Introduction à JFlex

licence info S5 et GMI S6
documentation COMPIL – mars 2005, revu août 2007

FIL

Pour toutes remarques, questions, suggestions : `mirabelle.nebut@lifl.fr`

JFlex[1] est un générateur d'analyseurs lexicaux pour Java : il génère des analyseurs lexicaux écrits en Java. Le F de JFlex signifie « Fast » : JFlex est une récriture du générateur d'analyseurs lexicaux Jlex[2]. JFlex et Jlex diffèrent par leur mise en œuvre mais les fichiers de spécification et les techniques d'analyse utilisées sont très similaires. Leur ancêtre commun est Lex, un générateur d'analyseur lexical pour C. Ils sont destinés à être utilisés en collaboration avec le générateur d'analyseurs syntaxiques Cup[3]. On doit alors utiliser des classes du paquetage `java_cup.runtime` fourni par Cup. Au M5 vous trouverez ce paquetage dans le jar : `/usr/share/java/cup.jar`. N'oubliez pas de l'ajouter dans votre `CLASSPATH`.

Ce document est un résumé des caractéristiques principales de JFlex (celles dont vous servirez en TP). Pour plus d'information au M5, consulter la documentation au format html ou pdf :

```
/usr/share/doc/jflex/
```

Lancer JFlex Au M5 le binaire `jflex` se trouve dans le répertoire `/usr/bin/jflex`. Pensez à l'ajouter dans votre variable d'environnement `PATH`. On lance JFlex par la ligne de commande :

```
jflex <options> <fichierDeSpecification>
```

L'option `-d <directory>` indique que le fichier généré est écrit dans le répertoire `<directory>`.

On peut aussi utiliser la classe `JFlex.Main` :

```
java JFlex.Main <fichierDeSpecification>
```

Le fichier de sortie est généré dans le répertoire courant. Au M5 cette classe se trouve dans

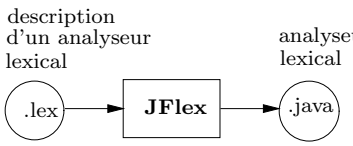
```
/usr/share/java/JFlex.jar
```

Table des matières

[1] `http://jflex.de/index.html`
[2] `http://www.cs.princeton.edu/~appel/modern/java/JLex/`
[3] `http:///www2.cs.tum.edu/projects/cup/`

1 Principes

Comme tous les générateurs d'analyseurs lexicaux, JFlex prend en entrée un fichier qui contient une description de l'analyseur lexical à générer. Il génère un fichier `.java` (cf section 3.1.1 pour le nom de ce fichier) contenant une classe Java de même nom qui contient l'analyseur lexical. Il faudra compiler cette classe avec `javac` (figure 1(a)).



(a) Génération d'une classe

```
<typeRetour> methAnalyse() throws <typeException> {
  while true {
    amener l'AFD de l'état initial à un état final ;
    effectuer une <action> ;
  }
}
```

(b) Code généré

Fig. 1 – Principes de JFlex

Le cœur de l'analyseur est une méthode appelée par la suite « méthode d'analyse », dont le code schématique est donné figure 1(b). Cette méthode extrait du texte à analyser le prochain symbole. JFlex permet de configurer le nom, le type de retour de cette méthode, et le type d'exception levée (cf section 3.1.2). Il est aussi possible de configurer son corps en associant une *action* à une description de symbole. Cette action sera effectuée dans l'état final correspondant au symbole. Dans l'utilisation habituelle d'un analyseur lexical, la méthode d'analyse est destinée à être appelée répétitivement par un analyseur syntaxique jusqu'à la fin du flot de caractères (fig. 2). L'action consiste alors le plus souvent à retourner le symbole reconnu, ce qui termine l'exécution de la méthode (la boucle infinie permet au contraire d'ignorer certains symboles). Dans d'autres utilisations, les actions pourront par exemple calculer une valeur qui sera retournée à la fin de l'analyse.

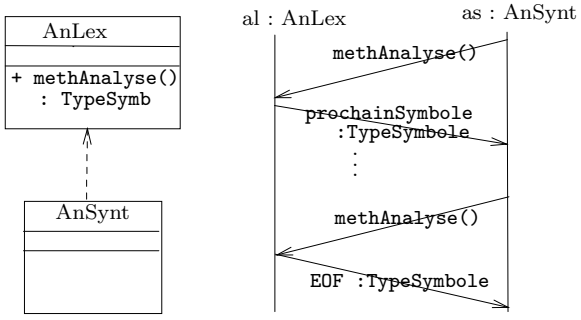


Fig. 2 – Collaboration avec un analyseur syntaxique

Structuration d'un fichier de spécification Un fichier JFlex est composé de trois sections séparées par des `%%` :

```
<section1>
%%
<section2>
%%
<section3>
```

La suite de ce document décrit succintement le contenu de ces trois sections :

- section 1 : code utilisateur ;
- section 2 : options et déclarations ;
- section 3 : règles lexicales.

