Corrigé du TD de L2 N°3

Patrick Poulingeas.

- **Exercice 1.**

Grammaire de grammaire :

<grammaire> ::= <règle><suite de règles>
<suite de règles> ::= ^ | <règle><suite de règles>

<règle> ::= '<' <identificateur> '>' '::=' <définition d'une règle> '#'
<définition d'une règle> ::= <alternative><suite d'alternatives>

<alternative> ::= '^' | <symbole><suite de symboles>
<suite d'alternatives> ::= ^ | '|' <alternative><suite d'alternatives>

<symbole> ::= <terminal> | <non terminal>
<suite de symboles> ::= ^ | <symbole><suite de symboles>

<terminal> ::= <identificateur> | <entier> | <caractère spécial>
<caractère spécial> ::= '(' | ')' | ';' | '.' *{ Sans les symboles '<', '^' et '|' }*
<non terminal> ::= '<' <identificateur> '>'

On a :
V_T = {<, >, ::=, #, ^, |, (,), ;, ., <identificateur>, <entier>}

La grammaire est LL(1).

Remarque sur la nécessité de finir l'écriture des règles par un symbole spécial (Ici, le #) :
<R1> ::= <R2> | b#
<R2> ::= ^ | a<R3>#
<R3> ::= c#
Si on ne met pas le #, l'analyseur lexicographique renverra comme unités syntaxiques b puis <R2>. Donc on verra b<R2> comme le début d'une alternative de <R1>.
On rappelle que l'analyseur lexicographique ignore les blancs (dont le passage à la ligne suivante) dans sa recherche d'un début d'unité syntaxique.

Insérons à présent les actions sémantiques dans la grammaire.
Leur rôle est de mettre à jour nb_règles, tab_règles, nb_symboles et tab_alter.

<grammaire> ::= **f1**<règle><suite de règles>
<suite de règles> ::= ^ | <règle><suite de règles>
<règle> ::= '<' <identificateur>**f2** '>' '::=' <définition d'une règle> '#'
<définition d'une règle> ::= <alternative><suite d'alternatives>
<alternative> ::= '^'**f4** | **f6**<symbole><suite de symboles>
<suite d'alternatives> ::= ^ | '|'**f3** <alternative><suite d'alternatives>
<symbole> ::= <terminal>**f4** | <non terminal>
<suite de symboles> ::= ^ | <symbole><suite de symboles>
<terminal> ::= <identificateur> | <entier> | <caractère spécial>

<caractère spécial> ::= '(' | ')' | ';' | '.'
<non terminal> ::= '<' <identificateur>**f5** '>'

Décrivons succintement les fonctions remplies par les diverses actions sémantiques :
f1 : Initialisation
f2 : Stockage de la règle dans tab_règles
f3 : Mise à jour du champ alternative_suivante dans tab_alter
f4 : Ajout d'un symbole terminal ou de '^' dans tab_alter
f5 : Ajout d'un symbole non-terminal dans tab_alter (Traitement spécial pour insérer les caractères '<' et '>')
f6 : Mise à jour de certaines variables au début d'une alternative

Ecrivons maintenant les différentes actions sémantiques sous forme algorithmique :

```
Action f1
    nb_règles ← 0
    nb_symboles ← 0
FinAction

Action f2
    { On vérifie tout d'abord que la règle n'est pas déjà stockée en mémoire
      (On s'interdit de mettre les alternatives sur plusieurs lignes) }
    indice ← 0
    erreur ← faux
    Tant que indice ≤ nb_règles et non(erreur) faire
        Si tab_règles[indice].nom = '<' + US0 + '>' alors      { US0 désigne l'US précédente }
            erreur ← vrai
        sinon
            indice ← indice+1
        FinSi
    FinTantque
    Si non(erreur) alors
        nb_règles ← nb_règles+1
        tab_règles[nb_règles].nom ← '<' + US0 + '>'
        tab_règles[nb_règles].première_alternative ← nb_symboles+1
    sinon
        Erreur ('<'+US0+'> : Règle déjà déclarée')
    FinSi
FinAction

Action f3
    Pour indice de indice_début_alternative à nb_symboles faire
        tab_alter[indice].alternative_suivante ← nb_symboles+1
    FinPour
FinAction

Action f4
    nb_symboles ← nb_symboles+1
    tab_alter[nb_symboles].symbole ← US0
    tab_alter[nb_symboles].symbole_suivant ← 0
    tab_alter[nb_symboles].alternative_suivante ← 0
```

```
        Si non(premier_symbole) alors
                tab_alter[nb_symboles-1].symbole_suivant ← nb_symboles
        sinon
                premier_symbole ← faux
        FinSi
FinAction

Action f5
        nb_symboles ← nb_symboles+1
        tab_alter[nb_symboles].symbole ← '<'+US0+'>'
        tab_alter[nb_symboles].symbole_suivant ← 0
        tab_alter[nb_symboles].alternative_suivante ← 0
        Si non(premier_symbole) alors
                tab_alter[nb_symboles-1].symbole_suivant ← nb_symboles
        sinon
                premier_symbole ← faux
        FinSi
FinAction

Action f6
        premier_symbole ← vrai
        indice_début_alternative ← nb_symboles+1
FinAction
```

