Rapport de projet de PDS

BARBOT Malo
BEUREL Luca

Pour mener à bien ce TP, nous avons procédé par partie sur un modèle de développement itératif en commençant à coder les différentes parties de la manière la plus simple possible. Nous savions qu'en codant ce cette manière nous allions avoir beaucoup de choses a revoir au fur et à mesure de l'avancée du projet et de l'implémentation des fonctionnalités plus avancées.

Dans l'avancement actuel du projet, nous avons toute la partie des Instructions jusqu'aux appels de fonctions qui sont faites et qui créent la séquence LLVM attendue, les fonctions "READ" et "PRINT" sont dans l'ASD et le parser mais ne génèrent pas de Llvm et ne se Pretty Print pas. Les tableaux n'ont pas du tout été implémentés.

Un des exemples de code que nous avons dû entièrement reprendre a été la conception des Blocs. Nous avions d'abord pensé créer les blocs comme une simple liste d'instructions ce qui nous "économisait" du travail. Cette technique portait ses fruits dans un premier temps mais nous posait de gros problèmes au moment de PrettyPrint et surtout, cela entrait en conflit avec notre gestion de la table des symboles. En effet, le passage de la table des symboles courante à la table des symboles parent était très compliqué avec notre gestion et cela impactait la créations de notre LLVM.
.

La première partie de l'ordre conseillé par le sujet de TP portant intégralement sur les instructions, nous avons créée une classe abstraite dont héritent toutes les autres instructions. L'intérêt principal de cette classe abstraite est de pouvoir traiter les instructions de manière indifférente, cela a été principalement utile au moment d'implémenter les blocs. En effet, les blocs prennent une liste d'instructions et chacune sera traitée en fonction de son sous-type au moment voulu. Au niveau de notre architecture nous avons donc l'affectation, la déclaration, le if, le while, le return et les blocs qui sont de type Instruction.

Néanmoins, le plus gros problème que nous ayons eu a été la gestion de la table des symboles. En effet, nous avions du mal a savoir ou l'appeler et comment bien propager la table entre les différents éléments pour être sûr de ne pas avoir de désynchronisation. En effet, cela a été un bug très récurrent lors du développement : la table n'était pas à jour et donc incomplète par rapport à ce qu'on attendait d'elle.

Un autre grand problème que nous avons rencontré lors de la conception a été la vérification de type. Nous devions savoir quand il était pertinent de le faire et ou. Dans l'état actuel du projet nous faisons une vérification de type pendant l'affectation et l'expression car ce sont les principaux points où nous voyions des potentiels problèmes à l'exécution.

```
if(!ret.type.equals(identType)) {
    throw new TypeException("type mismatch: have " + ret.type + " and " + identType);
}
```

Ainsi sur cet exemple tiré de notre Affectation, nous vérifions bien que le type de la valeur à affecter est bien du même type que ce qui est attendu. Dans le cas contraire, nous soulevons une erreur.