Compte-rendu

Compilateur

Mathias DUGUÉ Théo GIRAUDET

I. Introduction

Le but du projet ponctuant le semestre PDS est un compilateur VSL+ réalisé en Java ou en OCaml pour créer du code LLVM exécutable.

Ce projet synthétise le module en nous demandant de générer un AST à partir d'un fichier source en VSL+ pour ensuite effectuer de la vérification de type ainsi que de la génération de code. Ceci demande d'avoir une grammaire attribuée.

Nous avons choisi de travailler en Java et de créer notre grammaire à l'aide d'ANTLR. Cet outil nous permet d'insérer du code Java au sein même de la grammaire, puis d'ensuite la compiler vers un ensemble de classes Java.

II. Grammaire

Dans la suite nous ne parlerons que du parser car le lexer ne contient que les mots-clés du langage VSL+ ainsi que la reconnaissance des ident, du texte et des entiers.

A. Règles

Les règles de grammaires peuvent être décomposées en 3 grandes catégories :

- Fonctions (déclarations de fonctions, prototypes etc…)
- Statements (block, affectation, structures de contrôle, appel de fonctions...)
- Expressions (règles permettant d'évaluer les expressions et de gérer les priorités opératoires)

Un programme VSL+ est considéré comme une liste de iFunction, une iFunction étant la déclaration d'une fonction ou d'un prototype, les deux définitions étant similaires car une fonction prend juste un statement (qui est son corps) en fin de déclaration

La structure d'un bloc permet de déclarer les variables du bloc au début de ce dernier pour ensuite opérer au moins statement a l'intérieur de celui-ci. De plus, considérer les blocs comme des statement a permis de pouvoir passer n'importe quel statement en tant que corps d'une fonction ou d'une structure de contrôle.

L'appel de fonction est notre le cas le plus particulier car cela peut être un statement ou une expression. En effet une fonction effectuant des opérations sans rien rendre ne peut pas faire partie d'une expression mais une fonction rendant quelque chose peut très bien être utilisée dans une expression et cela nous semblait la manière la plus logique de gérer cela.

B. Points de génération

L'attribution de cette grammaire est effectuée par de la génération du code Java dans les règles. Ces points de génération permettent d'articuler les règles de notre grammaire avec les types créés pour implémenter notre ASD.

Ceci nous a permis beaucoup de choses, comme la création d'une liste d'éléments de types pour considérer des listes de déclaration de fonctions ou de paramètres et surtout de gérer dynamiquement l'héritage et la synthétisation des attributs à travers les règles, ainsi que d'instancier les types de l'ASD.

C. Tests

Afin de tester la grammaire, nous avons rédigé des programmes VSL+ grammaticalement corrects mais qui pouvaient échouer ou non au niveau de la compilation pour divers problèmes afin de réutiliser ces programmes pour tester la compilation et l'exécution du code LLVM.

Nous avons donc utilisé l'outil de visualisation d'ANTLR sous IntelliJ qui calculait un AST de notre programme VSL+ à partir de notre ASD

Par exemple avec le programme :

```
FUNC VOID main() {
        INT a
        a := 0
        PRINT a
}
```

Nous obtenons l'AST de la figure 1.

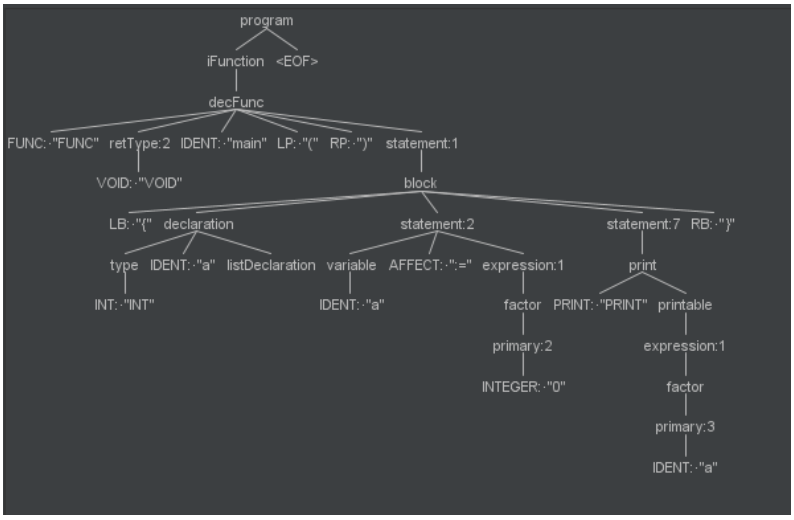

Figure 1. AST obtenu

III. Implémentation

Pour l'implémentation, nous avons suivi la structure proposée avec tout de même quelques modifications sur les classes fournies.

