# TP – Construction d'analyseurs lexicaux avec JFlex

## 1 Introduction

Un analyseur lexical découpe un flot d'entrée de caractères en *tokens*. L'écriture d'analyseurs lexicaux est souvent un exercice pénible, c'est la raison pour laquelle des outils de génération d'analyseurs lexicaux ont été développés. Le plus connu de ces outils est très certainement l'utilitaire **lex**, disponible sous **unix** qui permet de générer les sources en langage **C** de programmes d'analyse lexicale. Cette génération se fait à partir d'un fichier de spécifications qui doit suivre une certaine syntaxe.

L'utilitaire **JFlex** est un programme écrit en java d'analyseurs lexicaux à partir de fichiers de spécification. Par défaut, l'analyseur lexical généré est une classe java de nom **`Yylex`**, qui possède une méthode **`Yytoken yylex()`**. Le type **`Yytoken`** est à définir par le concepteur de l'analyseur lexical.

Vous n'utiliserez vraisemblablement pas toutes les fonctionnalités proposées par **JFlex** dans ce TP. Le prochain paragraphe décrit le contenu d'une spécification **JFlex**.

## 2 Fichier de spécification JFlex

Ce fichier de spécification se compose de trois parties, séparées par `%%` en début de ligne.

- La première partie contient du code Java qui se retrouvera dans le fichier généré, copié tel quel en début de fichier. On y retrouve en général une déclaration de paquetage, ou des importations de classes externes.
- La deuxième partie contient des directives de génération du code, des déclarations de macros qui permettent de donner des noms à des expressions régulières, et des déclarations d'états.
- La troisième partie contient les règles basées sur des expressions régulières. Une règle à base d'expression régulière a pour syntaxe :
  `expression { action }` ou bien `liste_etats expression { action }`

Voici un exemple de fichier de spécification (`uncomment.lex`), qui permet de créer un analyseur lexical qui supprime les commentaires commençant par `//`.

```
import java.io.* ;

class uncomment { // classe principale
  public static void main(String arg[]) throws IOException {
    // créer un Yylex qui va prendre ses entrées dans le fichier de nom arg[0]
  Yylex yy = new Yylex(new BufferedReader(new FileReader(arg[0]))) ;
  Yytoken token ;
    // la fin de fichier est codée par un token null
  while ((token = yy.yylex()) != null) System.out.print(token.image());
```

```
  }
}

// la classe Yytoken
class Yytoken {
  private String image ;
  public Yytoken(String image) {this.image = image ;}
  public String image() {return image ;}
}

%%

%unicode


NON_DEBUT_COMMENTAIRE=[^/]|"/"[^/]

%%

{NON_DEBUT_COMMENTAIRE}* {return new Yytoken(yytext()) ;}

"//".* {}
```

### 2.1 Fonctionnement général de JFlex

Lorsque la méthode `yylex` est appelée, l'analyseur lexical recherche une règle à appliquer. Pour cela, parmi les expressions régulières présentes dans les règles, il recherche à quelle expression appartient un début d'entrée le plus long possible. On appelle cela du *pattern matching* ou *filtrage*. En fonction du choix de cette expression, l'action associée (code java) est exécutée. Voici comment les règles sont choisies :

- Si une règle a spécifié une liste d'états, alors la règle ne peut s'appliquer que si l'état courant appartient à cette liste. Il y a toujours un état (initial) `YYINITIAL`. Pour changer d'état, utiliser l'instruction `yybegin(nouvelEtat) ;`. La directive `%state` permet de déclarer les états.
- Si plusieurs règles peuvent s'appliquer, l'analyseur choisit celle qui consommera le plus de caractères dans l'entrée.
- Si plusieurs règles s'appliquant consomment le même nombre de caractères, alors l'analyseur choisit celle qui apparaît en premier dans le fichier de spécification.
- Si aucune règle ne peut s'appliquer, l'analyseur déclenche une erreur.

Les actions associées aux règles correspondent à du code java exécuté lorsque la règle s'applique. Plus précisément :

- Si une action comporte une instruction `return` *objet*, le résultat de la méthode `yylex()` est cet objet.
  Par exemple, la règle `{NON_DEBUT_COMMENTAIRE}* {return new Yytoken(yytext()) ;}` signifie que l'analyseur va lire la plus longue séquence $s$ possible de caractères qui ne sont pas des débuts de commentaires, et va renvoyer un objet `Yytoken` contenant la chaîne $s$.
- Si une action ne comporte pas d'instruction `return`, l'analyseur boucle et va donc chercher de nouveau une règle à appliquer. Par exemple lorsque la règle `"//".* {}` est appliquée, l'analyseur lit tous les caractères à partir de `//` jusqu'à la fin de la ligne, et cherche ensuite à appliquer une règle.

- Dans une action, on peut utiliser la méthode `yytext()` qui renvoie la chaîne de caractères correspondant au motif repéré.
- Dans une action, on peut utiliser les variables entières `yyline`, `yycolumn`, ou `yychar` à condition d'avoir précisé les directives correspondantes (voir la section sur les directives de génération).

## 2.2 directives de génération

Par défaut, l'analyseur lexical généré est une classe java de nom `Yylex`, qui possède une méthode `Yytoken yylex()`. Le type `Yytoken` est à définir par le concepteur de l'analyseur lexical (donc vous !).

On peut paramétrer la génération de l'analyseur lexical, grâce à des directives écrites dans la seconde partie de la spécification.

