Syntaxe et sémantique
Project : Compilateur

Emil NOIRHOMME, Geoffroy LARUE

May 9, 2020

Contents

1 Description de la démarche

Pour les 3 étapes du projet, nous avons commencé par une analyse du problème et une recherche de solution individuelle. Nous avons ensuite discuté ensemble de la solution à adopter. Finalement le travail a été distribué en fonction des agendas de chacuns.

2 Table des symboles

La structure de la table des symboles est celle donnée par Terence Parr dans l'ouvrage: *Language Implementation Patterns Create Your Own Domain-Specific and General Programming Languages*. À cette structure, nous avons ajouté les éléments nécessaire à l'interprétation, par exemple, le corps des méthodes et leurs paramètres, et les valeurs que prennent les variables durant l'évaluation du programme.

La table des symboles (cf. Figure 1) est divisée en 2 grandes classes: les Symboles et les Scopes.

Les **symboles** représentent les entités d'un programme, ceux-ci ont 3 propriétés principales: un nom (qui les identifie), un type (entier, booléen, caractère,...) et une catégorie (variable, tableau, record, ...).

Ces entités (symboles) n'existent, et ne sont donc accessibles, qu'à des endroits spécifiques dans le programme. Ces endroits, ce sont les **scopes**. Ils sont de 2 types, le scope global et les scopes locaux. Les symboles contenu par le scopes global sont donc accessibles partout dans le programme. Certains symboles, tels que les méthodes et les structures, sont également des scopes. En effet, ils peuvent contenir des symboles qui n'existent qu'au sein de leur de leur bloc.

3 Description de l'architecture

Le processus de compilation du code Slip est divisé en 4 étapes principales:

- Parsing: création de l'AST
- Création de la table des symboles et vérification des types
  - Création de la table globale
  - Création de la table locale et vérification des types
- Evaluation du code
- Génération du code nbc

Ces différentes étapes sont lancées à partir de méthode compile() de Main.java dans la mesure où l'étape précédente ne contenait pas d'erreur.

Le projet est divisé en 3 packages: `checking`, `nbc` et `symboltable`.

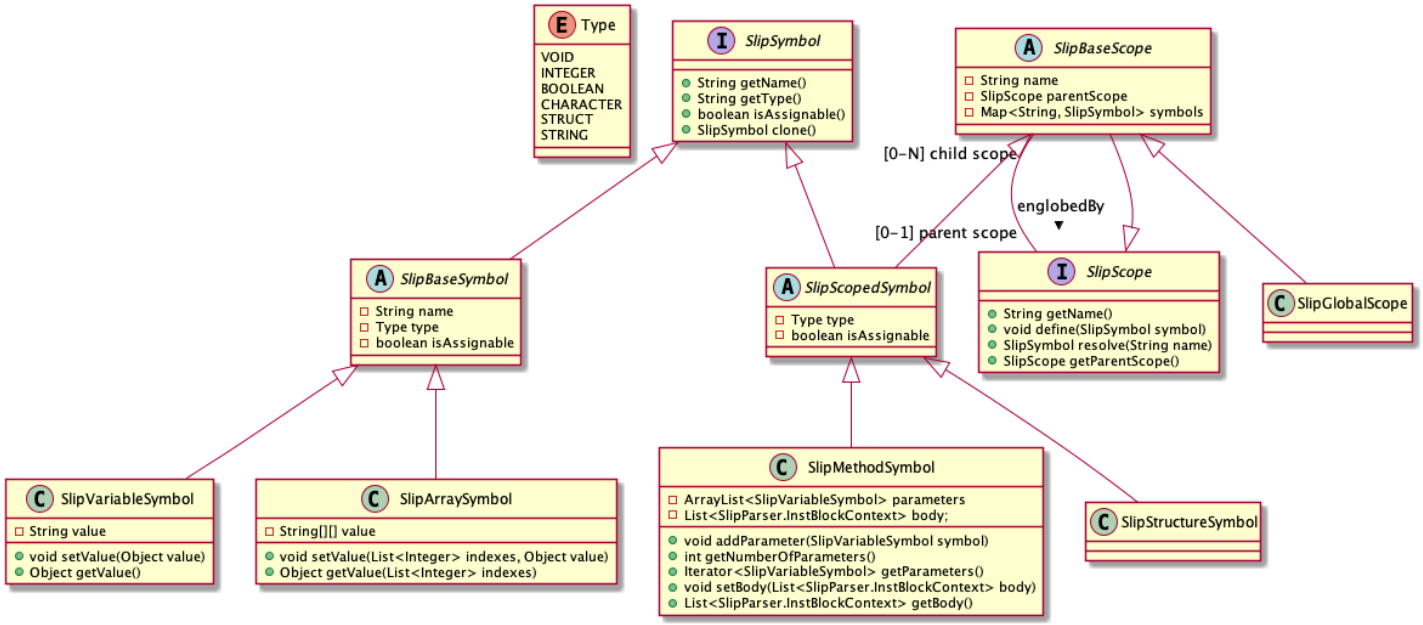

Figure 1: Table des symboles - Diagramme de classe

3.1 Package checking

Ce package gère l'étape de création de la table des symboles et de la vérification des types. Il contient 2 classes principales: `GlobalDefinitionPhase` et `CheckPhaseVisitor`.

