Projet : Compilateur SLIP vers NBC
IHDC B332 - Théorie des langages : Syntaxe et sémantique

Nicolay Matthias
Evrard Cédric

UNamur

1 Introduction

Dans ce rapport, nous allons présenter les différents aspects techniques du projet de syntaxe et sémantique ainsi que les choix techniques que nous avons effectués lors de la création du compilateur SLIP.

2 Démarche générale

Notre groupe était composé de deux personnes. La démarche que nous avons adoptée a été de laisser Cédric commencer à travailler en chaque début de partie afin de créer l'architecture de l'application ainsi que la structure des différents packages. Une fois que cette structure était créée, une répartition était faite du travail.

Il a parfois été difficile de découper la réalisation du travail de manière équitable. En effet, une dépendance assez importante dans la grammaire à la partie *expression droite* a fait que celle-ci devait souvent être réalisée avant tout le reste et conditionnait énormément la structure du projet.

Avant la fin de la phase 2, un gros travail de refactoring a été effectué afin de rendre le code plus lisible et d'en faciliter la maintenance et l'écriture du code. Cela nous a aussi permis de travailler plus rapidement sur la phase 3 en reprenant le même principe de structure.

De manière générale, la réalisation du projet s'est bien passée même si nous aurions pu mieux répartir du travail entre nous. Malheureusement, la charge de travail en dehors du projet ainsi que le commencement des phases de façon un peu tardif a fait que nous avons dû aller assez vite pour réaliser chacune d'elle et donc se répartir le travaille sur celui qui pouvait avancer le plus rapidement.

3 Structure de la table des symboles

La structure utilisée se trouve dans la *class* `Context.java`.

La "Table des symboles" en elle même est composée de quatre `Hashtables` qui sont accompagnées d'un *array* pour chaque symbole.

Les quatre symboles différents sont :

- VariableBase
- Function
- Record
- Enumeration

Notre choix d'utiliser quatre `Hashtable` est simple, notre point de vue est qu'il y a quatre *class* de symboles. Ainsi donc pour avoir une simplicité de codage, de stockage et d'accès aux symboles, nous avons utilisé cette implémentation. Cela permet aussi de facilement éviter les variables ou function avec des noms similaires.

Notre "Table des symboles" est donc utilisée comme suit:

- Lors du passage de notre visiteur, chaque symbole est stocké dans l'`Array` assigné à son symbole et référencé dans sa `Hashtable`. Le tout dans le contexte dans lequel se trouve ce symbole.

- Lors de la vérification de la sémantique du code, chaque `Function` est vérifiée par la validité de son type de retour et de la bonne présence des `VariableBase` dans leur contexte respectif pour qu'elles respectent leurs portées. Cette action est facilitée par la présence des `Hashtable`.
- Chaque symbole est donc restitué en code NBC en conservant chaque fonctionnalité du code **Slip** entré dans le compilateur.

Ceci est donc le résumé de l'implémentation de notre "Table des symboles".

4 Architecture du compilateur

Notre Compilateur est divisé en 6 p*ackages*:

- Application : qui contient toutes les classes liées à la sauvegarde des symboles
- Exception : qui contient l'exception `PlayPlusException`
- Language : qui contient le visiteur du language et tous les *helpers* qui aident le visiteur
- Main : qui contient la fonction main, le parseur *ANTLR* et le *listener* d'erreurs
- Map : qui contient le *Listener* de la carte
- NBC : qui contient le *Printer* NBC, le *Visiteur* NBC, les *Helper* et *Writer* qui vont aider les deux premières classes

4.1 Application

Voici les différentes *class* dans le *Package* Application.

4.1.1 Application

Cette *class* permet d'initialiser une nouvelle instance d'`Application`, elle contient aussi les méthodes permettant d'entrer dans le bon contexte et d'en sortir. Elle contient aussi les méthodes d'ajout des différents symboles et celles qui permettent d'y accéder.

4.1.2 Array

Cette *class* permet d'initialiser un `Array` qui sera utilisé dans un contexte.

