Université de Lorraine
Telecom Nancy 2eannée
(2016-2017)

Rapport du projet Clooc

Conception d'un compilateur à langage orienté objet

Nicolas Dubois
Guillaume Garcia
Laure Hinsberger
Gauthier Zambaux

Date de rendu : 19 mai 2017
Module de Projet de Compilation des Langages

Table des matières

Introduction

Ce projet de développement d'un compilateur pour le langage orienté objet LOOC s'inscrit dans la continuité du module de Traduction des langages, celui-ci faisant partie de la formation initiale de deuxième année à TÉLÉCOM Nancy. Il a pour objectif de mettre en application les connaissances travaillées dans le cadre de ce module.

Le langage LOOC est un langage de programmation spécialement conçu pour ce projet dont la caractéristique principale est d'être orienté objet. Il est de plus un peu plus simple qu'un langage plus répandu comme le C++ puisque les fonctionnalités d'écriture en d'entrée de données y sont intégrées et qu'il ne comprend que deux types natifs de données : les entiers et les chaînes de caractères.

De manière à étaler le développement du compilateur et assurer un suivi du projet, celui-ci a été divisé en trois étapes évaluées indépendamment :
- l'écriture de la grammaire du langage en suivant la syntaxe ANTLR,
- l'implémentation des contrôles sémantiques principaux,
- la génération de code assembleur.

Nous avons donc utilisé l'outil ANTLR pour faciliter la mise en place des contrôles syntaxiques et la génération de l'arbre correspondant au programme à analyser. Dans sa dernière phase, le compilateur transforme l'arbre en code assembleur MicroPIUP. L'implémentation des contrôles sémantiques et la génération du code assembleur se fait en utilisant le langage Java. Nous avons pour ce projet développé des scripts *shell* s'occupant de générer l'archive *jar* exécutable correspondant au programme développé et à laquelle nous faisons référence plusieurs fois dans ce rapport.

Ce document présente d'abord la mise en place de la grammaire et des analyses lexicale et syntaxique puis décrit la création de la table des symboles et l'implémentation des contrôles sémantiques. Il finit par développer la génération du code assembleur. Les annexes présentent par ailleurs la grammaire du langage en syntaxe ANTLR et la contribution de chacun des membres au projet.

1 Grammaire, analyseurs lexical et syntaxique

1.1 Construction de la grammaire LOOC

La définition complète de notre grammaire est disponible dans l'**annexe A** de ce rapport.
Nous sommes partis de la définition de la grammaire donnée dans le sujet, à savoir :

```
PROGRAM          ⟶  CLASS_DECL* VAR_DECL* INSTRUCTION+
CLASS_DECL       ⟶  class Idf [ inherit Idf ] = ( CLASS_ITEM_DECL )
CLASS_ITEM_DECL  ⟶  VAR_DECL* METHOD_DECL*
VAR_DECL         ⟶  var idf : TYPE ;
TYPE             ⟶  Idf | int | string
METHOD_DECL      ⟶  method idf ( METHOD_ARGS* ) {  VAR_DECL* INSTRUCTION+ }
                    | method idf ( METHOD_ARGS* ) :  TYPE {  VAR_DECL* INSTRUCTION+ }
METHOD_ARGS      ⟶  idf : TYPE {, idf : TYPE}*
INSTRUCTION      ⟶  idf := EXPRESSION ;
                    | idf := nil ;
                    | if EXPRESSION then INSTRUCTION [else INSTRUCTION] fi
                    for idf in EXPRESSION .. EXPRESSION do INSTRUCTION+ end
                    | { VAR_DECL* INSTRUCTION+ }
                    | do EXPRESSION . idf ( EXPRESSION {, EXPRESSION}* );
                    | PRINT
                    | READ
                    | RETURN
PRINT            ⟶  write EXPRESSION ;
                    | write cste_chaîne ;
READ             ⟶  read idf ;
RETURN           ⟶  return ( EXPRESSION );
EXPRESSION       ⟶  idf
                    | this
                    | super
                    | cste_ent
                    | new Idf
                    | EXPRESSION . idf ( EXPRESSION {, EXPRESSION}* )
                    | ( EXPRESSION )
                    | - EXPRESSION
                    | EXPRESSION OPER EXPRESSION
OPER             ⟶  + | - | * | < | <= | > | >= | == | !=
```

