, \beta \in V^*\}$

Détermination des suivants de X ($X \in V_N$) :
On considère les règles suivantes : $D ::= \ldots XA_1A_2...A_n$
$B ::= \ldots X$
On a :
$Suivant(X)=Suivant(B) \cup Premier(A_1A_2...A_n) \cup \Psi$
où : $\Psi = Suivant(D)$ si $\forall k \in \{1,...,n\}\ A_k \rightarrow^* \hat{}$
$\varnothing$ si $\exists k \in \{1,...,n\}$ non$(A_k \rightarrow^* \hat{})$

Exemple de calcul de l'ensemble des suivants :
On a les règles de production suivantes :
D ::= Z | W
Z ::= aXBC
B ::= ^ | b
C ::= ^ | c
W ::= ^ | Zt

En suivant les conventions que l'on s'est donné, l'axiome est D.
On cherche à déterminer Suivant(X).
On a :
Suivant(X)=Premier(B) $\cup$ Premier(C) $\cup$ Suivant(Z)
= {b}$\cup${c}$\cup${t}
= {b,c,t}
Preuve que t$\in$Suivant(X) :
D $\rightarrow$ W
$\rightarrow$ Zt
$\rightarrow$ aXBCt
$\rightarrow$ aXCt
$\rightarrow$ aXt

3.a)
Rappelons les règles données dans l'exercice 1 :
(1) <identificateur> ::= <lettre><suite de symboles>
(2) <suite de symboles> ::= <symbole><suite de symboles> | ^
(3) <symbole> ::= <lettre> | <chiffre> | _
(4) <lettre> ::= a | b | ... | z | A | B | ... | Z
(5) <chiffre> ::= 0 | 1 | ... | 9

Il n'y a pas de règles récursives à gauche ou de règles comprenant des alternatives commençant par le même symbole. On n'a donc pas de transformations à effectuer.

Examinons le caractère LL(1) de la grammaire :

Etudions la règle (2) :
La règle (2) ayant une alternative pouvant engendrer la chaîne ^, déterminons :
Premier(<symbole><suite de symboles>) $\cap$ Suivant(<suite de symboles>)
Premier(<symbole><suite de symboles>)
= Premier(<symbole>)
= Premier(<lettre>) $\cup$ Premier(<chiffre>) $\cup${_}
= {a,...,z,A,...,Z,0,...,9,_}
Suivant(<suite de symboles>) = $\varnothing$
Ainsi :
Premier(<symbole><suite de symboles>) $\cap$ Suivant(<suite de symboles>) = $\varnothing$
La règle (2) n'empêche donc pas la grammaire d'être LL(1).

Etudions la règle (3) :
On a :
Premier(<lettre>) = {a,...,z,A,...,Z}
Premier(<chiffre>) = {0,...,9}
Premier(_) = {_}
Ainsi :
Premier(<lettre>) $\cap$ Premier(<chiffre>) = $\varnothing$
Premier(<lettre>) $\cap$ Premier(_) = $\varnothing$
Premier(<chiffre>) $\cap$ Premier(_) = $\varnothing$
Ainsi, la règle (3) n'empêche pas la grammaire d'être LL(1).

De façon évidente les autres règles ne posent pas problème.

La grammaire est donc LL(1).

3.b)

Rappelons les règles données dans l'exercice 1 :

(1) <constante chaîne> ::= '<suite de caractères>'
(2) <suite de caractères> ::= <caractère><suite de caractères> | ^
(3) <caractère> ::= <lettre> | <chiffre> | " | @ | ...

On n'a aucune transformation à effectuer.

Regardons si la grammaire est LL(1).

Les intersections 2 à 2 des premiers des alternatives de la règle (3) sont vides, donc cette règle n'empêche pas la grammaire d'être LL(1).

Seule la règle (2) peut engendrer des difficultés.
On a :
Premier(<caractère><suite de caractères>)
= Premier(<caractère>)
= {a,...,z,A,...,Z,0,...,9,",@,...}
Suivant(<suite de caractères>) = {'}
Donc : Premier(<caractère><suite de caractères>) ∩ Suivant(<suite de caractères>) = ∅

Ainsi, la grammaire est LL(1).

