

# Génération outillée de planning HTML

## Analyse lexicale

Licence info S5  
TP COMPIL – 2012 - 2013

Génération outillée de planning HTML  
Analyse lexicale

**Objectif** Le but de ce TP est dans un premier temps d'apprendre à se servir de JFLEX, puis de spécifier un analyseur lexical pour les plannings. Ce sujet de TP s'étale sur 2 séances (rendu sur PROF à la fin de chaque séance). À l'issue de ce TP, vous devriez :

- avoir repéré le lexique des plannings ;
- avoir compris les principes de JFLEX ;
- commencer à intégrer le principe « générer puis compiler puis tester » ;
- commencer à intégrer un mode de développement itératif à itérations courtes ;
- commencer à intégrer le fait de conserver vos entrées de test ;
- commencer à vous y retrouver dans la masse de matériel fourni.

### Matériel fourni

- une ébauche de documentation sur JFLEX, l'original étant ici : <http://jflex.de/> ou au M5 : `/usr/share/doc/jflex/`
- un paquetage `init` contenant un analyseur lexical pour INIT donné à titre d'exemple ;
- un squelette de projet à compléter.

**Test du TP** Il vous est demandé de tester soigneusement vos TPs et de les **réaliser dans un mode de développement itératif à itérations courtes** : coder ou spécifier un petit peu, compiler, tester, recommencer. Ainsi, à tout moment, vous serez capable d'évaluer le travail effectué (ce qui est codé et testé avec succès). Le **mode de développement classique** consistant à tout coder, puis passer une heure à se débattre avec les erreurs de compilation (le code généré n'est pas facile à comprendre), puis passer des heures à déboguer **est à proscrire**. Comme le cours de compilation n'est pas le bon endroit pour apprendre à utiliser un cadre de test, des outils ad hoc vous sont fournis via des classes principales et des scripts de lancement.

**Important** Les scripts fournis supposent que votre variable d'environnement `CLASSPATH` contienne :

- le répertoire contenant vos `.class` (typiquement `classes`) ;
- le chemin d'accès aux `jar` de JFLEX (`/usr/share/java/JFlex.jar`) et CUP (`/usr/share/java/cup.jar`).

Il est fortement conseillé que votre variable `PATH` contienne aussi le répertoire `scripts`. Le plus simple est de vous faire un script `envCompil.sh` à stocker dans votre répertoire `COMPIL` et d'exécuter `source envCompil.sh` dans toute console servant à travailler sur votre projet (cf portail pour un exemple).

## 1 Familiarisation avec JFlex

Récupérer sur le portail l'archive du TP. Un peu de structuration étant nécessaire vu la diversité des fichiers de ce TP (java codé, java généré, tests, spécifications, scripts, etc), le répertoire `init` est composé des répertoires `src`, `classes`, `doc`, `spec`, `scripts` et `test` :

- `src`, `classes` et `doc` contient les sources, exécutables et javadoc du TP ;
- `spec` contient la description JFLEX d'un analyseur lexical (`anLexInit.lex`) et la description CUP `anSyntInit.cup` d'un analyseur syntaxique complètement bidon, qui ne sert à rien d'autre que de générer la description (type énuméré) des symboles de INIT ;
- `scripts` contient les scripts de lancement et test de l'analyseur lexical ;
- `test` contient des tests donnés à titre d'exemple.

### Description des scripts

Les scripts se lancent depuis le répertoire `init`, en tapant directement leur nom en ligne de commande (par exemple `execEnLigneAnalyseurLexical.sh`). La complétion par la touche `TAB` fonctionne, puisque que le répertoire `scripts` est dans votre `PATH`. **Ces scripts n'ont rien de magique et ne demandent que quelques compétences en shell**, vous pouvez les regarder pour apprendre, les améliorer, en faire d'autres.

Le script `compil.sh` permet de compiler tous les sources Java du projet, de manière très inefficace.

Le script `genererAnalyseurLexical.sh` permet de lancer JFLEX sur la description d'analyseur lexical `anLexInit.lex` : génération d'un fichier `ScannerInit.java` qui est placé au bon endroit dans la hiérarchie des paquetages.

Le script `genererAnalyseurSyntaxique.sh` permet de lancer CUP pour générer l'analyseur syntaxique `ParserInit.java` (qui ne nous sert à rien dans ce TP) et le fichier `TypeSymboles.java`, qui sont placés au bon endroit.

Le script `execEnLigneAnalyseurLexical.sh` attend sur l'entrée standard une suite de caractères qui sert d'entrée à l'analyseur lexical. La suite des symboles reconnue est affichée sur la sortie standard. Ce script permet de tester en détail l'analyseur, par exemple de vérifier que le mot-clé `program` est lexicalement correct, et est transformé en le symbole `PROG` et non en le symbole `IDENT`.