FIGURE 1 – Grammaire initiale

Cette grammaire produit une règle qui est récursive gauche : **expression**. Cela rend la grammaire non LL(1). Or c'est nécessaire pour pouvoir générer par la suite les Lexer et Parser (analyseurs lexical et syntaxique) avec ANTLR. Il a donc fallu dérecursiver la règle expression en la décomposant en deux sous règles (voir **annexe A**). Nous avons également remanié quelques règles afin de nous faciliter la construction de l'Arbre Syntaxique Abstrait (AST) et les contrôles sémantiques. Nous avons également autorisé syntaxiquement la déclaration de blocs anonymes et classes **vides**. Nous

avons également rendu possible le fait de déclarer des variables/méthodes même après des affectations (la structure de base était : |déclarations* | puis instructions/affectations* |, que nous avons rendu plus souple). Notre grammaire ignore bien les commentaires commençant par /* et se terminant par */. Nous forçons la grammaire à être LL(1) dans le bloc *options*.

Un autre problème de taille fut de gérer les priorités des opérateurs. En effet dans la grammaire de la **figure 1**, tous les opérateurs sont au même niveau, c'est-à-dire qu'ils ont tous la même priorité. Dans la **figure 2**, vous pouvez voir comment nous avons ordonné les sous règles de *expression* pour donner des priorités aux opérateurs. Vous pouvez également observer comment *expression* a été dérécursivée. (Il manque la règle *expressionbis*, que vous pouvez voir en annexe A).

```
expression:   'this' expressionbis
          |   'super' expressionbis
          |   STRING_CST expressionbis
          |   'new' IDFC expressionbis
          |   exprio1 expressionbis
          ;

exprio1 : exprio2 ( '+'^ exprio2 | '-'^ exprio2)* ;

exprio2 : exprio4 ( '*'^ exprio4)* ;

exprio4 : exprio7 ( '=='^ exprio7 | '!='^ exprio7 | '<'^ exprio7
          | '<='^ exprio7 | '>'^ exprio7 | '>='^ exprio7)* ;

exprio7 : ('-'^)? exprio8 ;

exprio8 : INT_CST
        | IDF
        | '(' expression ')' -> expression
        ;
```

FIGURE 2 – Opérateurs priorisés

Les priorités sont donc les suivantes, du moins prioritaire au plus prioritaire :

- Opérateurs binaires + et **-**, addition soustraction mais également OR et NOT logiques dans les expressions booléennes.
- Opérateur binaire *****, multiplication mais aussi AND logique dans les expressions booléennes.
- Opérateurs logiques binaires de comparaison : ==, >, <, >=, <=, **!**=.

— Opérateur unaire `-`, négation d'un entier, mais aussi le NOT logique dans les expressions booléennes.
— Les parenthèses, pas un "opérateur" à proprement parler mais permet de faire des expressions parenthésées.

Il est à noter que l'opérateur de division / n'était pas présent dans la grammaire fournie dans le sujet, et nous avons décidé de ne pas le rajouter, pour ne pas compliquer notre tâche. De plus, cette opérateur n'était pas absolument nécessaire pour comprendre comment gérer les opérateurs et leurs priorités tout au long de la conception du compilateur.

1.2 Génération des analyseurs lexical et syntaxique avec ANTLR

Les premières étapes de la compilation d'un langage consiste en l'analyse lexicale puis syntaxique du programme. Pour cela il nous faut deux analyseurs. ANTLR 3.3 est un logiciel écrit en *Java* permettant de générer automatiquement ces deux analyseurs directement à partir d'une grammaire (fichier.g), en respectant une syntaxe particulière pour la grammaire, notamment la notation légère de la BNF (Backus Naur Form), pour les expressions régulières. Pour cela, il suffit simplement de lancer la commande suivante en utilisant le jar d'ANTLR :

```
java -jar antlr-3.3-complete.jar Looc.g
```

Cette commande va générer les LoocParser.java et LoocLexer.java qui sont les analyseurs lexical et syntaxique en code source. Il faut ensuite compiler le tout (les analyseurs et le compilateur) avec la commande :

```
javac *.java
```

Ceci générera les .class pour lancer votre compilateur comprenant les 2 analyseurs.
Note : il faut avoir placé au préalable le path jusque le JAR d'ANTLR dans la variable d'environnement CLASSPATH :

```
export CLASSPATH=<path/to/ANTLR/jar>:.:$CLASSPATH
```

En fait, l'analyse lexical et l'analyse syntaxique ont lieu en même temps. Pour lancer le compilateur et donc les deux analyses préliminaires, il suffit de lancer par la suite la commande :

```
java Clooc
```

Et d'en observer la sortie. Si aucun message n'est produit, c'est qu'il n'y aucune erreur lexicale ni syntaxique (ni sémantique puisque dans l'état actuel notre compilateur possède un analyseur sémantique).

