Exercice 1 : Un automate qui fonctionne comme un an.lex.

Tiré de [Appel], exercice 2.8.

Q 1.1 : Donner un AFD qui reconnaît dans une chaîne de caractères les symboles :
- les opérateurs `+` et `-` ;
- les identificateurs à la Java composés uniquement de chiffres et de lettres (exemple : `ps2pdf`) ;
- les entiers non signés (exemple : `209`) ;
- un sous-ensemble des réels comportant syntaxiquement une suite de chiffres suivis de la lettre `e` suivis d'une suite de chiffres éventuellement signés (exemple : `12e-3`, `4e26`).

Noter l'absence de tout séparateur. □

Q 1.2 : Donner pour chaque symbole une description régulière qui le définit. □

Q 1. 3 : Comment l'analyseur lexical décompose-t-il les chaînes suivantes ? Dérouler l'analyse sur l'automate.
- $+-$;
- $aa+9$;
- $a9e+9$;
- $13e12e4$;
- $9e+a$.

□

Q 1.4 : Reprendre la question précédente en cherchant combien de caractères l'analyseur lexical doit lire après la fin d'un symbole avant de déclarer qu'il a reconnu ce symbole. Peut-on borner ce nombre ? □

Exercice 2 : Vive l'automatisation

Le but de cet exercice est de vous sensibiliser aux avantages de la génération de code comme solution aux tâches répétitives et fastidieuses. On considère un langage de programmation contenant les mot-clés `do`, `od` et les identificateurs à la Java.

Q 2.1 : Donner un AFD qui effectue l'analyse lexicale de ces trois classes de symboles. □

Q 2. 2 : Vous avez implanté à la main l'analyseur lexical à partir de l'AFD obtenu. Votre chef est content, vous aussi. Mais votre chef décide que tout compte fait, `done` serait un mot-clé plus agréable que `od` ! Recommencez... □

Exercice 3 : Un langage de commande

On souhaite écrire un analyseur lexical pour le langage de commandes suivant :
- une *commande* est composée d'un nom de commande, suivi d'une liste optionnelle d'arguments, suivie d'une liste facultative d'options ;
- une *liste d'arguments* est une suite d'arguments ;
- une *liste d'options* est une suite non vide d'options encadrée par `[` et `]`, à l'intérieur de laquelle les options sont séparées par `,` ;
- une *option* est un caractère précédé d'un tiret ;
- un *argument* est un identificateur, de même qu'un *nom de commande*.

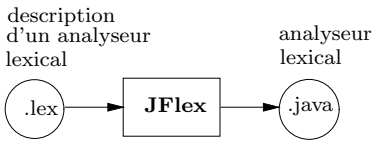
Par exemple, `macom arg1 arg2 [-a,-b]` est une commande, ainsi que `macom [-f]` et `macom`.

Q 3. 1 : Quelles sont les unités lexicales nécessaires à la description d'une commande ? Donner pour chacune d'entre elles une description régulière qui la définit. □

Exercice 4 : JFlex : un générateur d'analyseurs lexicaux pour Java

JFlex est un générateur d'analyseurs lexicaux pour Java. Voilà un extrait de la documentation qui vous sera founie pour les TPs.

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)).



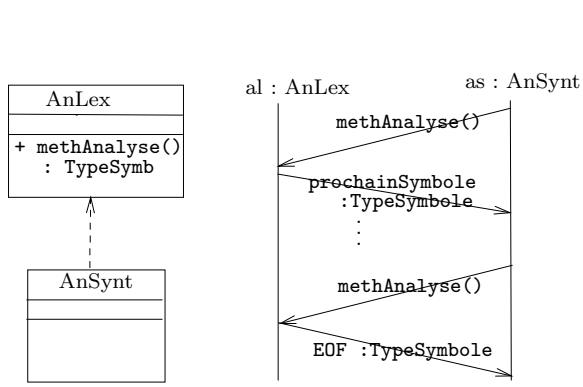
(a) Génération d'une classe

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

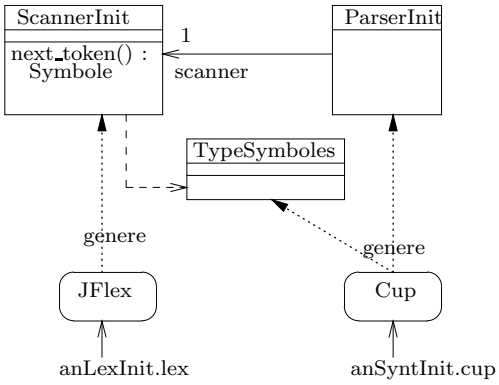
(b) Code généré

Fig. 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 extrait du texte à analyser le prochain symbole. 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

Fig. 2 – Collaboration avec un analyseur syntaxique

Q 4. 1 : Donner les classes de symboles nécessaires à l'analyse lexicale de Init, et pour chaque classe une expression ou description régulière qui la reconnaît. □

Q 4. 2 : En s'inspirant du modèle distribué, regarder comment mettre en œuvre avec JFlex votre réponse à la question précédente. □

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 le guillemet et le pourcent, `\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`. Pour utiliser ces caractères méta en tant que caractères intrinsèques, il faut les despécialiser en utilisant justement un guillemet. Par exemple le modulo sera représenté par `"%"`. Pour utiliser un `"`, on le despécialise avec un \ (`\"`). Et pour utiliser un \, on le despécialise avec lui-même (de quoi attraper une migraine !).

On considère les constructions de Ava suivantes.

- le mot-clé marqueur de fin de conditionnelle `end if` (le mot `end` et le mot `if` peuvent être séparés par un nombre quelconque de blancs et/ou retour à la ligne) ;
- le commentaire : indiqué par `--`, il s'étend jusqu'à la fin de la ligne ;
- la chaîne de caractère à la Java du type `"coucou"`. Comme en Java les guillemets despécialisés à l'intérieur d'une chaîne sont représentés par `\"` et le retour à la ligne est représenté par `\n` (par exemple `"il dit \"coucou\"\n"`). Une chaîne ne peut être coupée par un retour à la ligne.

Q 4.3 : Préparation aux points délicats du TP1 : donner pour ces constructions une description régulière dans la syntaxe de JFlex, ainsi que les flots d'entrées que vous jugez pertinents pour les tester. □