Chaque directive doit tenir sur une ligne, et doit commencer en début de ligne. Nous n'allons pas faire le catalogue de toutes les directives, pour cela, il faut se référer à la documentation. Voici quelques directives intéressantes :

- la directive `%unicode` permet d'utiliser des caractères accentués, avec le codage UNICODE.
- ```
  %eofval{
    code
  %eofval}
  ```
  permet de préciser le token que l'on veut renvoyer lorsque la fin de fichier est rencontrée. Par exemple, le code sera `return new TokenFin()`
- – la directive `%line` active le comptage des lignes. Dans ce cas, l'entier `yyline` donne le numéro de la ligne courante, sachant que la première ligne de l'entrée a le numéro 0.
  – de la même manière, la directive `%column` active le comptage des colonnes (i.e. le nombre de caractères depuis le début de la ligne). La variable `yycolumn` donne le numéro de la colonne courante,
  – la directive `%char` permet de compter les caractères depuis le début de la lecture, l'entier `yychar` donnant le numéro du caractère courant.
- ```
  %{
    code
  %}
  ```
  le code est inséré au début de la classe de l'analyseur lexical.
- la directive `%ignorecase` permet la génération d'un analyseur lexical insensible à la casse.
- ```
  %yylexthrow{
  exception1, exception2 ...
  %yylexthrow}
  ```
  permet d'insérer dans la signature de la méthode d'analyse lexicale la déclaration des exceptions `throws exception1, exception2, ....`
- Une directive peut aussi être la déclaration d'une macro. La syntaxe est `nomMacro=defMacro` où `nomMacro` est un identificateur et `defMacro` est une expression régulière.

### La syntaxe des expressions régulières :

Elle est très semblable à celle des expressions régulières Unix que vous avez utilisées au TP précédent.
  ```

- La plupart des caractères se filtrent eux-même. Par exemple, le motif `abc` filtre la chaîne de caractères `abc`.
- Les caractères entre guillemets se filtrent eux-même. Ainsi `"abc"` filtre la chaîne `abc`.
- Certains caractères représentent des opérateurs dans les expressions régulières :
  - `|` signifie *ou*
  - `*` signifie *zéro, une ou plusieurs instances de*
  - `+` signifie *une ou plusieurs instances de*
  - `?` signifie *zéro ou une instance de*
- Les parenthèses sont utilisées pour définir une plus forte priorité.
- Le caractère point filtre n'importe quel caractère, sauf la fin de ligne.
- Le caractère `$` représente la fin de ligne
- Comme en C, il y a des caractères spéciaux, précédés d'un backslash. Par exemple, `\n` signifie *newline*, `\b` représente *backspace*, `\t` représente *tab*, `\r` représente *carriage return*.
- Les caractères réservés peuvent être précédés de backslash si on veut leur enlever leur signification particulière. Par exemple `\"` représente le caractère guillemet.
- Les crochets permettent de définir une classe de caractères, c'est-à-dire un motif exprimant un caractère pris parmi tous ceux mis entre crochets. Par exemple `[abc]` représente un caractère choisi dans l'ensemble `{a,b,c}`. Entre ces crochets, on peut utiliser `^` pour exprimer le complémentaire. Ainsi, `[^abc]` représente n'importe quel caractère sauf `a`, `b`, ou `c`. Enfin, le symbole `-` peut être utilisé pour représenter un intervalle. Par exemple `[a-z]` représente n'importe quel caractère entre `a` et `z`.
- Le caractère espace termine l'expression régulière. C'est pourquoi, si on veut que le motif contienne des espaces, on les écrira entre guillemets, ou à l'intérieur d'une classe de caractères. Par exemple, les motifs `[a bc]` et `a|" "|b|c` sont équivalents.
- Si `m` est un nom de macro, alors `{m}` représente la définition de `m`.

# 3 Préparation de ce TP

Ce TP a pour but de vous faire manipuler l'outil **JFlex** afin de réaliser un interpréteur pour un petit langage de dessin appelé **Jade**[1].

Pour cela, nous nous appuyons sur une arborescence particulière afin de simplifier la manipulation les fichiers de spécification **JLex**. Cette dernière est disponible sous la forme d'une archive sur le portail.

> Téléchargez cette archive, et décompressez la dans un répertoire de votre choix. Dans la suite de ce sujet, nous nous référons à ce répertoire par le nom **racine**.

L'arborescence de ce répertoire est la suivante :

```
+ racine
|-- bin
|-- compilerLex
|-- compilerJava
```

[1] J'Apprends en DEssinant

