Yassine AMRAOUI
Mathias BAZIN
Nathan BONNARD
Adrien LAVILLONNIERE
Hien Minh NGUYEN

ET4 INFO
2017-2018

Projet compilation : Compte-rendu

Introduction

Voici le compte-rendu de notre projet de compilation. Dans ce rapport sera détaillé l'état d'avancement du programme ainsi que les contributions de chaque membre.

Grammaire, arbre et structures

L'arbre syntaxique correspondant à notre grammaire est fonctionnel et reconnaît l'ensemble des cas du langage. Il peut être affiché grâce à la fonction *afficherProgramme*().

Nous avons complété les structures existantes et ajouté des nouvelles afin de stocker toutes les expressions d'un programme.

Vérifications contextuelles

Les vérifications contextuelles sont presque toutes faites, sauf oubli de cas particuliers (voir plus bas). Nous avons rajouté un dossier contenant des fichiers tests que nous avons nous-même mis au point. Vous trouverez des commentaires à l'intérieur pour voir le résultat attendu et l'explication des erreurs.

Notre compilateur réalise les vérifications contextuelles suivantes :

- ❏ vérification de la bonne construction des classes :
    - ❏ vérification s'il y a une boucle d'héritage (arrête le programme pour éviter des boucles infinies).
    - ❏ vérification sur l'ensemble de la liste de classes (doublons de classe, vérifications blocs donc définitions de variables et méthodes, entre autre override, expressions, affectations, types cohérents, portée des variables.)
    - ❏ vérification de la bonne construction des objets.
- ❏ vérification du bloc principal :
    - ❏ vérification de la bonne définition des variables.
    - ❏ vérification de la portée des variables
    - ❏ vérification du typage lors des affectations
    - ❏ vérification de l'appel des méthodes (est défini dans la classe ou au dessus, bon paramètres...)

"Problèmes" connus de vérification :

- Si une variable est utilisée dans une Expression d'une classe alors qu'elle est pas définie, cette erreur va se répéter pour toutes les classes filles existantes.

- Il faut noter que certaines erreurs font effet “boule de neige”, avec une seule erreur on peut en générer beaucoup. nous avons essayé d’éviter au plus ces cas mais il en reste néanmoins.
- Il nous reste un cas pas traité : après un if then else dans lequel on aura associé à une variable x une valeur dans un des deux blocs, la variable sera considéré comme définie alors qu’elle n’aura pas forcément d’expression associée.

Génération du code intermédiaire

Concernant la génération de code, ont été traités :
- les expressions, instructions et listes d’instructions
- en particulier, les blocs ITE, les envois et les sélections
- les classes, leurs méthodes et leurs déclarations
- l’allocation d’objets
- la table virtuelle et les appels dynamiques

Des erreurs existent encore cependant pour l’envoi, la sélection, l’appel d’un constructeur et l’affectation. Le passage en paramètre de variables aux fonctions possède par moment un comportement indéfini.

La fonction *genCode()* donne en sortie un fichier nommé genCode qui peut être passé dans le programme interprète. Le code du fichier simple.txt peut être généré mais n’est pas complet. Seul le code du fichier simpleif.txt fonctionne et affiche le résultat voulu :

```
Programme de comparaison de valeurs !
valeur de y : 2
valeur de x : 9

Resultat de la comparaison :
y est inf a x
valeur de y : 2
valeur de x : 9
Au revoir
```

Contribution de chaque membre du groupe

Les éléments sont classés en fonction de la contribution du membre.

Tous :
- Construction et affichage de l’arbre syntaxique
- Tokens, précédence, associativité
- Résolutions des conflits de la grammaire et des bugs

Yassine :
- Grammaire, arbre syntaxique, génération de code : *codeConstructeur,* code*ConstructeurVersionStructure, codeObj, codeSelec, codeLDeclChampStructure, codeDeclChampStructure*
- Construction et remplissage des structures : *initClasse, makeClassesPrimitives, getClassePointer*
- Vérifications contextuelles : *checkArguments, checkHeritageClasse,*

checkOverrideMethode, checkOverrideLClasse, checkClassDefine, checkDoublonClasse

Mathias :

- Arbre syntaxique, structures et remplissage des structures
- Génération de code : instanciation, création et détermination de l'adresse d'une variable, gestion des environnement globaux, gestion de la pile de l'interprète

Nathan :

- Vérifications contextuelles : *checkPortee, verifLParam, checkHeritageClasse, checkArgumentOverride, checkOverrideMethode, checkOverrideLClasse, checkDoublonClasse, checkBoucleHeritage, checkMethodes*, checkDoublonMethodesLClasse, checkDoublonMethodes.
- Grammaire
- Construction des structures et remplissage des structures

Adrien :

- Vérification contextuelle : *checkClassDefine, checkPortee, getTypeMethode, checkExpr, checkSelection, getType, getTypeId, SetEnvirronementType, checkBlocClasse, envHerite, addEnv, addVarEnv, removeEnv, checkAff, getTailleListeVarDecl, getVarSelection, verifVarDeclDefinition, checkCast*.
- Analyse lexicale
- Remplissage des structures

Minh :

- Génération de code : instructions, expressions, envoi, sélection, affectation, print/println/toString, déclarations de variables et de méthodes, table virtuelle, fonctions de parcours de liste, fonctions renvoyant la structure recherchée pour les classes, méthodes et déclarations et diverses fonctions auxiliaires
- Analyse lexicale

Conclusion

Ce projet nous a permis de mieux comprendre les principes étudiés lors du cours de compilation. En travaillant sur toutes les subtilités du langage proposé, nous avons pu voir en détail le fonctionnement d'un compilateur.

Nous avons rencontré des difficultés dans la parallélisation des tâches, ce qui nous a considérablement ralenti dans notre progression.

Malgré cela, notre compilateur permet de générer un programme simple en code intermédiaire, et est à deux doigts de générer le code d'un programme plus complexe.