Introduction à JavaCC

Christian Rinderknecht

Mardi 18 mars 2003

1 Expressions régulières et analyse lexicale

Caractère, alphabet, mot, langage

Un *alphabet* est un ensemble fini non vide de *caractères*. On note souvent les alphabets Σ et les caractères a, b, c etc.

Un *mot* sur Σ est une suite, éventuellement vide, de caractères de Σ. Le mot vide est noté ε. Un mot non vide est noté par ses caractères séparés par un point (centré), par exemple $a \cdot b \cdot c$ avec $a, b, c \in \Sigma$. Le point dénote un opérateur dit de *concaténation*, que l'on peut généraliser simplement aux mots eux-mêmes : $x \cdot y$, où x et y sont des mots sur Σ. Remarques : (a) le mot vide ε est un élément neutre pour la concaténation des mots : $x \cdot \varepsilon = \varepsilon \cdot x = x$ pour tout mot x (b) la concaténation est une opération associative : $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ (c) on écrira simplement xy au lieu de $x \cdot y$

Un *langage* L sur Σ est un ensemble de mots sur Σ. La concaténation peut s'étendre aux langages : $L_1 L_2 = \{xy \mid x \in L_1, y \in L_2\}$. Nous pouvons ainsi définir inductivement l'ensemble Σ^n des mots de longueur n sur l'alphabet Σ :

$$\begin{cases} \Sigma^0 = \{\varepsilon\} \\ \Sigma^{n+1} = \Sigma \cdot \Sigma^n \end{cases}$$

L'ensemble de tous les mots sur Σ est alors $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$ et l'ensemble de tous les mots *non vides* sur Σ est $\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n$.

Préfixes, suffixes et sous-mots

- Soient x, y et w trois mots de Σ^* tels que $w = xy$. Alors x est un *préfixe* de w et y est un *suffixe* de w. Si $x, y \in \Sigma^+$, alors x est un *préfixe propre* de w, et y est un *suffixe propre* de w.
- Soient x, y, z et w quatre mots de Σ^* tels que $w = xyz$. Alors y est un *facteur* de w.

Langages réguliers

L'ensemble $\mathcal{R}$ des *langages réguliers* sur Σ^* est défini inductivement comme étant la plus petite famille de parties de Σ^* vérifiant les propriétés

- $\varnothing \in \mathcal{R}$
- $\{\varepsilon\} \in \mathcal{R}$
- $\forall a \in \Sigma . \{a\} \in \mathcal{R}$
- $\forall R_1, R_2 \in \mathcal{R} . R_1 \cup R_2 \in \mathcal{R}$
- $\forall R_1, R_2 \in \mathcal{R} . R_1 \cdot R_2 \in \mathcal{R}$
- $\forall R \in \mathcal{R} . R^* \in \mathcal{R}$

Expressions régulières

Une expression régulière est une notation compacte et simplifiée pour représenter des langages réguliers :

Expression régulière	Langage régulier	Mots du langage
$a \mid b$	$\{a, b\}$	a, b
ab^*a	$\{a\}\{b\}^*\{a\}$	aa, aba, $abba$ etc.
$(ab)^*$	$\{ab\}^*$	ε, ab, $abab$ etc.
$abba$	$\{abba\}$	$abba$

Analyse lexicale

Un analyseur lexical[1] est un programme qui lit un fichier contenant un texte et qui vérifie que son contenu peut s'interpréter comme une suite d'unités lexicales (ou *lexèmes*[2]) appartenant à un lexique défini au préalable. C'est à peu près l'équivalent d'un vérificateur orthographique pour la langue française. Les expressions régulières servent à définir ce lexique.

Les analyseurs lexicaux constituent la première phase d'un compilateur. Souvent, le lexique d'un langage de programmation se divise ainsi :

- les espaces, tabulations et sauts de pages et de lignes ;
- les commentaires ;
- les identificateurs ;
- les mots-clés (un sous-ensemble des identificateurs) ;
- les constantes entières, flottantes, caractères, chaînes de caractères ;
- les symboles (par exemple d'opérateurs arithmétiques, mais aussi de ponctuation) ;

Généralement, les espaces et autre marques invisibles servent à séparer d'autres éléments (comme deux identificateurs successifs) mais ne sont pas considérées comme des lexèmes. Il en est de même des commentaires. Si l'analyseur lexical peut découper tout le texte en lexèmes valides (par rapport au lexique), alors il renvoie la suite de lexèmes (donc sans les espaces et les commentaires) pour la phase suivante de compilation, dite *analyse syntaxique*, sinon il y a une erreur lexicale et il s'arrête.