2 Première section : code utilisateur

Le code placé dans cette section est inclus tel quel dans le fichier `.java` généré, avant la déclaration de la classe générée. On placera donc ici typiquement les instructions de déclaration de paquetage (clause Java `package`) et d'importation de paquetages et classes (clause Java `import`). Même si c'est déconseillé on peut aussi inclure dans cette section des classes dont l'utilisateur a besoin et qu'il définit lui même, typiquement celle déclarant les classes de symboles utilisées. Pour suivre les recommandations concernant un bon style de programmation Java, il vaut mieux mettre chacune de ces classes dans son propre fichier `.java`.

3 Seconde section : options et déclarations

3.1 Options

On ne décrit ici que les options qui serviront en TP. Dans la syntaxe JFlex une déclaration d'option commence par un `%` qui doit *impérativement* se trouver en début de ligne. Les options qui concernent la classe générée et la méthode d'analyse sont très utiles pour mettre au point l'*interface* entre l'analyseur lexical généré et un analyseur syntaxique avec lequel il collabore.

3.1.1 Options concernant la classe générée

Ces options affectent le nom de la classe générée, ainsi que ses différents composants.

- `%class <nomClasse>` demande à JFlex de générer un fichier `<nomClasse>.java` contenant une classe `<nomClasse>`. Par défaut ce nom est `Yylex`. Sans utilisation de l'option `-d` en ligne de commande, ce fichier sera généré dans le répertoire courant.
- `%implements <nomInterface1> [<nomInterface2>, ... , <nomInterfaceN>]` demande à ce que la classe générée implante les interfaces `<nomInterfacei>` (clause Java `implements`). Pour permettre l'interfaçage avec Cup elle doit implanter `java_cup.runtime.Scanner`[4].
- `%{`
  `...`
  `%}`
  Le code Java inclus entre `%{` et `%}` est recopié tel quel dans le corps de la classe générée. On peut s'en servir pour définir des attributs et des méthodes de l'analyseur.
- `%init{`
  `...`
  `%init}`
  Le code Java inclus entre `%init{` et `%init}` est recopié tel quel dans les constructeurs de la classe générée. On peut initialiser ici les attributs déclarés entre `%{` et `%}`, créer une table des symboles vide, etc.
- `%initthrow <exception1> [, <exception2>, ... , <exceptionN>]` indique que les constructeurs de la classe générée déclarent les exceptions `<exceptioni>` (clause Java `throws`).
- `%public` indique que la classe générée est déclarée publique.

3.1.2 Options concernant la méthode d'analyse

Ces options affectent la signature de la méthode d'analyse qui retourne le prochain symbole.

- `%function <nomMethode>` indique que cette méthode s'appellera `<nomMethode>`. Par défaut elle s'appelle `yylex`. Pour permettre l'interfaçage avec Cup elle doit s'appeler `next_token`.
- `%type <type>` indique que le type de retour de cette méthode est `<type>`. Pour permettre l'interfaçage avec Cup ce type doit être `java_cup.runtime.Symbol`.
- `%yylexthrow <exception1> [, <exception2>, ... , <exceptionN>]` indique que cette méthode déclare les exceptions `<exceptioni>` (clause Java `throws`).

[4] Il faudra alors un `import java_cup.runtime.* ;` en première section.

3.1.3 Options concernant le traitement de la fin du flot

On spécifie par ces options ce que l'analyseur retourne quand la fin du flot à analyser est atteinte, et quel code spécifique est exécuté.

- `%eofval{`
  `...`
  `%eofval}`
  Le code inclus entre `%eofval{` et `%eofval}` est recopié tel quel dans la méthode d'analyse, et exécuté chaque fois que la fin du flot est atteinte (en général une seule fois). Pour permettre l'interfaçage avec CUP ce code doit être
  `return new <TypeSymbole> (<DefTypeSymbole>.EOF);` où
  `<TypeSymbole>` est du type `java_cup.runtime.Symbol` et `<DefTypeSymbole>` est une classe qui définit chaque symbole par un entier (par défaut CUP génère la classe `sym`). Par exemple :
  `return new java_cup.runtime.Symbol(sym.EOF);`
  `EOF` est (le code d')un symbole généré automatiquement par CUP.
- `%eofclose` charge JFLEX de fermer le flot d'entrée après analyse. Cette option m'a posé souci pour les lectures de données au clavier demandées par l'exécution du programme analysé, enchaîné à la suite de son analyse. Je ne l'utilise pas : je ne comprends pas bien comment faire proprement.

