Sémantique et TDL.

Typage : Collecte des informations et Vérification

Considérons la grammaire décrivant les instructions du langage Bloc :

1. $S \rightarrow B$
2. $B \rightarrow \{\ LI\ \}$
3. $LI \rightarrow I\ LI$
4. $LI \rightarrow \Lambda$
5. $I \rightarrow \textbf{const}\ T\ \textbf{id} = V\ ;$
6. $I \rightarrow T\ \textbf{id} = E\ ;$
7. $I \rightarrow \textbf{id} = E\ ;$
8. $I \rightarrow \textbf{if}\ (\ E\ )\ B\ \textbf{else}\ B$
9. $I \rightarrow \textbf{if}\ (\ E\ )\ B$
10. $I \rightarrow \textbf{while}\ (\ E\ )\ B$
11. $I \rightarrow \textbf{print}\ E\ ;$
12. $T \rightarrow \textbf{int}$
13. $T \rightarrow \textbf{boolean}$
14. $T \rightarrow < T\ ,\ T >$
15. $E \rightarrow \ldots$

Le diagramme de classe (simplifié) de l'AST est rappelé dans la figure 1.
Soit le programme :

```
test {
  int i = 1;
  const int j = 2;
  < int, int> p = < 3, 4>;
  int k = fst p;
  if ( i < 5 ) {
    int j = 5;
    j = i * (snd p);
    i = j + 1;
    while ( k < 10 ) {
      int p = 3;
      k = k + i;
    }
  } else {
    if ( i + j > 10 ) {
      const boolean p = false;
      print p;
    }
    print p;
  }
  print j;
}
```

Nous désirons implanter le mécanisme de typage pour Bloc en respectant les mêmes principes que C par un parcours de l'arbre abstrait.

Il s'agit d'une part de construire la partie de l'arbre abstrait représentant les informations de typage, et d'autre part d'exploiter ces informations pour vérifier que les programmes sont