- **Exercice 2.**

a)

Exemple de déclarations :

```
entier b;
entier c,d,a[20],e;
```

On suppose que l'on ne peut pas initialiser les variables au moment de leur déclaration. On se limitera par ailleurs à des tableaux de dimension 1.

Grammaire pour les déclarations du langage LSP :

```
<liste de déclarations>     ::=   <déclaration><suite de déclarations>
<suite de déclarations>     ::=   ^ | <déclaration><suite de déclarations>
<déclaration>               ::=   entier<variable><suite de variables>;
<suite de variables>        ::=   ^ | ,<variable><suite de variables>
<variable>                  ::=   <identificateur><dimension>
<dimension>                 ::=   ^ | [<entier sans signe>]
```

Avec les exemples donnés précédemment, on obtient la table des identificateurs suivante :

Ident	***Type***	***Taille***	***NbElements***	***EmplRel***
b	entier	2	1	0
c	entier	2	1	2
d	entier	2	1	4
a	entier	2	20	6
e	entier	2	1	46

Les valeurs des identificateurs (i.e. dans le langage LSP des variables) sont stockées en mémoire centrale (MC) à partir de l'adresse DébutIdent. Le champ EmplRel indique la translation à effectuer à partir de DébutIdent pour obtenir l'adresse où est stocké un identificateur.

Insérons maintenant dans notre grammaire des actions sémantiques qui vont mettre à jour la table des identificateurs.

<liste de déclarations> ::= **f1**<déclaration><suite de déclarations>
<suite de déclarations> ::= ^ | <déclaration><suite de déclarations>
<déclaration> ::= entier<variable><suite de variables>;
<suite de variables> ::= ^ | ,<variable><suite de variables>
<variable> ::= <identificateur>**f2**<dimension>
<dimension> ::= ^ | [<entier sans signe>**f3**]

```
Action f1
        { Initialisations }
        AdrRelative ← 0
        nb_identificateurs ← 0
FinAction

Action f2
        Indice ← ChercherTable(Ident,US0)
        { On vérifie que l'identificateur n'a pas déjà été déclaré }
        Si Indice ≠ 0 alors
                Erreur (US0+' a dèjà été déclaré')
        sinon
                Indice ← RangerTable(Ident,US0)
                Type[Indice] ← 'entier'
                Taille[Indice] ← 2
                NbElements[Indice] ← 1
                EmplRel[Indice] ← AdrRelative
                AdrRelative ← AdrRelative+2
        FinSi
FinAction

Action f3
        NbElements[Indice] ← ConversionNumérique(US0)
        AdrRelative ← AdrRelative+Taille[Indice]*(NbElements[Indice]-1)
FinAction
```

Donnons à présent des algorithmes pouvant correspondre aux fonctions ChercherTable et RangerTable.

```
Fonction ChercherTable(Ident,Chaîne) retourne entier
        Si nb_identificateurs ≠ 0 alors
                Pour indice de 1 à nb_identificateurs faire
                        Si Ident[indice] = Chaîne alors
                                retourner (indice)
                        FinSi
                FinPour
```

```
        FinSi
        retourner (0)
FinFonction

Fonction RangerTable(Ident,Chaîne) retourne entier
        { précondition : on range un nouvel identificateur }
        nb_identificateurs ← nb_identificateurs+1
        Ident[nb_identificateurs] ← Chaîne
        retourner(nb_identificateurs)
FinFonction
```

Remarques :

- On ne génère pas de code machine TAC ou de code intermédiaire postfixé. On se contente de gérer la table des identificateurs.
- Pour les actions ChercherTable et RangerTable, on pourrait utiliser plusieurs tables et une fonction de h-code.

b)

* Etudions l'instruction d'affectation (ainsi que l'évaluation d'une expression arithmétique).

