TDM 3 - Automates et Langages

octobre 2018

Analyse lexicale avec JFlex - 1ère partie

Un **analyseur lexical** est un composant logiciel qui découpe un flux de données composé de caractères en une suite d'entités de niveau supérieur : les **unités lexicales** (ou « **tokens** »). Une unité lexicale est spécifiée par une expression régulière.
Un analyseur lexical est aussi appelé « **lexical parser** », « **tokenizer** », ou « **scanner** ».
Nous utiliserons un générateur d'analyseur lexical nommé **JFlex** qui produit un composant JAVA.

1 Analyse lexicale : principe et exemple

1.1 Les tokens

Un **token** (« unité lexicale ») est une portion du texte qui correspond à un motif préalablement spécifié par une expression régulière.
Par exemple si l'on définit 3 motifs :

- entier : `[0-9]+`,
- identificateur : `[A-Za-z][A-Za-z0-9]*`
- opérateur : `[-+*/]`

le texte `alpha+321*x5` sera découpé en une suite de 5 tokens :

IDENT	OPE	ENTIER	OPE	IDENT
`alpha`	`+`	`321`	`*`	`x5`

1.2 L'analyseur lexical

L'analyseur lexical est un composant logiciel intégré à une application. Il offre aux autres composants une méthode permettant de lire des données token par token plutôt que caractère par caractère.
Avantages :

- le reste de l'application n'a pas à se soucier de la syntaxe des tokens. Une éventuelle modification de la syntaxe des tokens ne concernera donc que l'analyseur lexical .
- un même analyseur lexical peut être intégré à plusieurs applications.

1.3 Générateur d'analyseur lexical

Un analyseur lexical peut être développé directement, mais on peut aussi utiliser un outil appelé **générateur d'analyseur lexical**.
Le plus connu est l'utilitaire historique nommé **lex** qui génère un analyseur lexical écrit en langage C.
Nous utiliserons **JFlex**, qui produit un analyseur lexical écrit en JAVA.
Le développeur écrit un fichier de **spécification des tokens**, le « source JFlex » (extension `.lex`).
À partir de ce fichier, JFLEX produit une **classe JAVA** qui implémente l'analyseur lexical. Le développeur intègre cette classe à son application.

2 JFlex par l'exemple

2.1 Le fichier de spécifications (.lex)

```
/* Exemple du sujet - version 0 */

%%

%unicode

%%
```

```
[0-9]+
  { return new Yytoken(TokenType.ENTIER); }
[A-Za-z][A-Za-z0-9]*
  { return new Yytoken(TokenType.IDENT); }
[-+*/]
  { return new Yytoken(TokenType.OPERATEUR); }
[\s\n]
  {}
```

En observant cet exemple on constate :

- le fichier comporte 3 parties, chacune séparée de la précédente par une ligne ne contenant que `%%`
- **la partie 1 :** (avant les premiers `%%`) est ici vide. Elle pourrait contenir du code java à inclure **avant** le code de l'analyseur (déclaration de paquetage, clauses `import` ...)
- **la partie 2 :** peut contenir
  - des **options** ; ici une seule option : `%unicode` qui indique qu'il faut prendre en compte les caractères unicode (recommandé)
  - des **définitions de « macros »**. Il n'y en a pas dans notre exemple.
- **la partie 3 :** une suite de **spécifications** (expressions régulières). Chaque expression est suivie de l'**action** (code JAVA) à exécuter quand un texte lui correspondant est rencontré.
  L'exemple comporte 4 expressions. Pour 3 d'entre elles (les tokens) l'action JAVA se termine par un `return` suivi de la valeur associée au token (nous reviendrons plus loin sur le type de cette valeur). La 4ème expression indique qu'à la rencontre d'un espace ou d'une fin de ligne l'analyseur ne doit rien faire (action vide).

2.2 Recherche de token : principe

À chaque progression élémentaire (recherche de token) l'analyseur cherche **le plus long texte** préfixe qui correspond à l'une des expressions spécifiées. En cas de succès, l'action correspondante est alors exécutée.

- si cette action se termine par `return` (un token a été trouvé) la progression élémentaire s'arrête et la valeur du token est renvoyée.
- si l'action ne comporte pas de `return`, le texte trouvé est ignoré (après exécution de l'action) et l'analyseur reprend aussitôt la recherche d'un token selon le même principe.
- si la fin des données est rencontrée, une valeur spéciale est renvoyée (en général : `null`)
- si un texte ne correspondant à aucune des expressions est rencontré, une `Error` est déclenchée.

