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

L'outil Cup[1] — raccourci pour Java-Based Constructor of Useful Parsers — est un générateur d'analyseurs syntaxiques LALR (cas particulier d'analyseurs LR) à partir de spécifications. Inspiré de Yacc, il est écrit en Java et génère des analyseurs écrits en Java. Il est conçu pour travailler avec le générateur d'analyseurs lexicaux JFlex[2]. Un fichier de spécification pour Cup décrit une grammaire algébrique LALR (analyse syntaxique) éventuellement décorée par des actions sémantiques (analyse sémantique). Cette documentation n'aborde que l'analyse syntaxique, sans décorations.

Au M5 Cup se trouve ici : `/usr/share/java/cup.jar`. On ne décrit ici que les caractéristiques de Cup qui serviront en TP. La documentation complète se trouve dans `/usr/share/doc/cup/manual.html`.

## 1 Principes

Chaque analyseur syntaxique généré contient trois classes :
- la classe `parser` (par défaut) contient l'analyseur lui-même, avec ses tables, elle étend la classe `java_cup.runtime.lr_parser` ;
- la classe `sym` (par défaut) contient la définition de chaque terminal de la grammaire sous la forme d'un entier ; cette définition est utilisée par l'analyseur lexical ;
- la classe `CUP$action`, interne à la classe `parser` et privée, encapsule toutes les actions sémantiques présentes dans la grammaire.

Il est possible d'adapter ces trois classes en paramétrant le fichier de spécification.

## 2 Fichier de spécification

Un fichier de spécification pour Cup se comporte de 5 parties.

### 2.1 Paquetages

La première partie (optionnelle) débute à la première ligne du fichier et est copiée telle quelle en tête du fichier `parser.java`. Elle permet de déclarer que :
- le paquetage auquel appartient la classe `parser` est `<name>` par :
  `package <name> ;`
- la classe `parser` importe les classes du paquetage `<name>` par exemple par :
  `import <name>.* ;`

### 2.2 Inclusion de code utilisateur

La seconde partie (optionnelle) comporte quatre sous-parties optionnelles qui permettent d'inclure du code utilisateur à divers endroits de `parser.java`. On trouve dans l'ordre :

```
action code {: <code Java> :} ;
parser code {: <code Java> :} ;
init with {: <code Java> :} ;
scan with {: <code Java> :} ;
```

`action code` permet d'inclure du code utilisateur dans la classe `CUP$action`. `parser code` permet d'inclure du code utilisateur dans la classe `parser` : déclaration d'attributs, redéfinition de méthodes de `java_cup.runtime.lr_parser` (par exemple de `public void syntax_error(Symbol cur_token)` qui est appelée en cas d'erreur syntaxique et qui par défaut se contente d'afficher un message sur `System.err`), ajout de méthodes. `init with` permet d'inclure du code qui sera effectué avant l'analyse syntaxique (initialisations diverses). `scan with` retourne un symbole, par défaut celui retourné par l'appel à `next_token` sur l'analyseur lexical associé.

[1] `http:///www2.cs.tum.edu/projects/cup/`
[2] `http://jflex.de/index.html`

## 2.3 Liste des symboles terminaux et non-terminaux

La troisième section (obligatoire) a pour objectif la déclaration de tous les terminaux et non-terminaux de la grammaire, dans un ordre quelconque.

```
terminal <nom1> ;
terminal <nom2> , ... , <nomn>  ;
```

déclarent respectivement le terminal `nom1` et les terminaux `nom2`, ..., `nomn`.

```
nonterminal <nom1> ;
nonterminal <nom2> , ... , <nomn> ;
```

déclarent respectivement le non-terminal `nom1` et les non-terminaux `nom2`, ..., `nomn`. Cette section contient autant de directives `terminal` et `nonterminal` qu'on veut, dans un ordre quelconque. Pour une meilleure lisibilité de la spécification, on a tout de même intérêt à regrouper les terminaux avec les terminaux et les non-terminaux avec les non-terminaux. Les noms `code`, `action`, `parser`, `terminal`, `non`, `nonterminal`, `init`, `scan`, `with`, `start`, `precedence`, `left`, `right`, `nonassoc`, `import` et `package` sont réservés pour CUP. Chaque terminal et non-terminal est représenté à l'exécution par un objet de type `java_cup.runtime.Symbol`.