JavaCC et les expressions régulières

Les expressions régulières sont utilisées dans de nombreux outils de recherche d'occurrences dans du texte, tels les éditeurs de texte (Emacs, vi) et la commande Unix grep, mais aussi dans JavaCC. Ce dernier est un constructeur d'analyseurs lexico-syntaxiques (analyseur lexical et analyseur syntaxique). Les expressions régulières sont notées de façon particulière en JavaCC :

Expression régulière mathématique	Expression régulière JavaCC
$a \mid b$	`a | b`
ab	`ab`
a^*	`(a)*`
a^+	`(a)+`
$a \mid \varepsilon$	`(a)?`
$"a" \mid "b" \mid \ldots \mid "z"$	`["a"-"z"]`
$"a" \mid \ldots \mid "z" \mid "A" \mid \ldots \mid "Z"$	`["a"-"z","A"-"Z"]`

Par exemple, en JavaCC, on peut définir des identificateurs commençant par une lettre ou un souligné, et suivi d'une suite (éventuellement vide) de lettres ou chiffres ou souligné par :

```
["a"-"z","A"-"Z","_"] (["a"-"z","A"-"Z","_","0"-"9"])*
```

1. *lexer* en anglais.
2. *token* en anglais

Cette expression régulière reconnaît les identificateurs `"a"`, `"_ab"`, `"a__b01"`, mais pas `"1"`, `"1a"`.

2 Grammaires BNF et analyse syntaxique

La seconde phase de compilation est l'analyse syntaxique. L'analyseur syntaxique prend une suite de lexèmes, fournis par l'analyseur lexical, et vérifie si ces lexèmes satisfont une grammaire définie à l'avance. Grosso modo, cela correspond à un correcteur grammatical de la langue française. Une façon courante de définir une grammaire pour des langages (par exemple de programmation) est d'employer la *Backus-Naur Form*. On défini formellement une grammaire G par le quadruplet $(\mathcal{T}, \mathcal{N}, \mathcal{P}, Z, \textbf{eof})$, où

— $\mathcal{T}$ est un ensemble de symboles dits *terminaux* (qui correspondent aux lexèmes) ; on représente les terminaux avec des lettre minuscules ou des chiffres, ou encore entre guillemets ;
— $\mathcal{N}$ est un ensemble de symboles dits *non-terminaux* ($\mathcal{N} \cap \mathcal{T} = \varnothing$) ; on représente les non-terminaux avec des lettres majuscules ;
— $\mathcal{P}$ est un ensemble de paires de la forme $\mathcal{N} \times (\mathcal{N} \cup \mathcal{T})^*$, appelées *productions* ; par commodité d'écriture, on sépare la première composante d'une production (appelée *partie gauche*) et la seconde (appelée *partie droite*) par le symbole `::=`, par exemple N `::=` $a\,B\,C$;
— Z est un non-terminal particulier ($Z \in \mathcal{N}$), appelé *point d'entrée* ;
— **eof** est un terminal particulier, appelé *fin de fichier* mais qui n'appartient pas à $\mathcal{T}$ par convention.

Par exemple, une grammaire pour les expressions arithmétiques est :

$$
\begin{array}{lll}
Z & \texttt{::=} & E\ \textbf{eof} \\
E & \texttt{::=} & E\ \texttt{"+"}\ E \\
E & \texttt{::=} & E\ \texttt{"-"}\ E \\
E & \texttt{::=} & E\ \texttt{"*"}\ E \\
E & \texttt{::=} & E\ \texttt{"/"}\ E \\
E & \texttt{::=} & \texttt{"("}\ E\ \texttt{")"} \\
E & \texttt{::=} & \mathsf{digit}
\end{array}
$$

où digit est le nom d'une expression régulière : digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9, dont la définition fait partie du lexique (donc pas de la grammaire proprement dite).

On simplifie l'écriture des productions du même non-terminal (en partie gauche) à l'aide d'un opérateur de disjonction `|`. Ainsi, l'exemple précédent s'écrit plus simplement :

$$
\begin{array}{lll}
Z & \texttt{::=} & E\ \textbf{eof} \\
E & \texttt{::=} & E\ \texttt{"+"}\ E \\
 & \texttt{|} & E\ \texttt{"-"}\ E \\
 & \texttt{|} & E\ \texttt{"*"}\ E \\
 & \texttt{|} & E\ \texttt{"/"}\ E \\
 & \texttt{|} & \texttt{"("}\ E\ \texttt{")"} \\
 & \texttt{|} & \mathsf{digit}
\end{array}
$$

Ainsi les textes ((2*(1+3))-5) et 7 sont conformes à cette grammaire.