3.1.4 Autres options

- `%unicode` indique à JFLEX que les caractères du fichier à analyser seront encodés selon la norme Unicode (sur 16 bits).
- `%line` ajoute dans la classe générée un attribut `yyline` qui contient le numéro de la ligne du premier caractère du dernier symbole reconnu.
- `%column` ajoute dans la classe générée un attribut `yycolumn` qui contient le numéro de la colonne du premier caractère du dernier symbole reconnu.

3.1.5 Interfaçage avec Cup

L'option `%cup` permet l'interfaçage avec CUP. Son utilisation équivaut à celle des quatre options suivantes qui ne doivent alors pas être utilisées dans le fichier de spécification :

- `%implements java_cup.runtime.Scanner`
- `%function next_token`
- `%type java_cup.runtime.Symbol`
- `%eofval{`
  `return new java_cup.runtime.Symbol(sym.EOF);`
  `%eofval}`
- `%eofclose`

3.2 Déclarations de macros et expressions régulières

On a vu que l'option `%{ ... %}` permet de déclarer du code utilisateur. On peut aussi déclarer dans cette section des macros. Une *macro* est une abbréviation pour une expression régulière. Si son nom est bien choisi, elle rend la description de l'analyseur lexical beaucoup plus lisible. L'utilisation des macros est donc fortement recommandée ! Une définition de macro est de la forme :

```
<idMacro> = <exprReg>
```

Les identificateurs de macro ont la même syntaxe que les identificateurs Java. Les expressions régulières de JFLEX sont détaillées en section 3.2.1.
Attention : une macro n'est pas un non-terminal de grammaire et ne peut être utilisée récusivement ! Elle peut être utilisée dans d'autres macros (on entourera alors son identificateur d'accolades), mais toute dépendance cyclique sera rejetée par JFlex.

3.2.1 Les expressions régulières de JFlex

Les caractères `| ( ) { } [ ] < > \ . * + ? ^ $ / " ~ !`
sont appelés caractères *méta* : ce sont les opérateurs ou *caractères spéciaux* de JFlex pour les expressions régulières. Les autres caractères sont appelés « *non méta* ». Pour représenter un symbole qui est un caractère méta, il faut le despécialiser en le faisant précéder de \, ou en l'entourant de `" "` (voir plus bas les chaînes de caractères despécialisés).

Les séquences de caractères `\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`~` (jusqu'à) est l'expression régulière qui représente « tout jusqu'à a incluse ».

On donne maintenant les éléments de base des expressions régulières ainsi que leur sémantique.

- Le caractère méta `.` représente tout caractère sauf `\n` (donc tout sauf une fin de ligne).
- Une *séquence d'échappement* représente respectivement :
  - `\t` : le caractère tabulation ;
  - `\b` : le caractère *backspace* ou retour arrière ;
  - `\n` : le caractère *line feed* ;
  - `\r` : le caractère retour à la ligne (*carriage return*) ;
  - `\f` : le caractère *form feed* (passage à une nouvelle page pour une imprimante).

  Une terminaison de ligne sera donc représentée par`\n` sous Unix, par `\r\n` sous Windows.
- un *caractère non méta* représente ce caractère (par exemple `a` représente le caractère 'a') ;
- un *caractère méta despécialisé* par \ perd sa signification : on représente le signe + par `\+`, le guillemet par `\"`, l'anti-slash par `\\`.
- une *classe de caractères* est une suite de caractères non méta 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 méta et de séquences d'échappement entre `[^` et `]`. Elle représente tout caractère qui n'est pas dans la classe. Par exemple `[^\n]` représente tout caractère qui n'est pas une terminaison de ligne. `[^]` représente n'importe quel caractère.
- une *chaîne de caractères despécialisés* est une suite de caractères non vide sans \ ni ", entourée de ". Elle représente exactement cette suite de caractères despécialisés : tous les méta catactères (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>`.

Il existe aussi des classes de caractères prédéfinies, certaines en relation avec le langage Java :

- `.` représente tous les caractères sauf `\n` ;
- `[:jletter:]` représente tout caractère susceptible de débuter un identificateur Java, c'est à dire toute lettre de l'alphabet et le caractère `_` ;
- `[:jletterdigit:]` représente tout caractère susceptible de figurer dans un identificateur Java sauf pour sa première lettre, c'est à dire toute lettre de l'alphabet ou chiffre, ou le caractère `_` ;
- `[:letter:]` représente les lettres de l'alphabet ;
- `[:digit:]` représente tout chiffre.

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 ligne, `" "` représente le caractère blanc, mais les quatre

expressions régulières suivantes `a | b`, `a| b`, `a |b` et `a|b` sont équivalentes et désignent le caractère `a` ou le caractère `b`.