```
|-- executer
|-- lib
|   '-- drawing.jar
|   '-- JFlex.jar
|-- src
|   |-- InterpreteurJade.java
|   |-- TabSymboles.java
|   |-- TestDessin.java
|   |-- TestLexic.java
|   |-- Token.java
|   |-- Yytoken.java
|   |-- uncomment.lex
|   '-- uncommentall.lex
'-- test
    |-- testFile.txt
    '-- testFile2.txt
```

Dans cette arborescence :

1. le répertoire **bin** contiendra toutes les classes compilées lors de ce TP;
2. le fichier **compilerLex**, qui nous facilite la compilation d'analyseurs syntaxiques;
3. le fichier **compilerJava**, qui nous facilite la compilation de l'analyseur syntaxique de Jade;
4. le fichier **executer**, qui nous facilite l'exécution d'un programme écrit dans le cadre de ce TP;
5. le répertoire **lib**, qui contient les librairies nécessaires à la réalisation de ce TP;
6. le répertoire **src**, qui contient les fichiers sources qui seront écrit au cours de ce TP;
7. le répertoire **test**, qui contient des fichiers permettant de tester les programmes que vous créez.

Avant de créer notre interpréteur pour Jade, la section qui suit vous fera manipuler des fichiers de spécification **JFlex**, afin de vous familiariser avec leur syntaxe.

# 4 Familiarisation avec JFlex

Deux fichiers d'exemples de spécifications pour **JFlex** sont disponibles dans le répertoire **`src`** de votre répertoire **`racine`** : **`uncomment.lex`** et **`uncommentall.lex`**

Le premier permet de construire un programme qui retire les commentaires de type **`//`** d'un fichier source java donné en argument. Le second permet de construire un programme qui retire les commentaires de type **`//`** ainsi que ceux de type **`/* ...*/`** d'un source java.

## 4.1 Première manipulation

> Observez le contenu du fichier **`testFile.txt`** du répertoire **`test`** à l'aide d'un éditeur de texte, ou à l'aide d'un terminal en vous plaçant dans le répertoire **`racine`** et en écrivant la commande **`more test/testFile.txt`**

Comme vous pouvez le remarquer, ce fichier contient à la fois des commentaires de type **`//`** et des commentaires de type **`/* ...*/`**.

Nous allons maintenant compiler un analyseur lexical permettant de retirer tous les commentaires de type **`//`**. Cet analyseur lexical est décrit dans le fichier **`uncomment.lex`** du répertoire **`src`**. Pour créer

l'exécutable qui permettra de retirer les commentaires, vous pouvez utiliser le fichier **compilerLex** disponible dans le répertoire **racine**, en tapant la commande suivante dans un terminal, à partir du répertoire **racine** :

```
./compilerLex src/uncomment.lex
```

Une fois compilé, vous pouvez utiliser l'analyseur syntaxique créé, en utilisant le fichier **executer** du répertoire **racine**. Dans le cas de l'analyseur **uncomment.lex**, il est nécessaire de fournir en plus en argument le nom du fichier que vous souhaitez analyser. Si ce fichier est le fichier **testFile.txt** du répertoire **test**, la commande permettant de lancer l'analyse lexicale est :