On propose la grammaire suivante qui tient compte de la priorité usuelle des opérateurs arithmétiques :

<instruction d'affectation>	::=	<identificateur>**f1**<indice tableau>**f2** ← <EA>**f3**
<indice tableau>	::=	^ | [<entier>**f4**]
<EA>	::=	<EA1><suite1>
<suite1>	::=	^ | +<EA1>**f5**<suite1> | -<EA1>**f6**<suite1>
<EA1>	::=	<EA2><suite2>
<suite2>	::=	^ | *<EA2>**f7**<suite2> | /<EA2>**f8**<suite2>
<EA2>	::=	(<EA>) | <opérande>
<opérande>	::=	<identificateur>**f1**<indice tableau>**f2** | <entier>**f9**

On s'est limité pour les indices des tableaux à des littéraux entiers, alors qu'en toute généralité les indices pourraient être des expressions arithmétiques.

Ecrivons à présent les actions sémantiques.

Remarque :

On dispose de plusieurs types dans le langage : le type entier et des types tableau d'entiers (de diverses tailles).

Ainsi, avec la déclaration suivante :

```
entier a,b[3],c[7],d[7];
```

a est de type entier,
b est de type tableau d'entiers de taille 3,
c est de type tableau d'entiers de taille 7,
d est de type tableau d'entiers de taille 7.

Aussi, les affectations suivantes :

```
a ← b;
b ← c;
c ← b;
```

sont toutes sémantiquement incorrectes (dans un langage de typage fort style Ada), mais :

```
c ← d;
c ← c+d;
```

sont toutes correctes.

Pour simplifier les choses, nous allons interdire l'utilisation d'un tableau dans sa globalité (Ceci devra être vérifié par les actions sémantiques). Avec cette convention, tout se passe comme s'il n'y avait plus qu'un seul type intervenant dans les instructions du langage : le type entier.

Commençons par les actions sémantiques engendrant du code postfixé.

```
Action f1
        Indice ← ChercherTable(Ident,US0)
        Si Indice = 0 alors
                Erreur (US0+' n"a pas été déclaré')
        FinSi
        Ranger(Arbre,US0)
        Si NbElements[Indice] > 1 alors
                { On a affaire à un tableau }
                { On ignore le cas d'un tableau à 1 élément }
                IndiceTableauAttendu ← vrai
        sinon
                IndiceTableauAttendu ← faux
        FinSi
FinAction

Action f2
        Si IndiceTableauAttendu alors
                Erreur ('Impossible de manipuler un tableau dans son intégralité')
        FinSi
FinAction

Action f3
        Ranger(Arbre,'←')
FinAction

Action f4
        Si  IndiceTableauAttendu alors
                Ranger(Arbre,US0)
                { Remarque : On ne teste pas si l'indice du tableau est valide }
                Ranger(Arbre,'#IT#')  { L'opérateur #IT# fournit l'emplacement de l'élément du
                                        tableau dont l'indice est rangé sous forme de code postfixé
                                        juste avant cet opérateur }
                Ranger(Arbre,'#T#')   { L'operateur #T# est l'opérateur de tableau }
                { Ici, comme nous n'avons que des tableaux à une seule dimension, on pourrait
                  n'introduire qu'un seul opérateur pour le code postfixé généralisé }
                IndiceTableauAttendu ← faux
        sinon
                Erreur('La variable n"est pas un tableau')
        FinSi
FinAction

Action f5
        Ranger(Arbre,'+')
FinAction
```

```
Action f6
        Ranger(Arbre,'-')
FinAction

Action f7
        Ranger(Arbre,'*')
FinAction

Action f8
        Ranger(Arbre,'/')
FinAction

Action f9
        Indice ← ChercherTable(Const,US0)
        Si Indice = 0 alors
                { C'est la première fois que l'on rencontre cette constante dans le source }
                { On la range dans la table des constantes }
                RangerTable(Const,US0)
        FinSi
        Ranger(Arbre,US0)
FinAction
```

Ecrivons maintenant des actions sémantiques générant du code machine TAC.