Le script `execSurFichierAnalyseurLexical.sh` prend en paramètre sur la ligne de commande le nom d'un fichier dont le contenu sert d'entrée à l'analyseur lexical. L'affichage est le même que pour `execEnLigneAnalyseurLexical.sh`.

Le script `execTestsAnalyseurLexical.sh` permet d'analyser d'un coup l'ensemble des fichiers `*.init` présents dans les répertoires `test/OK` et `test/KO`. Les fichiers de `test/OK` doivent être analysés avec succès par l'analyseur lexical, ceux de `test/KO` doivent être refusés (on peut ainsi vérifier que l'analyseur accepte `:=` mais pas `: ni =`). L'affichage sur la console indique pour chaque fichier si le test a passé ou non. Ce type de test binaire ("passe / passe pas") est rapide à lancer, mais il est peu informatif pour un analyseur lexical. En effet il ne vous préviendra pas si votre analyseur accepte `program` (le teste passe) mais le transforme en le symbole `IDENT`. Il est néanmoins recommandé de stocker dans `test` tous les tests pratiqués en ligne, d'une part pour se rappeler et stocker les tests pertinents, d'autre part pour que l'enseignant puisse évaluer vos tests.

### Description du paquetage init

**paquetage init.analyseurs** La classe `ScannerInit` est l'analyseur lexical généré par JFLEX à partir de `anLexInit.lex`. Les classes `ParserInit` et `TypeSymboles` sont des fichiers générés par CUP à partir de `anSyntInit.cup`, respectivement l'analyseur syntaxique et la classe définissant le codage par des entiers des symboles de INIT. La classe `Symbole` décrit ce qu'est un symbole. `ScannerException` est levée par l'analyseur lexical quand le flot d'entrée est lexicalement incorrect.

**paquetage init.executeurs** La classe principale `LanceurAnalyseurLexical` lance l'analyse lexicale et affiche chaque symbole reconnu. Elle est exécutée par les scripts `execEnLigneAnalyseurLexical.sh` et `execSurFichierAnalyseurLexical.sh`.

**paquetage init.testeurs** La classe principale `TesteurPositifAnalyseurLexical.java` lance l'analyse lexicale sur un fichier, mais n'affiche qu'un message "Test positif OK" si l'analyse se termine avec succès, "Test positif KO" sinon. La classe principale `TesteurNegatifAnalyseurLexical.java` a un comportement inverse : "Test négatif OK" si l'analyse lexicale échoue (levée de `ScannerException`), "Test négatif KO" sinon. Ces deux classes sont exécutées par le script `execTestsAnalyseurLexical.sh`.

### Travail à faire

- lister le contenu de `spec` et `src/init/analyseurs` ;
- générer l'analyseur lexical, l'analyseur syntaxique, penser à bien lire les affichages en console pour repérer les erreurs<sup>1</sup> ;
- lister à nouveau le contenu de `src/init/analyseurs` ;
- compiler ;
- lancer le script `execEnLigneAnalyseurLexical.sh` : reconnaître un entier, un identificateur, un mot-clé, des blancs, des retour-chariots, un caractère lexicalement incorrect, etc ;
- essayer les scripts `execTestsAnalyseurLexical.sh` et `execSurFichierAnalyseurLexical.sh` ;

1. Attention si vous écrivez un script qui génère l'analyseur lexical, puis l'analyseur syntaxique, puis compile, puis lance les tests : les messages d'erreur de JFLEX et/ou CUP seront cachés tout en haut de la console et vous vous demanderez bien ce qui ne marche pas.

- observer dans `anLexInit.lex` la création d'un symbole `IDENT`, modifier ce fichier pour associer de même aux entiers une valeur de type `Integer` ;
- générer les analyseurs, compiler le tout et re-tester.

## 2 Analyseur lexical pour les plannings

**Vous devez réaliser la version complète du dsl**, en gardant à l'esprit qu'il vaut mieux rendre un travail incomplet mais correct plutôt qu'un travail terminé mais bâclé. Votre projet doit notamment compiler et s'exécuter. **N'oubliez pas de relire la description du dsl sur le portail.**

Le répertoire `planning` est composé des répertoires `src`, `classes`, `doc`, `spec`, `test`, et `scripts`, suivant le même principe que pour `INIT`. **Il est demandé de conserver cette structure.** Vous pouvez par contre adapter les classes fournies comme bon vous semble.

Écrire (répertoire `spec`) une spécification JFLEX d'analyseur lexical `lexiquePlanning.lex` pour les plannings et une spécification CUP `syntaxePlanning.cup` pour générer le type des symboles, en **testant au fur et à mesure**. Les sources et scripts fournis sont tous prévus pour les noms `ScannerPlanning`, `ParserPlanning` et `TypeSymboles` du paquetage `planningMaster.analyseurs`.

### À rendre sur PROF

Pas de panique : ce TP met en place des outils que vous ne connaissez pas et c'est normal que ça soit difficile au début ! Votre analyseur lexical sera réévalué quand vous rendrez votre analyseur syntaxique, car écrire un analyseur syntaxique peut amener à réviser son analyseur lexical.

