Université de Caen Basse-Normandie
U.F.R. de Sciences
Département d'informatique

Bâtiment Sciences 3 - Campus Côte de Nacre
F-14032 Caen Cédex, France

TD-TP $n^o 3$

Niveau	**L3**
Parcours	**Informatique**
Unité d'enseignement	**ULI5B - Méthodes et modèles pour le logiciel**
Élément constitutif	**ECI54 - Conception et programmation par objets**
Responsable	**Emmanuel Cagniot** **Emmanuel.Cagniot@ensicaen.fr**

1 Exercice

Dans un TD-TP précédent, nous avons écrit les classes `Jeton` et `Lexical` permettant de représenter respectivement un jeton et un analyseur lexical de la grammaire suivante :

```
Expression     ::=  [Additif] Terme { Additif Terme }
Terme          ::=  Facteur { Multiplicatif Facteur }
Facteur        ::=  Entier | '(' Expression ')'
Additif        ::=  '+' | '-'
Multiplicatif  ::=  '*' | '/'
Entier         ::=  Chiffre { Chiffre }
Chiffre        ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Le rôle de l'analyseur lexical est de transformer un flot d'entrée composé de caractères en un flot de sortie composé de jetons. Ce dernier est consommé par l'analyseur syntaxique dont le rôle est de vérifier que l'axiome de la grammaire (ici `Expression`) peut effectivement se dériver en la suite de jetons constituant le texte à analyser.

La grammaire présentée ci-dessus est de classe $\mathcal{LL}(1)$, c'est à dire que la partie gauche de chaque règle de production est unique. En conséquence, chaque jeton lu détermine immédiatement la règle de production concernée. Plus généralement, une grammaire est dite de classe $\mathcal{LL}(k)$ si k jetons ($k > 0$) sont nécessaires pour déterminer la règle de production concernée.

Les analyseurs syntaxiques $\mathcal{LL}(1)$ sont récursifs descendants simples (sans backtracking). L'idée est d'associer une méthode à chaque règle de production qui n'est pas prise en charge par l'analyseur lexical. Le rôle de cette méthode consiste à vérifier la concordance du texte analysé avec l'une des parties droites de la règle. Ainsi, dans notre exemple, une méthode sera associée aux non terminaux `Expression`, `Terme` et `Facteur`.

Nous souhaitons écrire l'analyseur syntaxique de la grammaire présentée ci-dessus. La figure 1 présente son diagramme de classes.

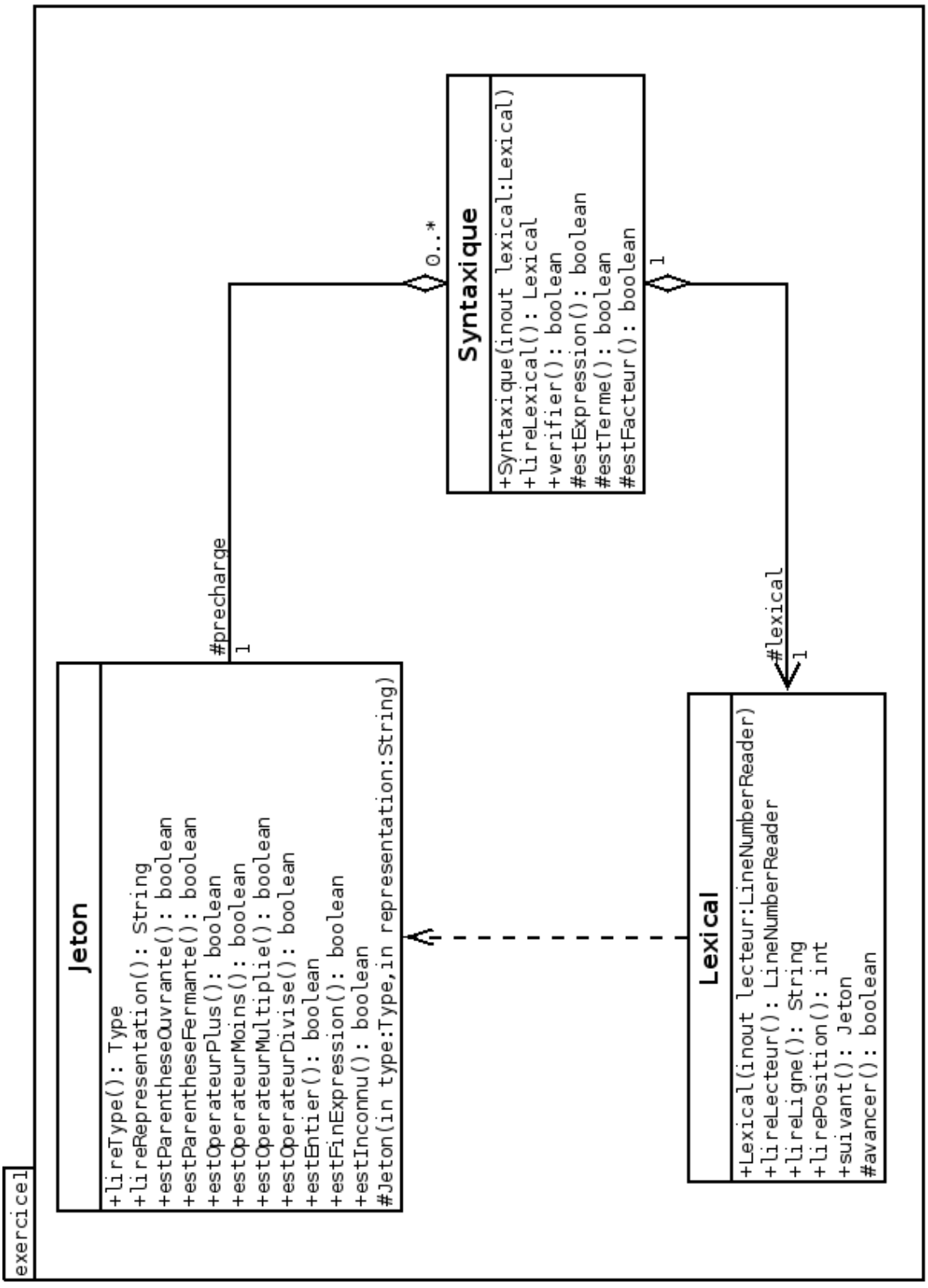

FIGURE 1 – Diagramme de classes de l'analyseur syntaxique.

1.1 Question

Écrivez la définition de la classe `Syntaxique`. Sa méthode `verifier` indique si le texte auquel est connecté l'analyseur lexical est syntaxiquement correct.

1.2 Question

Écrivez une classe `DemoSyntaxique` représentant le programme de démonstration de l'analyseur syntaxique. Cette application récupèrera le nom d'un fichier texte contenant une expression via la ligne de commandes. Après avoir connecté un analyseur syntaxique à cette expression, elle lancera sa vérification puis affichera le verdict sur la sortie standard. En cas d'expression incorrecte, elle affichera le numéro de la ligne fautive ainsi que cette dernière avec le jeton inconnu entre crochets.