4.1.3 Context

Class principale qui initialise un `Context` qui va être utilisé dans une `Application`. Ce `Context` contient nos 4 `Hashtable` dans lesquels nous sauvegardons nos symboles.

Ces 4 `Hashtables` sont les suivantes:

- variableSymbols : qui va contenir tous les symboles des variables.
- functionSymbols : qui va contenir tous les symboles des fonctions.

- recordSymbols : qui va contenir tous les symboles des records.
- enumSymbols : qui va contenir tous les symboles des énumérations.

Elles sont épaulées par 4 `Array`:

- variables : qui contient les différentes variables du programme.
- functions : qui contient les différentes fonctions du programme.
- records : qui contient les différentes structures du programme.
- enums : qui contient les différentes énumérations du programme.

De plus, la carte est aussi ajoutée dans cette *class*.

Finalement, en plus de cela, la *class* contient les méthodes qui permettent d'ajouter et retourner les `Variable`,`Array`,`Function` et `Enum`.

4.1.4 Enumeration

Cette classe permet d'initialiser une *Enumeration*.

4.1.5 Function

Cette classe permet d'initialiser une *Function*. Elle *extends* `Context` car elle doit pouvoir stocker les mêmes symboles qu'un contexte. En plus du contexte, elle doit stocker ses arguments et son type de retour. Toutes les méthodes liées à cette classe servent à ajouter/retourner les divers symboles spécifiques à la fonction.

4.1.6 Record

Cette classe permet d'initialiser un `Record`.

4.1.7 VariableBase

Cette classe permet d'initialiser une `VariableBase`. Qui est le type de base d'une variable.

4.1.8 Variable

Cette classe *extends* `VariableBase` et ajoute juste un attribut `isConstant` qui est là pour savoir si la variable est une constante ou non.

4.2 Exception

Ce package ne contient que `PlayPlusException` qui *extends RuntimeException* et qui est lancé quand il y a une exception non attendue dans le *Parser*.

Nous utilisons une `RuntimeException` afin de ne pas avoir à l'ajouter dans les signatures des méthodes surchargées.

4.3 Language

Ce package contient le `LanguageVisitor` qui va visiter l'arbre syntaxique et en extraire les symboles utiles. Le package contient un sous-package *Helper* qui contient toutes les méthodes qui vont être utilisées pour récupérer les symboles.

ActionExpression Cette classe contient les méthodes qui servent à *parser* les `ActionExpression`.

ArrayHelper Cette classe contient la méthode qui sert à convertir x noeuds terminaux `NUMBER` en un tableau d'entiers.

BooleanExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle expression booléenne.

CharExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle expression de caractère.

ConstantExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle expression de constante.

DeclarationExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelles Déclaration et Instruction.

EnumExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle énumération.

ForExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle expression `for`.

FunctionExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle expression de fonction, argument, appel de fonction ou déclaration de fonction.

GenericExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle expression gauche, de comparaison ou de parenthèses.

IfExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle instruction `if`.

IntegerExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle expression d'`integer`.

RepeatExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle expression `repeat`.

StructureExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle structure.

Tuple Cette classe permet de créer un `Tuple` de deux items de type différent.

VariableExpression Cette classe contient les méthodes qui servent à *parser* les instructions, les définitions, les initialisations de variables et de tableaux.

VariableHelper Cette classe contient les méthodes qui servent à ajouter une variable, un tableau et une structure. Elles sont utilisées dans `VariableExpression` et `ConstantExpression`.

WhileExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle expression `while`.

4.4 Main

Ce package contient la fonction `Main` le *paser ANTLR* et le *listener* d'erreurs.

4.5 Map

Ce package ne contient qu'une classe qui est le `MapListener` qui va *parser* la carte , sauvegarder ses symboles dans un tableau et vérifier qu'elle est correcte.

4.6 NBC

Ce package contient deux classes principales `NBCVisitor` et `NBCPrinter` qui vont respectivement visiter notre programme et le convertir en instructions *NBC*. Les classes aidant sont réparties en deux packages.

