Analyse sémantique: Traduction dirigée par la syntaxe

Les aspects lexicaux et syntaxiques d'un langage ne sont qu'*un support pour l'essentiel*, à savoir la sémantique véhiculée par les phrases du langage.

Par exemple, le fragment de code C++ :

```
int i = 2.567;
```

est correct lexicalement et syntaxiquement, mais il contient une erreur sémantique. On ne peut en effet affecter une valeur flottante à une variable entière dans ce langage. C'est au niveau de l'analyse sémantique qu'on détecte ce type d'erreur.

L'analyse sémantique se base sur la définition du langage à compiler, qui précise quelles phrases bien formées syntaxiquement ont un sens. Elle s'appuie sur des structures de données représentant le code source en cours de compilation. On notera qu'il n'est pas strictement nécessaire de s'appuyer sur une forme source textuelle : il est tout à fait possible de faire l'analyse sémantique d'informations stockées dans des structures de données, sans devoir passer par analyse lexicale et analyse syntaxique d'un texte. L'analyse sémantique effectue les vérifications de sémantique, c'est-à-dire de signification, sur le code source en cours de compilation.

Il s'agit d'ajouter des séquences de traitement qui sont exécutées au cours de l'analyse syntaxique pour effectuer des vérifications sémantiques et construire progressivement un code intermédiaire qui correspond au programme. Cette phase se base sur une grammaire appelée grammaire sémantique (liée à la grammaire de l'analyse syntaxique) avec une analyse soit ascendante soit descendante. Le traitement sémantique est le même dans les 2 types d'analyse, la différence se fait au niveau de la transformation de la grammaire lorsqu'on ajoute les appels aux routines sémantiques. Ces routines sémantiques sont des procédures qui effectuent les traitements sémantiques.

Vérifications sémantiques:

Lors de l'analyse sémantique on a plusieurs taches:

- Donner un sens aux entités syntaxiques utilisées,
- Génération automatique de la forme intermédiaire du code source,
- Remplissage d'informations de la table de symboles,
- Vérification des déclarations des variables (double déclaration,....)
- Vérification des déclarations des étiquettes (double déclaration, réferencées,
- Vérification dans l'utilisation des tableaux,
- Vérification de la comptabilité des types lors de l'affectation par ex,
- Vérification du type de certaines variables ou de certaines expressions, par exemple l'instruction for i:=expr to ....
- Remplissage des informations des goto vers une position antérieure
- traitement du goto vers position ultérieure.

-etc....

Erreurs sémantiques:

Cette phase peut échouer en détectant les erreurs sémantiques. Les erreurs sémantiques qu'on peut détecter sont:

- Incompatibilité de type lors des affectations, comparaison, etc..
- Var utilisée mais non déclarée (lorsque c'est obligatoire)
- étiquette utilisée mais non déclarée (lorsque c'est obligatoire)
- etc....

Transformation de la grammaire:

Soit une règle $A \rightarrow \alpha\beta$, supposons qu'on veuille effectuer un traitement sémantique entre α et β. Pour cela on doit transformer cette règle suivant le mode d'analyse, on rajoute un nouveau non terminal B pour appeler la routine:

Méthode ascendante: on transforme cette règle en

$A \rightarrow B\beta$

$B \rightarrow \alpha$

et lorsqu'on réduit α en B on exécute la routine sémantique nécessaire.

Méthode descendante: on transforme cette règle en

$A \rightarrow \alpha B\beta$

$B \rightarrow \epsilon$

et lorsqu'on appelle B on exécute la routine sémantique nécessaire.

