Université de Lille 1 SCIENCES ET TECHNOLOGIES

TDM 4 - Automates et Langages

Générateur d'analyseur lexical : JFlex

octobre 2015

1 Introduction

Un **analyseur lexical** découpe un flot d'entrée de caractères en **tokens**. Un token (ou encore «unité lexicale») est un portion du texte qui doit correspondre à un motif préalablement défini.
Par exemple si l'on définit 3 motifs : entiers simples (`[0-9]+`), identificateurs (`[A-Za-z][A-Za-z0-9]*`) et opérateurs (`[+-*/]`),

le texte `alpha+321*x5` se décomposera en 5 tokens :

`alpha`	identificateur
`+`	opérateur
`321`	entier
`*`	opérateur
`x5`	identificateur

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. Un générateur d'analyse lexicale est une application qui

- reçoit en entrée un fichier de **spécifications** comportant notamment la liste des motifs qui définissent la forme des tokens à rechercher.
- produit un **composant logiciel** (l'analyseur lexical) capable de parcourir un texte en le décomposant en une suite de tokens. Ce composant est produit sous la forme d'un fichier source qui peut être intégré à un projet de développement.

Le plus connu de ces outils est l'utilitaire historique nommé **lex**, disponible sous unix, qui génère les sources en langage C d'analyseurs lexicaux.

JFlex est l'un de ses homologues conçu, lui, pour engendrer des analyseurs lexicaux en **java** à partir d'un fichier de spécifications, le « source JFlex » (extension `.lex`).

2 Yylex : la classe produite, et son utilisation

JFlex génère un seul fichier, nommé `Yylex.java`, contenant une classe de même nom.
Voici de façon très schématique la structure de la classe Yylex engendrée :

```
// ...
class Yylex {
  // ...
  /* Constructeur
     paramètre : le flux de lecture des données à analyser (texte)
  */
  Yylex(java.io.Reader in) { // ...
  }

  /* Principale méthode publique de la classe.
     Lit le prochain 'token' dans le texte à analyser
  */
  public Yytoken yylex() throws java.io.IOException {
   // ...
  }
  /*
    renvoie le texte correspondant au dernier token lu.
```

```
  */
  public final String yytext() {
   //...
  }
  // ...
}
```

Notons qu'il existe quelques autres méthodes publiques non évoquées ici, et surtout de nombreuses méthodes privées.

La principale méthode publique de cette classe s'appelle `yylex()`. Elle est en principe destinée à être appelée de manière répétitive. Chaque invocation de la méthode va chercher dans le texte un «préfixe» (parmi les caractères non encore lus) correspondant à l'un des motifs recherchés puis renvoyer un objet de type `Yytoken` contenant des informations sur le préfixe trouvé. En fin de texte, la méthode renvoie la valeur `null`.
Voici l'exemple d'un code java qui va lire dans un fichier la totalité d'un texte et afficher entre crochets les tokens trouvés. (on suppose ici que la classe YYtoken dispose d'une méthode `image()` indiquant le texte du token)

```
Yylex yy = new Yylex(new BufferedReader(new FileReader(arg[0]))) ;
Yytoken token ;
while ((token = yy.yylex()) != null)
          System.out.print("["+token.image()+"]");
```

Si YYlex implémentait l'analyseur donné précédemment en exemple, le résultat serait
`[alpha][+][321][*][x5]`
En résumé l'utilisateur de JFlex doit

- concevoir un **fichier de spécifications (`.lex`)**
- fournir une classe (ou interface) **Yytoken** permettant de représenter les tokens (résultat de la méthode `yylex()`
- ainsi que des **classes utilisatrices**, au minimum une classe contenant `main(...)`

3 Le fichier de spécifications JFlex

Il détermine le fonctionnement de l'analyseur. La classe **Yylex** est fabriquée uniquement à partir des spécifications fixées dans ce fichier.
Il est composé de 3 parties (séparées par une ligne ne contenant que `%%`)

- Partie 1 : du code java qui sera repris tel quel au début du fichier `yylex.java`. On s'en servira essentiellement pour y préciser le nom du paquetage, ainsi que des clauses `import` si besoin est. Ce code **précède** la classe Yylex, on ne peut donc **pas** y définir d'attribut ou de méthode à ajouter à cette classe (ce qui serait une mauvaise idée, de toutes façons).
- Partie 2 : contient des **définitions de «macros»** JFlex, et/ou des **déclarations d'état** et/ou contient des **directives de génération de code**.
- Partie 3 : la spécification des tokens, chacun par un expression régulière. À chaque token est associé un bloc d'instructions java se terminant dans la majorité des cas par un `return valeur;` où valeur est un objet de type `Yytoken`. Ce sera la valeur renvoyée par la méthode `yylex()` à la rencontre de ce token.