3 BNF étendue

La notation habituelle pour les grammaire est la BNF. On l'étend souvent par des opérateurs inspirés des expressions régulières et définis ainsi :

- A^+ est la concaténation d'au moins une fois de tous le mots de A et est donc équivalent à A A^*.
- A^* est la concaténation, éventuellement vide, de tous les mots de A et est donc équivalent à $\varepsilon \mid A^+$.
- (A) est équivalent à A.
- [A] est une abréviation pour $\varepsilon \mid$ A.

Soit la grammaire G définie en EBNF de la façon suivante :

S	::=	P **eof**
P	::=	"(" [P] ")"

Cette grammaire décrit le langage dont les mots sont constitués d'une suite de parenthèses ouvrantes suivie d'autant de parenthèses fermantes, c'est-à-dire le langage $L(G) = \{"("^n")"^n \mid \forall n \geq 1\}$. Par exemple ce langage contient les mots `()`, `(())` etc. mais pas `(()())`, `(`, `(()` etc. *Ce langage n'est pas régulier*, c'est-à-dire qu'il n'existe pas d'expression régulière décrivant tous les mots. Il faut donc nécessairement un analyseur syntaxique (pour les grammaires algébriques).

4 Un langage simple

JavaCC est un programme qui prend une spécification sous forme d'une grammaire et d'un lexique écrite dans un fichier avec l'extension `.jj`, et produit des fichiers Java qui constituent l'analyseur lexico-syntaxique correspondant. Donc JavaCC combine la production de l'analyseur lexical et de l'analyseur syntaxique. En JavaCC la grammaire G se transcrit ainsi :

```
void S() : {} { P() <EOF> }
void P() : {} { "(" [P()] ")" }
```

En JavaCC, le non-terminal en partie gauche d'une production est écrit comme une déclaration de méthode, par exemple `void S()`, suivi d'un deux-points, puis d'un bloc entre accolades de déclarations de variables locales Java (ici vide), puis d'un bloc contenant la partie droite de la production. Les non-terminaux dans cette partie droite sont notés comme des appels de méthodes statiques, par exemple `P()`. Le terminal **eof** est noté `<EOF>`.

4.1 Par1.jj

Si nous voulons spécifier pour JavaCC le petit exemple ci-dessus, il nous faut le compléter et le placer dans un fichier nommé par exemple Par1.jj :

```
PARSER_BEGIN(Par1)
public class Par1 {
  public static void main(String args[]) throws ParseException {
    Par1 parser = new Par1(System.in);
    parser.S();
  }
```

```
}
PARSER_END(Par1)

void S() : {} { P() ("\n" | "\r")* <EOF> }
void P() : {} { "(" [P()] ")" }
```

Les caractéristiques générales d'une spécification sont ici illustrées :

- La spécification débute par une section encadrée par `PARSER_BEGIN` et `PARSER_END`, qui contient du code source Java, donc en particulier une classe publique de même nom que le fichier (ici `Par1`).
- Comme toujours en Java, la classe publique doit définir une méthode `main` standard, sauf qu'il faut déclarer qu'elle peut lancer une exception `ParseException` (ce sera toujours ce nom-là), en cas d'erreur de syntaxe dans le fichier analysé.
- Le paramètre de `PARSER_BEGIN` et `PARSER_END` doit être le nom de la classe publique.
- Le point d'entrée de la grammaire est donné par l'appel `parser.S()`.

Remarque Nous avons écrit `("\n" | "\r")* <EOF>` au lieu de `<EOF>`. Cela est rendu nécessaire par la façon dont est gérée la fin de ligne et la fin de fichier sous Unix, à la fois dans stdin et dans un fichier proprement dit.
Quand on donne ce fichier Par1.jj à JavaCC :

```
$ javacc Par1.jj
Java Compiler Compiler Version 3.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file Par1.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
```