3.c)

Rappelons les règles de production données dans l'exercice 1 :

(1) <déclaration de constantes> ::= CONST <suite de constantes>
(2) <suite de constantes> ::= <déclaration de constante>;
| <déclaration de constante>;<suite de constantes>
(3) <déclaration de constante> ::= <identificateur> = <valeur numérique>
| <identificateur> = '<caractère>'
| <identificateur> = <constante chaîne>
| <identificateur> : Integer = <constante entière>
| <identificateur> : Real = <constante réelle>
| <identificateur> : Char = '<caractère>'
| <identificateur> : String = <constante chaîne>
| <identificateur> : String[<nombre positif>] =
<constante chaîne>
(4) <valeur numérique> ::= <constante entière> | <constante réelle>
(5) <constante entière> ::= <signe><nombre positif> | <nombre positif>
(6) <signe> ::= + | -
(7) <nombre positif> ::= <chiffre> | <chiffre><nombre positif>
(8) <constante réelle> ::= <constante entière><exposant>
| <constante entière>.<nombre positif><exposant>
(9)<exposant> ::= ^ | E<constante entière>

La règle (2) possède des alternatives commençant par le même symbole. On doit donc la transformer :

(2') <suite de constantes> ::= <déclaration de constante>; <suite1>
(2") <suite1> ::= ^ | <suite de constantes>

La règle (3) possède des alternatives commençant par le même symbole. Transformons-là :
(3') <déclaration de constante> ::= <identificateur><suite2>
(3") <suite2> ::= = <valeur numérique>
| = '<caractère>'
| = <constante chaîne>
| : Integer = <constante entière>
| : Real = <constante réelle>
| : Char = '<caractère>'
| : String = <constante chaîne>
| : String[<nombre positif>] = <constante chaîne>

On doit de nouveau traiter la règle (3") à cause du même problème. On lui substitue les règles suivantes :
(3") <suite2> ::= =<suite3> | :<suite4>
(3'") <suite3> ::= <valeur numérique>
| '<caractère>'
| <constante chaîne>
(3^4) <suite4> ::= Integer = <constante entière>
| Real = <constante réelle>
| Char = '<caractère>'
| String<suite5>
(3^5) <suite5>::= = <constante chaîne>
| [<nombre positif>] = <constante chaîne>

La règle (7) possède des alternatives commençant par le même symbole. On la remplace par les règles suivantes :
(7') <nombre positif> ::= <chiffre><suite6>
(7") <suite6> ::= ^ | <nombre positif>

Pour la même raison, la règle (8) doit être changée.
(8') <constante réelle> ::= <constante entière><suite 7>
(8") <suite7> ::= <exposant> | .<nombre positif><exposant>

Examinons à présent le caractère LL(1) de la grammaire.

Etudions la règle (2") :
Premier(<suite de constantes>) = Premier(<déclaration de constante>)
= Premier(<identificateur>)
= Premier(<lettre>)
= {a,...,z,A,...,Z}
Suivant(<suite1>) = Suivant(<suite de constantes>)
= ∅
Ainsi, Premier(<suite de constantes>) ∩ Suivant(<suite1>) = ∅, donc la règle (2") n'empêche pas la grammaire d'être LL(1).

Etudions la règle (3") :
Premier(=<suite3>) = {=}
Premier(:<suite4>) = {:}

D'où : Premier(=<suite3>) ∩ Premier(:<suite4>) = ∅
Donc la règle (3") n'empêche pas la grammaire d'être LL(1).

Examinons la règle (3'") :
Premier(<valeur numérique>) = Premier(<constante entière>) ∪
Premier(<constante rélle>)
= Premier(<signe>) ∪ Premier(<nombre positif>)
= {+,-} ∪ Premier(<chiffre>)
= {+,-,0,...,9}
Premier('<caractère>') = {'}
Premier(<constante chaîne>) = {'}
Ainsi Premier('<caractère>') ∩ Premier(<constante chaîne>) ≠ ∅.
La grammaire que nous avons proposée n'est donc pas LL(1). Pour rendre celle-ci LL(1), nous supprimons l'alternative '<caractère>' de (3'") (C'est l'alternative <constante chaîne> qui gérera tous les cas) et nous remplaçons l'alternative Char='<caractère>' de (3^4) par Char=<constante chaîne>. C'est l'analyse sémantique qui vérifiera que l'on affecte bien un littéral de 1 caractère à une constante de type caractère.
Examinons la règle (3'") que nous venons de modifier :
Premier(<valeur numérique>) = {+,-,0,...,9}
Premier(<constante chaîne>) = {'}
Premier(<valeur numérique>) ∩ Premier(<constante chaîne>) = ∅
Par conséquent, la règle (3'") n'empêche pas la grammaire d'être LL(1).

