UE 4TIN603U – Compilation – Licence 3 – 2020-2021

TD5 - Contrôle de type et syntaxe abstraite

Première partie : une grammaire d'expressions typées

Soit un langage de programmation simplifié qui décrit une séquence de calculs.

Par exemple, avec ce langage, il est possible d'écrire

```
pi := 3.1416; r := 35; 2*pi*r
```

Ce qui veut dire ceci : `pi` est une variable réelle dont la valeur 3,1416, `r` est une variable entière qui vaut 35. On réalise la multiplication de `2*pi*r` avant de l'afficher.

Grammaire G Soit la grammaire suivante :

```
axiom:
        affs expression
        ;

affs:
        aff affs
        | aff
        ;

aff:
        lhs ":=" expression ';'
        ;

expression:
        IDENTIFIANT
        | ENTIER
        | CHAINE
        | expression '+' expression
        | expression '-' expression
        | '-' expression
        | expression '*' expression
        | expression '/' expression
        | '(' expression ')'
        ;
```

Exercices

1. Réaliser un programme reconnaissant le langage de G complet. Pour cela, on fera en sorte que la grammaire soit analysable avec `Bison` en utilisant les opérateurs de précédence.
   On utilisera la sémantique habituelle des opérateurs $+ - */$ sur les entiers et les réels. L'opérateur `+` sera un opérateur de concaténation des chaines de caractères.
2. En utilisant une grammaire attribuée, représenter pour chaque expression son type.
   On utilisera pour cela
   - L'interface `Type` du miniprojet que l'on adaptera,
   - La classe `TypeExpr` du miniprojet que l'on adaptera.

Deuxième partie : syntaxe abstraite

Il s'agit maintenant de représenter le calcul de chaque expression dans une structure arborescente. Ceci dans la perspective de construire un compilateur plus tard...

1. En modifiant la grammaire attribuée précédente, représenter pour chaque expression une structure arborescente qui en décrit sa sémantique.

   On utilisera pour cela

   — L'interface `Tree` qu'il faudra créer,

   — La classe `TreeExpr` qu'il faudra créer.

2. Ajouter un champ `Type` à la classe `TreeExpr` qui permet d'enregistrer le type de chaque expression.
3. Maintenant, pour chaque affectation, il faut enregistrer l'expression correspondante dans un environnement, et pour chaque variable trouvée dans une expression, il faut retrouver le type correspondant.

   On utilisera pour cela

   — L'interface `Environment` du miniprojet que l'on adaptera,

   — La classe `MapEnvironment` du miniprojet que l'on adaptera.

   On fera l'évaluation du type de chaque expression pour vérifier que les opérateurs sont correctement utilisés.

   Cette évaluation du type aura aussi pour intérêt d'utiliser leur bonne implémentation. Par exemple `x + y` pourra être l'addition de deux nombres ou la concaténation de deux chaînes de caractères.

   Enfin les types numériques pourront être utilisés comme chaînes de caractères par un mécanisme de coercition implicite. Dans ce cas, l'expression induite sera l'application d'une routine permettant d'afficher le nombre (`toString()`). Cette routine devra être ajoutée à la structure arborescente de l'expression.

   Exemple :

   ```
   pi := 3.1416; r := 35; n := 62; "module: " + 2*pi*r/62
   ```