On comprend que JavaCC produit un fichier Par1.java, ainsi que d'autres fichiers Java auxiliaires. Il est maintenant possible de compiler ce fichier (et ceux produits en support) :
`$ javac Par1.java`
L'analyseur lexico-syntaxique est prêt à l'emploi maintenant :
`$ java Par1`
On saisit alors une expression suivie de Enter (qui correspond à `"\n"`) et Ctrl-D (qui correspond à `<EOF>` pour stdin). S'il n'y a rien d'affiché, alors tout va bien : la syntaxe était correcte. Mais si on avait saisit par exemple `(`, alors on aurait lu le message suivant :

```
Exception in thread "main" TokenMgrError: Lexical error at line 1,
column 1. Encountered: "(" (40), after : ""
        at Par1TokenManager.getNextToken(Par1TokenManager.java:148)
        at Par1.jj_consume_token(Par1.java:141)
        at Par1.P(Par1.java:38)
        at Par1.S(Par1.java:9)
        at Par1.main(Par1.java:5)
```

Il est possible aussi de mettre les parenthèses dans un fichier, par exemple a.txt :
`$ java Par1 < a.txt`

4.2 Par2.jj

Voici une extension de l'exemple précédent, dans le fichier Par2.jj :

```
PARSER_BEGIN(Par2)
public class Par2 {
  public static void main(String args[]) throws ParseException {
    Par2 parser = new Par2(System.in);
    parser.S();
  }
}
PARSER_END(Par2)

SKIP : { " " | "\t" | "\n" | "\r" }

void S() : {} { P() <EOF> }
void P() : {} { "(" [P()] ")" }
```

Il est maintenant possible d'appuyer sur Enter parmi les parenthèses. En effet, cette nouvelle spécification précise l'analyse lexicale par un bloc `SKIP`. Dans ce bloc il y a quatre expressions régulières : espace, tabulation, fin de ligne et retour à la ligne. Cela signifie que lorsque ces caractères sont reconnus, ils sont écartés de l'analyse syntaxique.

4.3 Par3.jj

Le fichier Par3.jj est la dernière amélioration de notre analyseur :

```
PARSER_BEGIN(Par3)
public class Par3 {
  public static void main(String args[]) throws ParseException {
    Par3 parser = new Par3(System.in);
    int count = parser.S();
    System.out.println("Nesting level is " + count);
  }
}
PARSER_END(Par3)

SKIP : {" " | "\t" | "\n" | "\r"}
TOKEN : { <LPAR: "("> | <RPAR: ")"> }

int S() :
{int nesting;}
{
  nesting=P() <EOF> { return nesting; }
}

int P() :
{int nesting=0;}
{
  <LPAR> [nesting=P()] <RPAR>  {return ++nesting;}
}
```

Cet exemple illustre l'emploi d'un bloc lexical (comme l'est `SKIP`) pour spécifier des noms de lexèmes : il s'agit d'un bloc `TOKEN`. Dans notre cas, on l'utilise

pour donner le nom LPAR (respectivement RPAR) à l'expression régulière "(" (respectivement ")"). Ces noms peuvent alors être employés entre chevrons, en référence à leur expression régulière. Typiquement de telles spécifications de lexèmes sont utilisées pour des lexèmes complexes tels que des identificateurs et des constantes. Les lexèmes qui sont de simples chaînes étaient laissés tels quels dans l'exemple précédent. Pourquoi donc compliquer les choses ? La raison est liée à la façon dont sont compilées les spécifications par JavaCC : dans le cas général, le fait de partager la définition des lexèmes à l'aide de clauses TOKEN conduit à un code Java plus efficace.

Cet exemple illustre aussi l'usage d'*actions* dans les productions grammaticales. Les actions sont un bloc contenant du code Java, et sont insérées dans la partie droite des productions. Elle sont exécutées lorsque la partie de la production juste avant a été reconnue. Dans notre exemple, les actions comptent le niveau d'imbrication des parenthèses. Notez aussi l'usage du bloc de déclaration (auparavant vide) pour déclarer les variables nesting. Notez aussi comment le non-terminal P retourne sa valeur comme une méthode, c'est-à-dire à l'aide de l'instruction Java return — il y a donc un type de retour int pour chaque non-terminal.