## 2.4 Déclaration de la priorité et de l'associativité des opérateurs

La quatrième partie (optionnelle) permet de déclarer la priorité et l'associativité d'éventuels opérateurs de la grammaire. Cette caractéristique de CUP est très pratique, car elle permet de spécifier une grammaire à opérateurs ambiguë !

Chaque déclaration spécifie un niveau de priorité *et* un type d'associativité. Le mot-clé `precedence` introduit un nouveau niveau de priorité, *supérieur au niveau précédent*. Les mot-clés `left` et `right` désignent respectivement l'associativité gauche et droite[3]. On utilisera deux types de déclaration, chacun faisant intervenir des terminaux de la grammaire représentant des opérateurs :

```
precedence left <terminal> ;
```

déclare un opérateur associatif à gauche,

```
precedence left <terminal>, ... , <terminal> ;
```

déclare un ensemble d'opérateurs associatifs à gauche *de même priorité*.

```
precedence right <terminal> ;
```

déclare un opérateur associatif à droite,

```
precedence right <terminal>, ... , <terminal> ;
```

déclare un ensemble d'opérateurs associatifs à droite *de même priorité*.

Tout terminal non déclaré dans cette partie se voit attribuer le degré de priorité le plus bas (niveau 0). Par exemple, les déclarations :

```
precedence left PLUS, MOINS;
precedence left DIV, MULT;
```

spécifient que les opérateurs `PLUS`, `MOINS`, `DIV` et `MULT` sont associatifs à gauche, et que le `PLUS` et le `MOINS` (niveau de priorité 1) sont moins prioritaires que le `MULT` et le `DIV` (niveau de priorité 2).

[3] Le mot-clé `nonassoc` existe aussi mais on ne l'utilisera pas. Il sert à interdire deux occurrences consécutives et de même priorité d'un terminal. Par exemple si `==` est déclaré `nonassoc` alors `6 == 3 == 2` génèrera une erreur.

### 2.4.1 Grammaire

La cinquième et dernière partie (obligatoire) décrit les productions de la grammaire algébrique qui sous-tend l'analyseur syntaxique. Elle débute éventuellement par la déclaration de l'axiome de cette grammaire :

```
start with <non-terminal> ;
```

Si cette déclaration est omise, le non-terminal apparaissant en partie gauche de la première production est choisi comme axiome. On trouve ensuite une suite de productions, chaque production étant de la forme :

```
<non-terminal> ::= <terOuNon-ter1> <terOuNon-ter2> ... <terOuNon-tern> ;
```

pour décrire `<non-terminal>` → `<terOuNon-ter1>` `<terOunOn-ter2>` ... `<terOuNon-tern>`, et

```
<non-terminal> ::=  ;
```

pour décrire `<non-terminal>` → $\epsilon$. On peut factoriser deux productions de même membre gauche par `|`. On écrira par exemple pour *listeIdent* → *IDENT* | *IDENT SEPVAR listeIdent* (NB : grammaire non LL(1) !) :

```
listeIdent ::= IDENT | IDENT SEPVAR listeIdent ;
```

### 2.4.2 Conflits et priorités

Chaque production possède un niveau de priorité : c'est celui du terminal le plus à droite dans sa partie droite. Si la production ne contient pas de terminaux alors elle a le niveau de priorité le plus bas. Il est possible de forcer le niveau de priorité d'une production en faisant précéder le `;` qui la termine par `%prec <terminal>`. La production a alors le niveau de priorité de `<terminal>`.

Les niveaux de priorité des terminaux et productions sont utilisés pour tenter de résoudre les conflits réduire/réduire et lire/réduire : la plus forte priorité gagne. Ils sont aussi utiles quand un même symbole a des priorités différentes selon les contextes. Par exemple `MOINS` a une priorité plus forte que celle de `PLUS`, `MULT` et `DIV` si c'est un moins unaire, et égale à celle de `PLUS` sinon. On introduit alors un terminal `MOINS-U` bidon qui a la priorité du moins unaire, et on écrit :

```
precedence left PLUS, MOINS;
precedence left DIV, MULT;
precedence left MOINS_U;
...
exp ::= ...
       | exp MOINS exp
       | MOINS exp %prec MOINS-U
;
```

## 2.5 Actions sémantiques

CUP permet d'attribuer la grammaire LALR(1) : association d'attributs aux terminaux et non-terminaux, et d'actions aux productions.

