Rapport de Projet - Compilation

BROCARD, Thimotée
thimotee.brocard@etu.u-bordeaux.fr | A2

EVEN, Melvin
melvin.even@etu.u-bordeaux.fr | A4

LESNÉ, Nathan
nathan.lesne@etu.u-bordeaux.fr | A3

TOY-RIONT–LE-DOSSEUR, Maëlle
maelle.toy-riont-le-dosseur@etu.u-bordeaux.fr | A3

2 mai 2019

Table des matières

1 Introduction

Le projet consiste à analyser le langage impératif Lea et à le traduire en code intermédiaire et enfin de générer le code pour une machine Y86.

L'objectif étape par étape est donc :

— De créer analyseur syntaxique pour le langage Lea qui vérifie que le texte est bien engendré par la grammaire de Lea.

— De créer un analyseur sémantique pour le langage Lea qui vérifie et construit le typage de chaque expression dans le cas ou il est valide.

— De compiler le langage Lea vers le code intermédiaire.

— De générer le code machine Y86 à partir de ce code intermédiaire.

2 Plan du programme

2.1 Scanner

Le rôle du scanner est de lire le code Lea fourni et de vérifier que ce qui est écrit est syntaxiquement correct ("mot" bien formé) et de générer les tokens correspondant a ce qu'il a lu pour que le Parser les traite.

Le scanner ressemble beaucoup aux scanners fait lors des TD précédents, nous avons ajouté quelques particularités liées au langage ou pour nous faciliter quelques tâches :

- Mode débug (affichage des tokens détectés si on passe $-debug$ en argument).
- On convertis les nombres $hexa$ / oct à cette étape.
- On utilise un état spécial pour la gestion des $String$.

2.2 Parser, environnements et analyse sémantique

Le rôle du parser est de vérifier que le code fourni est grammaticalement correct, il va vérifier la cohérence entre les différents tokens (qui sont dans les $\%terminals$) qui lui sont fournis lors de la lecture afin de bien typer. Il agence ensuite tout cela en noeuds (node) formant un arbre syntaxique qui va nous servir à générer le code intermédiaire. Il est associé à trois environnements qui nous serviront à stocker les différents types de chaque symbole (dans une $HashMap$). Il y a un environnement pour stocker les types, un pour les procédures et un pour les variables.

En outre les environnements nous permettent d'associer les noms d'une variable ou d'une fonction à leur type et de vérifier l'existence d'une variable, ainsi que sa portée.

C'est ici que nous vérifions certaines erreurs comme :

— La **redéclaration** : pour chaque ajout de variable ou de fonction on doit vérifier dans l'environnement si elle existe déjà, cependant une variable locale doit pouvoir être redéclarée en une variable globale. (ie : une variable dans un bloc peut avoir le même nom qu'une variable en dehors du dit bloc, cette redéclaration est correcte).

— La bonne **déclaration** d'une variable avant son utilisation. Une variable doit être déclarée avant d'être utilisée.

— Le **typage correcte** des différentes expressions.

2.3 Code intermédiaire

Au sein du package $node$ on dispose d'une vingtaine de classes $NodeYYY$. Ces nodes nous permettent de générer un arbre de code intermédiaire à partir des méthodes $generateIntermediateCode$. En effectuant un parcours en profondeur sur cet arbre, on stocke ses noeuds dans une liste afin d'obtenir une linéarisation.

3 Difficultés et solutions

3.1 Parser

Nous avons implémenté le parser surtout au début du projet, puis nous avons continué de le modifier en fonction de nos besoins au cours du projet.

Les problèmes que nous avons rencontrés puis résolus au cours de la création du parser sont les suivant :

— Redéclaration de types/variables/fonctions, et redéfinitions de fonctions :
nous utilisons les environnements pour détecter les symboles déjà existants ou non existants. Pour les fonctions, nous utilisons un attribut de $TypeFunct$ pour détecter si la fonction a déjà été définie. Nous lançons également des $Exceptions$ quand nous détectons des cas d'erreurs de redéfinition.