Si vous ne souhaitez qu'effectuer les étapes d'analyse lexical et syntaxique avec notre exécutable *clooc.jar*, veuillez utiliser l'option **-a** ("analysis") :

```
java -jar clooc.jar -a fichier.looc
```

1.3 Conception de l'Arbre Syntaxique Abstrait

L'arbre abstrait (dit aussi AST) est une étape essentielle de la compilation : il est nécessaire à l'analyse sémantique et à la génération de code. Dans les compilateurs classiques il est généré habituellement à partir de l'analyse syntaxique. Ici, c'est ANTLR qui gère les analyses lexicale et syntaxique, et la génération de l'AST est effectuée en même temps que les deux analyses (qui sont simultanées rappelons-le).

La conception de l'arbre abstrait s'est faite en deux étapes : Les réécritures des règles dans la grammaire *Looc.g* pour ANTLR et créer un petit générateur d'arbre au format *DOT* à l'aide des bibliothèques ANTLR et JAVA pour afficher l'arbre si on le souhaite. L'ensemble des réécritures de règles est disponible dans la grammaire Looc.g dans l'**annexe A**.

Le compilateur, sous l'option **-T**, convertit l'AST, qui est de type CommonTree par défaut, en DOTTree, c'est à dire au format dot. Il écrit cet arbre en DOT sous un fichier *.dot*, puis fait appel au programme externe **dot** de la suite *graphviz* pour le convertir en image au format *PNG*. Enfin, il affiche cette image avec le programme externe **eog** (Eye of GNOME Image Viewer).

Il est possible de réécrire des règles dans ANTLR pour "personnaliser" la construction de l'AST. On peut rajouter des noeuds imaginaires qui serviront à l'analyse sémantique, au remplissage de la TDS mais également à la génération de code. Les réécritures permettent aussi d'éliminer des token inutiles pour l'AST comme des signes de ponctuation (point-virgule, parenthèse). Voici un exemple de réécriture de règle :

```
var_decl:   'var' IDF ':' type ';' -> ^(VARDEC IDF type);
```

Cette règle reconnaît les déclarations de variables en LOOC : le mot clé "var" suivi du nom de la variable (produit par la règle IDF : ...), un double-point, le type, et un point-virgule. Les deux signes de ponctuation sont inutiles pour l'AST, et nous souhaitons changer le token "var" par "VARDEC" donc notre AST. Nous réécrivons donc la règle avec le signe "->" suivie d'une expression de la forme `^(ROOT CHILD1 CHILD2 ...)` où ROOT, le noeud courant de l'arbre, est "VARDEC" et les enfants dont le nom de la variable et son type.

Voici donc un exemple d'arbre abstrait en **figure 3**, obtenu à partir de notre programme de test *progtestlvl2.looc*, le 2e exemple fourni dans le sujet.