On peut despécialiser tous les caractères méta sauf `\` et `"` en les entourant de `"`. On peut despécialiser tous les caractères méta en les préfixant par `\`. Ainsi `\+` et `"+"` sont deux expressions régulières équivalentes reconnaissant le signe +. Mais l'expression `"""` n'est pas une expression régulière correcte.

Exemples

- Les commentaires commençant par `//` et se terminant en fin de ligne peut être reconnus par l'expression régulière `"//".*` (rappel : `.` représente tout caractère sauf `\n`). On n'est alors pas obligé de taper un retour-chariot pour terminer le commentaire, un `ctrD` sous Unix suffit. Pour faire propre, JFlex recommande `"//"[\r\n]*(\r|\n|\r\n)` ;
- Les commentaires Java *non imbriqués* balisés par `/*` et `*/` peuvent être reconnus par `"/*" ~ "*/"`.
- Les identificateurs Java peuvent être reconnus par `[:jletter:][:jletterdigit:]*`.
- Une chaîne constante de caractères à la Java est une suite de caractères sans `"` ni fin de ligne, encadrée par des `"`. Pour être représenté dans cette suite, un guillemet doit être despécialisé et représenté par `\"`. Si on définit la macro : `interieurChaine = [^\n\r\"]`, on pourra représenter les chaînes Java par : `\" ({interieurChaine} | \\\" )* \"`.

4 Troisième section : 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*.

4.1 Syntaxe

Les *expressions régulières* utilisées sont celles présentées en section 3.2.1. On peut faire précéder une expression régulière par `^` pour spécifier que cette expression ne doit être reconnue qu'en début de ligne. Une *action* est de la forme `{ <codeJava> }`. On associe une action *action1* à une expression *expression1* par la ligne :
expression1 action1
On peut associer une même action à plusieurs expressions en utilisant l'action spéciale `|` :
expression1 `|`
expression2 action1et2
Attention : il faut impérativement que l'expression régulière et l'action soient séparées par un blanc.

Il existe aussi un mécanisme d'état lexical (bien utile pour reconnaître des commentaires imbriqués) et un mécanisme de lecture d'une expression en avant (*look ahead*) qu'on ne détaillera pas ici.

4.1.1 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()`.
- `int yycharat(int pos)` retourne sous forme entière le caractère à la position `pos` de `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 reconnue (disponible si l'option `%column` a été utilisée).

4.2 Comment l'entrée est reconnue

Reconnaissance du plus long préfixe 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 = [:jletter:][:jletterdigit:]*`

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*.

Mécanisme de priorité Si deux expressions régulières reconnaissent une portion de l'entrée de même longueur, alors l'analyseur choisit la première expression rencontrée dans les règles lexicales[5]. Dans le cas ci-dessus le flot d'entrée `bool` sera reconnu comme un symbole de classe *Bool* (probablement un mot-clé) et non un symbole de classe *Ident*.

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

```
...
\n {\\ on ignore les fins de ligne}
... autres règles ici ...
. | \n {\\ 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 commençant par "+ yytext() + " ligne " + yyline + " colonne " + yycolumn) ;`, soit le retour d'un symbole spécial (délégation du traitement d'erreur à l'analyseur syntaxique).

4.2.1 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. Pour lire dans un fichier on peut :

– commencer par créer un objet `File` en utilisant son constructeur `File(String s)`, où `s` est le nom du fichier à lire ;
– puis créer un objet `FilerReader` en utilisant son constructeur `FileReader (File f)`. C'est cet objet `FilerReader` que l'on passe en paramètre du constructeur de l'analyseur.

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

Il est possible d'adapter les constructeurs par les options suivantes, déjà décrites en section 3.1.1 :

– `%init{`
  `...`
  `%init}`
– `%initthrow <exception1> [, <exception2>, ... , <exceptionN>]`

Méthode d'analyse La classe générée contient aussi une méthode d'analyse (dont le nom est par défaut `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 additionnelle concernant « la valeur » du symbole. Cette valeur est un *attribut* de type `Object`, second paramètre du constructeur.

La méthode d'analyse générée est 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

[5] Attention, l'ordre de définition des macros n'a lui aucune importance dans l'affaire.

action est effectuée quand un symbole de cette classe est reconnue. Si l'action ne contient pas de `return`, alors elle 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 lexical sans être communiqués à l'analyseur syntaxique (typiquement les commentaires), ce qui donnera une règle lexicale du genre :
  ```
  "//".* {// on a reconnu un commentaire, on l'ignore}
  ```
- Certains sont transmis à l'analyseur syntaxique sans qu'un attribut lui soit associé (typiquement les mot-clés et opérateurs), ce qui donnera une règle lexicale du genre :
  ```
  if {// on a reconnu le mot-clé IF, on retourne le symbole
  // correspondant
  return new Yytoken(ClasseSymbole.IF); }
  ```
  en supposant que la classe `ClasseSymbole` définit 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 une 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());}
  ```