Le code TAC est produit dans les actions Engendrer qui se veulent assez générales pour pouvoir être implémentées avec n'importe quel code machine.

```
Action f1
        Indice ← ChercherTable(Ident,US0)
        Si Indice = 0 alors
                Erreur (US0+' n"a pas été déclaré')
        FinSi
        Engendrer(Empiler(PileEv,DébutIdent+EmplRel[Indice]))
        Si NbElements[Indice] > 1 alors
                IndiceTableauAttendu ← vrai
        sinon
                IndiceTableauAttendu ← faux
        FinSi
FinAction

Action Engendrer(Empiler(PileEv,X))
        { La variable AdresseCourante indique l'adresse en mémoire centrale (MC) de la dernière
          instruction du code TAC produit par le compilateur }
        { La variable SommetPile (initialisée à 9995 par le compilateur) contient l'adresse
          du dernier élément de la pile PileEv }
        MC[AdresseCourante+1] ← 010000+SommetPile  { AQ ← MC[SommetPile] }
        MC[AdresseCourante+2] ← 130001             { AQ ← AQ+1 }
        MC[AdresseCourante+3] ← 020000+SommetPile  { MC[SommetPile] ← AQ }
        MC[AdresseCourante+4] ← 120000+X           { AQ ← X }
        MC[AdresseCourante+5] ← 180000+SommetPile  { MC[MC[SommetPile]] ← AQ }
        AdresseCourante ←  AdresseCourante+5
FinAction
```

```
Action f2
        Si IndiceTableauAttendu alors
                Erreur ('Impossible de manipuler un tableau dans son intégralité')
        FinSi
FinAction

Action f4
        Si IndiceTableauAttendu alors
                IndiceTableau ← ConversionNumérique(US0)
                { Remarque : On ne teste pas si l'indice du tableau est valide }
                Engendrer(Dépiler(PileEv,Adresse1))        { MC[Adresse1] est
                                                             l'adresse de début du tableau }
                Engendrer(MC[Adresse1] ← MC[Adresse1]+(IndiceTableau-1)*2)
                Engendrer(Empiler(PileEv,MC[Adresse1])
                IndiceTableauAttendu ← faux
        sinon
                Erreur('La variable n"est pas un tableau')
        FinSi
FinAction

Action Engendrer(Dépiler(PileEv,AdresseMC))
        MC[AdresseCourante+1] ← 170000+SommetPile  { AQ ← MC[MC[SommetPile]] }
        MC[AdresseCourante+2] ← 020000+AdresseMC   { MC[AdresseMC] ← AQ }
        MC[AdresseCourante+3] ← 010000+SommetPile  { AQ ← MC[SommetPile] }
        MC[AdresseCourante+4] ← 140001             { AQ ← AQ-1}
        MC[AdresseCourante+5] ← 020000+SommetPile  { MC[SommetPile] ← AQ }
        AdresseCourante ←  AdresseCourante+5
FinAction
```