Etudions à présent la règle (3^4) :
Premier(Integer = <constante entière>) = {Integer}
Premier(Real = <constante réelle>) = {Real}
Premier(Char = <constante chaîne>) = {Char}
Premier(String<suite5>) = {String}
Les intersections 2 à 2 de ces 4 ensembles sont vides, donc la règle (3^4) n'empêche pas la grammaire d'être LL(1).

Etudions maintenant la règle (3^5) :
Premier(= <constante chaîne>) = {=}
Premier([<nombre positif>] = <constante chaîne>) = {[}
Premier(= <constante chaîne>) ∩ Premier([<nombre positif>] = <constante chaîne>) = ∅, donc la règle (3^5) n'empêche aucunement la grammaire d'être LL(1).

Etudions la règle (4) :
Premier(<constante entière>) = Premier(<signe>) ∪ Premier(<nombre positif>)
= {+,-} ∪ Premier(<chiffre>)
= {+,-,0,...,9}
Premier(<constante réelle>) = Premier(<constante entière>)
= {+,-,0,...,9}
Donc, Premier(<constante entière>) ∩ Premier(<constante réelle>) ≠ ∅.
La grammaire n'est donc pas LL(1).
Afin qu'elle le devienne, on remplace la règle (4) par les règles suivantes :
(4') <valeur numérique> ::= <constante entière><partie réelle>
(4") <partie réelle> ::= ^ | .<nombre positif><exposant> | <exposant>
On doit maintenant étudier la nouvelle règle (4").
On a :
Premier(.<nombre positif><exposant>) = {.}

Premier(<exposant>) = {E}
Suivant(<partie réelle>) = Suivant(<valeur numérique>)
= Suivant(<suite3>)
= Suivant(<suite2>)
= Suivant(<déclaration de constante>)
= {;}

Les intersections 2 à 2 des 3 ensembles précédents sont vides, donc la nouvelle règle (4") n'empêche pas la grammaire d'être LL(1).

Examinons maintenant la règle (5) :
Premier(<signe><nombre positif>) = Premier(<signe>)
= {+,-}
Premier(<nombre positif>) = {0,...,9}
Premier(<signe><nombre positif>) ∩ Premier(<nombre positif>) = ∅, donc la règle (5) n'empêche pas la grammaire d'être LL(1).

De façon évidente, l'intersection des premiers des 2 alternatives de la règle (6) est vide, donc cette règle n'empêche aucunement la grammaire d'être LL(1).

Etudions la règle (7") :
Premier(<nombre positif>) = Premier(<chiffre>)
= {0,...,9}
Suivant(<suite6>) = Suivant(<nombre positif>)
= {]} ∪ Suivant(<constante entière>) ∪ Suivant(<suite6>)
∪ Premier(<exposant>) ∪ Suivant(<suite7>)
= {]} ∪ [Premier(<suite7>) ∪ Suivant(<constante réelle>)
∪ Suivant(<suite4>) ∪ Premier(<partie réelle>)
∪ Suivant(<valeur numérique>)]
∪ Suivant(<nombre positif>) ∪ {E}
∪ Suivant(<constante réelle>)
= {]} ∪ [{E,.} ∪ Suivant(<suite4>)
∪ Suivant(<valeur numérique>)]
= {],E,.} ∪ Suivant(<suite2>) ∪ Suivant(<suite3>)
= {],E,.} ∪ Suivant(<suite2>)
= {],E,.} ∪ Suivant(<déclaration de constante>)
= {],E,.} ∪ {;}
= {], E, ., ;}

Comme Premier(<nombre positif>) ∩ Suivant(<suite6>) = ∅, la règle (7") n'empêche pas la grammaire d'être LL(1).