Cette application devra vérifier que le fichier indiqué existe bel et bien. Elle devra également signaler toute impossibilité d'en lire le contenu. Une ligne de commandes vide indiquera que l'utilisateur demande de l'aide : il ne s'agit donc pas d'une erreur.

La figure 2 présente un exemple de fichier texte contenant une expression syntaxiquement incorrecte (à gauche) et la trace d'exécution obtenue pour cet exemple (à droite).

2 Exercice

Nous définissons un langage byte code pour une machine à pile, c'est à dire une machine dont le processeur est dépourvu de registres. La table suivante présente son jeu d'instructions :

Mnémonique	**Signification**
`push e`	Empile l'entier `e`
`print`	Dépile l'entier en sommet de pile et affiche sa valeur
`neg`	Dépile l'entier en sommet de pile et empile son opposé
`add`	Dépile les entiers `e1` puis `e2` et empile la somme `e2+e1`
`sub`	Dépile les entiers `e1` puis `e2` et empile la différence `e2-e1`
`mul`	Dépile les entiers `e1` puis `e2` et empile le produit `e2*e1`
`div`	Dépile les entiers `e1` puis `e2` et empile la division entière `e2/e1`

2.1 Question

Sachant que les priorités des opérateurs `+`, `-`, `*` et `/` sont les priorités usuelles, transformez la classe `Syntaxique` écrite dans l'exercice précédent, en la classe `Compilateur` qui représente un compilateur byte code pour la grammaire correspondante. Ce compilateur aura la double tâche de vérifier la correction syntaxique de l'expression tout en générant le byte code correspondant, le tout en un seul et même passage. Le byte code sera écrit sur un flot de sortie de sortie de classe `PrintStream` fourni en argument à la méthode principale `compiler` (remplaçante de `verifier`).

2.2 Question

Transformez la classe `DemoSyntaxique` écrite dans l'exercice précédent en la classe `DemoCompilateur`. La figure 3 présente un exemple de fichier texte contenant une expression syntaxiquement correcte (à gauche) et la trace d'exécution obtenue pour cet exemple (à droite). Le flot de sortie sur lequel est écrit le byte code est tout simplement la sortie standard.

05/03/2012 incorrecte.txt 1

```
-(3 + (2 *5)
      *
(-5 - 3) /
-2
)
```

05/03/2012 trace_syntaxique.txt 1

```
[cagniot@ciril028n exercice1]$ java tdtp3/exercice1/DemoSyntaxique incorrecte.txt
Erreur (entre crochets) en ligne : 8
[-]2

[cagniot@ciril028n exercice1]$
```

FIGURE 2 – Trace d'exécution (à droite) obtenue pour l'expression syntaxiquement incorrecte de gauche.

05/03/2012 correcte.txt 1

```
-(3 + (2 *5)
   *
(-5 - 3) /
2
   )
```

05/03/2012 trace_compilateur.txt 1

```
[cagniot@ciril028n exercice2]$ java tdtp3/exercice2/DemoCompilateur correcte.txt
push 3
push 2
push 5
mul
push 5
neg
push 3
sub
mul
push 2
div
add
neg
print
```

FIGURE 3 – Trace d'exécution (à droite) obtenue pour l'expression syntaxiquement correcte de gauche.