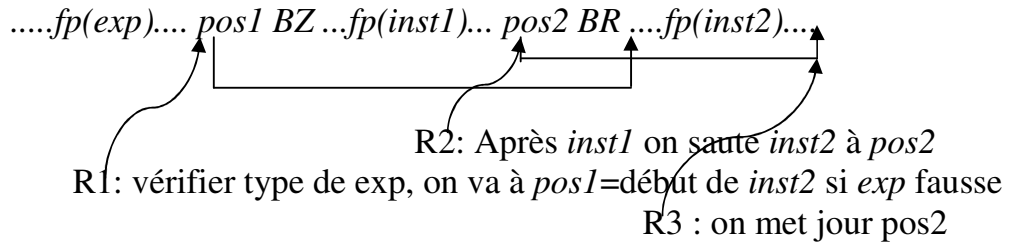
Démarche à suivre dans l'analyse sémantique:

- Donner une forme générale du code à générer
- Déterminer les endroits où on devait intervenir dans la grammaire
- Ecrire les traitements sémantiques à faire au niveau de chaque endroit
- Transformer la grammaire suivant le mode d'analyse ascendant ou descendant
- Ecrire les procédures des routines sémantiques correspondant aux traitements sémantiques

Exemple: Soit l'instruction if définie par sa grammaire comme suit:

<inst_if> → if <exp> then <inst1> else <inst2>

La forme générale du code de cette instruction si on veut générer la forme post fixée est donnée par:



Les endroits où on devait intervenir sont donc:

- Juste après l'expression booléenne pour vérifier le type de *<exp>*, voir si on exécute *<inst1>* ou *<inst2>* et donc générer "*pos1 BZ*" où *pos1* correspond au début du code de *<inst2>*, position qu'on ne connait pas pour l'instant.

- Après *<inst1>* pour générer un *"pos2 BR"* /* sauter *<inst2>*/
  pos2 est la position inconnue pour l'instant qui correspond à la suite après *<inst2>*.
  On met à jour l'inconnu *pos1* qui correspond au début de *inst2*

- Après *<inst2>* où on met à jour l'inconnu *pos2*

Les valeurs *pos1* et *pos 2* ne sont pas connues lors de la génération du *BZ* et du *BR*, c'est pour cela qu'on sauvegarde leurs positions et on les met à jour lorsque les valeurs des positions correspondantes sont connues.

On doit transformer la grammaire de l'instruction pour insérer ces traitements sémantiques, il y a donc 3 routines sémantiques: une après *<exp>*, une après *<inst1>* et la 3eme après *<inst2>*.

Rappel Transformation de la grammaire:

Soit la règle $A \rightarrow \alpha\beta$ avec une Routine Sémantique entre α et β.

Cas d'analyse descendante: *(elle devient* $A \rightarrow \alpha B\beta$ *et* $B \rightarrow \epsilon$*)*

<inst_if> → if <exp> then R1 <inst1> else R2 <inst2> R3

R1, R2, R3 $\rightarrow \epsilon$

A remarquer que les positions de R1 et R2 sont les memes si on les met respectivement avant le terminal *then* et le terminal *else*, puisque ces terminaux ne génèrent pas de code.

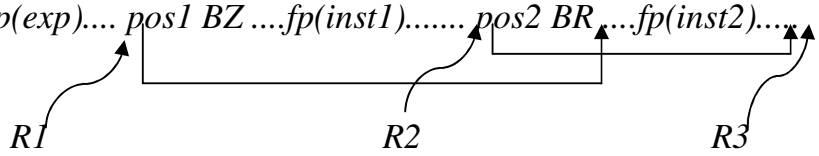
Cas d'une analyse ascendante: (elle devient $A \rightarrow B\beta$ $B \rightarrow \alpha$)

<inst_if> → P1 <inst2> "R3"

P1 → > *P2 <inst1> else "R2"*

P2 → if <exp> then "R1"

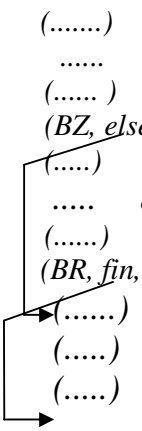
Les routines R_i sont exécutées lors des réductions des non-terminaux *<inst_if>, P1 et P2.*