Examinons la règle (8") :
Premier(<exposant>) = {E}
Premier(.<nombre positif><exposant>) = {.}
Donc Premier(<exposant>) ∩ Premier(.<nombre positif><exposant>) = ∅
Comme l'alternative <exposant> peut engendrer le vide, calculons Suivant(<suite7>) :
Suivant(<suite7>) = Suivant(<constante réelle>)
= Suivant(<suite4>) ∪ Suivant(<valeur numérique>)
= Suivant(<suite2>)
= Suivant(<déclaration de constante>)
= {;}

Ainsi :

Premier(<exposant>) ∩ Suivant(<suite7>) = ∅, et :
Premier(.<nombre positif><exposant>) ∩ Suivant(<suite7>) = ∅
Donc, la règle (8") n'empêche nullement la grammaire d'être LL(1).

Etudions enfin la règle (9) :
Premier(E<constante entière>) = {E}
Suivant(<exposant>) = Suivant(<suite7>) ∪ Suivant(<partie réelle>)
= Suivant(<constante réelle>) ∪
Suivant(<valeur numérique>)
= {;}
Ainsi Premier(E<constante entière>) ∩ Suivant(<exposant>) = ∅
Donc, la règle (9) n'empêche pas la grammaire d'être LL(1).

Conclusion :
En tenant compte des transformations que nous avons indiquées, la grammaire est LL(1).

3.f)

Rappelons les règles de productions que nous avons données au 1.f :

(1) <programme> ::= <titre><ensemble de déclarations><ensemble d'instructions>
(2) <titre> ::= Algo <identificateur> | ^
(3) <ensemble de déclarations>::= Variables <suite de déclarations> | ^
(4) <suite de déclarations> ::= <déclaration> | <déclaration><suite de déclarations>
(5) <déclaration> ::= <suite d'identificateurs> : <type>;
(6) <suite d'identificateurs> ::= <identificateur> | <identificateur>,<suite d'identificateurs>
(7) <type> ::= entier | réel
(8) <ensemble d'instructions>::= Début <suite d'instructions> Fin
(9) <suite d'instructions> ::= <instruction>; | <instruction>;<suite d'instructions>
(10) <instruction> ::= <affectation> | <conditionnelle> | <itération>
| <lecture> | <écriture>
(11) <affectation> ::= <identificateur> ← <EA>
(12) <conditionnelle> ::= Si <condition> alors <suite d'instructions><suite conditionnelle>
(13) <suite conditionnelle> ::= FinSi | Sinon <suite d'instructions> FinSi
(14) <itération> ::= Répéter <suite d'instructions> Jusqu'à <condition>
| Tant que <condition> faire <suite d'instructions> Fin Tant que
| Pour <identificateur> de <EA> à <EA> faire
<suite d'instructions> Fin Pour
(15) <lecture> ::= lire (<suite d'identificateurs>)
(16) <écriture> ::= écrire (<suite d'expressions>)
(17) <suite d'expressions> ::= <EA> | <EA>,<suite d'expressions>
(18) <EA> ::= <EA1> | <EA>+<EA1> | <EA>-<EA1>
(19) <EA1> ::= <EA2> | <EA1>*<EA2> | <EA1>/<EA2>
(20) <EA2> ::= (<EA>) | <signe><opérande>
(21) <opérande> ::= <identificateur> | <entier> | <réel>
(22) <signe> ::= ^ | + | -
(23) <condition> ::= <expression booléenne>
| (<expression booléenne>)
| <expression booléenne><connecteur><expression booléenne>
(24) <expression booléenne>::= <EA><comparaison><EA>
| (<EA><comparaison><EA>)
| non(<condition>)
(25) <connecteur> ::= et | ou

(26) <comparaison> ::= < | > | ≤ | ≥ | = | ≠

Eliminons les règles de productions récursives à gauche et celles commançant par le même symbole avec la méthode vue en cours. On obtient les transformations de règles suivantes :

