Compilation : analyse lexicale avec JFlex

TP Compilation

Analyse lexicale

Eric Ramat
ramat@lisic.univ-littoral.fr

9 mai 2014

Durée : 6 heures

---

1 Introduction

Le but de cet TP est de vous donner les bases nécessaires afin de pouvoir écrire votre analyseur lexical en Java. En C, on utilise généralement le générateur d'analyseur lexical flex de la fondation GNU `http://www.gnu.org/software/flex/manual/`. Pour Java, nous utiliserons JFlex (`http://jflex.de/`).

1.1 Analyseur lexical

L'analyseur lexical lit les caractères de l'entrée un à un afin de séparer les unités lexicales :
— séparateurs, commentaires, mots clés ;
— identificateurs, entiers, réels, symboles, ...

Ces entités correspondent à des expressions régulières. Ainsi, les symboles appartiennent à un ensemble fini de caractères, et les identificateurs correspondent par exemple à une suite finie de lettres. L'analyse lexicale correspond à la détection de ces unités lexicales.

Pour cela, l'ensemble des unités lexicales reconnaissables est donné au générateur d'analyseur lexical par le biais d'une grammaire composée d'expressions régulières. Cette grammaire distingue donc un nombre fini de classes d'unités lexicales, encore appelées *lexèmes* (token en anglais). On retient ensuite, pour chaque unité lexicale reconnue, sa valeur et sa classe qui seront utilisées dans la construction de l'arbre de syntaxe.

Exemples :
— le mot-clef "int" (INT) correspond à la chaîne *int* ;
— un entier (DIGIT) est donné par : $0, 1, 2, 3, 4, 5, 6, 7, 8, 9+$;
— un identificateur (ID) est donné par : $a, ..., z, A, ..., Z+$.

Remarque : il y a ici une ambiguïté avec l'unité lexicale int qui appartient à deux classes INT et ID. C'est l'ordre de définition des classes qui imposera le choix parmi les différentes classes auxquelles appartiennent l'unité lexicale (c'est la classe définie en première qui est retenue).

En pratique, on donnera à l'outil d'analyse syntaxique (en général flex, JFlex , flex++, ...) un fichier contenant la grammaire des lexèmes qu'il doit reconnaître. L'outil construira automatiquement à partir de

ce fichier du code dans le langage de programmation qui nous intéresse (C, C++, Java, ...) un analyseur lexical qui découpera l'entrée en une suite d'unités lexicales.

2 JFlex

2.1 Premier exemple

Un exemple de spécification JFlex est donné dans les exemples fournis avec JFlex . Il se trouve dans le répertoire *? ?/examples/standalone*. Le code du fichier standalone.flex est recopié ici :

```
%%

%public
%class Subst
%standalone

%unicode

%{
String name;
%}

%%

"name " [a-zA-Z]+ { name = yytext().substring(5); }
[ Hh ] "ello" { System.out.print(yytext() + " " + name + "!"); }
```

2.2 Format d'un fichier JFlex

Le code d'un fichier JFlex est divisé en trois parties comme indiqué ci-dessous :

```
Code de l'utilisateur
%%
Options et declarations de macros
%%
Regles lexicales
```

2.3 Code de l'utilisateur

Le code de l'utilisateur sera recopié tel quel dans l'entête du fichier généré par JFlex . Ce fichier généré sera un fichier Java, contenant une unique classe. Le code écrit ici contiendra essentiellement des chargements de packages et de bibliothèques.

2.4 Options et déclarations de macros

Les options permettant des passer des arguments à l'outil JFlex . Vous en trouverez quelques unes ci-dessous. Se référer au manuel JFlex pour plus de détails :

- `%class` toto demande à JFlex de nommer le fichier produit toto.java. La classe contenue dans ce fichier sera elle aussi nommée toto ;
- `%public`, `%final`, `%abstract`, `%implements interface1` demande à JFlex de déclarer la classe produite avec l'option correspondante ;
- %cup permet d'utiliser JFlex avec Cup le générateur d'analyseur syntaxique ;
- `%line %column` permet de compter les lignes et les colonnes dans les variables yyline et yycolumn respectivement ;

— `%standalone` permet d'utiliser JFlex seul. Ainsi, la classe générée contiendra une fonction main ;
— `%state` toto déclare un identifiant d'état de l'analyse lexicale.

Les déclarations de macros permettent des abréviations dans la définition des règles lexicales. Elles sont de la forme `idmacro1 = ER1`. Ainsi, on pourra utiliser par la suite l'expression régulière `ER1` en notant simplement `idmacro1`.

2.5 Les expressions régulières de JFlex

Les caractères | ( ) { } [ ] < > \. * + ? ^$ / "! sont les caractères spéciaux de JFlex pour les expressions régulières. Pour représenter un caractère qui est un caractère spécial, il faut le dé-spécialiser en le faisant précéder de \, ou en l'entourant de " ".

Les séquences \n \t \f \b \r sont appelées séquences d'échappement.

