Génération outillée de planning Latex
analyse syntaxique

Objectif Ce but de ce TP est de spécifier une première version d'analyseur syntaxique (sans analyse sémantique) pour les plannings en utilisant Cup. L'analyse lexicale sera effectuée par l'analyseur lexical que vous avez réalisé lors du précédent TP (qui doit donc être fini au moins dans les grandes lignes).

Matériel fourni Récupérer sur le portail l'archive `tp3.tgz` qui contient un répertoire principal `init` donné à titre d'exemple et un répertoire principal `planning`.

1 Matériel fourni : l'exemple d'Init

anSyntInit.cup Spécification d'une grammaire algébrique pour Init.

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 (y jeter un oeil pour voir comment faire fonctionner le parser).

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 doivent être rejetés syntaxiquement).

Scripts Le répertoire `scripts` contient toujours les scripts du TP précédent 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 `tp3.tgz`.

2.1 Découverte de Cup avec Init

Depuis le répertoire `init` :
- regarder la spécification `anSyntInit.cup` pour apprendre la syntaxe de Cup, notamment le `;` en fin de production, les alternatives, les productions vides, la déclaration des non-terminaux ;
- observer 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 !).
- expérimenter les scripts de lancement ;
- se familiariser avec les messages d'erreur de Cup en modifiant temporairement la grammaire pour obtenir une grammaire :
  - syntaxiquement incorrecte (par exemple suppression d'un `;` de fin de production) ;
  - ambiguë (message très clair, n'est-ce pas ?) ;
  - contenant un non-terminal inaccessible ;
  - contenant un non-terminal improductif ;

– contenant un terminal ou un non-terminal non déclaré.

2.2 Analyse syntaxique des plannings

NB : la syntaxe du dsl autorise les plannings restreints à la déclaration du master, de la salle et de la date.

Depuis le répertoire principal `planning` :
– recopier dans `spec` les spécifications du TP précédent sur l'analyse lexicale ;
– modifier la grammaire que contient le fichier `.cup` (celle donnée au TP1 était vide) jusqu'à obtenir une grammaire algébrique complète pour les plannings (version complète).
– à chaque modification regénérer l'analyseur syntaxique par un `genererAnalyseurSyntaxique.sh` et compiler ;
– 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 les déclarations pour un planning. Ensuite seulement on attaquera les soutenances, et les suites de plannings.

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

À rendre sur PROF N'oubliez pas d'écrire les noms du binôme dans les fichier .lex et .cup.

Rendre une archive de votre projet, incluant les tests, et un `README` expliquant l'avancement de votre travail, ce qui ne marche pas le cas échéant, et répondant aux questions suivantes :

1. que pensez-vous du traitement fait dans ce TP des blancs, retour-chariot, etc, par rapport à celui fait dans le TP utilisant les expressions régulières ? (d'ailleurs, avez-vous testé des plannings comportant des blancs partout et n'importe où ?)
2. pour le moment on a vu deux solutions pour l'analyse des plannings : la solution avec expressions régulières, et la solution avec JFlex et Cup. Faites la liste des avantages et inconvénients des 2 solutions et indiquer votre préférence.