Génération outillée de planning Latex
Analyse lexicale

Licence info S5
TP COMPIL – 2011 - 2012

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.

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 Pour mettre à jour vos variables d'environnement `CLASSPATH` (JFlex puis Cup) et `PATH` (scripts) il est impératif d'exécuter `source /home/enseign/COMPILS5/envm5.sh` **dans toute console** servant à compiler votre projet. Il est recommandé de copier ce fichier sur votre compte.

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` contiennent 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`.

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 negatif 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 [1] ;
- 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` ;
- supprimer les fichiers générés par JFlex et Cupavec une commande `ant`, constater en listant le répertoire `src/init/analyseurs` que ça fonctionne ;
- 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.

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.

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.

Écrire (répertoire `spec`) une spécification JFlex d'analyseur lexical `lexiquePlanning.lex` pour les plannings de master 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`.

Précisions sur le lexique des plannings et point délicat Concernant le lexique des plannings :
- on admet des blancs, tabulations, saut de ligne, etc partout et autant qu'on veut ;
- on restreint les noms de master aux noms existants : IAGL, TIIR, eServices et IVI ;
- on contrôle peu la date pour s'autoriser des écritures plus ou moins abrégées : des chiffres pour le jour et l'année, des lettres pour le mois ;
- on admet des chiffres et/ou des lettres pour la salle ;
- le créneau a le format heure-heure, une heure étant 1 ou 2 chiffres, puis `h`, puis 0 ou 2 chiffres.

Ces définitions n'étant pas exclusives, il faut bien réfléchir aux tokens qui seront définis (penser à la règle du plus long préfixe et aux priorités). Vous pourrez encore adapter vos choix quand vous réaliserez l'analyseur syntaxique.

Le point délicat est la description des cases du tableau pour l'identité de l'étudiant, son entreprise, et la description de sa soutenance. On peut en effet imaginer un peu n'importe quoi dans ces cases « fourre-tout », à commencer par des commandes Latex contenant des `\` et des `@`, pour structurer la case en tableau ou écrire en gras. Or, « n'importe quoi » ne définit pas une entité lexicale ! On doit aussi pouvoir accepter `eServices` comme description de soutenance. Pour ces 2 problèmes (entité lexicale fourre-tout et utilisation d'un mot-clé dans un autre contexte), les solutions standard les plus simples sont :
- la plus simple : on utilise un délimiteur qui n'est pas déjà utilisé dans le lexique (au choix, par exemple les guillemets ou les quotes) pour baliser la nouvelle entité lexicale. **C'est cette solution qu'on utilisera en TP**. Il faudra modifier en conséquence tous les fichiers de test. On écrira par exemple : `08h15-9h;  "Ali Georges" ; "setenv academy" ; "J2EE, maven, JUnit"`
  Cette solution implique un second problème : si le délimiteur peut apparaître à l'intérieur du texte balisé, il faut pouvoir « l'échapper », par exemple par `\"`.
- la plus élaborée : on utilise le mécanisme des états de JFlex [2]. Dans notre cas, on peut définir le contenu d'une case par `~;` (« tout jusqu'au point-virgule ») dans l'état `SOUTENANCE` uniquement, et entrer dans cet quand on reconnaît un créneau horaire. Il faut alors modifier la définition du dsl en rajoutant un `;` à la fin d'une ligne de soutenance, et aussi un marqueur pas utilisé dans le lexique (genre `#`) qui signifiera le retour dans l'état normal.

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.

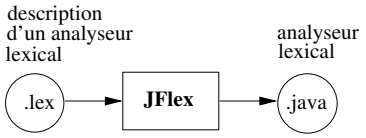
Vous devez rendre une archive contenant votre projet, **incluant un README** qui explique l'état d'avancement de votre travail, éventuellement ce qui ne marche pas, comment vos tests ont été réalisés, et répond aux questions suivantes :
- pensez-vous que le contrôle du jour, de l'année, ou de l'heure (par exemple : 38 n'est pas une heure) soit du ressort de l'analyse lexicale ?
- pourquoi n'est-il pas possible de définir une entité lexicale `CONTENU_CASE` comme « n'importe quoi jusqu'au prochain point-virgule » pour représenter le contenu d'une case fourre-tout ? (sans le mécanisme des états)

3 Quelques infos sur JFlex

JFlex est un générateur d'analyseurs lexicaux pour Java.

2. Tous les générateurs d'analyseurs lexicaux ne proposent pas ce mécanisme.

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)).



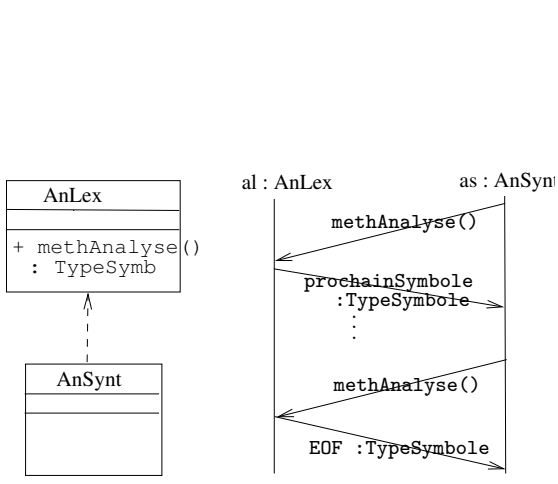
(a) Génération d'une classe

```
<typeRetour> methAnalyse() throws <typeException> {
  while true {
    amener l'AFD de l'état initial à un état final ;
    effectuer une <action> ;
  }
}
```

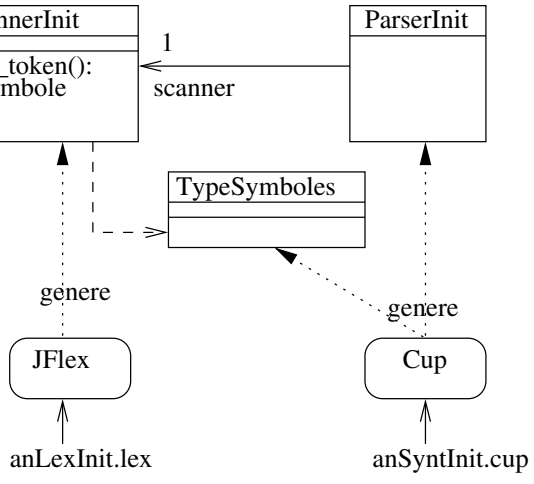
(b) Code généré

Figure 1 – Principes de JFlex

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 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.



(a) En général



(b) Dans le TP1

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 despécialiser en utilisant des guillemets ou un `\`. Par exemple le guillemet sera représenté par `\"`, et le `\` par `\\`.