Les opérateurs de composition des expressions régulières sont standard ; si a et b sont des expressions régulières :

— a|b (union) est l'expression régulière « a ou b » ;
— a b (concaténation) est l'expression régulière « a suivie de b » ;
— a* (clôture de Kleene) est l'expression régulière qui représente toute répétition de a y compris aucune ;
— a+ (itération) est équivalente à aa* ;
— a ? (option) est l'expression régulière « a ou rien » ;
— a{n} répétition de l'expression régulière a n fois ;
— (a) la même chose que l'expression régulière a ;
- [^a tout sauf l'expression régulière a

Voici maintenant les éléments de base des expressions régulières :

— le caractère . représente tout caractère sauf \n ;
— une séquence d'échappement représente respectivement :
— \t : le caractère tabulation ;
— \b : le retour en arrière ;
— \n : la fin de ligne ;
— \r : le retour à la ligne (attention, une terminaison de ligne sera donc représenté par \n sous Unix, par \r \n sous Windows) ;
— un caractère non spécial représente ce caractère (par exemple, a représente le caractère 'a') ;
— un caractère dé-spécialisé par \perd sa signification : on représente le signe + par \+, le guillemet par \", ...
— une classe de caractères est une suite de caractères non spécialisés et de séquences d'échappement entre crochets : elle représente n'importe quel caractère de cette classe. On peut aussi définir des intervalles de caractères. Par exemple [\ta-dAEIOU0-4] représente soit le caractère tabulation soit un des caractères suivants : a b c d A E I O U 0 1 2 3 4 ;
— une classe de caractères par complément est une suite de caractères non spéciaux et de séquences d'échappement entre [^et]. Elle représente tout caractère qui n'est pas dans la classe ;
— une chaîne de caractères dé-spécialisés est une suite de caractères non vides sans \ni \", entourée de ". Elle représente exactement cette suite de caractères dé-spécialisés : tous les caractères spéciaux (sauf \et \" qui sont interdits) perdent leur signification et sont considérés tels quels. Par exemple, "**" représente une balise ouvrante de commentaire Javadoc ;
— si on a défini une macro `mamacro` par `mamacro = <exprReg>`, alors on peut utiliser `mamacro` en l'entourant de et : `mamacro` représente l'expression régulière `<exprReg>`.

Remarques : Les blancs et les tabulations sont ignorés par JFlex (on peut donc les utiliser pour rendre les expressions régulières plus lisibles), sauf quand ils sont entre guillemets ou crochets. Par exemple [n] représente soit le caractère blanc soit la fin de la ligne, " " représente l'espace. On peut dé-spécialiser tous les caractères « spéciaux »sauf \et " en les entourant de ". On peut dé-spécialiser tous les caractères spéciaux en les préfixant par \.

3 Règles lexicales

Cette dernière section associe à des expressions régulières une action que l'analyseur généré doit effectuer quand il rencontre un symbole reconnu par une de ces expressions régulières. Une telle association est appelée *règle lexicale*.

3.1 Syntaxe

Les expressions régulières utilisées sont celles introduites en section 2.5. On peut faire précéder une expression régulière par ^pour spécifier que cette expression régulière ne doit être reconnue qu'en début de ligne.

Une action est de la forme <code Java>. On associe une action à une expression.

Méthodes et champs utilisables dans les actions JFlex fournit l'API suivante :

— String yytext() retourne la portion de l'entrée qui a permis de reconnaître le symbole courant ;
— yylength() retourne la longueur de la chaîne yytext() ;
— yyline retourne le numéro de la ligne du premier caractère de la portion de texte reconnue (disponible si l'option %line a été utilisée) ;
— yycolumn retourne le numéro de la colonne du premier caractère de la portion de texte (disponible si l'option %column a été utilisée).

3.2 Comment est reconnue l'entrée ?

Quand il consomme le flot d'entrée, le scanner détermine l'expression régulière qui reconnaît la portion de l'entrée la plus longue. Ainsi, si on a défini les macros :

Bool = bool
Ident = [:letter :][:letterdigit :]

et que l'on a les deux règles lexicales :

Bool System.out.println("Bool") ;
Ident System.out.println("Ident")

alors le flot d'entrée `boolTmp` sera reconnu comme un symbole `boolTmp` de la classe *Ident* et non comme le symbole bool de la classe *Bool* suivi du symbole `Tmp` de la classe *Ident*. Si deux expressions régulières reconnaissent une portion de l'entrée de même longueur, alors l'analyseur choisit la première expression rencontrée dans le fichier. Dans ce cas ci-dessus, le flot d'entrée `bool` sera reconnu comme comme un symbole de la classe *Bool* (probablement un mot-clef) et non un symbole de la classe *Ident*.

La dernière règle de la section servira donc au traitement d'erreur. On pourra par exemple écrire :

```
...
\textbackslash n { ; } // On ignore les fins de ligne
... autres regles ici ...
. { /* erreur */ }
```

Cette dernière règle reconnaît le premier caractère d'un flot d'entrée qui n'est reconnu par aucune des règles précédentes (rappel : . signifie « tout caractère sauf \n »). L'action correspondant à une erreur peut être soit le lancement d'une exception définie par l'utilisateur (par exemple, *throw new ScannerException("symbole inconnu")*), soit le retour d'un symbole spécial.

