TP2 : Analyseur syntaxique pour Ava

Objectif Ce but de ce TP est de spécifier une première version d'analyseur syntaxique (sans analyse sémantique) pour Ava en utilisant Cup. L'analyse lexicale sera effectuée par l'analyseur lexical que vous avez réalisé lors du TP1 (qui doit donc être fini avant d'attaquer le TP2).

Matériel fourni Récupérer sur le portail l'archive `tp2.tgz` qui contient un répertoire principal `init` donné à titre d'exemple et un répertoire principal `ava`.

1 Matériel fourni : l'exemple d'Init

anSyntInit.cup La spécification contient pour ce TP une grammaire algébrique pour Init. À noter :
- la syntaxe de Cup pour les règles de grammaire : ne pas oublier le `;` en fin de production ;
- le code inclus dans le parser généré :
  ```
  parser code {: ...
  public Symbol parse() throws Exception, ParserException {...}
  public void syntax_error(Symbol symboleCourant) {...}
  :} // parser code
  ```
  Ce code redéfinit des méthodes fournies par Cup pour rendre plus clairs la gestion des exceptions et les messages d'erreurs (il y a encore du boulot pour atteindre le retour d'erreur des vrais compilateurs !).

Fichiers sources De nouvelles classes sont apparues dans les paquetages `init.executeurs` et `init.testeurs` pour permettre le lancement et le test de l'analyseur syntaxique (pas besoin de les éditer).

Tests Le répertoire `test` contient deux répertoires :
- `OK` contient les fichiers de tests positifs (ils doivent être acceptés syntaxiquement) ;
- `KO` contient les fichiers de tests négatifs (ils ne doivent pas être acceptés syntaxiquement).

Scripts Le répertoire `scripts` contient toujours les scripts du TP1 permettant de lancer l'analyseur lexical si besoin. De nouveaux scripts sont apparus :
- `execEnLigneAnalyseurSyntaxique.sh` analyse un texte entré dans la console (exécution de `init.executeurs.LanceurAnalyseurSyntaxique`), rien ne s'affiche en cas de succès ;
- `execSurFichierAnalyseurSyntaxique.sh` prend en ligne de commande le nom d'un fichier à analyser, rien ne s'affiche en cas de succès ;
- `execTestsAnalyseurSyntaxique.sh` lance à la suite les tests de `test/OK` puis ceux de `test/KO` (exécution de `init.testeurs.TesteurPositifAnalyseurSyntaxique` et `init.testeurs.TesteurNegatifAnalyseurSyntaxique`).

2 Travail à réaliser

Ne pas oublier le `source envm5.sh`. Créer un répertoire pour ce TP pour y décompresser `tp2.tgz`.

Pour Init (répertoire principal `init`) : regarder la tête des spécifications puis expérimenter les scripts.

Pour Ava (répertoire principal `ava`) :
- recopier dans `spec` les spécifications du TP1 ;
- modifier la grammaire que contient le fichier `.cup` pour Ava (celle donnée au TP1 était vide) jusqu'à obtenir une grammaire algébrique complète pour Ava.
- à chaque modification regénérer l'analyseur syntaxique par un `ant genAnSynt` et compiler par `ant compil` ;
- alimenter au fur et à mesure les répertoires de test `OK` et `KO`.

Recommandations concernant l'organisation du projet Pour être terminé dans les délais impartis ce TP **doit** être réalisé en suivant une démarche *incrémentale* et *itérative* incluant des tests intensifs :

- démarche incrémentale : on modifie la grammaire petits bouts par petits bouts ;
- démarche itérative : on procède par cycles courts écriture_des_tests-codage-compilation-test ou codage-compilation-écriture_des_tests-test.

La bêtise *à ne pas faire* est d'écrire toute la grammaire d'un coup : il est très difficile ensuite de se dépêtrer des messages d'erreurs de Cup causés par des erreurs de syntaxe, et de trouver les erreurs dans la grammaire.

Il est recommandé de commencer «par le bas» de la grammaire. «Le bas» de la grammaire désigne les productions non-récursives (typiquement l'entête du programme, la lecture, les expressions réduites à un entier ou une variable, l'écriture). Ensuite seulement on attaquera les suites de déclarations de variables et les suites d'instructions.

Bien sûr, pour se rapprocher au plus tôt de programmes Ava courts, on commencera par ajouter dans la grammaire l'entête du programme. Par exemple, on écrira d'abord (à adapter à vos unités lexicales) :

```
start with program;
program ::= entete listeDecl listeInstr;
entete ::= PROGRAM CHAINE PV;
listeDecl ::= ; // on verra ensuite
listeInstr ::= ;
```

On peut alors reconnaître *et tester* des programmes du style `program "toto" ;` (correct) ou `program ;` (incorrect.)

