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 a che 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 *Le but de cet exercice est d'obtenir un analyseur syntaxique pour les polynômes en une inconnue. Une grammaire simple qui génère les polynômes peut être :*

$$S \mapsto a \mid c \mid (S + S) \mid (S * S) \mid (S - S)$$

où "a*" représente l'inconnue et "*c*" le coe cient. Le problème de cette grammaire est de ne pas être LL(1) (pourquoi ?). Trouvez une grammaire LL(1) qui génère le même language (*Suggestion : ajouter un symbole non terminal pour les opérateurs arithmétiques*). Construire l'analyseur correspondant (attention à la fin du mot), en permettant des messages d'erreur informatifs.*

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 lexical qui reconnaisse la grammaire LL(1) trouvée à l'exercice précédent pour des expressions arithmétiques sans inconnue.*

Vous compléterez le fichier arithmetic.flex *et écrirez les fichiers* Sym.java *et* Token.java *en conséquence. En particulier, les tokens (jetons) correspondant aux entiers et aux opérateurs devront disposer d'un champ* `value` *et d'une méthode correspondante.*

Nous allons connecter l'analyseur syntaxique à l'analyseur lexical au moyen d'un objet de la classe *LookAhead1*, dont le code vous est fourni. Dans cette classe, on dispose :

- d'un constructeur `LookAhead1` prenant en argument un lexeur,
- de deux champs privés `lexer` et `current` qui contiennent le lexeur et le token actuel,
- d'une méthode `boolean check(Sym s)` qui vérifie si le token actuel est bien de type *s*,
- 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 4 *Écrire un analyseur syntaxique correspondant à la grammaire, tel que dans la classe* Parser *:*

- *on 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*

Compléter enfin la 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 5 *Modifiez l' analyseur syntaxique pour que les méthodes intTerm et opTerm renvoient 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 6 *Vérifiez que votre code rejette bien les expressions contenues dans les fichiers* bad**. En particulier, avant de rendre le résultat du parsing, faites attention à ce que tout l'intégralité du flot de token ait été lu... En vous inspirant de ce que vous avez fait au première exercice, modifiez les exceptions rendues afin de les rendre plus expressives.*

(Suggestion : comment déclarer des exceptions dans un fichier .flex ? comment récupérer le numero de ligne, colonne, etc. qui peut étre la source d'un erreur pendant l'analyse lexicale ? Voir dans manuel de JFlex la description des options %char, %line, %column, %yylexthrow.)

Exercice 7 (Bonus) *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 , 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.