— Surcharge de $equals$ dans certains types pour avoir le bon comportement dans le $checkType$ de $NodeAssign$: avec les classes de base, il n'était pas possible de détecter certaines erreurs comme $l = m2$ dans le programme 9. Pour cela, la surcharge des fonctions equals avec certains types comme $TypeRange$ et $TypePointeur$ permet de définir ce que signifie deux instances équivalentes de ce type. Par exemple, deux instances de $TypeRange$ sont équivalentes si leurs attributs $first$ et $last$ le sont aussi.

— Si une variable déclarée est déjà présente dans l'environnement, une erreur doit être générée. Il faut donc analyser les variables globales mais aussi les variables locales (une variable globale et une variable locale doivent pouvoir avoir le même nom).

— Gestion du problème d'ambiguité "the dandling else" : une ambiguïté apparaissait dans le cas d'un **if a then if b then s else s2** (on peut faire soit **if a then (if b then s) else s2**, soit **if a then (if b then s else s2)**). Nous avons levé cette ambiguïté en modifiant légèrement la grammaire.

3.2 Environnement

Pour l'environnement des variables, nous utilisons une $ArrayList$ de $HashMap$ pour la pile de portée et non un $Stack$. En effet, le stack ne nous permettait pas d'itérer sur la pile efficacement de haut en bas.
Ainsi, pour chercher le type d'une certaine variable, on parcourt la pile de haut en bas et on retourne la première occurrence de cette variable.
Ici, une $HashMap$ représente une portée et la pile de $HashMap$ représente la priorité de ces portées.
Les variables globales sont donc stockées à la première couche de la pile.

3.3 Code intermédiaire

Les évolutions et problèmes rencontrés lors de l'écriture du code intermédiaire sont les suivants :

— Au lieu de retourner un $void$ comme c'était le cas au début, la fonction $generateIntermediateCode$ retourne une classe qui implémente $IntermediateCode$. Ainsi, chaque noeud génère son sous-arbre de code intermédiaire.

— $ExpList$ n'est plus une liste chaînée mais une $ArrayList$ pour simplifier le code.

— On utilise $ExpStm$ comme intermédiaire pour des lignes de code qui ne contiennent qu'une expression : par exemple, une ligne de code ne contenant qu'une expression (un println par exemple) ne peut pas être ajoutée directement à un Seq, on l'englobe donc dans un $ExpStm$ qui représente une expression dont la valeur est défaussée.

- On a ajouté une fonction $toDot$ pour afficher l'arbre de code intermédiaire : l'arbre généré représente principalement les instructions et non les expressions.
- On stocke les constantes $string$ statiquement en mémoire comme un tableau. Les constantes $string$ sont donc représentées par l'adresse mémoire du début du tableau.

Les problèmes non réglés sont les suivant :

— On ne génère pas de code intermédiaire pour $dispose$. En effet, nous pensons que l'utilisation de dispose dans le code intermédiaire est surtout intéressante dans le cas de l'utilisation du Y86, puisqu'elle représente un dépilement en Y86. Or nous n'avons pas eu le temps d'implémenter le Y86 et donc d'utiliser le code intermédiaire de $dispose$.

— Le code intermédiaire généré pour $NodeArrayAccess$ n'est pas complet (taille des éléments incorrecte).

3.4 Implémentation du Y86

Même si nous avons un bon aperçu de ce qu'il faut faire pour générer du Y86, nous n'avons pas travaillé dessus par manque de temps. Nous préférions aussi nous assurer d'avoir un code intermédiaire optimal et de bonne qualité plutôt que de commencer le Y86.

4 Autres

Dans la version actuelle de build.xml, on génère le code pour tous les programmes séquentiellement.

