Expressions avec Cup

Objectif Ce but de ce TP est de mettre en pratique les grammaires à opérateurs avec CUP : utilisation d'une grammaire ambiguë avec spécification des priorités et associativité des opérateurs. Ce TP permet aussi de réviser l'analyse lexicale et de concevoir et construire une structure de données.

Matériel fourni Récupérer sur le portail l'archive `tp5.tgz` qui contient la structure de projet habituelle.

Le DSL Il est volontairement très simple pour que le TP ne prenne pas trop de temps. Une version plus réaliste est disponible en fin de sujet. On souhaite décrire des affectations variable-expression. Les variables ne sont pas déclarées. Les expressions sont les expressions parenthésées sur des constantes entières, des identificateurs et les opérateurs `+`, `-`, `*`, `/`. L'opérateur unaire `-` est prioritaire sur les opérateurs `*` et `/`, qui sont prioritaire sur les opérateurs `+` et `-` binaire.

```
x := 3; y := 4;
z := - (100 + x * 32 - y);
```

Le but de l'attribution est d'afficher à la fin de la reconnaissance du fichier la valeur de chaque variable.

Les grammaires à opérateurs et Cup On a vu en cours que les grammaires à opérateurs sont naturellement ambiguës, et donc non automatisable. CUP les accepte néanmoins, en précisant par ailleurs les priorités et l'associativité des opérateurs. Dans cet exemple, l'opérateur `not` est prioritaire sur l'opérateur `and`, qui est prioritaire sur les opérateurs `or` et `xor`.

```
terminal OR, XOR, NOT, AND;
...
precedence left OR, XOR;
precedence left AND;
precedence left NOT;
```

Chaque déclaration débutant par `precedence` spécifie un niveau de priorité *et* un type d'associativité. Le mot-clé `precedence` introduit un nouveau niveau de priorité, *supérieur au niveau précédent*. Les mot-clés `left` et `right` désignent respectivement l'associativité gauche et droite [1]. Si l'opérateur est unaire il faut tout de même utiliser `left` ou `right` même si ça semble inutile. Tout terminal non déclaré avec `precedence` se voit attribuer le degré de priorité le plus bas (niveau 0). Les instructions de `precedence` sont à placer après la déclaration des terminaux et non-terminaux, et avant le `start with` (s'il y en a un).
Il reste à traiter le problème des symboles qui ont une double interprétation suivant qu'ils sont unaire ou binaire, comme le symbole `MOINS`. Le moins unaire a une priorité plus forte que celle du moins binaire, mais les deux ont la même représentation (`"-"`, symbole `MOINS`). Pour s'en sortir on utilise un mécanisme de CUP qui permet d'associer une priorité à une production. Par défaut chaque production possède un niveau de priorité : c'est celui du terminal le plus à droite dans sa partie droite. Si la production ne contient pas de terminaux alors elle a le niveau de priorité le plus bas. Il est possible de forcer le niveau de priorité d'une production en faisant précéder le `;` qui la termine par `%prec <terminal>`. La production a alors le niveau de priorité de `<terminal>`. On introduit donc un terminal `UNAIRE` bidon [2] qui a la priorité du moins unaire, et on écrit :

```
terminal UNAIRE;
precedence left ...
precedence left UNAIRE;
```

```
expr ::= ...
       | expr MOINS expr {: ... :}
       | MOINS expr {: ... :} %prec UNAIRE
;
```

1. Le mot-clé `nonassoc` existe aussi mais on ne l'utilisera pas. Il sert à interdire deux occurrences consécutives et de même priorité d'un terminal. Par exemple si `==` est déclaré `nonassoc` alors `6 == 3 == 2` génèrera une erreur.
2. Même si ce terminal ne sera jamais produit par l'analyseur lexical, il faut tout de même le déclarer en tant que terminal de CUP. Il apparaîtra donc inutilement dans `TypeSymboles.java`.

1 Travail à réaliser

1.1 Lexique et syntaxe

Écrire des fichiers de spécification pour JFlex et Cup, et des tests pour l'analyse syntaxique. Garder cette grammaire nue dans un fichier à part pour pouvoir la réutiliser plus tard.

1.2 Première attribution

Attribuer la grammaire pour afficher en fin de reconnaissance la valeur des variables, en imaginant une structure de données capable d'associer une valeur à une variable. On supposera dans un premier temps que le fichier d'entrée respecte les dépendances de données, par exemple on ne traitera pas le cas :

```
x := y;
y := 3;
```

1.3 Seconde attribution

Repartir de la grammaire nue. Attribuer la grammaire pour synthétiser une structure de données représentant *exactement* le fichier d'entrée. On soignera particulièrement la représentation des expressions : c'est de la COO. Afficher de même la valeur des variables en fin de reconnaissance. Deux options sont possibles :

- exécuter l'affichage dans l'attribution de la grammaire ;
- créer un nouveau lanceur de l'analyseur syntaxique attribué dans le paquetage `executeurs` pour récupérer l'attribut de l'axiome au niveau de l'appel de la méthode `parse` :
  `TypeAttributAxiome attributAxiome = parser.parse();`
  `traiter(attributAxiome);`
  où `TypeAttributAxiome` est le type de l'attribut associé à l'axiome dans le `.cup`.

1.4 Pour ceux qui ont fini

Ajouter à l'une ou l'autre des attributions un contrôle sémantique qui rejette les fichiers d'entrée qui ne respectent pas les dépendances de données.

1.5 Pour ceux qui s'ennuient

La version réelle du DSL contient des tests booléens. Ce fichier permet par exemple de se faire rapidement une idée de la formule d'abonnement la plus économique parmi celles proposées par l'éditeur pour abonner plusieurs sites d'une entreprise selon la durée de l'abonnement. Comme le contrôle des dépendances de données devient impossible statiquement, on se contentera d'afficher un `?` si la valeur à afficher n'existe pas.

```
input duree, nbabonnement

reduc := 10
duree = 6 and nbabonnement <= 10 : abonnement6mois := 120
duree = 6 and nbabonnement > 10 : abonnement6mois := 100
duree = 36 : abonnement36 := 500 - 500*reduc/100

print "nombre abonnements "  nbabonnement
print "\\n"
print "prix par an "
   duree = 6 : abonnement6*2*nbabonnement
   duree = 36 : abonnement36 / 3 * nbabonnement
```