(4) → (4') <suite de déclarations> ::= <déclaration><suite1>
(4") <suite1> ::= ^ | <suite de déclarations>
(6) → (6') <suite d'identificateurs> ::= <identificateur><suite2>
(6") <suite2> ::= ^ | ,<suite d'identificateurs>
(9) → (9') <suite d'instructions> ::= <instruction>;<suite3>
(9") <suite3> ::= ^ | <suite d'instructions>
(17) → (17') <suite d'expressions> ::= <EA><suite4>
(17") <suite4> ::= ^ | ,<suite d'expressions>
(18) → (18') <EA> ::= <EA1><suite5>
(18") <suite5> ::= ^ | +<EA1><suite5> | -<EA1><suite5>
(19) → (19') <EA1> ::= <EA2><suite6>
(19") <suite6> ::= ^ | *<EA2><suite6> | /<EA2><suite6>
(23) → (23') <condition> ::= <expression booléenne><suite7>
| (<expression booléenne>)
(23") <suite7> ::= ^ | <connecteur><expression booléenne>

Regardons maintenant si la grammaire est LL(1) ou non.

Etude de (2) :
Premier(Algo <identificateur>) = {Algo}
Suivant(<titre>) = Premier(<ensemble de déclarations>) ∪
Premier(<ensemble d'instructions>)
= {Variables} ∪ {Début}
= {Variables,Début}

Premier(Algo <identificateur>) ∩ Suivant(<titre>) = ∅, donc la règle (2) n'empêche pas la grammaire d'être LL(1).

Etude de (3) :
Premier(Variables <suite de déclarations>) = {Variables}
Suivant(<ensemble de déclarations>) = Premier(<ensemble d'instructions>)
= {Début}

Premier(Variables <suite de déclarations>) ∩ Suivant(<ensemble de déclarations>) = ∅, donc la règle (3) n'empêche pas la grammaire d'être LL(1).

Etude de (4") :
Premier(<suite de déclarations>) = Premier(<déclaration>)
= Premier(<suite d'identificateurs>)
= <identificateur>
Suivant(<suite1>) = Suivant(<suite de déclarations>)
= Suivant(<ensemble de déclarations>)
= {Début}

Premier(<suite de déclarations>) ∩ Suivant(<suite1>) = ∅, donc la règle (4") n'empêche pas la grammaire d'être LL(1).

Etude de (6") :
Premier(,<suite d'identificateurs>) = {,}

Suivant(<suite2>) = Suivant(<suite d'identificateurs>)
= {:,)}

Premier(,<suite d'identificateurs>) ∩ Suivant(<suite2>) = ∅, donc la règle (6") n'empêche pas la grammaire d'être LL(1).

L'intersection des premiers des 2 alternatives de la règle (7) est vide, donc cette règle n'empêche aucunement la grammaire d'être LL(1).

Etude de (9") :

Premier(<suite d'instructions>) = Premier(<instruction>)
= Premier(<affectation>) ∪ Premier(<conditionnelle>)
∪ Premier(<itération>) ∪ Premier(<lecture>) ∪
Premier(<écriture>)
= {<identificateur>,Si,Répéter,Tant que,Pour,lire,
écrire}

Suivant(<suite3>) = Suivant(<suite d'instructions>)
= {Fin}∪ Premier(<suite conditionnelle>)
∪ {FinSi}∪ {Jusqu'à,Fin Tant que,FinPour}
= {Fin}∪ {FinSi,Sinon}
∪ {FinSi}∪ {Jusqu'à,Fin Tant que,FinPour}
= {Fin,FinSi,Sinon,Jusqu'à,Fin Tant que,FinPour}

Premier(<suite d'instructions>) ∩ Suivant(<suite3>) = ∅, donc la règle (9") n'empêche pas la grammaire d'être LL(1).

Etude de (10) :

Premier(<affectation>) = {<identificateur>}
Premier(<conditionnelle>) = {Si}
Premier(<itération>) = {Répéter,Tant que,Pour}
Premier(<lecture>) = {lire}
Premier(<écriture>) = {écrire}

Les intersections 2 à 2 des ensembles précédents sont vides, donc cette règle n'empêche pas la grammaire d'être LL(1).