Voici ce qui se passe si le texte à analyser est id34 45 :

- **recherche d'un premier token :** le plus long préfixe convenable trouvé est id34. il correspond à la 2ème expression (ident). La valeur de ce token est renvoyée.
- **recherche d'un deuxième token :** le plus long préfixe convenable trouvé est l'espace. il correspond à la 4ème expression. L'action correspondante est exécutée : ici l'action est vide et la recherche reprend donc à partir du caractère suivant. Se trouve ensuite 45 qui correspond à la 1ère expression (entier). La valeur de ce token est renvoyée.
- **recherche d'un troisième token :** la fin des données est atteinte. La valeur `null` est renvoyée.

2.3 L'analyseur : la classe Yylex

JFlex génère le **code java** de l'analyseur : il s'agit **d'une seule classe** nommée (par défaut) `Yylex`. Une **instance** de la classe `Yylex` est dédiée à la lecture d'un unique flux de données (fichier) et permet sa lecture token par token. La classe doit donc être instanciée avec un `java.io.Reader`, connecté en général à un fichier.

- La principale méthode publique de la classe `Yylex` s'appelle `yylex()` :
  `public Yytoken yylex() throws java.io.IOException;`
  c'est cette méthode qui déclenche une progression élémentaire dans le flux de données à la recherche du prochain token

Elle envoie `null` en cas de fin de fichier ou déclenche une `Error` en cas d'erreur de lecture.

Le type du résultat s'appelle `Yytoken` : c'est au développeur (c'est à dire à vous) d'écrire cette classe `Yytoken`.

— Citons aussi la méthode
`public final String yytext();`
qui renvoie la chaîne de caractères correspondant au dernier token lu (cette méthode ne doit être invoquée qu'après une invocation de `yylex()`).

2.4 Comment utiliser Yylex ?

Voici un exemple simple d'utilisation de la classe `YYlex`. Remarque : on suppose dans notre exemple que la classe `Yytoken`, écrite par le développeur, possède une méthode `getType()` qui renvoie le type du token (ici un type énuméré)

```
  // instanciation de l'analyseur lexical (tokenizer) :
Yylex tokenizer = new Yylex(new BufferedReader(new FileReader("data.txt")));
 // recherche du premier token :
Yytoken token  = tokenizer.yylex();
 // traitement et lecture des suivants :
while ( token != null ) {
  System.out.println( "token : " + token.getType() );
  token =  tokenizer.yylex();  // recherche du  token suivant
}
```

Sur l'exemple de données déjà rencontré (`alpha+321*x5`) le résultat de l'exécution sera :

```
token : IDENT
token : OPERATEUR
token : ENTIER
token : OPERATEUR
token : IDENT
```

3 Exercices

3.1 L'exemple complet

L'exemple ci-dessus vous est fourni au complet dans le dossier `exsujet0`. Il comporte
- un fichier de spécification `.lex`
- une classe `Yytoken`
- d'autres classes de l'« application » :
  - un enum `TokenType`
  - une classe `ExSujet0` contenant le `main`

Exercice 1 :
Q 1 . Copiez cette arborescence dans votre espace disque et ouvrez ces différents fichiers dans un éditeur pour examiner leur contenu.
Dans un terminal, placez-vous dans le dossier `exsujet0`, puis exécutez les commandes suivantes
- `jflex src/exsujet0.lex` (génération de l'analyseur : création du fichier `src/Yylex.java`)
- `javac -cp src -d bin src/ExSujet0.java` (compilation java)
- `java -cp bin ExSujet0` (exécution de l'appli, lecture de l'entrée standard)
  ou `java -cp bin ExSujet0` <*nom de fichier*>

Essayez ensuite ce programme sur plusieurs données. Essayez notamment des données « incorrectes » (comportant du texte qui n'est ni un token, ni un espace).
Q 2 . Modification de la classe `Yytoken`. Elle comportera maintenant un 2ème attribut de type String contenant le texte source du token rencontré et son accesseur `public String getSource();`
La valeur du texte source sera fournie à l'instanciation.