N.B. Désormais, on se contentera parfois d'écrire dans les actions sémantiques simplement la signification du code TAC qui sera stocké en mémoire. On utilisera pour cela l'action Stocker_en_mémoire qui gèrera AdresseCourante et traduira en code machine TAC les paramètres qu'on lui fournira.

```
Action Engendrer( MC[X] ← MC[X]+(Y-1)*2)
         Stocker_en_mémoire (
                AQ ← 2,
                MC[Adresse] ← AQ,              { Adresse (=9999) est une adresse réservée }
                AQ ← Y,
                AQ ← AQ-1,
                AQ ← AQ*MC[Adresse],
                AQ ← AQ+MC[X],
                MC[X] ← AQ )
FinAction

Action Engendrer(Empiler(PileEv,MC[X])
         Stocker_en_mémoire (
                AQ ← MC[SommetPile],
                AQ ← AQ+1,
                MC[SommetPile] ← AQ,
                AQ ← MC[X],
```

```
                MC[MC[SommetPile]] ← AQ )
FinAction

Action f3
        Engendrer(Dépiler(PileEv,Adresse2))
        Engendrer(Dépiler(PileEv,Adresse1))
        { L'entier (résultat de l'évaluation de l'expression arithmétique) dont l'adresse
          est stockée dans Adresse2 est copié à l'adresse contenue dans Adresse1 }
        Engendrer(Affectation_entière(MC[MC[Adresse1]],MC[MC[Adresse2]])
FinAction

Action Engendrer(Affectation_entière(MC[MC[X]],MC[MC[Y]])
        { Attention ! Les entiers occupent deux cases mémoire de MC }
        Stocker_en_mémoire (
                { Recopie de la première case mémoire contenant l'entier }
                AQ ← MC[MC[Y]],
                MC[MC[X]] ← AQ,

                { Recopie de la seconde case mémoire contenant l'entier }
                { On incrémente les adresses contenues dans MC[X] et MC[Y] }
                AQ ← MC[X],
                AQ ← AQ+1,
                MC[X] ← AQ,
                AQ ← MC[Y],
                AQ ← AQ+1,
                MC[Y] ← AQ,
                { On fait la copie }
                AQ ← MC[MC[Y]],
                MC[MC[X]] ← AQ,
                { On remet MC[X] et MC[Y] à leurs valeurs initiales }
                AQ ← MC[X],
                AQ ← AQ-1,
                MC[X] ← AQ,
                AQ ← MC[Y],
                AQ ← AQ-1,
                MC[Y] ← AQ )
FinAction

Action f5
        Engendrer(Dépiler(PileEv,Adresse2))
        Engendrer(Dépiler(PileEv,Adresse1))
        { IndiceTravail indique la première case mémoire disponible dans la mémoire de
          travail (mémoire servant au stockage des valeurs intermédiaires) }
        Engendrer(MC[MC[IndiceTravail]] ← Somme_entière(MC[MC[Adresse1]],
                                                        MC[MC[Adresse2]])
        Engendrer(Empiler(PileEv,MC[IndiceTravail])
        Engendrer(MC[IndiceTravail] ← MC[IndiceTravail]+2)    { 2 est, par convention,
                                                                la taille d'un entier }
FinAction
```

```
Action Engendrer(MC[MC[Z]] ← Somme_entière(MC[MC[X]],MC[MC[Y]])
        Stocker_en_mémoire (
                { On rappelle que les entiers occupent deux cases mémoire de MC }
                { On suppose que la première case mémoire est la moins significative
                  et la seconde la plus significative }
                { On suit donc la convention little-endian }

                { Somme des premières cases mémoire contenant l'entier }
                AQ ← MC[MC[X]],
                MC[Adresse] ← AQ,                { Adresse (=9999) est une adresse réservée }
                AQ ← MC[MC[Y]],
                AQ ← AQ+MC[Adresse],
                MC[MC[Z]] ← AQ,

                { Somme des premières cases mémoire contenant l'entier en tenant compte
                  d'une retenue éventuelle engendrée par la première somme }
                { On stocke dans MC[Adresse] le résultat de la somme }
                MC[Adresse] ← Retenue,           { Retenue est égale à 0 ou 1 }
                { On récupère la valeur MC[MC[X]+1] et on l'ajoute à MC[Adresse]}
                AQ ← MC[X],
                AQ ← AQ+1,
                MC[AdresseRéservée] ← AQ,        { AdresseRéservée = 9994 }
                AQ ← MC[MC[AdresseRéservée]],
                AQ ← AQ+MC[Adresse],
                MC[Adresse] ← AQ,
                { On récupère la valeur MC[MC[Y]+1] et on l'ajoute à MC[Adresse]}
                AQ ← MC[Y],
                AQ ← AQ+1,
                MC[AdresseRéservée] ← AQ,
                AQ ← MC[MC[AdresseRéservée]],
                AQ ← AQ+MC[Adresse],
                MC[Adresse] ← AQ,
                { On sauvegarde finalement la valeur de MC[Adresse] en MC[Z]+1 }
                AQ ← MC[Z],
                AQ ← AQ+1,
                MC[AdresseRéservée] ← AQ,
                AQ ← MC[Adresse],
                MC[MC[Adresse_réservée] ← AQ )
FinAction
```

Les actions f6, f7 et f8 sont similaires à f5. On ne les détaillera pas.

```
Action f9
        Indice ← ChercherTable(Const,US0)
        Si Indice = 0 alors
                { C'est la première fois que l'on rencontre cette constante entière
                  dans le source du programme à compiler }
                { La fonction RangerTable(Const,US0) calcule la valeur numérique
                  de la constante placée dans US0, met à jour la table des constantes,
                  stocke la constante dans l'emplacement mémoire réservé aux constantes en MC
```

```
                    et renvoie finalement l'indice de la constante dans la table des constantes }
                Indice ← RangerTable(Const,US0)
        FinSi
        Engendrer(Empiler(PileEv,DébutConst+EmplRelC[Indice]))
FinAction
```

