Création de langages informatiques
INF600E

Corinne Pulgar
PULC26628901

Projet de session
Rapport

Travail remis
à
Étienne M. Gagnon

Université du Québec à Montréal
26 avril 2021

Règles sémantiques

1. Tous les éléments doivent avoir un `label` non vide.
2. La restriction d'une conclusion ne peut pas être une chaîne vide.
3. Un diagramme ne peut contenir qu'une seule `conclusion`.
4. Une `conclusion` ne peut mener vers un autre élément.
5. Il est impossible de faire un lien avec un élément non-déclaré.
6. Les `domain`, les `rationale`, les `support` et les `subconclusion` ne peuvent mener qu'à une `strategy`.
7. Une `strategy` peut seulement mener à une `conclusion` ou à une `subconclusion`.
8. Tous les éléments du diagramme doivent être liés.
9. Le diagramme doit être un graphe acyclique.

Dans l'implémentation présente, on valide le type des éléments de façon syntaxique seulement. Il serait possible de le faire aussi sémantiquement mais il s'agirait pour l'instant d'une redondance.

Les deux dernières règles ne sont pas encore implémentées.

Question 1

Dans l'interpreteur, la notion de `Scope` est remplacée par la notion de `Frame`. Un `Frame` contient toutes les informations relatives à une `FunctionInfo` associée et aux variables déclarées dans ce cadre. Les `Frames` sont consignés dans une pile : ils sont empilés à chaque appel de fonction et dépilés à la fin de ceux-ci. Un `Frame` de base en empilé au début du programme principal pour faire office de `main`. À l'appel d'une fonction, les paramètres de la fonctions sont ajoutés aux variables du nouveau `Frame` avec leur nom de déclaration et la valeur dans l'appel.

L'interpreteur ne regarde pas les déclaration de fonction et saute directement aux instructions du programme principal. On compte ici sur le fait que le vérificateur sémantique aura déjà créé les fonctions et vérifié leur validité. Au moment de l'appel, on récupéra le `funBody` depuis le `FunctionInfo` pour lancer l'exécution de la fonction.

Exemples des vérifications mise en place

Appel avec un paramètre du mauvais type

```
# proc printresult (1 param entier)
fun printresult(i: int) {
    print("The result is " + i);
}

# fonction max (2 params entiers, retourne entier)
fun max(i: int, j: int):int {
    if i > j then
    return i;
return j; }

# programme principal
var a = 3;
var b = "allo";
# ERREUR EST ICI (b est un string et doit etre un int)
printresult(max(a,b));

```

```
SEMANTIC ERROR: [16:9] expected type INT but got STRING for param 2
     of function max

```

Appel d'une fonction avec le mauvais nombre de paramètre

```
# proc printresult (1 param entier)
fun printresult(i: int) {
    print("The result is " + i);
}

# fonction max (2 params entiers, retourne entier)
fun max(i: int, j: int):int {
    if i > j then
    return i;
return j; }

# programme principal
var a = 3;
var b = 7;
var c = max(a,b);
# ERREUR EST ICI
printresult(a, b, c);

```

```
SEMANTIC ERROR: [17:1] expected call with 1 but got 3

```

```
# proc printresult (1 param entier)
fun printresult(i: int) {
    print("The result is " + i);
}

# fonction max (2 params entiers, retourne entier)
fun max(i: int, j: int):int {
```

```
    if i > j then
    return i;
return j; }

# programme principal
var a = 3;
var b = 7;
# ERREUR EST ICI
var c = max(a);
printresult(c);

```

```
SEMANTIC ERROR: [16:9] expected call with 2 but got 1

```

Ajout d'un return dans fonction sans retour (procédure)

```
# proc printresult (1 param entier)
fun printresult(i: int) {
    print("The result is " + i);
    return "test";
}

```

```
SEMANTIC ERROR: [4:5] cannot have a return in a procedure

```

Ajout d'un return dans fonction sans retour (procédure)

```
# proc printresult (1 param entier)
fun printresult(i: int) {
    print("The result is " + i);
    return "test";
}

```

```
SEMANTIC ERROR: [4:5] cannot have a return in a procedure

```

Fonction avec un type de retour sans return

```
# proc printresult (1 param entier, 1 type de retour ?)
fun printresult(i: int):int {
    print("The result is " + i);
}

```

```
SEMANTIC ERROR: [2:5] cannot have a function with a return type and
     no return.

```

Retour d'un mauvais type

```
# proc printresult (1 param entier)
# fonction max (2 params entiers, retourne entier)
fun max(i: int, j: int):int {
    if i > j then
    return "hello";
return j; }

```

```
SEMANTIC ERROR: [5:5] expected return to be INT but got STRING

```

Question 2

L'architecture de base du compilateur de code Java a été conservée. On aura seulement ajouté le code nécessaire au `CodeGenerator` pour que les déclarations de fonctions soient traitées proprement. La gestion de l'indentation est ajoutée pour chaque fonction ou bloc de code. Le langage original n'étant pas un langage orienté-objet, la génération du code Java se fait dans une classe `Main` et toutes les fonctions deviennent des méthodes `static`. Le code en Java garde la même présentation avec les définitions des fonctions avant le programme principal qui est dans la méthode `main`. La syntaxe initiale ignore les commentaires et la traduction en Java ne retient donc pas les commentaires.

Example de compilation en Java

```
fun printresult(i: int) {
    print("The result is " + i);
}

fun max(i: int, j: int):int {
    if i > j then
    return i;
return j; }

var a = 3;
var b = 7;
printresult(max(a,b));

```

```
class Main {

    public static void printresult(int i) {
        System.out.print(("The result is " + i));
    }

    public static int max(int i, int j) {
        if((i > j)) {
        return i;
        }
        return j;
    }

    public static void main(String[] args) {
        int a = 3;
        int b = 7;
        printresult(max(a, b))
    }
}

```