Modifiez également, dans le fichier `.lex`, les valeurs des `return` pour tenir compte de la nouvelle définition de `Yytoken`. NB : vous utiliserez ici la fonction `yytext()`
Enfin, vous mettrez à jour le programme principal pour obtenir un affichage du style :

```
token : IDENT (alpha)
token : OPERATEUR (+)
token : ENTIER (321)
...
```

3.2 Les macros

L'utilisation des macros est très fortement recommandée.
Définir une « macro » JFlex consiste à donner un nom à une expression régulière. Ce nom peut être ensuite utilisé en l'entourant par des accolades. Les macros se définisssent dans la partie 2 du fichier de spécifications. Exemple :

```
%%
%unicode
LETTRE=[a-zA-Z]
CHIFFRE=[0-9]
%%
{LETTRE}({LETTRE}|{CHIFFRE})*
    { return new Yytoken(TokenType.IDENT); }
// etc
```

Exercice 2 :
Q 1 . Modifiez le fichier de spécification pour définir et utiliser les macros : LETTRE, CHIFFRE, OP, IDENT, ENTIER. Mettez à jour l'application et testez.
Q 2 . Modifiez la syntaxe des tokens :
- — toutes les lettres, même accentuées, seront acceptées (utiliser une classe de caractère prédéfinie)
- — dans un identificateur on acceptera le underscore mais ni au début, ni à la fin, ni 2 underscores successifs.
- — idem dans les entiers

Recontruisez l'application et testez.

3.3 Des automates sous le capot

L'analyse lexicale utilise des automates. Partant des expressions réguilères, JFlex procède en 3 étapes
- — construction d'un automate non détermiste (NFA = Non-deterministic Finite Automaton)
- — déterminisation. Obtention d'un premier DFA (Deterministic Finite Automaton)
- — calcul d'un automate déterministe minimal

Lors de l'exécution de jflex, le nombre d'états de chacun des automates est d'ailleurs affiché.
Les automates eux-mêmes peuvent aussi être visualisés.

Exercice 3 :
Reconstruisez l'analyseur en utilisant l'option `-dot` de JFlex.
Ceci crée 3 fichiers : `nfa.dot, dfa-big.dot, dfa-min.dot`
La commande dot permet de convertir les .dot en .pdf : `dot -T pdf -O fichier.dot`
Visualisez les 3 automates.
NB : l'automate non déterministe n'est pas *exactement* du type de ceux vus en cours car il contient des ε-transitions.

3.4 Permettre les commentaires

Exercice 4 :
Le but de cet exercice est d'autoriser l'éditeur des données à y ajouter des commentaires. Les commentaires ne sont pas autorisés « à l'intérieur » d'un token.

Les commentaires ne seront pas considérés comme des token. L'analyseur lexical les autorisera mais ils n'apparaîtront jamais dans les résultats de `yylex()` (comme les espaces ou les fins de ligne, en résumé).
Les modifications que vous ferez dans cet exercice n'impacteront donc que l'analyseur lexical, pas les autres fichiers source.
Q 1 . Une portion de ligne comprise entre le premier `#` et la fin de ligne est considérée comme un commentaire.
Q 2 . Un texte commençant par `{` et se terminant par `}` est **aussi** considéré comme un commentaire. Ce genre de commentaires peut s'étendre sur plusieurs lignes. Il est bien précisé que chaque commentaire se termine au **premier** `}` rencontré.
Par exemple `123 {blabla} cd` présente une erreur de syntaxe : il comporte successivement un token ENTIER, un commentaire, et un token IDENT. Le dernier `>` doit provoquer une erreur.
Q 3 . On ajoute un troisième type de commentaire : un texte commençant par `<!-` et se terminant par `->` Ce genre de commentaires peut s'étendre sur plusieurs lignes. Il est bien précisé que chaque commentaire se termine au **premier** `->` rencontré.

4 Syntaxe JFlex

La structure du fichier `.lex` a été décrite par les exemples ci-dessus, nous n'y revenons pas.

4.1 Lien utile

Site et documentation jflex.

4.2 Les expressions régulières

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]`
- Classes de caractères prédéfinies : `[:letter:], [:uppercase:], [:lowercase:], [:digit:]`
- `\p{`*propriété unicode*`}` est aussi une classe de caractères, définie par une propriété unicode. Par exemple `\p{Punctuation}` qui désigne les caractères de ponctuation, `\p{Letter}` qui est équivalent à `[:letter:]` ou `\p{Greek}` qui désigne les lettres grecques.

Voir documentation JFlex, chapitre 4.3 : `http://jflex.de/manual.pdf#subsection.4.3`

4.3 Quelques directives et options JFlex

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.

— `%class` *name* : choisit le nom à donner à la classe engendrée (par défaut `Yylex`)
— `%function` *name* : choisit le nom à donner à la méthode principale (par défaut `yylex`)
— `%type` *name* : choisit le type du résultat de `yylex()` (par défaut `Yytoken`)

La liste complète est dans la documentation JFlex, chapitre 4.2 :
`http://jflex.de/manual.pdf#subsection.4.2`