Caractéristiques de l'analyseur généré

Constructeurs : la classe générée contient deux constructeurs : l'un prend en paramètre un flot d'entrée d'octets *java.io.InputStream* ; l'autre prend en paramètre un flot d'entrée de caractères *java.io.Reader*. Généralement, on lit un flot de caractères sur l'entrée standard (paramètre effectif *System.in*) ou dans un fichier.

Le tampon d'entrée est vidé à chaque terminaison de ligne, ce qui déclenche l'analyse des caractères qu'il contient.

Méthode d'analyse : la classe générée contient aussi une méthode d'analyse (dont le nom par défaut est Yylex) destinée à être appelée par l'analyseur syntaxique. Elle retourne des symboles, par défaut de type Yytoken. Les symboles sont décrits par une classe Java existante (par exemple *java.cup.runtime.Symbol*) ou par une classe écrite par l'utilisateur. Le constructeur d'un symbole prend typiquement en paramètre une valeur d'un type énuméré représentant les différentes classes de symboles. Lorsqu'une classe contient plusieurs symboles, l'analyseur syntaxique a souvent besoin d'une information supplémentaire concernant « la valeur »du symbole. Cette valeur est un attribut de type Object, second paramètre du constructeur.

La méthode d'analyse générée peut être vue comme une boucle dont le corps est une suite de conditionnelles associant à une classe de symbole l'action spécifiée par l'utilisateur dans la règle lexicale correspondante. Cette action est effectuée quand un symbole de cette classe est reconnue. Si l'action ne contient par de return, alors elle se termine et la méthode d'analyse continue en analysant le prochain symbole. Si l'action contient un return <expression> où expression désigne un symbole, alors la méthode termine en retournant ce symbole.

Lors de la construction d'un analyseur lexical, il faut donc bien réfléchir aux symboles à définir :

— certains sont consommés par l'analyseur sans être communiqués à l'analyseur syntaxique (typiquement les commentaires) ;

— certains sont transmis à l'analyseur syntaxique sans qu'un attribut lui soit associé (typiquement les mot-clés et les opérateurs), ce qui donnera une règle lexicale du genre :

```
if { // on a reconnu le mot-cle IF , on retourne le symbole
     // correspondant
     return new Yytoken (ClasseSymbole.IF);
}
```

en supposant que la classe ClasseSymbole définisse sous forme de type énuméré les différentes classes de symboles existantes, parmi lesquelles IF et ENTIER ;

— certains sont transmis à l'analyseur syntaxique avec un attribut associé (typiquement une valeur pour un classe numérique, un nom pour les identificateurs), ce qui donnera une règle lexicale du genre :

```
[:digit:]+ {// on a reconnu un entier, on retourne le symbole
            // correspondant et sa valeur
            return new Yytoken ( ClasseSymbole.ENTIER, new Integer(yytext()
```

4 Travail à réaliser

4.1 Traitement de factures

Une facture se présente ainsi :

```
FACTURE numéro facture
Libellé1    Quantité1    Prix1,
Libellé2    Quantité2    Prix2,
TOTAL       Prix.
```

Le numéro de facture est composé des deux premières lettres du mois et du numéro de la facture de ce mois. Les libellés sont des suites de lettres. Les quantités et les prix sont des entiers.

A l'aide de JFlex, créez une analyseur lexical qui reconnaît les unités lexicales suivantes :

(1) FACTURE = FACTURE
(2) NO = $lettre^2 chiffre^3$
(3) LIB = $lettre^+$
(4) NB = $chiffre^+$
(5) VIRG = ,
(10) TOTAL = TOTAL
(11) PT = .

A l'exécution, votre programme affichera la suite d'unités lexicales et le numéro de la ligne correspondante.

```
FACTURE JA001
Pomme    3    10,
Orange   5    7,
TOTAL    65.
```

L'analyseur affiche :

```
ligne 1 : FACTURE
ligne 1 : NO
ligne 2 : LIB
ligne 2 : NB
ligne 2 : NB
ligne 2 : VIRG
ligne 3 : LIB
ligne 3 : NB
ligne 3 : NB
ligne 3 : VIRG
ligne 4 : TOTAL
ligne 4 : NB
ligne 4 : PT
```

4.2 Calcul du nombre d'occurrences d'entiers et de réels

En utilisant JFlex, générer un analyseur lexical qui affiche et compte le nombre d'entiers et de réels dans un fichier texte.

4.3 Substitution et remplacement dans un fichier

En utilisant JFlex, générer un analyseur lexical qui, dans un fichier comportant sur la première ligne la déclaration suivante (`nom toto`), remplacera toutes les occurrences du mot `bonjour` dans le reste du texte par `bonjour toto`.

4.4 Evaluateur d'expressions postfixées

Écrire un programme JFlex permettant d'évaluer des expressions post-fixées :

— $n \in N$est une expression postfixée ;

— si s1 et s2 sont des expressions postfixées, *s1 s2 op* avec $op \in \{+, -, *, /\}$ est une expression postfixée.

Exemple :

$$72 * 31 - 7 * + \rightarrow 28$$
$$119 - 56/ * 2+ \rightarrow 3.66$$