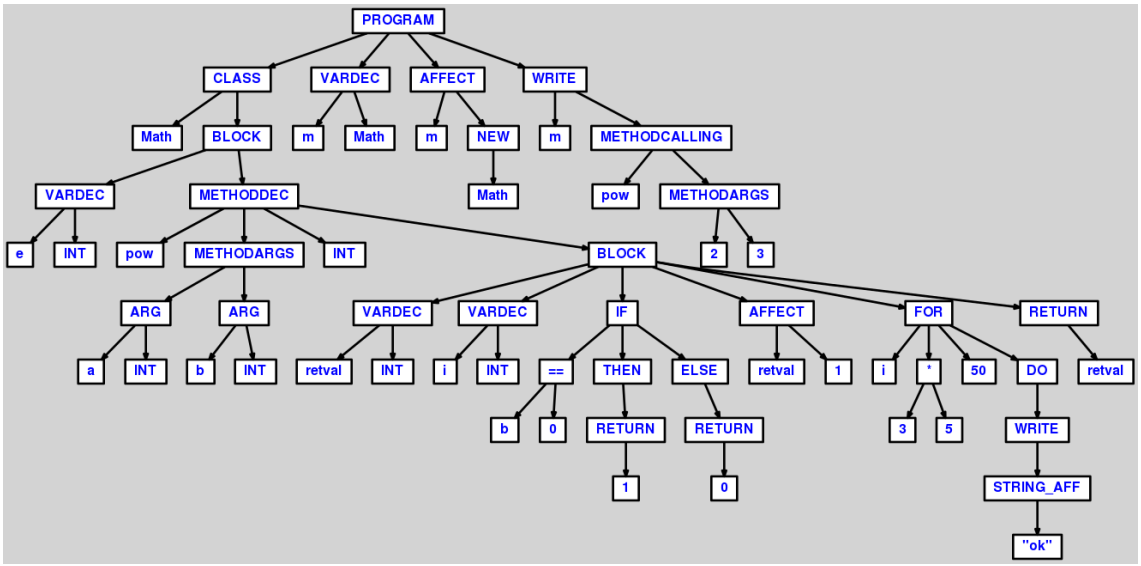

FIGURE 3 – AST de progtestlvl2.looc

Si vous souhaitez construire l'AST en format *DOT* puis le convertir en image *PNG* et l'afficher, utiliser l'option **-T** du compilateur ("Tree"). Cette option ne peut pas être combinée avec les autres et n'effectue que les étapes d'analyses syntaxique et lexicale en même temps d'afficher l'arbre. La commande :

```
java -jar clooc.jar -T prog.looc
```

produira un fichier *prog.ast.dot* (l'arbre en *DOT*) et un fichier *prog.ast.png* (l'image de l'AST en *PNG*) dans le dossier courant, et affichera l'AST.

2 Table des Symboles, analyse sémantique

2.1 Structure de la Table Des Symboles

La construction de la table des symboles (aussi appelée TDS), est une étape primordiale de la compilation d'un programme. Dans le compilateur Clooc, elle survient après l'étape des analyses lexicale et syntaxique. Elle est construite en même temps que l'analyse sémantique a lieu. Elle sert pour l'analyse sémantique mais aussi pour la génération de code en conservant le déplacement des variables. Chaque entrée dans la TDS correspond à une déclaration d'objet (variable, méthode, classe) dans le code. Chaque classe, méthode, bloc anonyme définissent un nouvel sous espace de noms et donc une nouvelle TDS.

La conception de la structure générale de la TDS fut longue et laborieuse, mais nous avons fini par créer une table robuste. Nous avons utilisé **3** types de structure de données différents pour la construire.

- La première, fut les **HashMap<String, LinkedList>**, qui constitue de façon brut les TDS et sous-TDS, chacune représentant un espace de nom.
- La seconde, fut les **LinkedList**, contenues dans les HashMap, représentant diverses informations sur les objets représentées par la **String** clé pointant la LinkedList. La structure générique de ces listes chainées est détaillée plus loin.
- Enfin, une structure conçue par nos soin qui représente un Arbre, que nous avons appelée **NodeTDS**. Cette objet représente un noeud de l'arbre ; ce noeud a un ID, contient une et une seule TDS (HashMap), et possède **1** (espace de nom parent) ou **2** (dans le cas d'une classe fille) parent(s) (seul le Node Root n'a pas de parent, c'est à cela qu'un le reconnaît), et peut avoir 0 ou + enfants (sous-espace de noms). L'ensemble des primitives de **NodeTDS** est consultable dans le fichier *analyseurs/NodeTDS.java* du git.

Dans la HashMap<String, LinkedList>, la String représente le nom de la variable / symbole / classe / etc ... et la LinkedList représente certaines informations à son sujet. La structure générique de ces informations est : `[Type d'entité, info1, info2, ...]`. Si bien que l'élément à l'index 0 sera toujours le type d'entité (Class, Variable, Method ...).

- Pour une classe ce sera : `["CLASS", MOTHERCLASS]`. Si la classe n'a pas de classe mère, l'élément en index 1 sera initialisé à *null*, sinon au

nom de sa classe mère.
- Pour une variable, ce sera : `["VAR", TYPE, VALUE]`. L'élément TYPE vaut soit "INT" ou "STRING", les deux types natifs du LOOC, ou alors il vaut le nom d'une classe définie auparavant (type objet).
- Pour une méthode ce sera : `["METHOD", RETURN_TYPE, ARGS]`. Si la méthode n'a pas de type de retour, RETURN_TYPE vaut "void" sinon son type de retour. ARGS est une LinkedList contenant le **type** de chaque argument, dans l'ordre d'écriture. Si la méthode n'a pas d'argument, cette liste est initialisée à *null*.