Ecriture des routines sémantiques: / Pc position courante dans la fp)*/*

```
Routine R1 /* après <exp>*/
debut
    type.exp:=eval_type(<exp>)      /* eval_type est une procédure qui nous permet d'évaluer*/
    si type.exp <> bool             /*le type de exp */
      alors erreur ("erreur de type expression")
     sinon debut
              sauv.pos1:=Pc; ;/* sauver la position pour la mettre à jour plus tard*/
            Pc++;
             fp(Pc):="BZ";Pc++
          fin
fin

Routine R2 /* après inst1 */
debut
     sauv.pos2:=Pc;          /* sauver cette position pour la mettre à jour plus tard*/
    Pc++;
    fp(Pc):="BR";Pc++;   /*générer un BR vers Pos2 - après inst2*/
    fp(sauv.pos1):=Pc;       /*Mise à jour de Pos1 sauvegardé dans sauv.pos1 en R1*/
end

Routine R3 /*Après inst2*/
début
     fp(sauv.pos2):=Pc;  /* Mise à jour de pos2 du BR sauvegardé dans sauv.pos2 en R2*/
end
```

Exemple: Meme exemple <*inst_if*> avec les quadruplets. La forme générale du code est donnée par (qd pour quadruplet):



```
(.......)
 .....      qd générés par  <exp> dont le résultat est dans T.cond
(..... )
(BZ, else, , T.cond)          /* R1 : on génère un BZ vers début des quadruplets de <inst2> */
(.....)                          /* ce début des quadruplets n'est pas connu maintenant*/
 .....    qd générés par <inst1>
(......)
(BR, fin, , )   /*R2 : génère un BR vers le quadruplet après <inst 2>  et MAJ de la valeur du else*/
(.....)
 (.....)   qd générés par <inst2>
 (.....)
                    /*R3 :  MAJ de la valeur de fin */
```

La transformation de la grammaire est la meme que l'exemple 1.

Les Routines sémantiques sont: (Qc pour quadruplet courant)

Routine R1

```
debut
    Type.exp:=eval_type(<exp>)
   si Type.exp incompatible
     alors erreur
     sinon debut
            qd(Qc):=(BZ,  ,  , T.cond) ; /*générer BZ vers else:  position inconnu */
            sauv.else := Qc;   /* sauvegarder la position du quadruplet BZ incomplet*/
           Qc++;                    */pour le metre a jour plus tard */
        end;
end
```

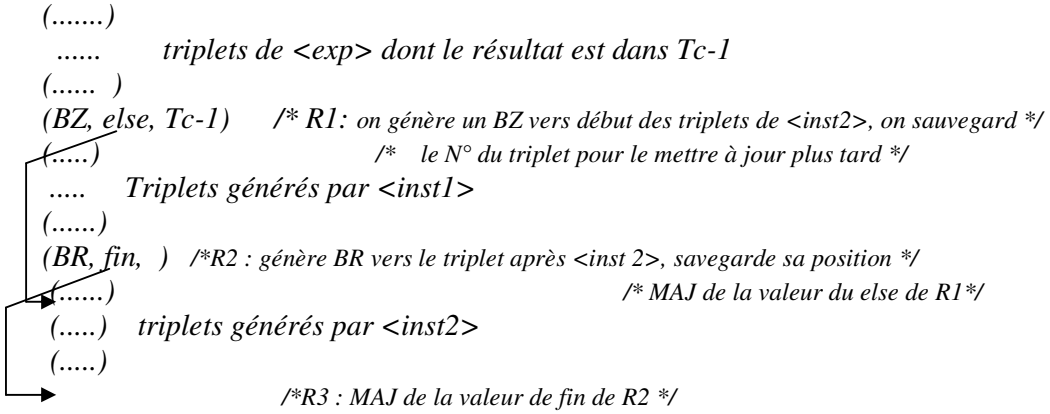
Routine R2;

```
debut
    qd(BR,  , ,  ); /*générer un BR vers une position inconnue pour l'instant */
    sauv.BR:=Qc;  Qc++;
   qd(sauv.else,2):=Qc;/* MAJ du champ 2 du quadruplet BZ sauvagardé ds sauv.else*/
end;
```