```
./executer uncomment test/testFile.txt
```

Veuillez noter que pour l'exécution, nous n'utilisons pas le suffixe *".lex"* de notre analyseur syntaxique.

Testez l'analyseur syntaxique **uncomment** :

1. Compilez le fichier de spécification **uncomment.lex** du répertoire **src**;
2. Exécutez l'analyseur syntaxique nouvellement créé, appelé **uncomment**, sur le fichier de test **testFile.txt** du répertoire **test**;
3. Constatez la disparition des commentaires de type **//**.

## 4.2 Analyse d'un fichier de spécification

Intéressons nous maintenant au contenu du fichier **uncomment.lex** et à son sens.

Ouvrez à l'aide d'un éditeur de texte le fichier **uncomment.lex** du répertoire **src**.

Vous pouvez constater que ce fichier commence par des déclarations de code java. Ces déclarations ne seront utilisées que dans cette première partie du TP. En effet, dans le cas d'applications complexes, il est préférable de séparer le fichier de spécification de l'analyseur syntaxique du reste du programme.

Le code java ici présent est utilisé pour créer un fichier exécutable nommé **uncomment.class**, qui permet d'exécuter l'analyseur syntaxique. Il contient en particulier une méthode **public static void main(String[] arg)** qui ouvre le fichier dont le nom est fourni en paramètre lors de l'exécution de la commande, puis lance une analyse syntaxique à l'aide d'une boucle.

La deuxième partie de ce fichier commence au niveau du couple de caractères **%%**. Cette deuxième partie sert à déclarer les options utilisées dans l'analyseur syntaxique (en l'occurrence **%unicode**), ainsi que d'éventuelles macros simplifiant la lecture du fichier de spécification. Dans notre cas, il s'agit d'une macro intitulée *"NON_DEBUT_COMMENTAIRE"*, qui représente un raccourci vers l'expression régulière **[^/]|"/"[^/]**. Cette expression régulière reconnait les lettres que l'on retrouve dans toutes les lignes que ne sont pas des commentaires de type **//** : soit un caractère différent de /, soit un caractère / suivi d'un autre caractère qui n'est pas /.

La dernière partie de ce fichier commence au niveau de deuxième couple de caractères **%%**. Cette dernière partie sert à déclarer les règles utilisées pour analyser syntaxiquement des données. Dans le fichier **uncomment.lex**, ces règles sont au nombre de deux :

1. **"//".*** {} est une règle qui spécifie que pour toute expression lue correspondant à l'expression régulière **"//".*** (deux caractères "/", suivis d'un nombre quelconques de caractères sur une même ligne), le code java {} est exécuté. Autrement dit, si l'on rencontre un commentaire de type **//** dans le fichier, on ne fait rien;

2. {**NON_DEBUT_COMMENTAIRE**}***** {**return new Yytoken(yytext()) ;**} est une règle qui spécifie que pour toute expression lue correspondant à l'expression régulière {**NON_DEBUT_COMMENTAIRE**}***** (autrement dit **([^/]|"/"[^/])***, car NON_DEBUT_COMMENTAIRE est une macro), le code java {**return new Yytoken(yytext()) ;**} est exécuté. Ce code crée un *token*, dont le contenu est la ligne lue. Autrement dit, à chaque fois qu'une ligne du fichier traité n'est pas un commentaire, un *token* est créé pour ensuite être manipulé par le programme.

Comme le programme utilisant cet analyseur syntaxique est la méthode **main** mentionnée plus haut, à chaque fois qu'un *token* est créé, il est affiché dans le terminal.

### 4.3 Analyse d'un deuxième fichier de spécification

Intéressons nous maintenant au contenu du fichier **uncommentall.lex** et à son sens.

| Ouvrez à l'aide d'un éditeur de texte le fichier **uncomment.lex** du répertoire **src**. |
|---|

Vous pouvez constater que la première partie de ce fichier est identique à la première partie du fichier **uncomment.lex**.

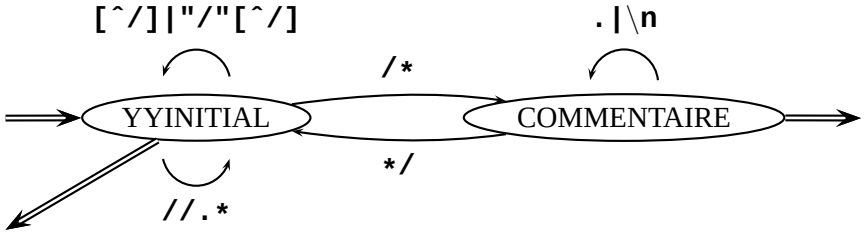
Dans la seconde partie, deux éléments ont été ajoutés :

1. **private int compteurCommentaire = 0;** qui déclare une variable java qui pourra être utilisée dans les règles;
2. **%state COMMENTAIRE** qui déclare un nouvel état qu'il sera possible d'utiliser pour écrire les règles.

Chaque ligne de la troisième partie commence maintenant par du texte mis entre chevrons <>. Ces chevrons signifient que la règle qui suit ne pourra être déclenchée que si l'analyseur syntaxique est dans un état particulier. Par exemple, la ligne "**<COMMENTAIRE> "*"+"/"** {**yybegin(YYINITIAL) ;**}" signifie que la règle **"*"+"/"** {**yybegin(YYINITIAL) ;**} n'est exécutée que si l'analyseur est dans l'état intitulé **COMMENTAIRE**.

L'analyseur syntaxique est initialement dans l'état **YYINITIAL**. Il ne change d'état que lorsque le code d'une règle contient un appel à la méthode **yybegin(état)**. Par exemple, un appel à **yybegin(YYINITIAL);** spécifie que l'analyseur syntaxique change d'état pour **YYINITIAL**.

En conséquence, si on pose $X$ l'alphabet utilisé, et si on suppose qu'une transition peut lire une séquence de caractères, l'ensemble des règles écrites ci-dessus peuvent être interprétées comme l'automate qui suit :




### 4.4 Deuxième manipulation

Nous nous intéressons maintenant à l'analyseur syntaxique **uncommentall.lex**, avec lequel nous souhaitons aussi supprimer les commentaires de type **/* ...*/**.

Test de l'analyseur **`uncommentall.tex`** existant :

1. Compilez l'analyseur syntaxique **`uncommentall.tex`**;
2. Testez cet analyseur syntaxique sur les fichiers **`testFile.txt`** et **`testFile2.txt`** du répertoire **`test`**;
