Université de Lille 1 SCIENCES ET TECHNOLOGIES

TDM 5 - Automates et Langages

Le projet Jade (avec JFlex)

octobre 2015

1 Jade, langage de dessin

Les instructions de Jade commandent les mouvements d'un crayon sur une feuille de papier. Le crayon ne peut se déplacer que dans 4 directions orthogonales (**nord, sud, est, ouest**). La longueur d'un déplacement élémentaire est **un pas**. Une instruction permet de fixer la longueur d'un pas (initialement, elle vaut 5).
Le crayon peut être soit levé soit baissé. Un déplacement dessine un trait si (et seulement si) le crayon est baissé.

1.1 But du projet

Réaliser un interpréteur Jade, c'est à dire un logiciel capable de lire une suite d'instruction Jade et de produire le dessin correspondant. Cet interpréteur utilisera plusieurs composants

- Un analyseur lexical réalisera le découpage en une suite de tokens
- L'interpréteur lui-même vérifiera la validité de l'enchaînement des tokens et invoquera des méthodes de dessin.
- Un dernier composant simulera les mouvements du crayon et produira un dessin.

2 Jade Niveau 1

2.1 Syntaxe du langage

Tous les mots clés du langage pourront être utilisés soit entièrement en majuscules, soit entièrement en minuscules, même si dans le présent document ils figurent seulement en minuscules.

2.1.1 Instructions élémentaires

- `lever` : lève le crayon. Les déplacements se font alors sans dessiner
- `baisser` : baisse le crayon. Les déplacements qui suivent provoquent alors un dessin
- `nord` : se déplace d'un pas vers le haut.
- `sud` : se déplace d'un pas vers le bas.
- `est` : se déplace d'un pas vers la droite.
- `ouest` : se déplace d'un pas vers la gauche.
- `pas` *n* : donne au pas la valeur de l'entier *n*.
- `aller` *x y* : réalise un saut jusqu'au point de coordonnées x et y, sans dessiner. x et y sont des entiers.

2.1.2 Opérateur de répétition

- *n* `fois` où n est un entier positif : cet opérateur est nécessairement suivi d'une instruction élémentaire qui est alors exécutée n fois. Par exemple : « `ouest 3 fois nord est` » indique de faire 1 pas vers l'ouest, puis 3 pas vers le nord et enfin 1 pas vers l'est.

2.1.3 Séparateurs et commentaires

Les espaces et fins de ligne seront admis partout entre les éléments du langage.
Le langage admettra les commentaires à la java :

- `//` débute un commentaire qui s'étend jusqu'à la fin de ligne

— `/*` débute un commentaire éventuellement multiligne qui s'étend jusqu'au marqueur de fin `*/`

Les commentaires peuvent se situer **entre** les instructions mais **jamais à l'intérieur** d'une instruction (jamais entre le mot "pas" et l'entier qui suit, par exemple).

2.2 Architecture du projet

Le projet que vous développerez se découpera en plusieurs composants

— Un **analyseur lexical** engendré par JFlex, chargé du découpage du texte en une suite de tokens. Les classes relatives à l'analyse lexicale figureront dans le paquetage `jadelex`.

— Un **analyseur syntaxique** : cette classe utilise l'analyseur lexical, vérifie que la séquence de tokens lue est correcte et envoie à une **machine Jade** les commandes à exécuter.

— Une **machine Jade** est une machine capable d'exécuter les commandes basiques de Jade. Formellement il s'agira d'une classe dont les spécifications sont fixées par l'interface `JadeMachine`. À des fins de débogage, une machine Jade vous est fournie (elle ne réalise pas vraiment de dessin mais fabrique un «log» des commandes exécutées). Vous implémenterez une autre machine Jade qui, elle, dessine vraiment.

— Enfin, vous utiliserez le **dessinateur** qui vous est fourni. Ce composant, qui implémente l'interface `Drawing`, est capable d'ouvrir une fenêtre de dessin et d'y tracer des segments de droite.

Vous serez guidés dans la réalisation qui se fera de manière progressive. Dans un premier temps il s'agira de construire l'analyseur lexical et de le tester séparément pour valider son bon fonctionnement.

2.3 Analyse lexicale

Le type renvoyé par l'analyseur lexical est défini par une `interface Yytoken` (fournie). Un type énuméré `enum TokenType` (fourni) fixe la liste des «types» de token (unité lexicale). Pour chaque type, on construira une classe capable de représenter les unités lexicales de ce type. Toutes ces classes devront implémenter l'interface `Yytoken`. Vous trouverez en outre dans le paquetage `jadelex` :

— la classe `BaseToken` : une implémentation minimale de Yytoken, uniquement destinée à être enrichie par d'autres classes (son constructeur n'est d'ailleurs pas public).