Remarque : Après avoir évalué l'expression arithmétique, on peut supprimer de la mémoire de travail les résultats des calculs intermédiaires effectués lors de l'évaluation. Pour cela, on ajoute à la fin de l'action de f3 :

Engendrer(MC[IndiceTravail] ← MemTravail)

* Etudions à présent la structure conditionnelle Si/alors/sinon/FinSi.

On propose la grammaire suivante :

<instruction conditionelle> ::= Si <condition> **f1** alors <liste d'instructions> <fin instruction conditionnelle>

<fin instruction conditionnelle> ::= FinSi **f3** | sinon **f2** <liste d'instructions> FinSi **f3**

Remarque : Nous écrirons ensuite une grammaire avec des actions sémantiques pour la gestion d'une condition.

Ecrivons maintenant les actions sémantiques qui produisent du code postfixé.

```
Action f1
        Ranger(Arbre,'#BF#')
        Empiler(PileSi,IndArbre)      { IndArbre est une fonction renvoyant la position où
                                        sera inséré le prochain élément dans l'arbre sémantique }
        Ranger(Arbre,0)
FinAction

Action f2
        Dépiler(PileSi,PositionArbre)
        Ranger(Arbre,'#B#')
        Empiler(PileSi,IndArbre)
        Ranger(Arbre,0)
        Arbre[PositionArbre] ← IndArbre
FinAction

Action f3
        Dépiler(PileSi,PositionArbre)
        Arbre[PositionArbre] ← IndArbre
FinAction
```

Ecrivons maintenant des actions sémantiques engendrant du code machine TAC.

```
Action f1
        Engendrer(Dépiler(PileEv,Adresse))          { Adresse = 9999 }
        Engendrer(Si non(MC[MC[Adresse]]) alors aller en 0)
        Empiler(PileSi,AdresseCourante)             { AdresseCourante est l'adresse de la
                                                      dernière instruction TAC actuellement
                                                      générée par le compilateur }
FinAction
```

```
Action Engendrer(Si non(MC[MC[X]]) alors aller en 0)
        MC[AdresseCourante+1] ← 170000+X      { AQ ← MC[MC[X]] }
        MC[AdresseCourante+2] ← 040000        { Si AQ = 0 alors CO ← 0000 }
                                              { CO est le compteur ordinal. CO contient
                                                 l'adresse de la prochaine instruction
                                                 à exécuter }
         AdresseCourante ← AdresseCourante+2
FinAction

Action f2
        Dépiler(PileSi,Adresse_en_MC)         { Adresse_en_MC est une variable utilisée
                                                 lors de la compilation }
        Engendrer(Aller en 0)
        Empiler(PileSi,AdresseCourante)
        Partie_Adresse(MC[Adresse_en_MC]) ← AdresseCourante+1
FinAction

Action Engendrer(Aller en 0)
        MC[AdresseCourante+1] ← 030000        { CO ← 0000 }
        AdresseCourante ← AdresseCourante+1
FinAction

Action f3
        Dépiler(PileSi,Adresse_en_MC)
        Partie_Adresse(MC[Adresse_en_MC]) ← AdresseCourante+1
FinAction
```

Traitons à présent la génération de code pour une condition.

Nous utiliserons la grammaire suivante qui introduit des priorités entre les opérateurs logiques et arithmétiques :

<condition> ::= <condition1> | <condition> ou <condition1> **f1**
<condition1> ::= <condition2> | <condition1> et <condition2> **f2**
<condition2> ::= <condition3> | non[<condition3>] **f3**
<condition3> ::= [<condition>] | <relation>
<relation> ::= <EA><fin relation>
<fin relation> ::= = <EA> **f4** | ≠ <EA> **f5** | ≤ <EA> **f6** | ≥ <EA> **f7** | < <EA> **f8** | > <EA> **f9**

Remarque : Cette grammaire n'est pas LL(1), mais pour l'insertion et le codage des actions sémantiques ceci est de peu d'importance. Il faudra bien sûr ensuite rendre la grammaire LL(1) en “transportant” les actions sémantiques quand on fera des modifications dans les règles et les alternatives.