3. Ouvrez les fichiers **`testFile.txt`** et **`testFile2.txt`** à l'aide d'un éditeur de texte, et identifiez la différence entre ces fichiers.

Vous pouvez constater que l'exécution de l'analyseur syntaxique sur le fichier **`testFile2.txt`** produit une erreur. En effet, l'analyseur syntaxique est pour le moment écrit pour ne reconnaitre que les commentaires que l'on peut retrouver en java : il n'est pas possible d'imbriquer des commentaires du type **`/* ...*/`**, ce qui est bien dommage. En effet, cela permettrait, par exemple, de mettre rapidement en commentaire toute une méthode, même si celle-ci contient déjà des commentaires.

Nous souhaitons remédier à cela en modifiant le fichier **`uncommentall.tex`** en conséquence.

Modifiez les règles décrites dans le fichier **`uncommentall.tex`** afin de prendre en compte les commentaires imbriqués de type **`/* ...*/`**.
*Indication : dans la règle commençant par* ***`<COMMENTAIRE> "*"+"/"`****, utilisez un compteur pour savoir quand vous êtes sorti des commentaires imbriqués.*

# 5 Réalisation d'un interpréteur de langage de dessin

Nous allons réaliser un interpréteur pour un petit langage de dessin appelé Jade inspiré du langage Logo, qui avait pour objectif d'initier les enfants à la programmation.

## 5.1 Instructions

On veut réaliser un interpréteur du langage `JADE`, langage de dessin au trait. `JADE` permet de déplacer un *crayon* dans un repère orthonormé, en suivant les verticales ou les horizontales. La position initiale du crayon vaut $(0, 0)$ par défaut. Un déplacement élémentaire se fait de la longueur d'un *pas*. La valeur de ce pas est $5$ par défaut. Initialement, le crayon est *baissé* et tout déplacement provoque un dessin. Il est possible de se déplacer sans dessiner, en *levant* le crayon. Il est aussi possible de placer le crayon à une position donnée par un *point* (couple de coordonnées $(x, y)$).

Voici les **instructions élémentaires** de `JADE` :

- `pas` $n$ : donne au pas la valeur de l'entier $n$.
- `origine` $p$ : positionne le crayon sur le point $p$.
- `nord` : trace un trait de 1 pas vers le haut.
- `sud` : trace un trait de 1 pas vers le bas.
- `est` : trace un trait de 1 pas vers la droite.
- `ouest` : trace un trait de 1 pas vers la gauche.
- $n$ `fois nord` : trace un trait de $n$ pas vers le haut.
- $n$ `fois sud` : trace un trait de $n$ pas vers le bas.

- $n$ `fois est` : trace un trait de $n$ pas vers la droite.
- $n$ `fois ouest` : trace un trait de $n$ pas vers la gauche.
- `lever` : lève le crayon. Les déplacements se font alors sans dessiner
- `baisser` : baisse le crayon. Les déplacements provoquent alors un dessin.

Dans un programme `JADE`, on peut écrire des commentaires commençant par `//` et se terminant à la fin de la ligne.

Dans un deuxième temps, on ajoutera des **instructions complexes** au langage :

- `repeter` $n$ `fois` *suiteInstr* `fin repeter` : permet de répéter $n$ fois une suite d'instructions **élémentaires**.
- `definir` *ident suiteInstr* `fin definir` : permet de donner un nom (l'identificateur) à une suite d'instructions élémentaires.
- *ident* : exécute la figure de nom *ident* (à condition que cette figure ait été définie auparavant).
- $n$ `fois` *ident* : exécute $n$ fois la figure de nom *ident*.

Remarquons au passage que le langage `JADE` est un langage rationnel sur l'alphabet des unités lexicales. Chaque unité lexicale correspond elle-même à un langage rationnel sur l'alphabet des caractères ASCII.

## 5.2 Structure de l'application réalisée

L'interpréteur Jade est un programme qui :

1. lit des instructions écrites par l'utilisateur dans un terminal;
2. utilise un analyseur syntaxique pour reconnaitre les mots clés du langage Jade (présenté dans la section qui suit);
3. interprète les mots clés lus afin de faire un dessin.


Pour réaliser cet interpréteur, nous allons diviser le programme source de l'application en sous-fichiers, ayant chacun un rôle spécifique :

1. l'**analyseur syntaxique** sera spécifié dans un fichier intitulé **`analyseurJade.lex`**;
2. l'analyseur syntaxique produit des *token* que l'on appelle par la suite **unités lexicales**. Les unités lexicales seront des classes qui implémenteront l'interface **`Yylex.java`**;
3. la **fenêtre de dessin** sera décrite dans un fichier appelé **`FenetreJade.java`**. Cette classe disposera de méthodes correspondant aux instructions élémentaires du langage. Par exemple :
   - **`public void nord()`** pour aller d'un pas vers le nord;
   - **`public void nFoisEst(int n)`** pour aller de n pas vers l'est;
   - **`public void lever()`** pour lever le crayon;
   - ...

   Chacune de ces méthodes aura pour rôle de modifier le dessin affiché dans la fenêtre de dessin (par exemple, un appel à la méthode **`nord`** tracera un trait vers le nord si le crayon est baissé);
4. l'**interpréteur Jade** sera spécifié dans un fichier appelé **`InterpreteurJade.java`**.

Tous ces fichiers seront placés dans le répertoire **`src`**.

## 5.3 Première étape : analyse lexicale simplifiée du langage Jade

Dans un premier temps, on ne s'intéresse pas aux instructions complexes. Pour écrire l'analyseur, il s'avère d'abord nécessaire de définir les unités lexicales que l'analyseur syntaxique enverra à l'interpréteur.

## 5.4 Unités lexicales

Les unités lexicales sont représentées par des classes qui implémentent l'interface **`Yylex`** fournie dans l'archive. Voici le code de cette interface :