Etude de (13) :

Premier(FinSi) = {FinSi}
Premier(Sinon<suite d'instructions>Finsi) = {Sinon}

Premier(FinSi) ∩ Premier(Sinon<suite d'instructions>Finsi) = ∅, donc la règle (13) n'empêche pas la grammaire d'être LL(1).

Etude de (14) :

Les intersections 2 à 2 des premiers des alternatives de la règle (14) sont clairement vides, donc cette règle n'empêche pas la grammaire d'être LL(1).

Etude de (17") :

Premier(,<suite d'expressions>) = {,}
Suivant(<suite4>) = Suivant(<suite d'expressions>)
= {)}

Premier(,<suite d'expressions>) ∩ Suivant(<suite4>) = ∅, donc la règle (17") n'empêche pas la grammaire d'être LL(1).

Etude de (18") :

Premier(+<EA1><suite5>) = {+}

Premier(-<EA1><suite5>) = {-}
Suivant(<suite5>) = Suivant(<EA>)
= Suivant(<affectation>) ∪ {à} ∪ {faire}
∪ Premier(<suite4>) ∪ Suivant(<suite d'expressions>)
∪ Premier(<comparaison>)
∪ Suivant(<expression booléenne>) ∪ {)}
= Suivant(<instruction>) ∪ {à,faire} ∪ {,} ∪ {)}
∪ {<, >,≤ ,≥,=,≠} ∪ Premier(<connecteur>)
∪ Suivant(<condition>)
= {;} ∪ {à, faire, ',',), <, >, ≤, ≥, =, ≠} ∪ {et,ou} ∪ {alors}
∪ Suivant(<itération>) ∪ {faire}
= {;, à, faire, ',',), <, >, ≤ , ≥, =, ≠, et, ou, alors}

Les intersections 2 à 2 des ensembles précédents sont vides, donc la règle (18") n'empêche pas la grammaire d'être LL(1).

Etude de (19") :
Premier(*<EA2><suite6>) = {*}
Premier(/<EA2><suite6>) = {/}
Suivant(<suite6>) = Suivant(<EA1>)
= Premier(<suite5>) ∪ Suivant(<suite5>) ∪ Suivant(<EA>)
= {+,-} ∪ Suivant(<EA>)

Les intersections 2 à 2 des ensembles précédents sont vides, donc la règle (19") n'empêche pas la grammaire d'être LL(1).

Etude de (20) :
Premier((<EA>)) = {(}
Premier(<signe><opérande>) = Premier(<signe>) ∪ Premier(<opérande>)
= {+,-} ∪ {<identificateur>,<entier>,<réel>}
= {+,-,<identificateur>,<entier>,<réel>}

Premier((<EA>)) ∩ Premier(<signe><opérande>) = ∅, donc la règle (13) n'empêche pas la grammaire d'être LL(1).

Etude de (21) :
Les intersections 2 à 2 des premiers des alternatives de la règle (21) sont vides, donc cette règle n'empêche pas la grammaire d'être LL(1).

Etude de (22) :
Premier(-) = {-}
Premier(+) = {+}
Suivant(<signe>) = Premier(<opérande>)
= {<identificateur>,<entier>,<réel>}

Les intersections 2 à 2 des ensembles ci-dessus sont vides, donc cette règle n'empêche pas la grammaire d'être LL(1).

Etude de (23') :
Premier(<expression booléenne><suite7>) = Premier(<EA>) ∪ {(} ∪ {non}
= Premier(<EA1>) ∪ {(,non}
= {(} ∪ {+,-} ∪ Premier(<opérande>)
∪ {(,non}
= {<identificateur>,<entier>,<réel>,+,-,(,non}
Premier((<expression booléenne>)) = {(}

L'intersection des 2 ensembles ci-dessus est non vide. Pour l'instant, la grammaire n'est pas LL(1).

Le problème apparaît quand on a à analyser des instructions du style :

Si (x>y) alors ...
⇑
expression booléenne

Si (x+1)>y alors..
⇑
expression arithmétique

Pour rendre la grammaire LL(1), on modifie les règles (23') et (24). Elles deviennent :
(23') <condition> ::= [<expression booléenne><suite7>]
(24) <expression booléenne> ::= [<EA><comparaison><EA>]
| non[<expression booléenne>]
La règle (23') ne pose maintenant plus de problème.