4.6.1 Helper

Premier package qui contient toutes les classes d'aide.

ActionExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle action en code *NBC*.

ActionInterface Classe qui permet d'exécuter le *Writer*.

ArrayExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle *Array* en code *NBC*.

AssignationExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle assignation en code *NBC*.

ComparisonExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle comparaison en code *NBC*.

DeclarationExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle déclaration en code *NBC*.

ForExpression Cette classe contient les méthodes qui servent à *parser* n'importe quel `for` en code *NBC*.

FunctionExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle fonction en code *NBC*.

IfExpression Cette classe contient les méthodes qui servent à *parser* n'importe quel `if` en code *NBC*.

IntegerExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle expression entière en code *NBC*.

IntegerHelper Cette classe sert à renvoyer le bon symbole lors d'une opération entière.

LeftExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle expression gauche en code *NBC*.

MapExpression Cette classe contient les méthodes qui servent à importer la carte.

RepeatExpression Cette classe contient les méthodes qui servent à *parser* n'importe quel `repeat` en code *NBC*.

RightExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle expression droite en code *NBC*.

VariableExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle variable en code *NBC*.

VariableHelper Cette classe sert à renvoyer le bon symbole dépendant le type de variable.

WhileExpression Cette classe contient les méthodes qui servent à *parser* n'importe quelle expression *While* en code *NBC*.

4.6.2 Writer

Ce package contient toutes les classes servant à écrire le code *NBC*.

ActionWriter Cette classe permet d'écrire n'importe qu'elle type d'action.

FunctionWriter Cette classe permet d'écrire n'importe qu'elle type de fonction.

IfWriter Cette classe permet d'écrire n'importe qu'elle type de `if`.

LogicWriter Cette classe permet d'écrire n'importe qu'elle type de comparaison logique.

LoopWriter Cette classe permet d'écrire n'importe qu'elle type de boucle.

NBCCodeTypes Cette classe renvoie la bonne écriture du type de variable en code *NBC*.

NBCIntCodeTypes Cette classe renvoie la bonne écriture du type d'opération entière en code *NBC*.

NBCOpCodeTypes Cette classe renvoie la bonne écriture du type d'opérateur logique en code *NBC*.

NBCWriter Cette classe écrit le code *NBC* de base inhérent au fonctionnement du robot.

OperationWriter Cette classe permet d'écrire n'importe qu'elle type d'opération en code *NBC*.

PreprocessorWriter Cette classe permet d'écrire les informations pour le préprocesseur en code *NBC*.

VariableWriter Cette classe permet d'écrire n'importe qu'elle type de variable en code *NBC*.

5 Conclusion

Nous sommes assez contents de notre compilateur, particulièrement l'architecture de celui-ci qui, suivant le contexte ANTLR, redirige vers la partie qui s'occupe de la gestion de ce contexte. Cela nous a permis de facilement réutiliser nos codes et de gérer la plupart des parties du compilateur de manière indépendante.

Une faiblesse de celui-ci pourrait être l'utilisation importante de `instanceof`. Nous aurions pu regarder à une alternative suivant les informations qui étaient contenues dans le contexte ANTLR sur lequel on travaillait pour voir si celui-ci ne contenait pas une information qui nous aurait permis d'éviter l'`instanceof`.

Une amélioration que nous aimerions apporter est un *refactoring* de la grammaire. En effet, nous avons pu remarquer lors de l'écriture du compilateur que nous avions à passer par certains contextes qui ne faisaient rien. Nous pourrions donc retirer certains éléments de la grammaire pour améliorer celle-ci.

Le projet nous a appris l'écriture d'une grammaire assez complète ainsi que la rigueur dans la compilation d'un code vers un autre. De fait, il y a vite beaucoup d'éléments à gérer dans le code NBC de sortie.

Une difficulté du projet a été de ne pas pouvoir tester le code compilé sur un vrai robot LEGO®, car le simulateur *Bricx* n'est pas complet et donc nous ne pouvions savoir si ce que nous générions était correct et correspondait au comportement attendu.