3.1 Passons à un exemple

Ouvrez le fichier `exemple1.lex` et examinez son contenu.

- la partie 1 contient seulement la déclaration de paquetage qui permettra d'affecter la class Yylex au paquetage `exemple1`
- la partie 2 contient une directive indiquant que l'analyseur devra accepter les caractères unicode, puis deux définitions de macro (`ENTIER_SIMPLE` et `MOT_USUEL`). Une macro est un moyen de nommer une expression régulière dans l'intérêt de rendre plus lisible la partie 3.
- dans la partie3, se trouvent deux expressions régulières chacune suivie d'une action. L'analyseur (méthode `yylex()`) doit vérifier que le texte à analyser commence par un préfixe correspondant à l'une de ces deux expressions

— si le préfixe trouvé correspond au premier motif (`{MOT_USUEL}|{ENTIER_SIMPLE}`), alors la méthode `yylex()` s'arrête et renvoie un objet de type `Yytoken`
— si le préfixe trouvé correspond au deuxième motif : on remarque que l'action associée ne comporte pas de `return`. La méthode `yylex()` va alors recommencer aussitôt une nouvelle recherche (au cours de la même exécution)
— s'il n'existait pas de préfixe correspondant à l'un de ces motifs, un exception serait déclenchée. Notez que dans notre exemple cela ne peut pas se produire (les 2 motifs ont été choisis de façon à ce que tout texte comporte un préfixe correspondant à l'un d'eux)
— en fin de texte, la méthode `yylex()` renvoie `null`

3.2 Génération, compilation, test

L'exemple 1 est composé du fichier de spécifications `exemple1.lex` et de 2 classes du paquetage `exemple1` : `Yytoken` et `TestEx1`. Consultez et examinez le contenu de ces 2 classes.
L'application **JFlex** est elle-même écrite en java et figure dans un fichier `jar` «exécutable». On l'appelle par la commande :

`java -jar jflex-1.6.1.jar` *fichier.lex*

La classe Yylex sera créée dans le même répertoire que le fichier `.lex` (l'option -d permet de choisir une autre destination)
Placez vous dans le répertoire `exemple1` (au dessus de `src`) puis effectuez les commandes suivantes :
— génération de l'analyseur : `java -jar ../jflex-1.6.1.jar src/exemple1/exemple1.lex`
— compilation java : `javac -cp src -d bin src/exemple1/TestEx1.java`
— exécution de l'analyseur sur le fichier de test : `java -cp bin exemple1.TestEx1 test.txt`

3.3 Exercice

Modifiez l'exemple pour lui faire prendre en compte les motifs identificateur, entier et opérateur.

4 Un peu plus formellement

4.1 Syntaxe des expressions rationnelles

Voici les principales différences avec les syntaxes que vous connaissez déjà (javascript, egrep).
— Le caractère guillemets (`"`) est un caractère spécial servant à encadrer une suite de caractères. À l'intérieur des guillemets les caractères «spéciaux» deviennent «normaux», à l'exception de `\` et de `"` lui-même. par exemple `"a (et) b"` désigne le mot `a (et) b`
— Les classes de caractères peuvent être imbriquées : `[[a-z][0-9]]` équivaut à `[a-z0-9]`
— Les classes prédéfinies sont sensiblement différentes. Par exemple `[:letter:]` désigne une lettre (au sens unicode). Elles peuvent être utilisées avec un seul niveau de crochet.

4.2 Comment yylex() choisit le motif

Souvent, pour une liste de motifs donnée, il existe plusieurs façons de découper le texte. Dans l'exemple ci-dessus le texte `x5` peut être vu comme composé d'un seul token (identificateur `x5`) ou de 2 tokens (identificateur `x`, entier `5`). `yylex()` cherchera toujours le token **le plus long possible** (donc ici `x5`). Si plusieurs motifs conduisent à des tokens de même longueur, et seulement dans ce cas, `yylex()` prendra en compte le premier des ces motifs, selon l'ordre où ils figurent dans la spécification. Rappelons que si le texte ne commence par aucun des motifs, une erreur est déclenchée.

