Université Paris Diderot
L2 Informatique et Math-Info

Analyse de Données Structurées
Année 2014-2015

TP Noté n°6 sujet 4

Analyse lexicale et syntaxique dans Java

Ce TP est divisé en trois sections (section 1 : *échauffement*, section 2 : *on ajoute la multiplication* , section 3 : *on calcule*). **À la fin de chaque section**, vous devez soumettre sur Didel dans le travail associé à votre groupe de TD votre solution dans une archive compressée `tar.gz` nommée : *nom-section<numerosection>*`.tar.gz`, où *nom* est votre nom et *section<numerosection>* est le numero de section. Par exemple, si vous vous appelez Ralf Treinen et vous appartenez au groupe Info4, une fois terminée la section 1 vous devrez soumettre sur Didel dans le travail *Info4.Tp note - section1* une archive nommée `treinen-section1.tar.gz`.

Attention, la soumission est permise jusqu'à 16h.
Après ce délai aucune soumission ne sera plus acceptée

Il est possible de mettre à jour (jusqu'à 16h) un travail déjà soumis. On vous conseille donc de soumettre la solution à une section dès que vous l'avez trouvée et ensuite de la mettre à jour dans le cas où vous feriez de modifications.

Le but de ce TP est de construire un analyseur lexical et syntaxique pour un langage d'expressions arithmétiques sur les chaînes de caractères, défini par la grammaire suivante :

```
   <int> ::= [1-9][0-9]*
<string> ::= "\""[^\"]*"\""
  <expr> ::= <string> |"("<expr>"+"<expr>")" | "("<expr>"-"<expr>")"
           | "("<expr>"*"<int>")"
```

FIGURE 1 – Expressions arithmétiques sur les chaînes de caractères

1 Échauffement

Les premiers trois exercices sont sur une version simplifiée de la grammaire de Figure 1, ne contenant pas le non-terminal `<int>` ni la multiplication `*`.

Exercice 1 *Complétez le fichier* `string.flex` *pour obtenir, à travers* `JFlex`*, un analyseur lexicale qui découpe l'entrée en des jetons. Les jetons sont spécifiés dans les fichiers* `Sym.java` *et* `Token.java`*. Vous ferez attention à tenir compte des espaces ou sauts de lignes pouvant séparer les mots.*

Exercice 2 *Complétez le fichier* `Parser.java` *pour obtenir un analyseur syntaxique qui reconnaît, parmi les flots de jetons générés par* `string.flex` *, ceux qui appartiennent au langage défini par la grammaire suivante*

$$S \mapsto E\ \$ \qquad E \mapsto \texttt{STR} \mid \texttt{LPAR}\ E\ O\ E\ \texttt{RPAR} \qquad O \mapsto \texttt{PLUS} \mid \texttt{MOINS}$$

L'analyseur syntaxique traitera le flot de jetons grâce aux méthodes `boolean check(Sym s)` *et* `void eat(Sym s)` *de la classe* `LookAhead1.java`*, dont le code vous est fourni.*

Exercice 3 *La classe* `Arithmetic` *contient la méthode* `main` *qui permet de tester votre code sur un fichier. Compilez et exécutez le programme sur les trois exemples* `good*` *et les trois exemples* `bad*` *qui se trouvent sur le sous-répertoire* `Section1` *du répertoire* `TP6_Info4_MathInfo` *de Didel.*

2 On ajoute la multiplication

Exercice 4 *Modifiez les fichiers* `string.flex` *,* `Sym.java` *et* `Token.java` *pour que l'analyseur lexicale accepte les entiers et la multiplication. Faites attention à ce qu'un jeton associé à un entier garde la valeur de l'entier, et que désormais les jetons pour les chaînes de caractères garde aussi la chaîne correspondante.*

Exercice 5 *Modifiez le fichier* `Parser.java` *pour obtenir un analyseur syntaxique qui reconnaît les flots de jetons qui appartiennent à la grammaire définie en Figure 1 au début de ce TP. Attention : la grammaire de Figure 1 n'est pas LL(1). Il faut donc trouver une grammaire LL(1) équivalente et ensuite l'implémenter. Vous écrirez votre grammaire en commentaire au début de votre fichier* `Parser.java`*. Aussi, rappelez-vous de prendre en compte le symbole* `$` *de fin de flot.*

Exercice 6 *Vérifiez que votre code rejette bien les expressions contenues dans les fichiers* bad* *et accepte ceux des fichiers* good* *qui se trouvent dans le sous-répertoire* `Section2` *du répertoire* `TP6_Info4_MathInfo` *de Didel.*

3 On calcule

Exercice 7 *Modifiez l'analyseur syntaxique afin qu'il affiche à la fin de l'analyse la valeur de l'expression arithmétique qu'il vient d'analyser, dans le cas où l'on n'a pas de soustraction* `(s-t)`*. Si* `s` *est une chaîne de caractères et* `n` *un entier, on définit* `s*n` *comme la chaîne produite par n concaténations de* `s` *avec elle-même (en particulier,* `"s"*2="s"+"s"`*). Par exemple,*

— *il affiche* `Bonjour monde` *si l'expression est :*

```
(("Bon"+("jou"+"r")) + " monde")
```

— *il affiche* `cocorico` *si l'expression est :*

```
(("co"*2)+"rico")
```

— *et il déclenche une erreur si la proposition est :* `2*"aa"+"a"`

Attention, il est nécessaire de modifier la classe `LookAhead1` *pour permettre à* `Parser.java` *d'accéder à la valeur d'un jeton associé à un entier ou une chaîne de caractères.*

Exercice 8 (Bonus) *Modifiez l'analyseur syntaxique afin qu'il effectue aussi le calcul pour les expressions* `(s-t)`*. Si* `s` *est une chaîne de caractères et* `c` *un caractère, on définit* `s-c` *comme la chaîne* `s` *à laquelle on a retiré la première occurence de* `c` *s'il y en a une*[1]*. Enfin, si* `s` *et* `t` *sont des chaînes de caractères, on définit* `s-t` *comme la chaîne* `(((s-t[0])-t[1])-...-t[l])` *où* `t[0]...t[l]` *sont les caractères composant* `t`*.*
Par exemple, il affiche `Bonjour monde` *si l'expression est* `(("Bronijouz " - "riz") + "monde")` *et il affiche* `cocorico` *si l'expression est* `(("rico"*3)-("riz"*2))`

Exercice 9 (Bonus) *Traitez les exceptions pour que les messages d'erreur de votre programme soient informatifs.*

1. Vous pourrez vous servir de la méthode `removeFirstChar` qui vous est fournie dans `Parser.java`