**N'oubliez pas de mettre les noms du binôme dans chaque fichier que vous aurez modifié.** Vous devez rendre une archive contenant votre projet, **incluant un README** qui explique l'état d'avancement de votre travail, éventuellement ce qui ne marche pas, comment vos tests ont été réalisés, et répond aux questions suivantes **en justifiant de manière concise vos réponses**.

**Q 1 :** À votre avis :

1. est-il facile de modifier l'un des mots clés ? (ex : remplacer "master" par "diplome") comment faire ? est-ce plus facile que pour le TP1 ?
2. est-il facile de rajouter un mot clé ? (ex : "responsable") comment faire ? est-ce plus facile que pour le TP1 ?
3. est-il facile de gérer les lignes blanches dans le fichier d'entrée ? comment est-ce fait ? est-ce plus facile que pour le TP1 ?
4. en cas d'erreur de lexique, le message produit par votre application permet-il de retrouver facilement l'erreur ? est-ce plus facile que pour le TP1 ?
5. discuter les avantages et inconvénients de l'approche JFLEX par rapport à l'approche « tout par expressions régulières » du TP1.

□

## 3 Quelques infos sur JFlex

JFLEX est un générateur d'analyseurs lexicaux pour Java. Comme tous les générateurs d'analyseurs lexicaux, JFLEX prend en entrée un fichier qui contient une description de l'analyseur lexical à générer. Il génère un fichier `.java` contenant une classe Java de même nom qui contient l'analyseur lexical. Il faudra compiler cette classe avec `javac` (figure 1(a)).

Le cœur de l'analyseur est une méthode appelée par la suite « méthode d'analyse », dont le code schématique est donné figure 1(b). Cette méthode reconnaît dans le texte à analyser le prochain symbole et le retourne. JFLEX permet de configurer le nom, le type de retour de cette méthode, et le type d'exception levée. Il est aussi possible de configurer son corps en associant une *action* à une description de symbole. Cette action sera effectuée dans l'état final correspondant au symbole. Dans l'utilisation

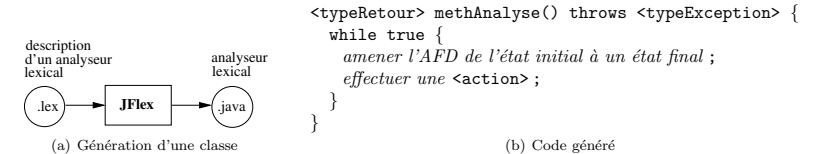


FIGURE 1 – Principes de JFLEX

habituelle d'un analyseur lexical, la méthode d'analyse est destinée à être appelée répétitivement par un analyseur syntaxique jusqu'à la fin du flot de caractères (fig. 2). L'action consiste alors le plus souvent à retourner le symbole reconnu, ce qui termine l'exécution de la méthode (la boucle infinie permet au contraire d'ignorer certains symboles). Dans d'autres utilisations, les actions pourront par exemple calculer une valeur qui sera retournée à la fin de l'analyse.

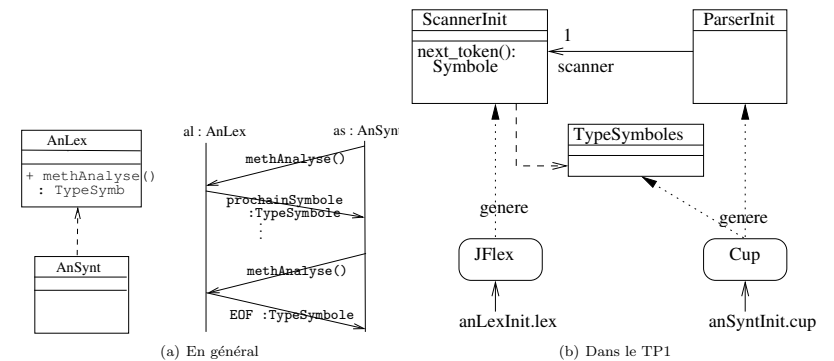


FIGURE 2 – Collaboration avec un analyseur syntaxique

La syntaxe de JFLEX utilise de nombreux caractères méta ou spéciaux (qui ont une signification particulière dans JFLEX, tout comme l'étoile des expressions régulières signifie la clôture de Kleene et non le caractère étoile), par exemple :

- %, ", \
- \n pour signifier le line-feed, \r pour le carriage-return,
- le . pour signifier « tout caractère sauf \n », [abc] pour signifier un caractère parmi a, b ou c et [^abc] pour signifier s'importe caractère sauf a, b, et c.
- ~ puis un caractère pour signifier « tout jusqu'à la prochaine occurrence de ce caractère »

Pour utiliser ces caractères méta en tant que caractères intrinsèques, il faut les déspecialiser en utilisant des guillemets ou un \. Par exemple le guillemet sera représenté par \", et le \ par \\.