Ensuite on peut écrire une production pour reconnaître une instruction de lecture (à adapter à vos symboles) :

```
listeInstr ::= lecture;
lecture ::= READ IDENT PV;
```

et on peut reconnaître et *tester* des programmes du type `program toto ; read x ;`

Il est *absolument nécessaire* de tester au plus tôt (écrire l'entête, tester l'entête, écrire la lecture, tester la lecture...) avec des programmes courts qui seront stockés dans `OK` pour les programmes corrects et `KO` pour les programmes incorrects. On gardera ces programmes de test. On les réexecutera à chaque modification de la grammaire (test de non régression).

Pour être sûr d'avoir bien compris la spécification de Ava il est réellement profitable d'écrire les tests *avant* de coder la grammaire.

Il n'est pas nécessaire d'écrire des tests correspondant à des programmes sémantiquement corrects (variables déclarées, etc), même si on ne pourra plus réutiliser ces tests pour tester l'analyseur sémantique.

3 Le cas des expressions

Cette section est à lire avant le TP à titre de préparation. Elle explique à partir d'un exemple comment écrire la grammaire des expressions de Ava (grammaire à opérateurs écrite sous une forme ambiguë avec des indications de priorité et d'associativité des opérateurs pour Cup).

3.1 Les expressions de Ava (rappel de la spécification)

Une *expression* Ava est de type entier ou booléen. Une *expression entière* est une expression arithmétique parenthésée habituelle, à résultat entier (combinaison des opérateurs d'addition `+`, soustraction `-`, multiplication `*`, division `/`, opposé `-`, modulo `mod`). L'opérateur `mod` est prioritaire sur `*` et `/`, eux-même prioritaires sur `+` et `-`. Une expression est construite uniquement au moyen de variables déclarées de type entier et de constantes de type entier (par ex. `52`).

Une expression booléenne est construite à partir des opérateurs classiques (`and`, `or`, `not`), de parenthèses, de variables déclarées de type booléen, des constantes `true` et `false`, et de comparaisons entre expressions entières par les opérateurs classiques (égalité `=`, différence `/=`, `<=`, `<`, `>`, `>=`). L'opérateur `not` est prioritaire sur `and` qui est prioritaire sur `or`. Les opérateurs arithmétiques sont prioritaires sur les opérateurs de comparaison qui sont prioritaires sur les opérateurs booléens.

3.2 Syntaxe de Cup

À titre d'exemple, voilà la grammaire de Init écrite en Cup (cf l'archive pour le fichier `anSyntInit.cup`).

```
terminal PROG,INT,READ,IDENT,ENTIER,AFF,FININSTR,VIRG;
terminal SEPVAR;
non terminal programme,entete,listeDecl,decl,listeIdent;
non terminal listeInstr,instr,affect,lect;
start with programme;

programme ::= entete listeDecl listeInstr
;
entete ::= PROG IDENT FININSTR
;
listeDecl ::=
            | decl listeDecl
;
decl ::= INT listeIdent FININSTR
;
listeIdent ::= IDENT
             | IDENT SEPVAR listeIdent
;
listeInstr ::=
             | instr listeInstr
;
instr ::= affect
        | lect
;
affect ::= IDENT AFF ENTIER FININSTR
;
lect ::= READ IDENT FININSTR
;
```

Noter le ; de fin de production et la lisibilité d'une grammaire bien indentée.

3.3 Expressions et typage pour Ava

Au niveau lexical les identificateurs Ava ne sont pas typés : il est impossible de distinguer un identificateur de type entier d'un identificateur de type booléen. C'est le travail de l'analyse sémantique d'assigner des types et de les vérifier, lors du contrôle de type. Cette impossibilité a des répercutions sur la grammaire algébrique des expressions bâties à partir de ces identificateurs.

Supposons qu'on essaie de différencier syntaxiquement les expressions entières des expressions booléennes :

```
exprEnt ::= ENTIER | IDENT | expr PLUS expr | expr MOINS expr | expr MULT expr
| expr DIV expr | expr MOD expr | MOINS expr | PARO expr PARF
;
exprBool ::= IDENT | TRUE | FALSE | exprBool AND exprBool
| exprBool OR exprBool | NOT exprBool | comparaison | PARO exprBool PARF
;
comparaison ::= exprEnt EGAL exprEnt | exprEnt SUPEGAL exprEnt | exprEnt SUP exprEnt
    | exprEnt INF exprEnt | exprEnt INFEGAL exprEnt | exprEnt DIFF exprEnt
;
```