3.1.1 GlobalDefinitionPhase

Cette classe a pour objectif de répertorier les variables et fonctions déclarées globalement. Cette étape est nécessaire afin que le langage accepte les forward references. Elle consiste en un visiteur de l'AST. À chaque fois qu'elle rencontre un symbole (variable ou fonction), elle vérifie que le symbole n'existe pas déjà et elle ajoute ce dernier à la table des symboles dans le scope global.

3.1.2 CheckPhaseVisitor

Cette classe fait la même chose que `GlobalDefinitionPhase` (le code commun est repris dans la classe `CheckSlipVisitor` dont elles héritent) mais cette fois pour les variables locales. En plus elle va vérifier les types des expressions. Par exemple lors d'une assignation, elle va vérifier que le type de l'expression à gauche est bien le même que celui de l'expression à droite de `:=`.

3.2 Package nbc

Ce package est responsable de la traduction du code Slip en code nbc. Cette traduction se fait en 2 étapes: l'interprétation du code Slip et la génération d'un code nbc simplifié ne reprenant que la sémantique du code Slip. À noter que nous traduisons le Slip de cette manière à cause d'un changement dans l'énoncé de la 3ème partie intervenu après que nous ayons implémenté l'interpréteur. En effet, afin de ne pas jeter le code de celui-ci nous l'avons utilisé pour générer un code simplifié. Le package contient 2 classes principales: `Evaluator` et `NbcCompiler`.

3.2.1 Evaluator

Ce visiteur va réaliser l'interprétation de tout le code afin de déterminer les valeurs nécessaires à la compilation.

Il va commencer par visiter à nouveau toutes les définitions de variables (dans le scope global et les scopes des fonctions), afin d'évaluer et stocker dans la table des symboles la valeur des variables si une valeur initiale est donnée. Il ne va pas s'occuper des définitions de fonctions, dont le corps est stocké par la `GlobalDefinitionPhase` dans la table des symboles. Il va ensuite exécuter le code de la fonction `main` du programme source : ligne par ligne, il va "exécuter" les instructions :

- dans le cas d'instruction d'assignation : il va évaluer l'expression droite et placer le résultat dans la table des symboles à la variable correspondante (dans le cas d'un tableau, à la case correspondante du tableau "interne" du symbole.

- si l'expression droite est un appel de fonction, l'évaluateur va créer un scope d'appel de fonction, contenant toutes les variables clonées du scope lié à la déclaration de fonction[1]. Ensuite il va visiter, dans le contexte de ce nouveau scope, les instructions qui constituent le corps de la fonction.
- si l'instruction est une instruction d'action, il va l'ajouter à une file (deque) qui sera utilisée par le compilateur `NbcCompiler` pour générer le code.

3.2.2 NbcCompiler

Cette classe génère le code nbc. Elle reçoit de l'`Evaluator` la liste des actions que le code nbc devra exécuter. Elle parcourt la liste en ajoutant au main du nbc les appels aux fonctions. Elle termine en ajoutant uniquement les subroutines nécessaires à l'éxécution du programme.

3.3 Package symboltable

Ce package est détaillé dans la section "Table des symboles".

4 Conclusion

4.1 Forces et faiblesses

Notre expérience assez limitée de la programmation orientée objet se ressent dans l'organisation du code. Nous n'avons pas toujours été capables de découpler autant que souhaité les différentes parties du compilateur.

4.2 Amélioration

Notre implémentation porte un peu trop le poids de la révélation par paliers des détails différentes étapes du projet (consignes par étapes). Par exemple, si nous avions su dès le départ que nous allions interprèter le code, nous aurions fait la vérification des types lors de la phase d'interprétation. Dès lors certaines parties peuvent apparaître quelque peu comme du "bricolage".

4.3 Apprentissage

A travers ce projet, nous avons pu mettre en pratique et mieux cerner les enjeux de la théorie des langages de programmation. Cela nous servira non seulement pour réaliser de petits DSL à l'occasion, mais nous a également permis d'élargir notre compréhension du fonctionnement interne des langages plus généralistes que nous utilisons quotidiennement.

[1] Nous sommes obligés de cloner ce scope pour éviter les partages de références en cas d'appels récursifs à l'intérieur de la fonction.

L'ampleur du projet a été également l'occasion de tenter de mettre en pratique les notions théoriques vues au cours de *CPOO*. Nous avons aussi pu approfondir la connaissance et l'utilisation du design pattern Visiteur.

Enfin, nous avons appris à utiliser la libraire *ANTLR*, que nous utiliserons certainement encore par la suite, pour réaliser d'autres langages avec des fonctionnalités étendues.

4.4 Commentaire constructif

Une chose qui nous aurait aider à voir dès le début comment le langage Slip fonctionnait aurait été d'avoir un exemple de code Slip correct reprenant toutes les structures du langage. Cela nous aurait permis de ne pas devoir revenir en arrière à certains moment.

Parfois les consignes et les tests ne correspondaient pas, et il fallait un peu découvrir la réalité des spécifications en tatônnant, et analysant le résultat des tests. Cela empêche une vraie anticipation et rend difficile une organisation cohérente du code.

Le langage ne permet pas d'utiliser: les fonctions de type `void`, les énumérations, le type `String`.