TP `JavaCC` — Corrigé

David A. Madore

25 janvier 2018

INF105

Git : `af7f0fd Thu Jan 25 11:22:37 2018 +0100`

JavaCC est un programme permettant de créer automatiquement un analyseur syntaxique (=parseur, *parser* en anglais) pour un langage décrit sous forme de grammaire hors contexte. Le principe est le suivant : on écrit un fichier `.jj` qui contient un mélange de description du langage (sous forme de productions dans une syntaxe évoquant les expressions régulières) et de code Java à exécuter lorsque ces différentes productions sont rencontrées. Le programme `javacc` convertit ce fichier `.jj` en un fichier `.java` qui contient le code de l'analyseur ; on peut ensuite compiler ce fichier avec `javac` comme n'importe quel code Java (puis l'exécuter avec `java`).

La documentation de JavaCC est accessible en ligne sur `https://javacc.org/doc` mais il est probablement plus simple d'apprendre par imitation à partir du fichier proposé.

Sur `http://perso.enst.fr/~madore/inf105/Calculatrice.jj` (à télécharger, ou à recopier directement depuis `~madore/Calculatrice.jj` dans les salles de TP) on trouvera le code d'une calculatrice très simple écrit en JavaCC. On peut compiler et lancer ce code (après l'avoir recopié chez soi) en lançant successivement :

```
~madore/bin/javacc Calculatrice.jj
javac Calculatrice.java
java Calculatrice
```

On peut alors taper des expressions arithmétiques telles que `3+4` ou `2*6` (terminer chaque calcul en tapant Enter, et terminer le programme en tapant Control-D). Vérifier que cela fonctionne correctement.

Initialement, la grammaire de la calculatrice définie dans `Calculatrice.jj` est donnée par :

$$\begin{aligned} \mathit{expression} &\to \mathit{element}\ (\text{“+”}\ \mathit{element} \mid \text{“−”}\ \mathit{element} \mid \text{“⋆”}\ \mathit{element} \mid \text{“/”}\ \mathit{element})* \\ \mathit{element} &\to \mathit{Number} \\ \mathit{Number} &\to [\text{“0”}{-}\text{“9”}]+\ (\text{“.”}\ [\text{“0”}{-}\text{“9”}]*)? \end{aligned}$$

Il s'agit d'une grammaire « étendue » au sens où le membre de droite des règles de production autorise l'utilisation de métacaractères « $|$ » (disjonction), « $*$ » (étoile de Kleene), « $+$ » et « $?$ » de manière semblable aux expressions rationnelles : c'est-à-dire que JavaCC note $X|Y$ pour un choix entre X et Y, ainsi que $X*$ pour un nombre quelconque de répétitions de X et $X+$ pour la variante imposant au moins une répétition et $X?$ pour un X factultatif (les parenthèses permettent par ailleurs de grouper). En principe, on pourrait éliminer l'utilisation de ces métacaractères en s'inspirant de la manière dont on démontre que tout langage rationnel est algébrique (par exemple $X \to Y*$ peut être remplacé par $X \to \varepsilon \mid YX$), mais l'analyseur LL de JavaCC ne sera pas forcément capable de gérer la grammaire ainsi transformée, donc il est préférable d'utiliser les métacaractères lorsque c'est possible.

Le code important se situe dans `Calculatrice.jj` au niveau de la ligne `double expression():` ; le bloc qui suit définit la production *expression* ainsi que le code à exécuter quand elle est rencontrée.

(1) Que se passe-t-il quand on demande de calculer `3+4*5` à la calculatrice ? Pourquoi ?

Modifier la grammaire pour avoir une priorité correcte des opérations (`*` et `/` doivent être prioritaires sur + et −, et en particulier `3+4*5` doit renvoyer `23.0`). Pour cela, on peut par exemple introduire un nouveau nonterminal *term* représentant un terme d'une somme ou différence, et qui est lui-même formé de produits ou quotients. Tester la calculatrice ainsi modifiée.

Corrigé. La calculatrice renvoie `35.0` quand on lui demande de calculer `3+4*5`, c'est-à-dire qu'elle analyse ce mot comme `(3+4)*5`. La raison en est que la production *expression* produit un arbre d'analyse « plat », et le code est alors exécuté séquentiellement (on part de `a=3`, on exécute ensuite `a+=4` et enfin `a*=5`).

Pour corriger ce problème, on va modifier la grammaire de la manière suivante :

$$\begin{aligned} \textit{expression} &\to \textit{term}\ (\text{“+”}\,\textit{term} \mid \text{“−”}\,\textit{term})* \\ \textit{term} &\to \textit{element}\ (\text{“}*\text{”}\,\textit{element} \mid \text{“/”}\,\textit{element})* \end{aligned}$$

(le reste est inchangé) : ceci forcera l'arbre d'analyse de `3+4*5` à comporter deux *term*, en l'occurrence `3` et `4*5`, qui s'évalueront dans les bonnes valeurs.

Voici à quoi peut ressembler le code JavaCC modifié pour refléter la grammaire en question :

```
double expression():
{ double a,b; }
{
    a=term()
    (
      "+" b=term() { a += b; }
    | "-" b=term() { a -= b; }
    )* { return a; }
}

double term():
{ double a,b; }
{
    a=element()
    (
      "*" b=element() { a *= b; }
    | "/" b=element() { a /= b; }
    )* { return a; }
}
```

On vérifie bien que la nouvelle calculatrice renvoie `23.0` quand on lui demande de calculer `3+4*5`. ✓

(2) La calculatrice ne gère pas les parenthèses : ajouter cette fonctionnalité et tester. Par exemple, `(3+5)/(2*2)` doit renvoyer `2.0`. Pour cela, on pourra ajouter une nouvelle production pour *element* correspondant à une *expression* entourée de parenthèses.