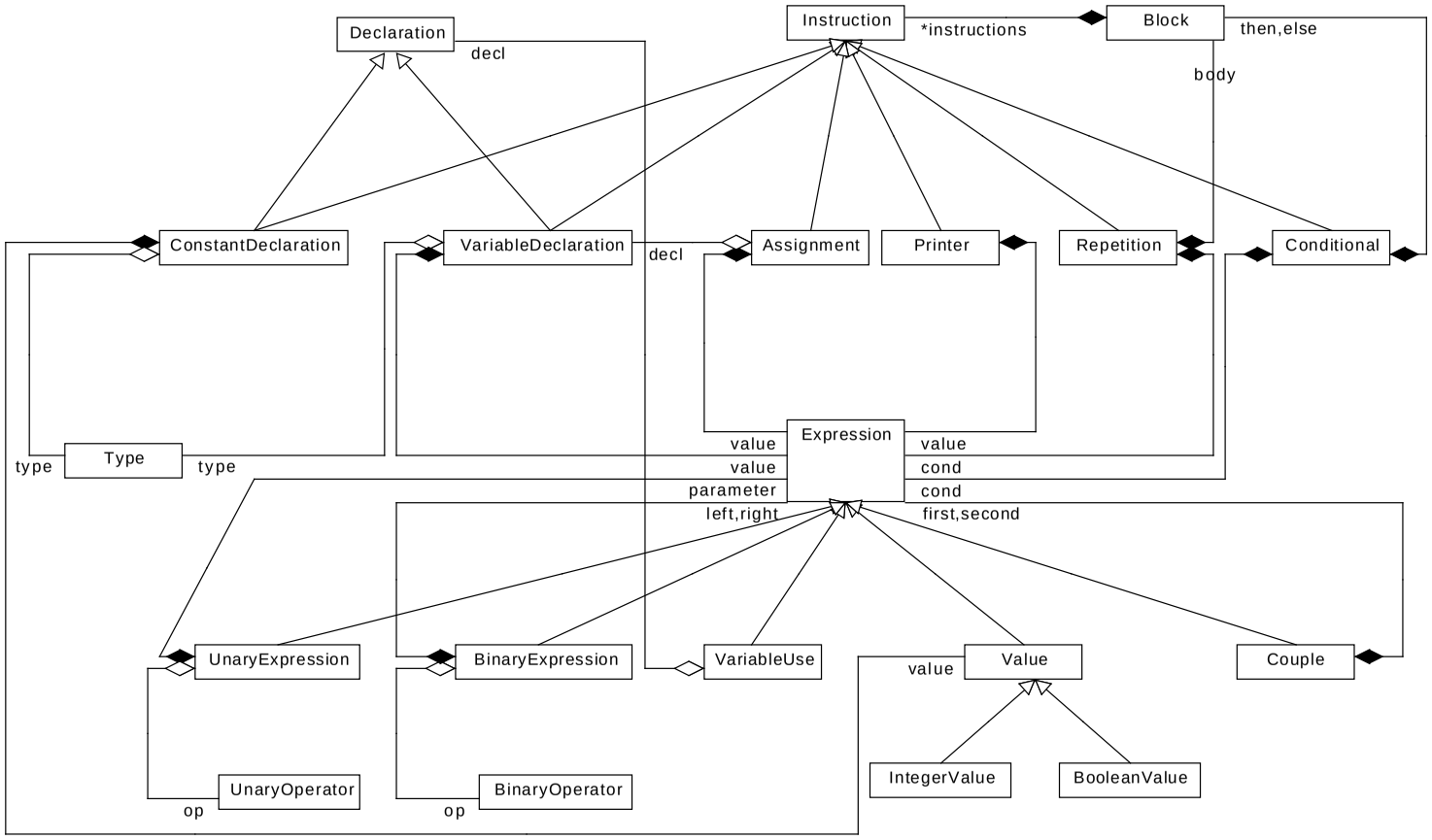

Figure 1: Diagramme de classe (simplifié) de l'AST

bien typés, c'est-à-dire qu'ils ne produiront pas d'erreurs de typage lors de leur exécution. Cette approche est semblable au typage spécifié pour `miniML`.

Cette sémantique étends celle des expressions étudiée en cours.

1 Typage des instructions

1. Définir les activités de vérification de typage qui doivent être effectuées pour le programme précédent. Vous indiquerez celles-ci pour chaque ligne du programme.
2. Compléter le diagramme de classe de l'AST pour ajouter les classes correspondant aux informations de typage.
3. Définir le traitement sémantique nécessaire pour vérifier qu'un programme en langage Bloc est bien typé.

2 Définition de type

Nous étendons le langage Bloc par une instruction de définition d'identificateurs de type.

16. $I \rightarrow$ **typedef** T **id** ;
17. $T \rightarrow$ **id**

Soit un exemple associé :

```
test {
   typedef int Entier;
   Entier i = 1;
   typedef < Entier, Entier> PaireEntier;
   PaireEntier p = < i, 2>;
}
```

1. Compléter le diagramme de classe de l'AST pour ajouter la ou les classes nécessaires à la gestion des types définis.
2. Décrire la sémantique de construction de l'arbre abstrait.
3. Définir le traitement sémantique nécessaire pour la résolution des identifiants (collecter les définitions de type dans la table des symboles et remplacer les identificateurs de type par leur définition).
4. Définir le traitement sémantique nécessaire pour vérifier qu'un programme en langage Bloc ainsi modifié est bien typé.

3 Type enregistrement

Nous étendons le langage Bloc par la notion de type enregistrement. Celui-ci est composé de champs typés. Un nom symbolique local est associé à un type enregistrement pour permettre la construction de définitions récursives (par exemple pour les structures chaînées).

18. $T \rightarrow$ **struct id** { LC }
19. $LC \rightarrow C\ LC$
20. $LC \rightarrow \Lambda$
21. $C \rightarrow T$ **id** ;
22. $E \rightarrow E$. **id**
23. $E \rightarrow$ { LE }
24. $LE \rightarrow E$, LE
25. $LE \rightarrow E$

Soit un exemple associé :

```
test {
   struct Point { int x; int y} p = { 1, 2};
   Entier i = p.x + p.y;
}
```

1. Compléter le diagramme de classe de l'AST pour ajouter la ou les classes nécessaires à la gestion des enregistrements.
2. Décrire la sémantique de construction de l'arbre abstrait.
3. Définir le traitement sémantique nécessaire pour la résolution des identifiants.
4. Définir le traitement sémantique nécessaire pour vérifier qu'un programme en langage Bloc ainsi modifié est bien typé.