4.4 Liste d'identificateurs

Voici à quoi ressemble la description d'une suite d'identificateurs.

```
PARSER_BEGIN(IdList)
public class IdList {
  public static void main(String args[]) throws ParseException {
    IdList parser = new IdList(System.in);
    parser.Ids();
  }
}
PARSER_END(IdList)

SKIP : {" " | "\t" | "\n" | "\r"}
TOKEN : { <ID: ["a"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])*> }

void Ids() : {} { (<ID>)+ <EOF> }
```

5 Calculatrice

Nous souhaitons trouver une spécification JavaCC qui reconnaisse ou rejette une expression arithmétique sans identificateurs. Comme toujours, il nous faut d'abord trouver une grammaire qui décrive le langage que nous voulons, puis s'assurer qu'elle possède les propriétés requises par JavaCC. Considérons d'abord la grammaire qui décrit directement le langage d'expressions voulu :

(1)	S	::=	E **eof**
(2)	E	::=	E "+" E
(3)		|	E "-" E
(4)		|	E "*" E
(5)		|	E "/" E
(6)		|	"(" E ")"
(7)		|	int

Il y a une raison majeure de rejeter cette grammaire pour l'implantation avec JavaCC (et l'immense majorité des générateurs d'analyseurs syntaxiques) : elle est *ambiguë*. Une grammaire est dite ambiguë lorsqu'elle autorise la reconnaissance d'un même texte d'au moins deux façons différentes (en d'autres termes, il y a plus d'un arbre de syntaxe concrète pour le même mot des feuilles). Si on note $\overset{i}{\Longrightarrow}$ l'usage de la production (i), alors on a par exemple deux façons d'interpréter le texte `7-5-2` avec cette grammaire :

$$\begin{array}{rcl} \mathrm{E} & \overset{3}{\Longrightarrow} & \mathrm{E}\ \texttt{"-"}\ \mathrm{E} \\ & \overset{3}{\Longrightarrow} & (\mathrm{E}\ \texttt{"-"}\ \mathrm{E})\ \texttt{"-"}\ \mathrm{E} \\ & \overset{7}{\Longrightarrow} & (\mathsf{int}\ \texttt{"-"}\ \mathsf{int})\ \texttt{"-"}\ \mathsf{int} \\ \mathrm{E} & \overset{*}{\Longrightarrow} & \texttt{"7-5-2"} \end{array}$$

mais aussi :

$$\begin{array}{rcl} \mathrm{E} & \overset{3}{\Longrightarrow} & \mathrm{E}\ \texttt{"-"}\ \mathrm{E} \\ & \overset{3}{\Longrightarrow} & \mathrm{E}\ \texttt{"-"}\ (\mathrm{E}\ \texttt{"-"}\ \mathrm{E}) \\ & \overset{7}{\Longrightarrow} & \mathsf{int}\ \texttt{"-"}\ (\mathsf{int}\ \texttt{"-"}\ \mathsf{int}) \\ \mathrm{E} & \overset{*}{\Longrightarrow} & \texttt{"7-5-2"} \end{array}$$

En l'absence de construction d'arbre de syntaxe abstraite, le premier cas aboutira à l'évaluation de l'expression $(7-5)-2 = 0$ et dans le second cas $7-(5-2) = 4$. Malheureusement, *le problème consistant à déterminer si une grammaire BNF donnée est ambiguë ou non est indécidable.* Pour chaque grammaire particulière, il faut donc s'assurer soi-même qu'elle n'est pas ambiguë, en général en prouvant qu'elle appartient à une classe restreinte de grammaires.

Comme nous l'avons dit, il n'y a pas de solution générale au problème de trouver une grammaire équivalente [3] non-ambiguë [4]. Dans le cas qui nous occupe ici, il est possible de trouver une grammaire de telle sorte que les opérateurs de plus faible priorité apparaissent uniquement en haut de l'arbre de syntaxe concrète et ceux de plus haute priorité en bas — sauf si un opérateur apparaît entre parenthèses, car il n'y a alors pas d'ambiguïté. Par exemple

```
(1)  S  ::=    E eof
(2)  E  ::=    E "+" F
(3)            | E "-" F
(4)            | F
(5)  F  ::=    F "*" G
(6)            | F "/" G
(7)            | G
(8)  G  ::=    "(" E ")"
(9)            | int
```

n'est pas une grammaire ambiguë et elle engendre le même langage que la précédente. (Attention : il n'est pas possible de déduire une grammaire non-ambiguë