Ecrivons les actions sémantiques engendrant du code intermédiaire postfixé.

```
Action f1
        Ranger(Arbre,'ou')
FinAction
```

Les actions f2 à f9 sont similaires (seul l'opérateur inséré dans l'arbre sémantique change).

Ecrivons à présent les actions sémantiques engendrant du code machine TAC.

```
Action f1
        Engendrer(Dépiler(PileEv,Adresse2))
        Engendrer(Dépiler(PileEv,Adresse1))
        Engendrer(MC[MC[IndiceTravail]] ← MC[MC[Adresse1]] ou MC[MC[Adresse2]])
        Engendrer(Empiler(PileEv,MC[IndiceTravail]))
        Engendrer(MC[IndiceTravail] ← MC[IndiceTravail]+Taillelogique)
        { Taillelogique = 1 car on suppose qu'une valeur booléenne est stockée
          dans une seule case mémoire }
FinAction

Action Engendrer(MC[MC[Z]] ← MC[MC[X]] ou MC[MC[Y]])
        { Convention pour les valeurs booléennes :        0  ↔ faux
                                                          entier ≠ 0 ↔ vrai }
        Stocker_en_mémoire (
                AQ ← MC[MC[X]],
                Si AQ = 0 alors CO ← Prendre_seconde_condition,
                MC[MC[Z]] ← AQ,          { On stocke la valeur "vrai" à l'adresse MC[Z] }
                CO ← Fin,
        Label Prendre_seconde_condition,
                AQ ← MC[MC[Y]],
                MC[MC[Z]] ← AQ,
        Label Fin)
FinAction
```

Remarque : La sémantique choisie pour le "ou" est celle d'une évaluation paresseuse.

```
Action f2
        Engendrer(Dépiler(PileEv,Adresse2))
        Engendrer(Dépiler(PileEv,Adresse1))
        Engendrer(MC[MC[IndiceTravail]] ← MC[MC[Adresse1]] et MC[MC[Adresse2]])
        Engendrer(Empiler(PileEv,MC[IndiceTravail]))
        Engendrer(MC[IndiceTravail] ← MC[IndiceTravail]+Taillelogique)
FinAction

Action Engendrer(MC[MC[Z]] ← MC[MC[X]] et MC[MC[Y]])
        Stocker_en_mémoire (
                AQ ← MC[MC[X]],
                Si AQ = 0 alors CO ← Fin,
                AQ ← MC[MC[Y]],
        Label Fin,
                MC[MC[Z]] ← AQ)
FinAction

Action f3
        Engendrer(Dépiler(PileEv,Adresse))
        Engendrer(MC[MC[IndiceTravail]] ← non(MC[MC[Adresse]]))
        Engendrer(Empiler(PileEv,MC[IndiceTravail]))
        Engendrer(MC[IndiceTravail] ← MC[IndiceTravail]+Taillelogique)
FinAction
```

```
Action Engendrer(MC[MC[Y]] ← non(MC[MC[X]]))
        Stocker_en_mémoire (
                AQ ← MC[MC[X]],
                Si AQ = 0 alors CO ← Passer_à_vrai,
                AQ ← 0,
                CO ← Fin
        Label Passer_à_vrai,
                AQ ← 1,
        Label Fin,
                MC[MC[Y]] ← AQ)
FinAction

Action f4
        Engendrer(Dépiler(PileEv,Adresse2))
        Engendrer(Dépiler(PileEv,Adresse1))
        Engendrer(Si MC[MC[Adresse1]] = MC[MC[Adresse2]] alors
                        MC[MC[IndiceTravail]] ← vrai
                    sinon
                        MC[MC[IndiceTravail]] ← faux
                    FinSi)
        Engendrer(Empiler(PileEv,MC[IndiceTravail]))
        Engendrer(MC[IndiceTravail] ← MC[IndiceTravail]+Taillelogique)
FinAction

Action Engendrer(Si MC[MC[X]] = MC[MC[Y]] alors
                        MC[MC[Z]] ← vrai
                    sinon
                        MC[MC[Z]] ← faux
                    FinSi)
        Stocker_en_mémoire (
                AQ ← MC[MC[X]],
                MC[Adresse] ← AQ,
                AQ ← MC[MC[Y]],
                Si AQ > MC[Adresse] alors AQ ← AQ-MC[Adresse]
                                    sinon AQ ← 0,
                Si AQ = 0 alors CO ← Continuer_test,
                AQ ← 0,
                MC[MC[Z]] ← AQ,
                CO ← Fin,
        Label Continuer_test,
                AQ ← MC[MC[Y]],
                MC[Adresse] ← AQ,
                AQ ← MC[MC[X]],
                Si AQ > MC[Adresse] alors AQ ← AQ-MC[Adresse]
                                    sinon AQ ← 0,
                Si AQ = 0 alors CO ← Egalité,
                AQ ← 0,
                MC[MC[Z]] ← AQ,
                CO ← Fin,
```

```
        Label Egalité,
                AQ ← 1,
                MC[MC[Z]] ← AQ,
        Label Fin)
FinAction
```

