Université Paris Diderot — Analyse de Données Structurées
L2 Informatique et Math-Info — Année 2014-2015

TP n°8

Expressions arithmétiques

Le langage décrit par la grammaire suivante permet de produire les opérations arithmétiques élémentaires : l'addition, la multiplication, la soustraction et la division. Chaque mot du langage doit être produit par une dérivation commençant par le non-terminal `Exp`.

```
      Exp ::= int | (Exp ExpSuite)
      int ::= 0 | "-"?[1-9][0-9]*
 ExpSuite ::= + Exp | - Exp | / Exp | * Exp
```

Le but de ce TP est d'écrire un interpréteur pour ce langage. Plus précisément, le programme final doit prendre en entrée un fichier contenant une série d'opérations élémentaires. Il doit d'abord transformer le programme en un flot de jetons avant de vérifier que la syntaxe est correcte puis de construire l'arbre de syntaxe abstraite. Une fois l'arbre de syntaxe abstraite construit, le programme doit le parcourir afin de calculer le résultat obtenu.

1 Les expressions arithmétiques

Exercice 1 *Dans un premier temps, déterminez quels sont les di érents jetons que le programme doit générer.*

Exercice 2 *Téléchargez les fichiers sur DidEl et complétez `expression.flex` et et `Token.java` en vue de `Sym.java` afin de générer les jetons que vous avez choisis à l'exercice précédent. On se contentera de ne préciser que les entêtes des fonctions dans `Token.java`.*

Exercice 3 *Le fichier `AbstractSyntax.java` contient une implémentation des arbres de syntaxe abstraite en java. Complétez les fonctions de `Parser.java` afin de construire l'arbre de syntaxe abstraite associé au langage.*

Exercice 4 *Ajouter à la classe `Expression` une fonction récursive qui parcourt l'arbre et calcule puis retourne la valeur de l'expression encodée par l'arbre.*

2 Programmes avec variables

On introduit à présent une grammaire pour écrire des programmes simples avec variables. Toutes les variables sont censées garder des valeurs entières.

Voici la nouvelle grammaire :

Chaque mot du langage doit être produit par une dérivation commençant par le non-terminal `Prog`.

```
    Prog  ::= ε | Inst Prog
    Inst  ::=  VAR variable; | variable = Exp; | PRINT Exp; | FOR Exp TIMES {Prog}
variable ::= [a-z][a-zA-Z]*
     Exp ::= int | variable | (Exp ExpSuite)
     int ::= [1-9][0-9]*
ExpSuite ::= + Exp | - Exp | / Exp | * Exp
```

Exercice 5 *Téléchargez les fichiers sur DidEl et complétez `expression.flex`, `Sym.java` et `Token.java` afin de générer les jetons dont vous avez besoin.*

Exercice 6 *Le fichier `AbstractSyntax.java` contient une implémentation des arbres de syntaxe abstraite en java. Complétez les fonctions de `Parser.java` afin de construire l'arbre de syntaxe abstraire associé au langage.*

Exercice 7 *Completez la classe `ValueEnvionment` pour qu'elle puisse à tout moment contenir les variables déjà déclarées du programme et leurs valeurs.*

Exercice 8 *Ignorez pour le moment les `FOR`. Completez les fonctions `run()` et `exec()` des classes `Program` et `Instruction` pour qu'on puisse exécuter un programme ne contenant pas de boucle.*

Exercice 9 *Completez votre travail en tenant compte des boucle `FOR`. Ecrivez, dans le langage introduit ci-dessus, un programme qui calcule le factoriel de 10 et testez-le.*

3 Traduction des programmes

Maintenant, nous voulons traduire un programme écrit dans le langage ci-dessus en Java, et nous voulons le faire à partir de l'arbre de syntaxe abstraite — et non à partir de la grammaire concrète.

Exercice 10 *Créez, donc, une fonction récursive `String toJava(int indent)` qui produit un programme en Java à partir de l'arbre de syntaxe. Comme d'habitude, vous pouvez commencer par ignorer les boucles, pour les intégrez ensuite quand vous vous sentez plus à l'aise.*

Faite scrupuleusement attention à la bonne indentation et à la bonne gestion des variables, en particulier les variables de boucles. Assurez-vous que même les boucles imbriquées sont bien traitées.