Université Paris Diderot
L2 Informatique et Math-Info

Analyse de Données Structurées
Année 2014-2015

TP n°4

Introduction à l'analyse syntaxique dans Java

Exercice 1 *Sur Didel vous trouverez les fichiers vus en cours qui implémentent un analyseur syntaxique pour la grammaire : $F \mapsto a$, $S \mapsto F \mid (F + S)$. Téléchargez et compilez ces fichiers, faites des tests pour en comprendre le fonctionnement.*

1. *Est-ce que le programme marche comme attendu ? Testez-le sur le mot `(a+a))`...*
   Corrigez ce bug en ajoutant à la grammaire un nouveau symbole terminal `$` représentant la fin du mot : modifiez les règles en conséquence et le programme pour qu'il rejette les mots comme `(a+a))`.
2. *Si le mot rentré n'appartient pas au langage, le programme ne nous donne aucune information sur le type d'erreur qu'il a rencontré. Ceci peut être très limitant, notamment il est précieux de connaître la position du symbole où le programme s'est bloqué et quel était le non-terminal qu'il cherchait à réduire.*
   Apportez les modifications nécessaires pour que le programme affiche ces informations. Par exemple sur `(a+b)`, on doit avoir quelque chose comme :

```
It is not accepted.
ParserException:
Error at position 3
cannot reduce F
waiting for "a" or "("
```

   et sur `(a+aa` quelque chose comme :

```
It is not accepted.
ParserException:
Error at position 4
cannot reduce "a"
waiting for ")"
```

   (Suggestion : c'est l'exception `ParserException` qui doit permettre de communiquer les informations concernant l'erreur détectée.)

Exercice 2 *TODO : donner une grammaire pour les polynômes qui n'est pas LL(1) et demander de la transformer dans une grammaire LL(1) (Attention à la fin du mot). Construire l'analyseur correspondant, en gardant le message d'erreur.*

Modifiez le programme pour qu'il accepte la grammaire suivante :

$$I \mapsto a \mid b \qquad O \mapsto + \mid - \mid * \qquad S \mapsto I \mid (SOS)$$

Attention l'analyseur syntaxique doit contenir une méthode `term(char c)` qui consomme un symbole terminale du flot d'entrée et une méthode pour chaque symbole non-terminale.

Branchement

On se propose dans cette section de faire interagir un analyseur lexical (que l'on créera à l'aide de `JFlex`) et un analyseur syntaxique que l'on va construire à la main. L'objectif final sera de pouvoir évaluer des expressions arithmétiques bien parenthésées.

Exercice 3 *Écrire à l'aide de JFlex un analyseur lexicale qui reconnaisse la grammaire suivante :*

$$S \mapsto I \mid (SOS) \qquad I \mapsto n \in \mathbb{N} \qquad O \mapsto + \mid - \mid *$$

On écrira les fichiers Sym.java *et* Token.java *en conséquence. En particulier, les tokens correspondant aux entiers et aux opérateurs devront disposer d'un champ* `value` *et d'une méthode correspondante.*

Exercice 4 *En s'inspirant des fichiers vu au premier exercice, nous allons simuler le fonctionnement de* LookAhead1Reader. *Créer une classe* LookAhead1 *telle que :*

- *le constructeur* `LookAhead1` *prenne en argument un lexeur,*
- *on dispose de deux champs privés* `token` *et* `current` *qui contiennent le lexeur et le token actuel,*
- *on dispose d'une méthode* `boolean check(Sym s)` *qui vérifie si le token actuel est bien de type* s,
- *on dispose d'une méthode* `void eat(Sym s)` *qui consomme le token actuel s'il est bien de type* s, *et lève une exception sinon.*

Exercice 5 *Écrire un parser correspondant à la grammaire, de telle sorte que la classe* Parser *:*

- *le constructeur* `Parser` *prenne en argument un lexeur*
- *dispose d'un champ privé de type LookAhead1*
- *dispose d'une méthode pour chaque règle de réduction de la grammaire, que l'on nommera* `sNonTerm`,`intTerm` *et* `opTerm`.
- *les méthodes pour les terminaux consomment le token correspondant*

Écrire enfin une classe Arithmetic.java *qui contiendra le* `main`, *qui prendra un fichier en argument, en fera l'analyse lexicale et l'analyse syntaxique du flot de token ainsi produit.*

Exercice 6 *Modifiez le parser pour que les méthodes* `intTerm` *et* `opTerm` *renvoie la valeur du token consommé et que* `sNonTerm` *renvoie la valeur de l'expression arithmétique correspondante. On ajoutera pour cela des méthodes* `getIntValue` *et* `getOpValue` *dans la définition des tokens correspondant. Testez votre code sur les fichiers* good*.

Exercice 7 *Vérifiez que votre code rejette bien les expressions contenues dans les fichiers* bad*. *En vous inspirant de ce que vous avez fait au première exercice, modifiez les exceptions rendues afin de les rendre plus expressives.*

Exercice 8 *Si par hasard vous avez tout fini et avez peur de vous ennuyer d'ici la fin de la séance, n'ayez peur, il en reste. Réfléchissez à une façon d'adapter ce que vous venez de faire pour reconnaître un langage dont les seules instructions seraient des instanciations de variables entières :*

```
int a = 14+12*45;
int b = 11-23;
```

Puis essayez d'adapter cela à un langage dans lequel on aurait une première partie avec une ligne de déclaration par variables, puis une ligne par instanciation, en autorisant l'utilisation de variables déjà instanciées au sein des expressions arithmétiques :

```
int a = 42;
int b = 2*a-23;
int c = 2*b-a;
```

Vous pouvez vous servir d'une table de hashage (cf TP 2) et des conseils avisés de votre chargé de TP.