Routine R3;

```
debut
     qd(sauv.BR, 2):=Qc;/* MAJ du champ 2 du quadruplet BR sauvagardé ds sauv.BR*/
end;
```

Exemple: Meme exemple *<inst_if>* avec les triplets. La forme générale du code est donnée par (*Tc* pour triplet courant):



La transformation de la grammaire est la meme que l'exemple 1.
Les Routines sémantiques sont:

Routine R1

```
debut
     Type.exp:=éval_type(<exp>)
    si Type.exp incompatible
      alors erreur
     sinon debut
            Trp(Tc):=(BZ,   , Tc-1) ; /*générer BZ vers else:  position inconnu */
            sauv.else := Tc;Tc++;  /* sauvegarder la position du Triplet  BZ incomplet*/
          end;
end
```

Routine R2;

```
debut
     Trp("BR",   , );              /*générer un BR vers une position inconnue pour l'instant */
     sauv.BR:=Tc;  Tc++;
    qd(sauv.else,2):=Tc;      /* MAJ du champ 2 du triplet BZ sauvegardé dans sauv.else*/
end;
```

Routine R3;

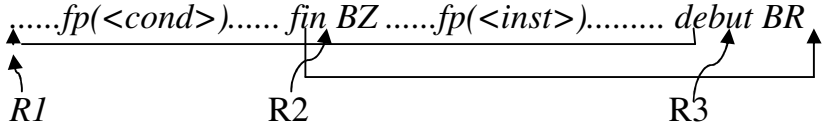
```
debut
    Trp(sauv.BR, 2):=Tc;       /* MAJ du champ 2 du triplet BR sauvagardé ds sauv.BR*/
end;
```

Exemple2: Soit l'instruction while définie par sa grammaire comme suit:

*<inst_while> → **while** <cond> **do** <inst>*

Donner le schéma de traduction en post fixée avec analyse descendante.

La forme générale du code post fixée pour cette instruction est:



On a 3 routines *R1, R2 et R3*

En *R1*, on sauvegarde la position du début du code post fixé de *<cond>*

En *R2*, on vérifie le type de *<cond>,* on sauvegarde la position de la case de *"fin"* dans *pos_BZ* et on génère un *BZ* vers la fin.

En *R3*, on génère un *BR* vers le début sauvegardé en *R1*, et on met à jour la case de *"fin"* du *BZ* sauvegardé dans *pos_BZ en R2*.

Transformation de la grammaire (analyse descendante):

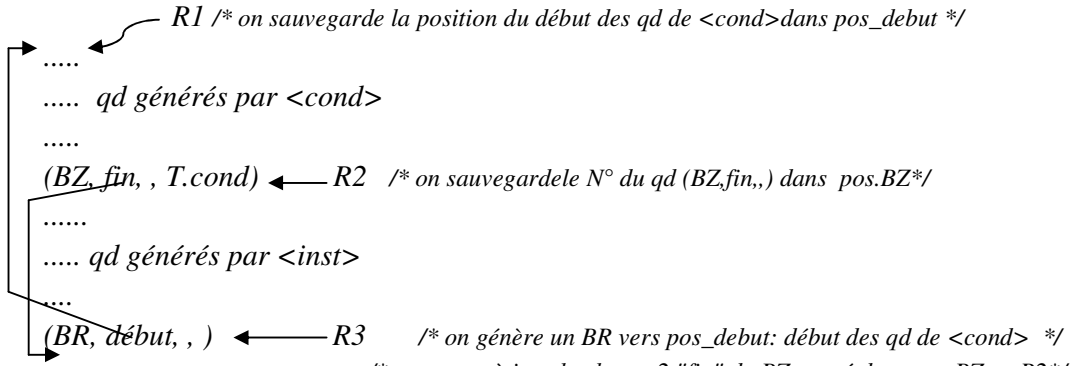
*<inst_while> → A **while** <cond> B **do** <inst>C*