Corrigé. On va maintenant modifier la production pour *element* et en ajouter une nouvelle, à savoir :

$$\textit{element} \to \textit{Number} \mid \text{“(”}\,\textit{expression}\,\text{“)”}$$

Le code JavaCC pour cette production peut ressembler à ceci :

```
double element():
{ Token t; double a; }
{
```

```
    t=<NUMBER> { return Double.parseDouble(t.toString()); }
|   "(" a=expression() ")" { return a; }
}
```

(ici, `Token` est un type interne à JavaCC utilisé pour les productions du lexer ; on ne touche pas à la première partie, on se contente d'ajouter une deuxième production). ✓

(3) La calculatrice ne gère pas le $-$ unaire (pour indiquer l'opposé d'un nombre) : ajouter cette fonctionnalité et tester. Par exemple, `-3*-6` doit renvoyer `18.0`, et `-1-2` doit renvoyer `-3.0`. On veut que le moins unaire marche aussi devant les parenthèses : par exemple, `-(2+3)` doit renvoyer `-5.0`. On pourra aussi ajouter le $+$ unaire (qui ne change pas le nombre). Pour cela, on pourra introduire un nouveau nonterminal $factor$ représentant un élément éventuellement précédé d'un $-$ (ou d'un $+$) unaire.

Corrigé. On modifie la grammaire comme suit :

$$term \to factor\ (\text{“}*\text{”}\, factor \mid \text{“}/\text{”}\, factor)*$$
$$factor \to element \mid \text{“+”}\, element \mid \text{“–”}\, element$$

et le code JavaCC correspondant pourrait être :

```
double term():
{ double a,b; }
{
    a=factor()
    (
      "*" b=factor() { a *= b; }
    | "/" b=factor() { a /= b; }
    )* { return a; }
}

double factor():
{ double a; }
{
    a=element() { return a; }
|   "+" a=element() { return a; }
|   "-" a=element() { return -a; }
}
```

(Plusieurs variations sont possibles : par exemple, si on fait suivre le $-$ unaire d'un $factor$ au lieu d'un $element$, cela permet d'autoriser `--3` comme façon d'écrire `+3`.) ✓

(4) La calculatrice ne gère pas l'opération puissance : introduire celle-ci sous la notation du caractère “^” (accent circonflexe) : attention, on veut que l'opération `^` soit prioritaire sur toutes les autres (addition, soustraction, multiplication, division, et opérations unaires), et on veut qu'elle soit associative *à droite*, c'est-à-dire par exemple que `2^3^2` doit calculer $2^{3^2} = 2^9 = 512$.

On pourra pour cela introduire un nouveau nonterminal $power$ représentant un $element$ suivi facultativement d'un `^` et d'un nouveau $power$.

La méthode Java permettant de calculer a^b s'écrit `Math.pow(a,b)`.

Tester notamment les calculs suivants : `2^3^2` ⇝ `512.0`; `-1^2` ⇝ `-1.0`; `(-1)^2` ⇝ `1.0`; `(2+3)^2` ⇝ `25.0`; `2^-1` ⇝ `0.5`; `2*3^2*3` ⇝ `54.0`.

Corrigé. Pour obtenir l'associativité *à droite*, on va définir un nonterminal $power$ pour lequel on va définir une production faisant qu'une $power$ peut s'analyser comme un $element$ suivi d'un “^” suivi

facultativement d'une nouvelle $power$, c'est-à-dire qu'on modifie la grammaire ainsi :

$$factor \rightarrow power \mid \text{“+”}\, power \mid \text{“−”}\, power$$
$$power \rightarrow element(\text{“\^{}”}\, factor)?$$

avec le code JavaCC suivant :

```
double factor():
{ double a; }
{
    a=power() { return a; }
|   "+" a=power() { return a; }
|   "-" a=power() { return -a; }
}

double power():
{ double a,b; }
{
    a=element()
    (
      "^" b=factor() { a = Math.pow(a,b); }
    )? { return a; }
}
```

Remarquez l'utilisation du métacaractère « ? » pour marquer le caractère facultatif de la partie “^”$factor$; on aurait pu tenter d'écrire la règle sous la forme $power \rightarrow element \mid element$“^”$factor$ pour un code JavaCC qui ressemblerait à « `{ a=element() { return a; } | a=element() "^" b=factor() { return Math.pow(a,b); } }` », mais ce code fait échouer JavaCC car au moment où il reconnaît le nonterminal $element$, l'analyseur ne peut pas encore savoir dans quelle branche de l'alternative il sera.

Voici la grammaire finale :

$$\begin{aligned}
expression &\rightarrow term\ (\text{“+”}\, term \mid \text{“−”}\, term)* \\
term &\rightarrow factor\ (\text{“}\star\text{”}\, factor \mid \text{“/”}\, factor)* \\
factor &\rightarrow power \mid \text{“+”}\, power \mid \text{“−”}\, power \\
power &\rightarrow element(\text{“\^{}”}\, factor)? \\
element &\rightarrow Number \mid \text{“(”}\, expression\, \text{“)”} \\
Number &\rightarrow [\text{“0”}\text{–}\text{“9”}]+\ (\text{“.”}\ [\text{“0”}\text{–}\text{“9”}]*)?
\end{aligned}$$

Le code JavaCC correspondant peut se trouver à l'adresse `http://perso.enst.fr/~madore/inf105/Calculatrice-final.jj` ✓