4.3 Quelques directives de compilation

Situées dans la partie 2 du fichier des spécifications, elles commencent par un caractère `%` qui doit se situer en tout début de ligne. Voici quelques directives utiles (liste non exhaustive)
— `%unicode` : prise en compte des caractères unicode
— `%line` : active le comptage des lignes (du fichier de texte). Le numéro de ligne courante est alors accessible dans un attribut `yyline`

— `%column` : idem pour le numéro de colonne (attribut `yycolumn`)
— `%char` : idem pour le décompte des caractères lus (attribut `yychar`)
— `%{`
```
  // code java
%}
```
Le code java est inséré **dans** la classe **Yylex** (afin de l'enrichir d'autres attributs et/ou méthodes)
— `%eofval{`
```
  // code java
%eofval}
```
définit le comportement quand l'analyseur arrive en fin de fichier (par défaut : renvoyer null). Typiquement, ce code peut contenir le «return» d'un objet particulier à renvoyer, plutôt que null.
— `%function "`*name*`"` : donne à la méthode principale un autre nom que `yylex`.

5 Exercice : enrichir la classe Yytoken

Nous allons concevoir un petit interpréteur d'expressions arithmétiques postfixées. Dans une expression postfixée, les opérandes précèdent l'opérateur. Par exemple la valeur de l'expression `50 7 +` est 57. Dans nos expressions, l'opérateur + est toujours un opérateur à 2 arguments, il consomme les deux opérandes et produit un résultat.
L'évaluation de l'expression `50 7 + 10 *` est 570 : l'addition produit le résultat 57 qui devient le premier opérande de la multiplication.
Un évaluateur d'expression arithmétique utilise un pile de valeurs. Il implémente l'algorithme :

tok ← lireToken() ;
while tok ≠ null **do**
  if tok est une valeur **then**
    Empiler la valeur de tok
  else
    if tok est un opérateur **then**
      Dépiler les arguments nécessaires à l'opérateur
      Empiler le résultat de l'opération
    end if
  end if
  tok ← lireToken() ;
end while
Renvoyer la valeur située en sommet de pile.

Nos expressions utiliseront les unités lexicales suivantes :
— Les entiers simples, non signés
— Les opérateurs à deux arguments notés `+, -, *, /` mais qui pourront aussi s'écrire `plus, minus,mult,q`
— L'opérateur à un seul argument : `opp` ("opposée de la valeur")

Les caractères d'espacement et de fin de ligne peuvent apparaître partout entre ces unités lexicales.
L'entité `Ytoken` sera cette fois une interface implémentée par plusieurs classes spécialisées pour les différents types de token.
Vous trouverez dans l'archive les fichiers
— `Yytoken.java` interface Yytoken
— `Valeur.java` classe implémentant les tokens valeurs entières
— `Operateur.java` classe abstraite implémentant les opérateurs
— `Plus.java` classe implémentant le PLUS
— `TestPost.java` classe avec main permettant de tester le découpage en token
— `Evaluateur.java` classe avec main évaluant une expression

Vous trouverez également une ébauche du fichier de spécifications `postfixees.lex`.
Complétez ce fichier de spécification et testez le avec la classe `TestPF`.
Complétez le développement des classes pour tous les types de tokens, en prenant exemple sur la classe `Plus`. Vous pouvez ensuite tester l'évaluateur (classe `Evaluateur`).

6 L'analyseur dans tous ses états

L'analyseur implémente la notion d'**état**. Par défaut, il existe un unique état prédéfini qui s'appelle `YYINITIAL` et l'analyseur est donc en permanence dans cet état.
L'utilisateur peut créer des états supplémentaires avec la directive `%state` suivie d'une liste de noms d'états.
Dans la partie 3 du fichier de spécification, la recherche de motif peut être rendue conditionnelle. Une règle telle que
`<YYINITIAL> [a-z]+ {return ...;}` signifie que le motif `[a-z]+` ne doit être recherché que si l'analyseur est dans l'état YYINITIAL.
Le changement d'état est déclenché par la méthode `yybegin(etat)`.
Voici par exemple comment s'opère un changement d'état :
`<YYINITIAL> [a-z]+ {yybegin(MON_ETAT); return ...;}`
Dans le fichier de spécification `postfixeesAvecCommentaire.lex` vous verrez une utilisation des états. Dans cette version, on introduit la possibilité de mettre des commentaires multilignes délimités par `/*` et `*/`, ce qui impose d'une part d'ignorer les autres motifs situés dans les commentaires et d'autre part de compter le nombre de `/* */` imbriqués pour déterminer la fin de la zone de commentaire.