A, B, C →∈

Les Routines:

```
Routine R1
début
     pos_debut:=Pc;            /* on sauvegarde la position du début de la fp */
end;

Routine R2
Début  Type:=eval_type(<cond>)
        si Type<>Booléen
          alors erreur
          sinon pos_BZ:=Pc; Pc++;        /* on sauvegarde la position de "fin" du BZ */
                fp(Pc):="BZ"; Pc++;      /* on génére un BZ vers la "fin" */
end.

Routine R3
début
      fp(Pc):=pos_debut; Pc++;        /* on génère un BR vers pos_debut  */
      fp(Pc):="BR"; Pc++;
      fp(pos_BZ):=Pc;          /* on met à jour la position sauvegardée dans pos_BZ de R2*/
  end.
```

Avec les quadruplets, on a les routines suivantes:
La forme générale du code est:



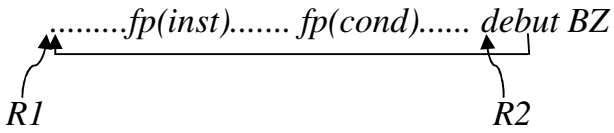
Les Routines:

```
Routine R1
début
      pos_debut:=Qc;
end.

Routine R2
début
      Type:=eval_type(<cond>)
      si Type<>Booléen
        alors erreur
       sinon qd(Qc):=(BZ, , , T.cond);  /* BZ vers la fin */
             pos_bz:=Qc;  /* on sauvegarde la position de qd pour MAJ son champ 2 plus tard/*
             Qc++;
end;

Routine R3
début
     qd(Qc):=(BR, pos_debut, , );Qc++;  /* on génère un BR vers pos_debut  */
     qd(pos_BZ).2:=Qc;      /* on met à jour le champ 2 du qd sauvegardée dans pos_BZ de R2*/
end.
```

Exemple: Soit l'instruction *repeat* définie par la grammaire, donner le schéma de traduction sous forme post fixée par une analyse ascendante:

<inst_repeat> → repeat <inst> until <cond>

La forme générale du code est:



En *R1*, on sauvegarde la position du début de la forme post fixée de *l'inst*.

En *R2*, on vérifie le type de *<cond>* s'il est booléen et on génère un branchement *"BZ"* vers le début de l'instruction si la condition est fausse.

Transformation de la grammaire pour une **analyse ascendante**:

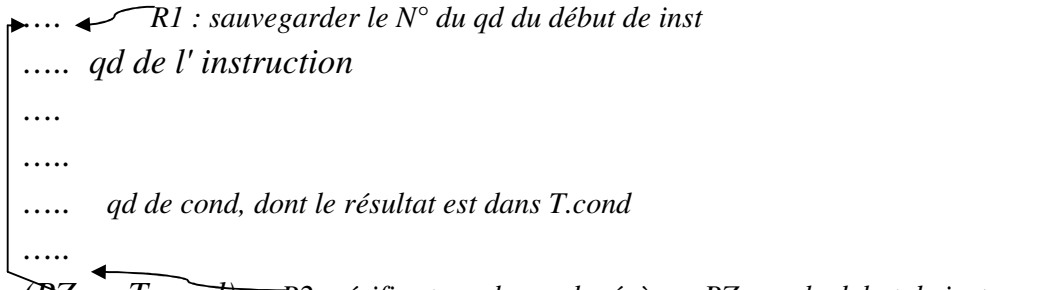
<inst_repeat> → A <inst> until <cond>"R2" *A → repeat "R1"*

```
Les Routines
Routine R1
début
    etiq:=Pc;   /*On sauvegarde le début du code de inst */
end.

Routine R2;
début
     Type.cond:=éval_type(<cond>)
     si type.cond <>bool
       alors erreur
       sinon fp(Pc) := etiq;   /*générer BZ vers le début de inst sauvegardé dans etiq de R1 */
             Pc++;
             fp(Pc):="BZ";Pc++;
 end;
```