### 2.5.1 Déclaration des attributs

Quand on écrit une grammaire attribuée, on a l'habitude d'indiquer de manière globale le nom des attributs associés aux symboles terminaux et non-terminaux de la grammaire, ainsi que leur type. Le procédé de spécification de CUP est différent : on explicite globalement le type des attributs lors de la déclaration des terminaux et non-terminaux, mais le nom des attributs est local à chaque production. Chaque terminal ou non-terminal ne peut posséder qu'un attribut, nécessairement de type objet (on utilisera par exemple le type objet Java `Integer` et non le type primitif `int`).

La syntaxe est la suivante :

```
terminal <type_attribut> <nom1> ;
terminal <type_attribut> <nom2> , ... , <nomn>  ;
```

déclarent respectivement le terminal `nom1` et les terminaux `nom2`, ..., `nomn` comme possédant un attribut de type `type_attribut`.

```
nonterminal <type_attribut> <nom1> ;
nonterminal <type_attribut> <nom2> , ... , <nomn> ;
```

déclarent respectivement le non-terminal `nom1` et les non-terminaux `nom2`, ..., `nomn` comme possédant un attribut de type `type_attribut`.

Par exemple, dans une grammaire des expressions arithmétiques, on associe à `expr` et `ENTIER` un attribut de type entier :

```
terminal Integer ENTIER ;
non terminal Integer expr;
```

L'attribut d'un terminal est suppposé être fixé par l'analyseur lexical.

### 2.5.2 Actions et accès aux attributs

Les actions sont de la forme `{: code Java :}` (ne pas oublier le `;` en fin d'instruction Java, à ne pas confondre avec le `;` de fin de production de CUP). Une action est placée en partie droite de production, et est exécutée quand le mot (composé de terminaux et non-terminaux) qui est à sa gauche est réduit. Le plus simple en TP est de toujours positionner les actions en fin de production, même si elles ne seront pas exécutées au plus tôt.

Dans l'exemple suivant `code affect` sera effectué quand `IDENT AFF expr` aura été réduit. Par contre `code read` sera effectué quand toute la partie droite de la production aura été réduite.

```
affect ::= IDENT AFF expr {: code affect :} PV ;
read ::= READ IDENT PV {: code read :} ;
```

L'accès à la valeur des attributs dans une production se fait de la manière suivante :
- par convention l'attribut du non terminal en partie gauche s'appelle `RESULT` ;
- en partie droite on nomme comme on le souhaite l'attribut d'un terminal ou non-terminal au moyen du caractère `:`.

Dans le cas des expressions arithmétiques :

```
expr ::= expr:e1 PLUS expr:e2    // Prod 1
  {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
       | ENTIER:e                // Prod 2
          {: RESULT = e; :}
       | MOINS expr:e            // Prod 3
          {: RESULT = new Integer(- e.intValue()) ;:}
          %prec MOINS-U
;
```

l'attribut de type `Integer` de `expr` est appelé `e1`, `e2` ou `e` selon les productions. Dans la production 1 on aurait écrit dans une grammaire attribuée classique `expr1:e PLUS expr2:e`. CUP choisit de toujours donner le même nom aux terminaux et non-terminaux, mais de modifier le nom de l'attribut : ça revient au même. À noter dans la production 3 : le `%prec` doit être placé **après** l'action, en fin de production.

# 3 Interfaçage avec JFlex

CUP et JFLEX sont prévus pour s'interfacer. Pour que CUP puisse exploiter un analyseur lexical celui-ci doit satisfaire les conditions suivantes :
- la méthode d'analyse s'appelle `next_token` et retourne des symboles de type `java_cup.runtime.Symbol` ;
- le symbole de fin de fichier s'appelle `EOF` ;
- la classe implémente l'interface `java_cup.runtime.Scanner`.

# 4 Utiliser Cup

On lance CUP par la ligne de commande :

```
java java_cup.Main <options> <spec.cup>
```

où `spec.cup` est le fichier de spécification pour CUP. Les fichiers `parser.java` et `sym.java` sont produits dans le répertoire courant. On construit une instance de `parser` en lui passant en paramètre un analyseur lexical :

```
// anLex est un analyseur lexical de type Scanner
parser p = new parser(anLex);
p.parse();
```

Les noms `parser` et `sym` peuvent être modifiés si on lance CUP avec respectivement l'option `-parser` et `-symbols` suivie d'un nom destiné à remplacer respectivement `parser` et `sym`. L'option `-dump` est particulièrement utile quand la grammaire est déclarée non LALR(1) : CUP envoie sur `System.err` (redirection avec `2>` sous Unix) une version textuelle de l'automate LALR.