```
public interface Yytoken {
   public Token getToken();
   public Object getValue();
   public String toString();
}
```

Une unité lexicale est donc caractérisée par au plus deux valeurs. La première est un élément de l'énumération **`Token`**, qui décrit ce à quoi correspond l'unité lexicale.

```
public enum Token {
nord, sud, est, ouest, fois, lever, baisser,
origine, pas, point, entier, identificateur, repeter, fin, definir, eof, erreur
}
```

La seconde représente une valeur associée à l'unité lexicale, par exemple si elle représente un entier, un identificateur, ou un point.

Créez les quatres classes suivantes dans le répertoire **`src`**, qui implémenteront l'interface **`Yytoken`** :

- **`ULMotClef`**, qui correspond à l'unité lexicale des mots clés. Elle dispose d'un attribut qui représente le token du mot clé, dont la valeur est définie par le constructeur, et renvoit systématiquement la valeur `null`;
- **`ULEntier`**, qui correspond à l'unité lexicale des nombres entiers. Son token est **`Token.entier`**, et sa valeur est mémorisée par un attribut de type **`Integer`**, dont la valeur est définie par le constructeur;
- **`ULPoint`**, qui correspond à l'unité lexicale des points. Son token est **`Token.point`**, et sa valeur est définie par une instance de la classe **`java.awt.Point`**. Son constructeur prendra en paramètres deux valeurs de type **`int`**, correspondant aux deux coordonnées du point;
- **`ULIdent`**, qui correspond à l'unité lexicale des identifiants. Son token est **`Token.identificateur`**, et sa valeur est mémorisée par un attribut de type **`String`**, dont la valeur est définie par le constructeur.

La méthode **`public String toString()`** permet d'afficher un texte qui sera utilisé pour tester l'analyseur lexical. Nous proposons ici que cette méthode retourne une chaine de caractères contenant : le nom de la classe (ULMotClef, *etc.*), la valeur retournée par la méthode **`public Token getToken()`** et la valeur retournée par la méthode **`public Object getValue()`**.

## 5.5 Analyseur lexical

Dans cette section, nous allons réaliser et tester un analyseur syntaxique pour Jade en utilisant les unités lexicales définies dans la section précédente.

> Créez un fichier nommé **`analyseurJade.lex`** dans le répertoire **`src`**, et remplissez-le par le code suivant :
> %%
>
> ```
> %unicode
> ```
>
> %%

### 5.5.1 Les commandes `nord`, `sud`, `est` et `ouest`

Dans un premier temps, nous allons chercher à reconnaitre les commandes permettant d'aller d'un pas vers le sud, le nord, l'est ou l'ouest.

> Complétez le fichier **`analyseurJade.lex`** en lui ajoutant quatre règles reconnaissant respectivement les mots *"nord"*, *"sud"*, *"est"* et *"ouest"*. Chaque règle retournera une unité lexicale du type **`ULMotClef`**.

Plutôt que d'écrire les autres règles, il est préférable de vérifier dans un premier temps que les règles déjà écrites sont correctes. Pour effectuer ce test, nous allons utiliser la classe **`TestLexic`** fournie par l'archive. Cette classe exécute l'analyseur lexical, en lisant ce que l'utilisateur écrit au clavier, et exécute la méthode **`public String toString()`** de chaque unité lexicale lue.

> Compilez et testez votre analyseur lexical :
>
> 1. Compilez votre fichier spécification JLex, à l'aide de la commande **`./compilerLex src/analyseurJade.lex`**;
> 2. Compilez le fichier de test **`TestLexic`**, à l'aide de la commande **`./compilerJava src/TestLexic.java`**;
> 3. Exécutez le programme de test, à l'aide de la commande **`./executer TestLexic`**;
> 4. Essayez d'enchainer les commandes de Jade que vous venez d'implémenter.

Vous constaterez qu'une erreur survient après la lecture de la première unité lexicale. Cette erreur est due à la lecture d'un caractère de retour à la ligne, alors qu'aucune règle de l'analyseur lexical ne permet d'en lire.

> Complétez votre analyseur lexical en ajoutant une règle reconnaissant les caractères de retour à la ligne, et ne faisant rien lorsqu'ils sont rencontrés, puis testez à nouveau votre analyseur syntaxique (en effectuant les étapes 2 à 4 de la manipulation précédente).

A force d'utiliser votre analyseur, vous pourrez remarquer qu'il n'est pas possible d'arrêter l'analyse à moins de provoquer une erreur. Pour remédier à ce problème, il se révèle nécessaire de prendre en compte la fin d'un programme Jade.

Complétez votre analyseur lexical, en lui ajoutant une règle qui reconnait le mot *"quitter"*, et qui retourne une unité lexicale du type **`ULMotClef`**, dont l'attribut token vaut **`Token.eof`**. Testez à nouveau votre analyseur lexical, en essayant d'utiliser la commande **`quitter`**.

### 5.5.2 Les commandes `lever` et `baisser`

Complétez votre analyseur lexical, les règles permettant de prendre en compte les commandes Jade **`lever`** et **`baisser`**.