On utilisera *exprEnt* par exemple dans les impressions avec `%i`, et *exprBool* par exemple dans les conditionnelles. Mais considérons l'affectation : on peut affecter à un identificateur une expression de n'importe quel type (sémantiquement on veut que les membres droit et gauche de l'affectation aient le même type, mais on ne peut pas le dire au niveau syntaxique, puisqu'on ne peut pas différencier un identificateur de type booléen d'un identificateur de type entier). On écrira donc :

```
affectation ::= IDENT AFF exprEnt | IDENT AFF exprBool
;
```

Et paf ! Le mot `IDENT AFF IDENT` est ambigu[1] ! Si on essaie de contourner la difficulté en supprimant `IDENT` de $L(exprEnt)$ ou de $L(exprBool)$, on ne décrira plus l'ensemble des programmes Ava.

Donc, la seule solution consiste à utiliser une grammaire pour les expressions qui confond type booléen et type entier :

[1] En effet, à la fois `exprEnt` et `exprBool` permettent de dériver `IDENT`.

```
expr ::= ENTIER | IDENT | expr PLUS expr | expr MOINS expr | expr MULT expr
| expr DIV expr | expr MOD expr | MOINS expr | PARO expr PARF
| TRUE | FALSE | expr AND expr | expr OR expr | NOT expr
| comparaison
;
comparaison ::= expr EGAL expr | expr SUPEGAL expr | expr SUP expr
    | expr INF expr | expr INFEGAL expr | expr DIFF expr
;
```

Dans cette approche, l'expression `ENTIER PLUS FALSE SUPEGAL IDENT` est déclarée *syntaxiquement* valide, et son invalidité sera détectée lors du contrôle de type (*analyse sémantique*). Néanmoins, cette grammaire est encore clairement ambiguë (grammaire à opérateurs classique).

3.4 Levée des ambiguïtés

Il est possible de lever les ambiguïtés en exprimant dans la grammaire la priorité et l'associativité des opérateurs (cf cours et TD). On obtiendrait alors une grammaire peu lisible. Cup offre une autre possibilité :

- écrire une grammaire ambiguë ;
- spécifier dans la spécification Cup la priorité et l'associativité des opérateurs ;
- laisser Cup générer un automate à pile correspondant à la grammaire équivalente non ambiguë.

Chaque déclaration débutant par `precedence` spécifie un niveau de priorité *et* un type d'associativité. Le mot-clé `precedence` introduit un nouveau niveau de priorité, *supérieur au niveau précédent*. Les mot-clés `left` et `right` désignent respectivement l'associativité gauche et droite[2]. Tout terminal non déclaré avec `precedence` se voit attribuer le degré de priorité le plus bas (niveau 0). Les instructions de `precedence` sont à placer après la déclaration des terminaux et non-terminaux, et avant le `start with` (s'il y en a un). On écrira par exemple pour les expressions arithmétiques ;

```
precedence left PLUS, MOINS;
precedence left DIV, MULT;
```

Il reste à traiter le problème du symbole `MOINS`. Le moins unaire a une priorité plus forte que celle du moins binaire, mais les deux ont la même représentation (`"-"`, , symbole `MOINS`). Pour s'en sortir on utilise un mécanisme de Cup qui permet d'associer une priorité à une production. Par défaut chaque production possède un niveau de priorité : c'est celui du terminal le plus à droite dans sa partie droite. Si la production ne contient pas de terminaux alors elle a le niveau de priorité le plus bas. Il est possible de forcer le niveau de priorité d'une production en faisant précéder le `;` qui la termine par `%prec <terminal>`. La production a alors le niveau de priorité de `<terminal>`. On introduit donc un terminal `UNAIRE` bidon[3] qui a la priorité du moins unaire, et on écrit :

```
precedence left PLUS, MOINS;
precedence left DIV, MULT;
precedence left UNAIRE;
```

```
expr ::= ...
       | expr MOINS expr
       | MOINS expr %prec UNAIRE
;
```

Une fois que vous avez compris comment ça marche pour les expressions arithmétiques, il reste à traiter les autres expressions en ne perdant pas de vue les priorités respectives des différents opérateurs, décrites dans la spécification de Ava.

[2] Le mot-clé `nonassoc` existe aussi mais on ne l'utilisera pas. Il sert à interdire deux occurrences consécutives et de même priorité d'un terminal. Par exemple si `==` est déclaré `nonassoc` alors `6 == 3 == 2` génèrera une erreur.

[3] Même si ce terminal ne sera jamais produit par l'analyseur lexical, il faut tout de même le déclarer en tant que terminal de Cup. Il apparaîtra donc inutilement dans `TypeSymboles.java`.