On a des actions sémantiques similaires à f4 pour f5, f6, f7, f8 et f9.

* Etudions maintenant l'instruction itérative Tantque.

On propose la grammaire suivante :

<instruction itérative> ::= Tantque **f1** <condition> faire **f2** <liste d'instructions> FinTantque **f3**

Ecrivons les actions sémantiques engendrant du code intermédiaire postfixé.

```
Action f1
        Empiler(PileTQ,IndArbre)
FinAction

Action f2
        Ranger(Arbre,'#BF#')
        Empiler(PileTQ,IndArbre)
        Ranger(Arbre,0)
FinAction

Action f3
        Ranger(Arbre,'#B#')
        Dépiler(PileTQ,PositionArbre2)
        Dépiler(PileTQ,PositionArbre1)
        Ranger(Arbre, PositionArbre1)
        Arbre[PositionArbre2] ← IndArbre
FinAction
```

Ecrivons à présent les actions sémantiques produisant du code machine TAC.

```
Action f1
        Empiler(PileTQ,AdresseCourante)
FinAction

Action f2
        Engendrer(Dépiler(PileEv,Adresse))
        Engendrer(Si non (MC[MC[Adresse]]) alors aller en 0)
        Empiler(PileTQ,AdresseCourante)
FinAction

Action f3
        Dépiler(PileTQ,Adresse2)
        Dépiler(PileTQ,Adresse1)
        Engendrer(Aller à Adresse1+1)
        PartieAdresse(MC[Adresse2]) ← AdresseCourante+1
FinAction
```

* Examinons l'instruction de choix pour le langage LSP.

Nous proposons les règles de production suivantes :

<instruction de choix> ::= Choix <identificateur> **f1** selon <liste de choix><fin choix>
<liste de choix> ::= <choix><suite de choix>
<suite de choix> ::= ^ | FinCas **f3** <liste de choix>
<choix> ::= <entier> **f2** : <liste d'instructions>
<fin choix> ::= Finchoix **f4** | autrechoix <liste d'instructions> Finchoix **f4**

Remarque : Dans la partie droite de la règle <choix>, on aurait pu remplacer <entier> par <EA> (et même remplacer <identificateur> par <EA> dans la première règle de production).

Ecrivons maintenant les actions sémantiques générant du code postfixé.

```
Action f1
        Empiler(PileVarChoix,US0)
        Empiler(PileNbChoix,0)
FinAction

Action f2
        Dépiler(PileVarChoix,variable)
        Ranger(Arbre,variable)
        Empiler(PileVarChoix,variable)
        Ranger(Arbre,US0)
        Ranger(Arbre,'=')
        Ranger(Arbre,'#BF#')
        Empiler(PileChoix,IndArbre)
        Ranger(Arbre,0)
FinAction

Action f3
        Ranger(Arbre,'#B#')
        Empiler(PileVersFinChoix,IndArbre)
        Dépiler(PileNbChoix,nb_choix)
        nb_choix ← nb_choix+1
        Empiler(PileNbChoix,nb_choix)
        Ranger(Arbre,0)
        Dépiler(PileChoix,PositionArbre)
        Arbre[PositionArbre] ← IndArbre
FinAction

Action f4
        Dépiler(PileNbChoix,nb_choix)
        { Remarque : nb_choix ≥ 1 }
        Pour indice de nb_choix à 1 faire
                Dépiler(PileVersFinChoix,PositionArbre)
                Arbre[PositionArbre] ← IndArbre
        FinPour
        { Nettoyage }
        Dépiler(PileVarChoix,variable)
FinAction
```