### 5.5.3 Les commandes `n fois nord`, `n fois sud`, `n fois est` et `n fois ouest`

Nous nous intéressons maintenant aux commandes de type **`n fois nord`**. Pour gérer cette commande, l'analyseur syntaxique doit pouvoir reconnaitre un nombre entier suivi d'un ou plusieurs espaces, puis du mot clé *"fois"*, d'un ou plusieurs autres espaces, et enfin d'un mot clé *"nord"*, *"sud"*, *"est"* et *"ouest"*. Bien entendu, un mot clé *"fois"* qui n'est pas précédé d'un nombre entier, ou qui n'est pas suivi d'une direction ne doit pas être accepté. Il en va de même pour un nombre entier qui n'est pas suivi du mot clé *"fois"*.

Pour commencer, nous allons écrire une règle permettant de reconnaitre les nombres entiers.

Complétez votre analyseur lexical, en lui ajoutant une règle qui reconnait les mots composés d'une suite de chiffres. Cette règle devra retourner une unité lexicale de type **`ULEntier`**, construite en lui fournissant en paramètre le nombre entier lu.
*Indication : vous allez devoir utiliser les méthodes* ***`yytext()`*** *et* ***`Integer.parseInt(String s)`***.

A la différence des commandes précédentes, il n'est pas possible ici de gérer toute la commande **`n fois nord`** avec une seule règle. Nous allons devoir utiliser des états. En supposant que *DIRECTIONS*, *ESPACES*, et *NOMBRE_ENTIER* sont des macros reconnaissant respectivement l'une des quatre directions, une suite d'espaces, et un nombre entier, l'automate que nous utiliserons est le suivant :


Complétez votre analyseur lexical en vous basant sur cet automate pour décrire les règles reconnaissant les expressions du type **`n fois nord`**. N'oubliez pas de tester vos nouvelles commandes à plusieurs reprises.
*Indication : vous pouvez vous inspirer du fichier* ***`src/uncommentall.lex`***.
*Indication : Pour éviter tout problème, n'oubliez pas que les règles que vous avez écrites dans les sections précédentes ne doivent être exécutées que si l'analyseur se trouve dans l'état YYINITIAL.*

### 5.5.4 La commande `pas n`

En vous inspirant du travail réalisé dans les sous-sections précédentes, complétez votre analyseur lexical pour prendre en compte la commande **`pas n`**, où n est un nombre entier.

### 5.5.5 La commande `origine(x,y)`

En vous inspirant du travail réalisé dans les sous-sections précédentes, complétez votre analyseur lexical pour prendre en compte la commande **`origine(x,y)`**, où x et y sont des nombres entiers.

## 5.6 La fenêtre de dessin

Maintenant que nous avons un premier analyseur lexical, nous nous intéressons à la fenêtre qui servira à dessiner le résultat des commandes Jade.

La classe **`DrawingFrame`** du package **`drawing`** permet de dessiner dans une fenêtre en utilisant un curseur, qui mémorise la position courante du "crayon" utilisé pour dessiner. Chaque dessin est fait à partir de cette position courante, et a pour effet de déplacer le curseur. Par exemple, si le curseur se trouve à la position $(1; 1)$, et si $p$ est une instance de la classe **`Point`** représentant la position $(2, 4)$, alors un appel à la méthode **`drawTo(p)`** aura pour effet de dessiner un trait entre les points de coordonnées $(1, 1)$ et $(2, 4)$, et de déplacer le curseur au point $(2, 4)$. La documentation de ce package est disponible depuis le portail.

Dans ce TP, le dessin dans une fenêtre sera géré à l'aide de la classe **`FenetreJade`**. Chaque instance de la classe **`FenetreJade`** aura un attribut de type **`DrawingFrame`**, afin de dessiner dans une fenêtre. **`Fenetre`** doit aussi disposer d'autres attributs, que vous devrez découvrir par vous même.

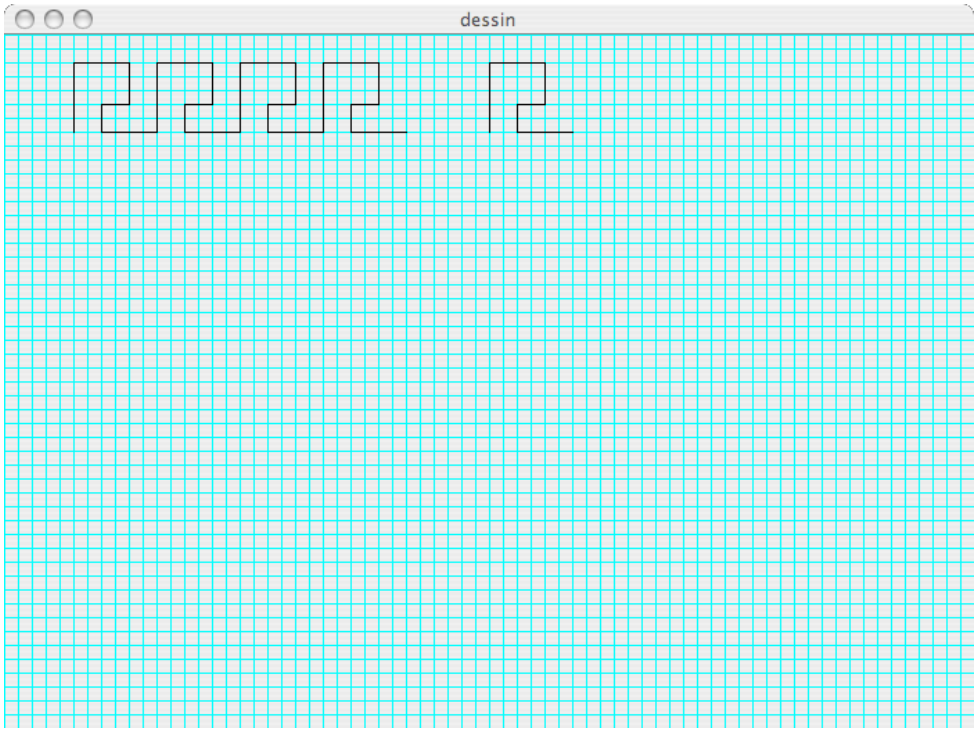
Construisez la classe **`FenetreJade`** :

1. Créez un fichier **`FenetreJade.java`** dans le répertoire **`src`**;
2. Complétez-le par l'entête de la classe **`FenetreJade`**;
3. Écrivez le constructeur de cette classe. Il ne doit pas avoir de paramètres;
4. Écrivez le code des méthodes **`public void nord()`**, **`public void sud()`**, **`public void est()`** et **`public void ouest()`**, dont l'effet est de dessiner un trait vers le nord, le sud, l'est ou l'ouest, si le crayon n'est pas levé. Sinon, elle ne font que déplacer le point courant de votre fenêtre (sans dessiner);
5. Écrivez le code de la méthode **`public void origine(Point p)`**, dont l'effet est de déplacer le curseur vers une nouvelle position (sans tracer de trait entre l'ancienne position du curseur et sa nouvelle position);
6. Écrivez le code des méthodes **`public void lever()`** et **`public void baisser()`**.

*Indications : N'oubliez pas d'initialiser votre attribut de type **`DrawingFrame`**; consultez la documentation de **`DrawingFrame`** pour écrire vos méthodes; par défaut, on veut que le crayon utilisé pour dessiner soit baissé; chaque méthode de la classe **`DrawingFrame`** peut lever une exception; la signature des méthodes de la classe **`FenetreJade`** doit donc être complétée par **`throws Exception`**; par défaut, on veut que le pas soit de 1 unité.*

Testez votre classe **`FenetreJade`** :

1. Compilez votre classe **`FenetreJade`** avec la commande **`./compileJava src/FenetreJade.java`**;
2. Compilez la classe de test **`TestDessin`** avec la commande **`./compileJava src/TestDessin.java`**;