Etude de (23") :
Premier(<connecteur><expression booléenne>) = {et,ou}
Suivant(<suite7>) = {]}
Premier(<connecteur><expression booléenne>) ∩ Suivant(<suite7>) = ∅, donc la règle (23") n'empêche pas la grammaire d'être LL(1).

Etude de (24) :
Premier([<EA><comparaison><EA>]) = {[}
Premier(non[<expression booléenne>]) = {non}
Premier([<EA><comparaison><EA>]) ∩ Premier(non[<expression booléenne>]) = ∅, donc la règle (24) n'empêche pas la grammaire d'être LL(1).

De façon évidente les règles (25) et (26) ne posent pas de difficultés.

En conclusion, avec les transformations indiquées, la grammaire est LL(1).

. <u>Exercice 4.</u>

Examinons tout d'abord quelques cas particuliers de récursivité à gauche.

- Double récursivité à gauche :

<R1> ::= <R1><R1>a | b
En effectuant la transformation habituelle, on obtient les nouvelles règles :
<R1> ::= b<R2>
<R2> ::= ^ | <R1>a<R2>
On remarque que <R2> n'est pas récursive à gauche.

- Récursivité à gauche portant sur plusieurs alternatives (ce qui ne correspond pas à la forme canonique de la transformation donnée en cours) :

<R1> ::= <R1>a | <R1>b | c
En effectuant la transformation pour la première alternative, on obtient :
<R1> ::= <R1>b<R2> | c<R2>
<R2> ::= ^ | a<R2>
La règle <R1> est encore récursive à gauche. On applique donc une nouvelle fois la

transformation usuelle et l'on a :

<R1> ::= c<R2><R3>
<R2> ::= ^ | a<R2>
<R3> ::= ^ | b<R2><R3>

Pour le stockage en mémoire des alternatives de la grammaire, on reprend les structures de données introduites à l'exercice 2.

Regardons sur un exemple les modifications à apporter à tab_règles et tab_alter quand on doit transformer une règle de production récursive à gauche.

On considère la règle de production suivante :
<R1> ::= <R1>ab | cd | e
En mémoire, elle est représentée par :

nom	***premiere_alternative***
<R1>	1

tab_règles

Indice de Tab_alter	***symbole***	***symbole_suivant***	***alternative_suivante***
1	<R1>	2	4
2	a	3	4
3	b	0	4
4	c	5	6
5	d	0	6
6	e	0	0

tab_alter

Après transformation, la règle <R1> est remplacée par :
<R1> ::= cd<R2> | e<R2>
<R2> ::= ^ | ab<R2>
On modifie tab_règles et tab_alter pour prendre en compte ces changements :

nom	***premiere_alternative***
<R1>	7
<R2>	12

tab_règles

Indice de Tab_alter	***symbole***	***symbole_suivant***	***alternative_suivante***
1	<R1>	2	4
2	a	3	4
3	b	0	4
4	c	5	6
5	d	0	6
6	e	0	0
7	c	8	10
8	d	9	10
9	<R2>	0	10
10	e	11	0
11	<R2>	0	0
12	^	0	13
13	a	14	0
14	b	15	0
15	<R2>	0	0

tab_alter

On a progressivement inséré les nouvelles alternatives en fin de tab_alter.