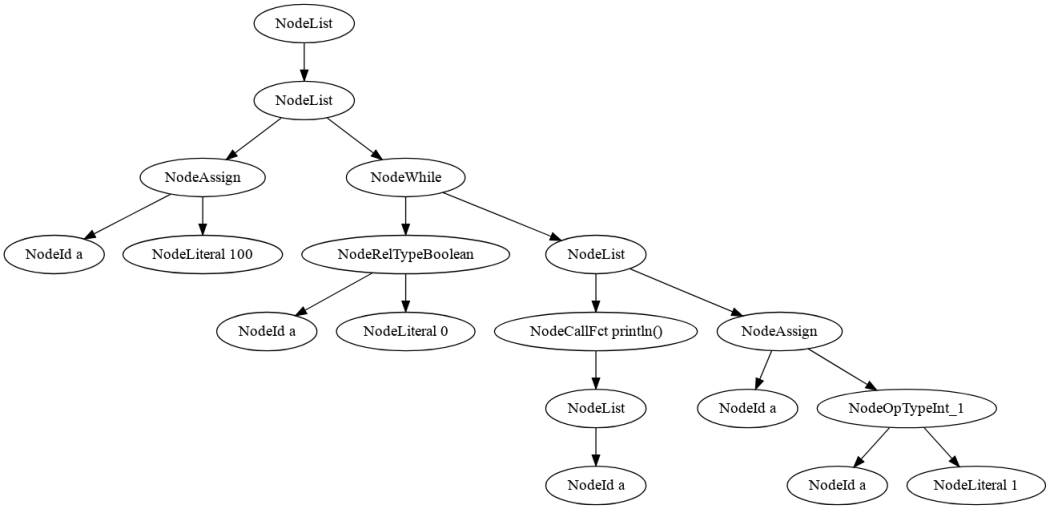

FIGURE 1 – Arbre syntaxique du programme 1

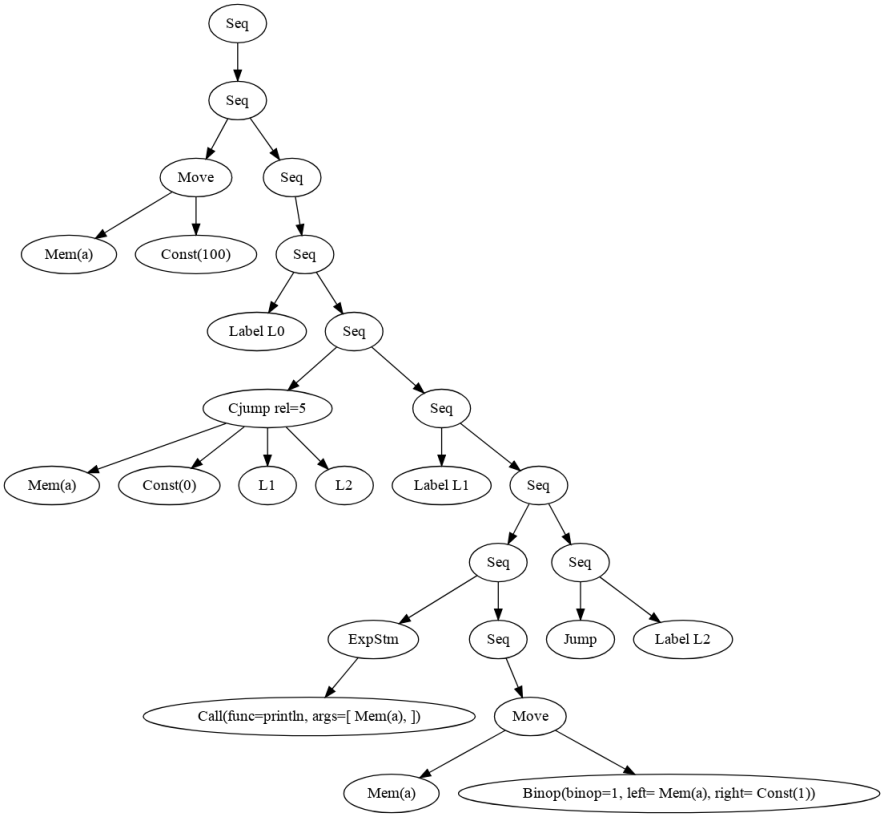

FIGURE 2 – Arbre de code intermédiaire du programme 1