3. Par définition, deux grammaires sont équivalentes si les langages qu'elles engendrent sont les mêmes.

4. Il existe même des langages simples qui ne peuvent être engendrés que par des grammaires ambiguës.

équivalente à une grammaire ambiguë donnée seulement en jouant sur les opérateurs algébriques. Il faut en trouver une autre par l'intuition et prouver leur équivalence. Ce dernier problème étant indécidable, la preuve devra se faire au cas par cas.) Le problème est que si l'analyseur syntaxique est *descendant*, c'est-à-dire qu'il applique les productions syntaxiques aux non-terminaux pour parvenir aux terminaux (lexèmes) à partir du terminal initial, alors la récursivité à gauche d'une production le fera boucler. Or notre nouvelle grammaire est récursive à gauche, donc JavaCC, qui implante une analyse descendante, la refuserait (pour que le code produit ne boucle pas). Heureusement, il nous est ici possible de construire une grammaire équivalente qui ne soit pas récursive à gauche. En effet, elle est équivalente à

```
S  ::=  E eof
E  ::=  E (("+" | "-") F)  |  F
F  ::=  F (("*" | "/") G)  |  G
G  ::=  "(" E ")"  |  int
```

et donc, en utilisant l'opérateur étoile, elle équivaut à

```
S  ::=  E eof
E  ::=  F (("+" | "-") F)*
F  ::=  G (("*" | "/") G)*
G  ::=  "(" E ")"  |  int
```

Cette grammaire est acceptée par JavaCC. La classe des grammaires acceptées par JavaCC est celle des grammaires analysables de façon descendante sans rebroussement, c'est-à-dire que l'analyseur ne s'engage pas dans une production pour en essayer une autre si la première n'aboutit pas. C'est pour cela que d'autres propriétés que la non-récursivité à gauche doivent être vérifiées par les grammaires soumises à JavaCC. La classe acceptée est nommée LL(k). Le k indique le nombre de lexèmes de prévision que s'autorise l'analyseur pour progresser. Par défaut, JavaCC suppose que l'analyseur ne choisit une production à appliquer à chaque étape de la reconnaissance que sur la base de la connaissance du seul lexème courant — pas les suivants. Parfois, augmenter la prévision permet d'accepter des grammaires plus compliquées, mais parfois non, car la grammaire peut ne pas être LL(k) pour aucun k (par exemple si elle est ambiguë). Ici, donc, notre grammaire est LL(1). La première étape consiste toujours à écrire une spécification JavaCC sans actions, pour vérifier justement que la grammaire est LL(1). Ainsi nous avons :

```
PARSER_BEGIN(Calc1)
public class Calc1 {
  public static void main(String args[]) throws ParseException {
    Calc1 parser = new Calc1(System.in);
    while (true) {
      System.out.print("Entrez une expression: ");
      System.out.flush();
      try {
        parser.S();
      } catch (ParseException x) {
          System.out.println("Exiting.");
          throw x;
```

```
        }
    }
  }
}
PARSER_END(Calc1)

SKIP  : { " " | "\t" | "\r" }
TOKEN : { <EOL: "\n"> }
TOKEN : { <PLUS: "+"> | <MINUS: "-"> | <TIMES: "*"> | <SLASH: "/"> }
TOKEN : { < INT: (<DIGIT>)+ > | <#DIGIT: ["0"-"9"]> }

void S() : {} { E() <EOL> }
void E() : {} { F() ((<PLUS> | <MINUS>) F())* }
void F() : {} { G() ((<TIMES> | <SLASH>) G())* }
void G() : {} { <INT> | "(" E() ")" }
```

On souhaite ici afficher un message invitant à la saisie d'une expression arithmétique (Enter Expression :), et aussi en cas d'erreur de syntaxe (Exiting.). On veut une boucle interactive (while (true)) pour saisir de façon répétée des expressions. S'il n'y a pas d'erreur de syntaxe, on invite l'usager à saisir une autre expression, sinon un message d'erreur est imprimé et le programme termine. Une des nouveautés ici est qu'on ne saute pas les fins de ligne (cf. bloc `SKIP`) et qu'on définit un lexème pour la fin de fichier (cf. `<EOF>`). La raison est que l'on s'attend à ce que l'expression soit saisie sur une seule ligne, donc il ne faut pas sauter ce caractère de contrôle (cf. la production `S`). On utilise aussi, dans un bloc `TOKEN` (définition de lexèmes), une nouvelle construction : `<#DIGIT: ["0"-"9"]>`. La présence du signe dièse indique qu'il ne s'agit pas d'un lexème proprement dit, mais d'une définition auxiliaire : ce pseudo-lexème sert en effet à définir le lexème `INT`. Autrement dit, `DIGIT` est le nom d'une expression régulière qui sert à définir d'autres expressions régulières, mais pas un lexème directement. Attention, la syntaxe (`#` et `|`) est plutôt trompeuse ici. L'étape suivante est l'ajout de l'évaluation des expressions arithmétiques.