— `PenMode` est l'une de ces classes, conçue pour représenter les unités lexicales (U.L.) de type `PEN_MODE` (instructions lever et baisser). À une U.L. de type `PEN_MODE` est associée une valeur booléenne : vrai pour un passage en mode «tracé» (inst. **baisser**) ou faux dans l'autre cas (inst. **lever**)

Comme la classe `PenMode`, toutes les classes destinées à représenter les unités lexicales devront redéfinir la méthode `toString()` de manière à ce qu'elle renvoie une chaîne sous la forme
<*type de token*>`[`*valeur associée*`]`

2.3.1 Un premier analyseur

Ouvrez le fichier `src/jadelex/jade1.lex`. Quelques directives ont été créées afin que la classe engendrée s'appelle `TokenizerV1` (au lieu de Yylex) et qu'elle implémente l'interface `jadelex.Tokenizer`. Cette première version de l'analyseur devra être capable de

— distinguer les mots clés `baisser` et `lever` et pour chacun d'eux renvoyer une instance de `PenMode` avec la valeur correspondante

— considérer comme séparateur tout caractère d'espacement ou de fin de ligne

— accepter comme commentaire un texte commençant par deux slashs et se terminant en fin de ligne

— considérer que tout autre texte est un token de type `UNKNOWN` et renvoyer une instance de la classe `Unknown`

Une fois ces spécifications écrites, utilisez JFlex pour produire la classe `TokenizerV1` puis compilez la classe Test. Ce programme Test lit un texte soit dans un fichier (si un nom de fichier lui est fourni sur la ligne de commande) soit saisi directement au clavier. Il affiche son découpage en tokens. La lecture du texte

```
baisser// bla bla bla
  lever lever baisser
```

devrait produire

`<PEN_MODE>[true]<PEN_MODE>[false]<PEN_MODE>[false]<PEN_MODE>[true]`

Ensuite, vous ajouterez le traitement des commentaires multilignes `/*  */` (reprenez ce que vous avez fait lors de la séance précédente).

2.3.2 Mouvements

Les instructions de mouvement sont des tokens de type `MOVE`. Sur le modèle de la classe `PenMode`, réalisez une classe `Move`. La valeur associée à ces unités lexicales est une direction (classe énumérée `jade.Direction`). La classe devra donc disposer d'une méthode `Direction getDirection()` et d'un constructeur ayant une direction comme paramètre.
Complétez le fichier de spécification `jade.lex` pour ajouter la reconnaissance des 4 instructions nord, sud, est, ouest.Testez son fonctionnement.

2.3.3 Changement de pas

Une instruction de changement de pas est un token de type `STEP_LENGTH`. avec pour valeur associée la longueur du pas. Écrivez une classe `StepLength` disposant d'une méthode `int getLength()` et d'un constructeur avec un argument entier.
Il vous faudra ensuite compléter les spécifications de l'analyseur. Attention, cette fois on ne peut pas renvoyer le token immédiatement après avoir rencontré le mot clé `pas`, mais seulement après avoir rencontré l'entier qui suit. Pour résoudre facilement ce cas, il vous est fortement conseillé de déclarer un **état** dans lequel l'analyseur passera après avoir rencontré le mot **pas**.

2.3.4 Sauts

Une instruction de saut est un token de type `JUMP` avec pour valeurs associées les coordonnées du point de destination. Écrivez une classe `Jump` disposant de méthodes `int getX()` et `int getY()`, ainsi que d'un constructeur avec deux arguments entiers x et y.
Dans l'analyseur il faut cette fois attendre d'avoir rencontré le mot clé puis les deux entiers avant de pouvoir renvoyer le token. Si l'on utilise des états, comme précédemment, il faut cette fois stocker le premier des entiers le temps de trouver le deuxième. Rappelons (voir fiche précédente : directives `%{` et `%}`) que l'on peut ajouter des attributs et méthodes dans la classe. Il est ainsi possible de déclarer un attribut privé `lastInt` permettant de stocker temporairement un entier.

2.3.5 Répétitions

L'opérateur de répétition se compose d'un entier suivi du mot `fois`. L'analyseur lexical en fait un token de type `REPEAT`.
Écrivez une classe `Repeat` disposant d'une méthode `int getOccurrences()` et d'un constructeur avec un argument entier $occurrences$ (nombre d'occurrences).
Dans l'analyseur, le cas se traite de manière analogue au cas précédent.

Une fois l'analyseur complété et testé, il est capable d'analyser lexicalement tout « programme » Jade niveau 1.

2.4 Jade Machine

Ce composant reçoit par invocation de méthodes les instructions élémentaires de la machine Jade et les exécute. Consultez la documentation de l'interface `JadeMachine`.
Un machine Jade `LoggingMachine` vous est fournie sous forme binaire (fichier .class). Cette machine se contente de générer un message pour chaque commande exécutée, ce qui permet de bien vérifier le déroulement d'une exécution .
Vous utiliserez en premier lieu cette machine pour tester votre analyseur syntaxique.

2.5 Analyseur syntaxique

Ce composant fait appel à un Tokenizer pour lire et le texte des commandes token par token. Il doit vérifier que les token sont bien ceux attendus puis, au fur et à mesure de leur lecture, envoyer à la machine Jade la commande correspondante.
Votre analyseur syntaxique devra implémenter l'interface `jade.JadeParser` ainsi qu'un constructeur avec les arguments indiqués dans la documentation de cette interface.
Pour lancer l'interprétation, vous pouvez utiliser la classe `JadeRunner`.

2.6 Une autre machine Jade

Dans le paquetage `drawing` (documentation en ligne sur le portail) vous avez les classes nécessaires pour dessiner des segments de droite dans une fenêtre graphique.
Vous pouvez ainsi développer votre propre `JadeMachine` qui réalisera graphiquement les dessins.