3. Exécutez le programme de test avec la commande **`./executer TestDessin`**;
4. Comparez le dessin obtenu avec le dessin décrit sur l'image qui suit. si votre dessin est identique, votre classe **`FenetreJade`** fonctionne correctement.




## 5.7 L'interpréteur de Jade

Le rôle de l'interpréteur Jade est de lire des commandes du langage Jade, et de modifier la fenêtre de dessin en conséquence. Dans cette section, nous allons écrire le code de cet interpréteur, dans une classe que nous appellerons **`InterpreteurJade`**.

Puisque le rôle de l'analyseur syntaxique est joué par la classe **`Yylex`**, et que le rôle de la fenetre de dessin est joué par la classe **`FenetreJade`**, la classe **`InterpreteurJade`** disposera d'un attribut de chacun de ces types. Elle disposera de plus de trois méthodes :

1. **`public Yytoken lireProchaineUniteLexicale()`**, qui consiste à récupérer la prochaine unité lexicale lue par l'analyseur lexical;
2. **`public void traiterUniteLexicale(Yytoken ul)`**, qui aura pour rôle de lire le contenu de l'unité lexicale, et de modifier la fenêtre de dessin en conséquence;
3. **`public static void main(String[] args)`**, qui aura pour rôle d'exécuter le programme de l'interpréteur Jade.

Le répertoire **`src`** contient le squelette de cette classe, dans lequel les méthodes **`main`** et **`lireProchaineUniteLexicale`** sont écrites. Votre travail consistera à écrire le contenu de la méthode **`traiterUniteLexicale`**.

Le principe utilisé dans la méthode **`traiterUniteLexicale`** consiste dans un premier temps à lire le *token* associé à l'unité lexicale (avec la méthode **`getToken()`**), et à identifier sa valeur, en la comparant aux différentes valeurs de l'énumération **`Token`**. A partir de cette valeur, vous pouvez savoir ce qu'il faut faire.

**Exemple :** Si le *token* lu est **`Token.nord`**, alors il faut exécuter la méthode **`nord()`** de la classe **`FenetreJade`**.

**Exemple :** Si le *token* lu est un entier, on se retrouve dans le cas de la commande **`n fois direction`** (où $n$ est un entier, et $direction$ soit "nord", soit "sud", soit "est", soit "ouest") va etre exécutée. Il faut alors mémoriser la valeur associée à l'unité lexicale, qui représentera la valeur $n$, puis lire une nouvelle unité lexicale pour lire le "fois", et enfin lire une dernière unité lexicale pour connaitre la direction du déplacement. Il faudra alors exécuter $n$ fois la méthode **`nord()`** de la classe **`FenetreJade`** si l'unité lexicale représentait le mot clé **`Token.nord`**, la méthode **`sud()`** pour le mot clé **`Token.sud`**, la méthode **`ouest()`** pour le mot clé **`Token.ouest`** et la méthode **`est()`** pour le mot clé **`Token.est`**.

Complétez la méthode **`traiterUniteLexicale`** de la classe **`InterpreteurJade`** :

1. Basez-vous sur les informations fournies dans cette section pour compléter le code de la méthode **`traiterUniteLexicale`**;
2. Compilez la classe **`InterpreteurJade`** à l'aide de la commande **`./compilerJava src/InterpreteurJade.java`**;
3. Testez votre interpréteur, à l'aide de la commande **`./executer InterpreteurJade`**. Vous pouvez alors écrire des commandes Jade dans votre terminal, et voir leur résultat sur la fenêtre de dessin. Vous pouvez en particulier essayer de reproduire les dessins décrits sur le portail.

**<u>Attention :</u> la suite de ce TP n'est pas aisée à effectuer. Il est donc recommandé de sauvegarder votre travail, et de continuer en travaillant sur une copie.**

## 6 Les instructions complexes

Les instructions complexes permettent de "factoriser" un ensemble d'instructions élémentaires. Dans le langage Jade, il est interdit d'imbriquer les instructions complexes.

1. instruction **`repeter`**
   La première instruction complexe est la répétition d'une séquence d'instructions élémentaires. Rappelons que sa syntaxe est :

   **`repeter`** *n* **`fois`** *suiteInstr* **`fin repeter`**

   Pour exécuter une telle instruction, il faut mémoriser dans une liste les objets **`yytoken`** qui correspondent à la séquence d'instructions, puis exécuter $n$ fois cette séquence d'instruction.

Modifiez les fichiers **`src/analyseurJade.lex`** et **`src/InterpreteurJade.java`** pour prendre en compte ce nouveau type d'instruction.

2. utilisation de macros
   On peut définir une macro par la syntaxe :

   **`definir`** *ident suiteInstr* **`fin definir`**

   qui permet de donner un nom (l'identificateur) à une suite d'instructions élémentaires.

   On utilisera une table de symboles pour associer à un nom de macro sa définition sous la forme d'une liste d'objets **`Yytoken`** qui correspondent à la séquence d'instructions élémentaires. Cette table des symboles est réalisée par la classe **`TabSymboles`** qui vous est fournie.

   Dans un programme Jade, toute occurrence d'un identificateur (donc nom de macro) sera remplacée par sa définition lors de l'interprétation du programme.

| |
|---|
| Modifiez les fichiers **`src/analyseurJade.lex`** et **`src/InterpreteurJade.java`** pour prendre en compte ce nouveau type d'instruction. |