Pour cela, il nous faut nous poser la question des paramètres des méthodes associées aux non-terminaux et de leur valeur de retour. Dans le premier cas on parle, en termes grammaticaux, d'*attributs hérités*, et dans le second on parle d'*attributs synthétisés*. Dans un premier temps nous n'avons indiqué aucun paramètre et aucune valeur de retour, comme on le voit sur l'extrait `void S()`. Une première contrainte provient néanmoins des productions `E` et `F`. En effet, la façon de lier un attribut synthétisé est la syntaxe Java habituelle. Par exemple, dans l'extrait de spécification `void S() : {} { e=E() <EOL> }`, la variable `e` désigne l'attribut synthétisé par le non-terminal E, ou, en termes équivalents, la valeur de retour de la méthode `E` à cet endroit précis. Or, si nous considérons la production E : `void E() : {} { F() ((<PLUS> | <MINUS>) F())* }`, nous constatons que nous ne pouvons lier (nommer) d'un coup toutes les occurrences de `F`, pour en faire la somme, à cause de l'opérateur étoile (répétition éventuellement nulle). Une solution de contournement consiste alors à ne pas employer d'attributs, mais à employer une variable globale, précisément une pile, qui stocke les arguments des opérateurs arithmétiques. Lorsqu'une opération est complètement reconnue (c'est-à-dire l'opérateur et ses arguments), nous dépilons le nombre d'arguments nécessaire, nous effectuons l'opération correspondante en Java et nous replaçons le résultat dans cette pile. À la fin de l'analyse syntaxique

nous aurons alors directement le résultat de l'évaluation dans la pile globale.

```
PARSER_BEGIN(Calc2)
public class Calc2 {
    static int total;
    static java.util.Stack argStack = new java.util.Stack();

    public static void main(String args[]) throws ParseException{
      Calc2 parser = new Calc2(System.in);
      while (true) {
        System.out.print("Enter Expression: ");
        System.out.flush();
        try {
          parser.S();
          System.out.println("Result: " + argStack.pop());
        } catch (ParseException x) {
            System.out.println("Exiting.");
            throw x;
          }
      }
    }
}
PARSER_END(Calc2)

SKIP  : { " " | "\t" | "\r" }
TOKEN : { <EOL: "\n"> }
TOKEN : { <PLUS: "+"> | <MINUS: "-"> | <TIMES: "*"> | <SLASH: "/"> }
TOKEN : { < INT: (<DIGIT>)+ > | <#DIGIT: ["0"-"9"]> }

void S() : {} { E() <EOL> }

void E() :
{Token x;}
{
 F()
 ((x=<PLUS> | x=<MINUS>) F()
  {
   int a = ((Integer) argStack.pop()).intValue();
   int b = ((Integer) argStack.pop()).intValue();
   if (x.kind == PLUS)
     argStack.push(new Integer(b + a));
   else
     argStack.push(new Integer(b - a));
  }
 )*
}

void F() :
{Token x;}
{
 G()
 ((x=<TIMES> | x=<SLASH>) G()
  {
   int a = ((Integer) argStack.pop()).intValue();
   int b = ((Integer) argStack.pop()).intValue();
```

```
   if (x.kind == TIMES)
     argStack.push(new Integer(b * a));
   else
     argStack.push(new Integer(b / a));
  }
 )*
}

void G() :
{}
{
 <INT>
 {
  try {
    int x = Integer.parseInt(token.image);
    argStack.push(new Integer(x));
  } catch (NumberFormatException ee) {
      argStack.push(new Integer(0));
    }
 }
| "(" E() ")"
}
```

Le calcul repose sur une pile d'opérandes. Celle-ci, nommée `argStack`, est vide initialement. À chaque fois qu'un entier est reconnu, il est empilé. Dès qu'un opérateur est reconnu, les arguments (un ou deux, en fonction de l'arité de l'opérateur) sont dépilés et l'opération arithmétique Java correspondante est effectuée et le résultat est à nouveau empilé. Si la syntaxe de l'expression est correcte, la pile contiendra alors à la fin la valeur de l'expression. Notons encore que la valeur d'un lexème est obtenue par `x=<PLUS>`, où `x` est de type `Token` (produit par JavaCC à partir des blocs lexicaux `TOKEN`). La classe `Token` possède un champ `kind` pour lire le type de lexème dont il est question, par exemple : `x.kind == TIMES`. Cette classe possède aussi un champ `image` qui permet d'obtenir la chaîne de caractères reconnue pour le lexème en question, par exemple : `int x = Integer.parseInt(token.image)`.