A. Modifications apportées

Tout d'abord, la classe "SymbolTable" a été renommée en "Context" de façon à être plus large, et représenter le contexte d'exécution des instructions et des expressions. Cette classe fonctionne toujours de la même façon, c'est-à-dire qu'il est toujours possible d'empiler les contextes, mais elle permet aussi de stocker la fonction courante du contexte. De plus, une fonction a été ajouté à la classe interne "FunctionSymbol" pour vérifier l'égalité avec un objet en paramètre tout en ignorant le paramètre "Defined". Pour finir, la classe "Expression" a été transformée en interface afin d'éviter d'utiliser l'héritage unique alors qu'il n'est ici pas nécessaire. Ces modifications réalisées, nous ont permis de développer le compilateur de façon plus propre et efficace.

B. Architecture

L'architecture du projet est basée sur le design pattern Interpreter. L'idée est donc d'avoir une classe pour chaque élément sémantique différent du langage, dont la forme globale forme un arbre tel qu'un nœud à pour enfant les éléments sémantiques qu'il possède. Ainsi, une expression représentant l'addition aura pour enfant deux sous-expressions.

- Nous possédons différents types pour représenter nos éléments sémantiques :
- Le type Type, représentant les types du langage (Int, Array, StrinV, Void)
- Le type Expression, représentant les expressions du langage (constante entière, addition, division, appel de fonctions…)
- Le type Statement, représentant les instructions du langage (if, while, affectation, appel de fonction…)
- Le type IFunction, représentant les fonctions du langage (fonction ou prototype)

Cette notion de type est représentée par le biais d'interfaces qu'implémentent la plupart de nos classes. Le fait de passer par des interfaces nous permet notamment de considérer le constructeur « FunctionCall » comme étant à la fois une Expression et un Statement.

D'autres interfaces ont été ajoutées, principalement pour marquer un rôle et généraliser nos fonctions (principes ouvert/fermé de et d'inversion de dépendances de SOLID). Parmi celles-ci, nos pouvons notamment citer « Printable » et « Readable » qui servent à dénoter quels types peuvent être écrits et lus sans avoir à énumérer chaque possibilité ou faire des « if » pour gérer ces comportements dans les classes Read et Print. Sur la même idée, l'interface « Constant » permet de définir quelles expressions

peuvent être considérées comme des constantes et donc être directement inclues dans le message du Print sans passer par des instructions de formatage (%s, %d). Ainsi, le code suivant :

```
a := 0
PRINT "Variable : ", a, ", Constante :", 5
```

Sera compilé de la sorte :

```
@.fmt1 = global [29 x i8] c"Variable : %d, Constante : 5\00"
store i32 0, i32* %b2.a
call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([29 x i8], [29 x i8]* @.fmt1, i64 0, i64 0), i32 %tmp5)
```

Cela permet à l'extension du compilateur d'ajouter deux nouvelles expressions affichables sans modifier la classe « Print ».

Dans la même logique, les instructions de formatage se trouvent directement dans les classes de type, avec l'opération de l'accesseur défini dans l'interface.

Pour représenter les tableaux, nous avons décidé côté Java de créer une classe qui encapsule un Type et une taille, cela permet d'être générique. Côté LLVM, tout est représenté par des pointeurs, ce qui permet de faire le passage de ceux-ci en paramètre.

Pour finir, nous séparons le concept d'accès en lecture et en écriture à une variable par deux ensembles de classes : Dereference et Reference.
Le déréférencement, comme son nom l'indique, déréférence une variable (référence) pour en obtenir la valeur. Ainsi, la classe représentant cela possède pour attribut une Reference qui sera appelée pour l'obtention de la référence vers la variable à déréférencée. Une fois la référence obtenue, il ne nous reste qu'à la déréférencer par le biais d'une instruction « load » dans le cas des variables classiques, ou d'une instruction « getelementptr » dans le cas d'un accès un à un tableau. Il s'agit du seul endroit où nous ne respectons pas entièrement la ligne directive que l'on s'est donné : rendre le code le plus générique possible. En effet, nous pouvons y retrouver directement des tests d'instance pour gérer quelle instruction renvoyée parmi les deux citées. Cela est principalement dû à un manque de temps qui nous a fait accélérer sur la fin pour obtenir un projet fini.

IV - Conclusion

Ce projet nous a pris beaucoup de temps, mais nous sommes très satisfaits du résultat. Cela nous a permis d'approfondir la notion de parsing et de compilation et de la mettre en application sur un projet bien plus complet que celui réalisé en L3.