Remarque : les éléments de 1 à 6 de tab_alter peuvent être récupérés par un processus de ramasse-miettes (garbage collection).

```
Action Traiter_récursivité
    { On suppose que la grammaire est en mémoire
      suite à l'appel de l'action Stocker_grammaire }
    indice_règle_testée ← 1
    Tant que indice_règle_testée ≤ nb_règles faire
        Si Règle_récursive(indice_règle_testée) alors
            { La variable  indice_alter_récursive est mise à jour
              dans la fonction booléenne Règle_récursive }
            Eliminer_récursivité(indice_règle_testée,indice_alter_récursive)
        sinon
            indice_règle_testée ←  indice_règle_testée+1
        Fin Si
    Fin Tant que
Fin Action

Fonction Règle_récursive(indice_règle_testée) retourne Booléen
    récursive ← faux
    indice ← tab_règles[indice_règle_testée].première_alternative
    Tant que indice ≠ 0 et non(récursive) faire
        Si tab_alter[indice].symbole = tab_règles[indice_règle_testée].nom alors
            récursive ← vrai
```

```
                indice_alter_récursive ← indice
            sinon
                indice ← tab_alter[indice].alternative_suivante
            Fin Si
        Fin Tant que
        Retourner(récursive)
Fin Fonction

Action Eliminer_récursivité(indice_règle_testée,indice_alter_récursive)
        nom_nouvelle_règle ← Créer_nom { Détermine un nom pour la nouvelle règle }

        { Traitement de la première règle }
        { Ajout des alternatives modifiées en fin de tab_alter }
        indice ← tab_règles[indice_règle_testée].première_alternative
        première_modification ← vrai
        Tant que indice ≠ 0 faire
            Si indice ≠ indice_alter_récursive alors

                Si première_modification alors
                    tab_règles[indice_règle_testée].première_alternative ←
                        nb_symboles+1
                    première_modification ← faux
                Fin Si

                { alter_à_suivre est un booléen indiquant s'il reste une alternative
                  à insérer après l'ajout de l'alternative courante modifiée }
                Si tab_alter[indice].alternative_suivante = indice_alter_récursive alors
                  { Sauter l'alternative récursive }
                  alter_à_suivre ←
                      tab_alter[tab_alter[indice].alternative_suivante].alternative_suivante≠0
                sinon
                  alter_à_suivre ← tab_alter[indice].alternative_suivante ≠ 0
                Fin Si

                Insérer_alternative_modifiée(indice,nom_nouvelle_règle,alter_à_suivre)

            FinSi
            indice ← tab_alter[indice].alternative_suivante
        Fin Tant que

        { Création de la nouvelle règle }
        nb_règles← nb_règles+1
        tab_règles[nb_règles].nom ← nom_nouvelle_règle
        nb_symboles ← nb_symboles+1
        tab_règles[nb_règles].première_alternative ← nb_symboles
        tab_alter[nb_symboles].symbole ←  '^'
        tab_alter[nb_symboles].symbole_suivant ← 0
```

```
        { Insertion de l'alternative modifiée dans tab_alter }
        Si tab_alter[indice_alter_récursive].symbole_suivant = 0 alors
                { Cas où la récursivité est sous la forme :
                  <R> ::= α | <R> }
                tab_alter[nb_symboles].alternative_suivante ← 0
        sinon
                tab_alter[nb_symboles].alternative_suivante ← nb_symboles+1
                Insérer_alternative_modifiée(tab_alter[indice_alter_récursive].symbole_suivant,
                                             nom_nouvelle_règle,faux)
        Fin Si
Fin Action

Action Insérer_alternative_modifiée(indice_début_alter,nom_règle,encore_une_alternative)
        indice_nouvelle_alternative ← nb_symboles+1

        { Recopie de l'alternative en fin de tab_alter }
        indice ← indice_début_alter
        Tant que indice ≠ 0 faire
                nb_symboles ← nb_symboles+1
                tab_alter[nb_symboles].symbole ← tab_alter[indice].symbole
                tab_alter[nb_symboles].symbole_suivant ← nb_symboles+1
                tab_alter[nb_symboles].alternative_suivante ← 0
                indice ← tab_alter[indice].symbole_suivant
        Fin Tant que

        { Ajout du symbole correspondant à la nouvelle règle à la fin de l'alternative }
        nb_symboles ← nb_symboles+1
        tab_alter[nb_symboles].symbole ← nom_règle
        tab_alter[nb_symboles].symbole_suivant ← 0
        tab_alter[nb_symboles].alternative_suivante ← 0

        { Mise à jour du champ alternative_suivante pour l'alternative que l'on vient de construire
          dans le cas où celle-ci doit être suivie par d'autre(s) alternative(s) }
        Si encore_une_alternative alors
                Pour indice de indice_nouvelle_alternative à nb_symboles faire
                        tab_alter[indice].alternative_suivante ← nb_symboles+1
                Fin Pour
        Fin Si

Fin Action
```