Quels sont les inconvénients de cette approche ? Le principal problème provient, comme souvent en programation, des variables globales. En effet, `argStack` pouvant être modifiée par tout le programme, les risques d'erreurs sont fortement augmentés (en général). Malheureusement, la forme de la grammaire EBNF choisie pour la spécification JavaCC implique l'usage d'une variable globale, il faut donc trouver une autre grammaire équivalente. Par exemple une variante récursive à droite de la grammaire initiale :

```
S  ::=   E eof
E  ::=   F "+" E
       | F "-" E
       | F
F  ::=   G "*" F
       | G "/" F
       | G
G  ::=   "(" E ")"
       | int
```

Tout d'abord, le fait qu'elle soit récursive à droite convient bien à l'analyse descendante implantée par JavaCC. Prouvons alors son équivalence avec la grammaire récursive à gauche présentée page 8. Puisque nous avons établi que cette dernière était équivalente à

$$
\begin{array}{lll}
\text{S} & ::= & \text{E } \textbf{eof} \\
\text{E} & ::= & \text{F } (\texttt{"+"} \mid \texttt{"-"})\,\text{F})^* \\
\text{F} & ::= & \text{G } (\texttt{"*"} \mid \texttt{"/"})\,\text{G})^* \\
\text{G} & ::= & \texttt{"("}\ \text{E}\ \texttt{")"} \mid \texttt{int}
\end{array}
$$

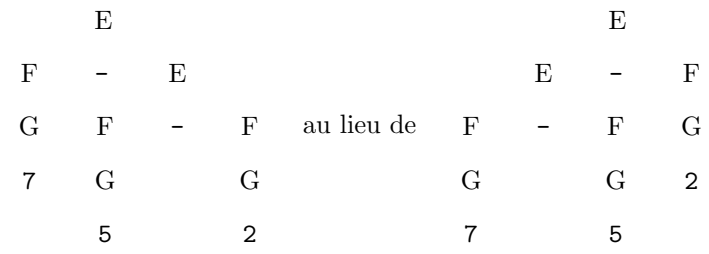
nous allons prouver que la nouvelle grammaire récursive à gauche est équivalente à cette grammaire-là. Considérons la production E, pour simplifier. Nous avons les sous-grammaires équivalentes :

$$
\begin{array}{lll}
\text{E} & ::= & \text{F}(\texttt{"+"} \mid \texttt{"-"})\,\text{E} \mid \text{F} \\
\hline
\text{E} & ::= & \text{F}[(\texttt{"+"} \mid \texttt{"-"})\,\text{E}] \\
\hline
\text{E} & ::= & \text{F E'} \\
\text{E'} & ::= & (\texttt{"+"} \mid \texttt{"-"})\,\text{E} \mid \varepsilon \\
\hline
\text{E} & ::= & \text{F E'} \\
\text{E'} & ::= & (\texttt{"+"} \mid \texttt{"-"})\,\text{F E'} \mid \varepsilon \\
\hline
\text{E} & ::= & \text{F E'} \\
\text{E'} & ::= & ((\texttt{"+"} \mid \texttt{"-"})\,\text{F})^* \\
\hline
\text{E} & ::= & \text{F } ((\texttt{"+"} \mid \texttt{"-"})\,\text{F})^*
\end{array}
$$

ce qui était à démontrer. Il suffit de faire de même avec F pour terminer le travail. Nous pouvons donc repartir avec notre grammaire récursive à droite, écrite sous une forme compacte :

$$
\begin{array}{lll}
\text{S} & ::= & \text{E } \textbf{eof} \\
\text{E} & ::= & \text{F } [(\texttt{"+"} \mid \texttt{"-"})\,\text{E}] \\
\text{F} & ::= & \text{G } [(\texttt{"*"} \mid \texttt{"/"})\,\text{F}] \\
\text{G} & ::= & \texttt{"("}\ \text{E}\ \texttt{")"} \mid \texttt{int}
\end{array}
$$

Puisque nous ne souhaitons pas ici construire d'arbre de syntaxe abstraite, nous devons appuyer la sémantique sur les arbres de syntaxe concrète. Or, puisque notre grammaire est récursive à droite, la soustraction sera de fait parenthésée à droite :



avec la grammaire récursive à gauche. Si la sémantique se base uniquement sur des attributs synthétisés (c'est-à-dire pas de paramètres pour les méthodes associées aux non-terminaux, seulement une valeur de retour), alors la soustraction devient associative à droite, ce qui est contraire aux conventions mathématiques. Comment donc implanter la sémantique en JavaCC avec cette grammaire récursive à droite ?