Si on veut le faire avec une **analyse descendante** en générant les quadruplets, la transformation de la grammaire nous donne:

<inst_repeat> → repeat R1 <inst> until <cond>R2 R1,R2 →∈

La forme générale du code est :




```
Routine R1;
début
      deb_inst := Qc; /* on sauvegarde le Qc dans deb_inst, debut des quadruplets de l'instr */
end;

Routine R2
 début
        type.cond:=eval_type(<cond>);
        si type.cond <>booléen
          alors erreur
         sinon qd(Qc):=(BZ, deb_inst, , T.cond); /*on génère BZ vers deb_inst=début de inst*/
                Qc++;
 end;
```

Exemple: Schéma de traduction de l'instruction suivante en générant les quadruplets avec analyse descendante:

<inst> → id := moyenne(<exp1>,<exp2>,.....,<expn>)

Cette instruction calcule la moyenne des résultats des expressions et met le résultat dans id. On écrit tout d'abord la grammaire de l'instruction:

<inst> → id := moyenne(<liste>)

<liste> → <exp>A *A →;<liste> / ∈*

La forme générale du code:

```
(:=, id, 0, );     /* Ces 2 quadruplets sont générés par une routine R0, qui consiste à initialiser  */
(:=, nbre, 0, )  /*id à 0 et le nbre d'expression à 0. R0 est juste après id, après avoir vérifié id    */
......
......   qd(exp1)
......
(+, id, T.exp, id);   /* Routine R1 juste après les qd de l'expression, elle ajoute le résultat de */
(+,nbre,1,nbre);    /*              l'expression à id, et incrémente le nombre d'expression */
......
....  qd(exp2)
.....                  /* la meme routine R1 après chaque expression */
......                 /* on répète ceci pour chaque expression */
.....  qd(expn)
......
(+, id, T.expn, id);      /* meme  routine R1 après la dernière expression*/
(+, nbre, 1, nbre);
(/, id, nbre, id);          /* Routine R2 qui calcule la moyenne, si nbre <>0 */
```

La transformation de la grammaire pour une **analyse descendante**:

<inst> → id R0 := moyenne(<liste>)R2 *R0,R1,R2 → ∈*

<liste> → <exp>R1 A *A → ;<liste> / ∈*

Les Routines

```
Routine R0
debut
   loockup(id,P);          /*vérifier si id est déclaré */
  si P=0
     alors erreur("id non déclaré")
     sinon Pnom:=P.nom;        /* récupérer le nom et le type de id */
           Ptype:=P.type;
          qd(Qc):=(:=,pnom,0, );Qc++;  /*initialiser la somme id à 0 */
          qd(Qc):=(:=,nbre,0, ,);Qc++;  /* initialiser nbre d'expr à 0 */
end;
```

```
Routine R1
début
        type.exp:=eval_type(<exp>)
       si ptype et type.exp incompatible
          alors erreur("type incompatible")
          sinon qd(Qc)=(+, pnom, T.exp, pnom);Qc++;  /* ajouter T.exp à la somme */
                qd(Qc)=(+, nbre, 1, nbre);Qc++;*/ on incrémente le nbre d'expr */
end;

Routine R2;
début
    si nbre=0
     alors erreur('nbre expression = 0")
    sinon qd(Qc):=(/, pnom, nbre, pnom);Qc++; /* on calcule la moyenne */
end;
```

Exemple: Soit l'instruction for suivant:

<inst_for>→ for id := <exp1> to <exp2> step <exp3> do <inst>

R1 R2 R3 R4 R5

Donner le schéma de traduction pour générer les quadruplets avec une analyse descendante (on suppose que *inst* peut modifier les *var* des *exp*). Le schéma général de la traduction est donné par:

```
    R1          /* verifier l'id et récupérer son nom et vérifier son type si int */
.....
..... qd(exp1)
....
qd(:=, id, T.exp1,)← R2   /*Routine2: vérifier le  type de exp1 et initialiser id  à exp1*/
.....                     /*     sauvegarder le début des quadruplet de exp2   */
qd(exp2)
.....
qd(BG,␣ , id, T.exp2) ←R3  /* vérifier type de exp2, tester si id > exp2 alors aller à la fin */
....                       /* sauvegardé le n° du qd (BG...) pour le mettre à jour */
qd(exp3)
....        R4: /* vérifier type de exp3, sauvegarder le résultat de exp3 dans T.exp3*/
.....
qd(inst)
.....
qd(+, id, T.exp3, id);←R5   /*:  incrémenter id de la valeur de exp3 */
qd(BR,␣, , )                /*          se brancher vers début qd de exp2 */
                            /*          mise à jour du qd BG    */
```

Les endroits où on doit intervenir sont:

<inst_for>→ for id := <exp1> to <exp2> step <exp3> do <inst>

R1 R2 R3 R4 R5

En *R1*, on vérifie l'id, on vérifie son type et on sauvegarde son nom.

En *R2*, on vérifie le type de *exp1*, on initialise *id* avec le résultat de *exp1*, on sauvegarde le début des quadruplets de *exp2*.

En *R3*, on vérifie le type de *exp2*, on sauvegarde le *Qc*, on génére un *BG* vers la fin

En *R4,* on vérifie le type de *exp3*, on sauvegarde le résultat de exp3 dans *Tmp.exp3*

En *R5*, on génère un qd pour incrémenter le id avec *Tmp.exp3*, on génère un *BR* vers qd de début de *exp2* (sauvegardé dans *R2*), on met à jour le quadruplet du *BG* (généré en *R3*).

Transformation de la grammaire: (analyse descendante)

<inst_for>→ for id R1 := <exp1>R2 to <exp2>R3 step <exp3>R4 do <inst>R5

R1,R2,R3,R4,R5 →∈

```
Routine R1
début
      loockup(id, P);
     si (p=0) ou (p.type<>entier)
       alors erreur
       sinon id.var:=P.nom     /* on sauvegarde le nom de l'id */
end;

Routine R2
début
      type.exp1:=eval_type(exp1)
     si type.exp1<>entier
       alors erreur
       sinon qd(Qc):=(:=,id.var, ,T.exp1);Qc++;/* on initialise id avec le resultat de exp1*/
             deb.exp2:=Qc;  /* on sauvegarde le début des qd de exp2 */
end

Routine R3
début
      type.exp2:=eval_type(exp2)
     si type.exp2<>entier
       alors erreur
       sinon sauv.BG:=Qc;        /*on sauvegarde le N° du qd du BG */
             qd(Qc):=(BG, ? ,id.nom, T.exp2); /*on génère un BG vers la fin si id>exp2*/
             Qc++;
 end;

Routine R4
début
      type.exp3:=eval_type(exp3)
     si type.exp3<>entier
       alors erreur
       sinon tmp.exp3:= T.exp3;/* on sauvegarde le résultat de exp3 dans T.exp3 */
end;

Routine R5
début
      qd(Qc):=(+, id.nom, tmp.exp3, id.nom);Qc++;/* on incrémente id de tmp.exp3 */
     qd(Qc):=(BR, deb.exp2, , );Qc++;/*génère un BR vers deb.exp2 sauvegardé dans R2*/
    qd(sauv.BG).2:=Qc;/*on met à jour le champ2 du qd du BG de R3 */
end.
```

Grammaire des expr arithmétiques