Projet de Compilation – Second devoir

Analyse sémantique

Le but de cette étape est de prendre une partie du projet précédent, et d'ajouter l'analyse sémantique qui contient

1. La gestion de tables de symboles
2. La vérification des types
3. La représentation des structures de contrôles
4. Le calcul des coercitions et surcharges

Nous n'implémenterons pas l'héritage, les restrictions d'accès (`public`, `private`, `protected`), les packages, les déclarations liées à la classe (`static`). Il est cependant possible de laisser les mots clefs qui correspondent à ces éléments de la programmation sans effet.

1 Gestion des tables de symboles

Les tables des symboles permettent d'enregistrer :
— Les variables, leur type, la taille mémoire allouée, la valeur initiale
— Les types déclarés
— Les classes, fonctions et procédures

1. Écrire une classe `StackEnvironments` et une classe `Environment`. La première permet de créer une pile d'environnements où le sommet désigne l'environnement courant. La seconde permet de créer un environnement, c'est-à-dire une table des symboles.
2. Faire en sorte que les variables soient enregistrées dans les environnements qui correspondent à leur portée. Nous rappelons que `CUP` ne traite que d'attributs synthétisés. Il est donc nécessaire d'associer un environnement à un bloc, non par attribut, mais par variable déclarée dans la classe `Parser`. En effet, vous ne pourrez pas passer l'attribut d'un bloc à une déclaration de variable enchâssée dans ce bloc directement en `CUP`.
3. Utiliser la même classe `Environment` pour déclarer les types, les classes, les procédures et les fonctions.

Le compilateur doit être en mesure d'identifier une erreur de déclaration (variable utilisée pour un type, une fonction ou un classe et inversement).

2 Vérification des types

1. Écrire une classe `Type` qui permet de créer une expression de type sous la forme d'un arbre binaire.

2. Y implémenter une méthode `String toString()` permettant d'afficher une expression de type dans une chaine.
3. Écrire une classe `TypeDiff` qui permet d'instancier un objet représentant un couple (x, y) de types. Dans cette classe, on implémentera la méthode `boolean isDiff()` qui retourne `true` si la différence entre x et y est nulle. Attention : deux objets différents ayant la même expression de type ont une différence nulle !
4. Implémenter une méthode `TypeDiff diff(Type otherType)` dans la classe `Type` qui fabrique un objet de `TypeDiff` représentant la première différence entre `this` et `otherType`.
5. Implémenter une méthode `Type unify(Type otherType)` dans la classe `Type` qui renvoie un type distingué `bottom` si l'unification échoue, ou alors le résultat de l'application du plus grand unificateur de `this` et `otherType`.
6. Utiliser cette méthode pour signaler les erreurs sémantiques
   — des affectations ne respectant pas les types des variables
   — des expressions ne respectant pas les types des variables et des fonctions
   — des passages de valeurs ne respectant pas la signature des fonctions

3 La représentation des structures de contrôles

L'écriture structurée dans ce langage de programmation évolué qui utilise les mots clefs `if`, `while`, `foreach`, `repeat`, etc. sera traduite sous la forme d'un arbre de syntaxe abstraite.

1. Écrire une interface `Statement` qui permet de créer un arbre.
2. Implémenter les classes de `Statement` pour les différentes structures de contrôle et les expressions de votre langage de programmation.
3. Y implémenter la méthode `String toString()` permettant d'afficher une expression de l'instruction dans une chaine. L'expression est laissé à votre discrétion. Par exemple l'instruction `if (a<6) a+=1;` aura cette représentation : `IF(<(a,6),AFF(a,+(a,1)))`
4. Ajouter une instance de type aux classes `Statement`.
5. Écrire une méthode de vérification de type `boolean check()` qui échoue et renseigne le type d'erreur en cas d'erreur de typage..

4 Le calcul des coercitions et surcharges

1. Si une opérande polymorphe (un nombre entier par exemple) reçoit un ordre de coercition (*casting*) implicite, ajouter à l'arbre de syntaxe abstraite l'appel de fonction correspondant.

   Exemple :

```
1 r: real;
2 i: integer;
3 print(r+i);
```

Si ce code est accepté et que la fonction `print` prend un argument de type `String`, alors il revient à ceci : `print(realToString(r + intToReal(i))`
Reprendre la méthode de vérification de type `boolean check()` pour lui ajouter un ordre de coercition implicite. Cette méthode aura donc un effet sur `this`.

2. Si un opérateur ou une fonction est polymorphe, il doit correspondre à des implémentations différentes selon le type des arguments ou du contexte.
   Exemple (impossible en Java) :

```
1     procedure setValue(i: integer){
2         //implementation (i)
3         this.nom = i.intToString();
4     }
5     procedure setValue(s: string){
6         //implementation (ii)
7         this.nom = s;
8     }
9     procedure string getValue(){
10        //implementation (iii)
11        return this.nom;
12    }
13    procedure integer getValue(){
14        //implementation (iv)
15        return this.nom.stringToInt();
16    }
17
18 [...]
19
20 object.setValue(36); //implementation (i)
21 object.setValue("36"); //implementation (ii)
22 s: string = object.getValue(); //implementation (ii)
23 i: integer = object.getValue(); //implementation (iv)
```

   (a) Ajouter aux tables des symboles un identificateur unique pour chaque fonction. Les différentes implémentations des fonctions auront un identificateur distinct.

   (b) Aux fonctions polymorphes, associer une variable de type. Par exemple `setValue` prend un argument de type variable et `getValue` rend une valeur de type variable.

   (c) Après vérification de type, si une fonction polymorphe ne reçoit pas un type constant, signaler l'erreur. Sinon, utiliser la signature calculée par unification pour identifier l'implémentation. Si aucune implémentation ne correspond (par exemple en appelant `setValue` avec un argument de type `real`), signaler l'erreur.