L'option **-v** du compilateur ("verbose") effectue les analyses lexicale et syntaxique, puis l'analyse sémantique et la construction de la TDS. Il affiche la composition de toutes les TDS construites (mais pas dépilées, par soucis de visualisation). Il annonce aussi les différentes étapes de la compilation.
La commande :

```
java -jar clooc.jar -v progtestlvl2.looc
```

va donc produire l'output suivant (seul l'affichage de la TDS est conservé ici) :

```
============================ TDS : root ==============================
Idf : Math || Type d'entité : CLASS || Herite de :  ||
Idf : m || Type d'entité : VAR || Type : Math || Valeur : null ||
======================================================================

============================ TDS : Math ==============================
Idf : e || Type d'entité : VAR || Type : INT || Valeur : null ||
Idf : pow || Type d'entité : METHOD || Type de retour : INT || Args : [INT, INT] ||
======================================================================

============================ TDS : pow ===============================
Idf : a || Type d'entité : ARG || Type : INT ||
Idf : b || Type d'entité : ARG || Type : INT ||
Idf : i || Type d'entité : VAR || Type : INT || Valeur : null ||
Idf : retval || Type d'entité : VAR || Type : INT || Valeur : null ||
======================================================================
```

2.2 Définition de notre analyseur sémantique

Notre analyseur sémantique, et également constructeur de TDS (à partir du parcours de l'AST), est défini comme un objet de type **TreeParser** de **TreeParser.java** contenant plus de 1300 lignes de code. Par soucis de clarté nous avons donc décidé d'implanter une interface **ITreeParser.java** permettant de visualiser facilement toutes les primitives de notre analyseur.

Nous avons mis en place un système d'exceptions Java pour "tester" la sémantique du code. L'analyseur parcourt l'AST et teste chaque déclaration, initialisation, expression ... Vous pourrez voir la liste détaillée des erreurs testées dans la partie suivante (**2.3**). Voici les exceptions créées pour le compilateur et leur utilisation :

- **MismatchTypeException** : Problème de concordance de type dans une expression calculatoire, ou à l'appel d'une fonction pour son type de retour ou le type de ses arguments.
- **NoSuchIdfException** : Le symbole utilisé n'a pas été défini auparavant dans la TDS ou ses TDS parentes.
- **NoSuchNodeException** : L'analyseur tente de faire référence à une TDS qui n'existe pas (une classe mère fictive par exemple).
- **NotInitializedVariableException** : Une expression tente d'utiliser une variable qui n'a pas été initialisée. Sert à certains warnings et erreurs.
- **NoPureStatementException** : L'expression n'est pas un INT pur ou une STRING pure, sert au MismatchType, mais aussi pour faciliter le calcul et la reconnaissance de certaines expressions.

Nous avons également implémenté certains contrôles sémantiques de type "Warning". Ces warnings sont détaillés en **2.3**. Ceux-ci ne sont pas affichés par défaut, il faut passer l'option **-W** au compilateur. La commande :

```
java -jar clooc.jar -W warning.looc
```

va sortir l'output de l'analyse sémantique (les erreurs) en tenant compte des warnings :

```
ligne 1 : Warning : la classe Empty est vide.
ligne 15 : Erreur : Une variable n'est ni un entier ni nil.
ligne 23 : Erreur : La borne inférieure de la boucle for n'est pas un entier.
ligne 23 : Erreur : La borne supérieure de la boucle for n'est pas un entier.
ligne 35 : Erreur : L'argument de write n'est pas un entier ou une chaîne de caractères
4 erreur(s) dans le fichier prog_test/progtestlvl2.looc.
```

On remarque que le compilateur affiche la ligne de l'erreur/warning, son type (si c'est un warning ou une erreur), et une courte explication du problème. A la fin de l'analyse sémantique, le compilateur affiche le nombre d'erreurs (seules) détectées. Si aucune erreur est détectée, rien n'est affiché et la traduction en assembleur peut commencer. Il est à noter que le compilateur effectue l'analyse sémantique de TOUT l'arbre, et ne s'arrête pas dès qu'il a trouvé une erreur.

2.3 Liste des contrôles sémantiques (erreurs) implémentés

Le tableau ci-dessous présente la liste des contrôles sémantiques implémentés par chacun.

Nicolas Dubois	- (Erreur) Vérifie le type des arguments de write - (Erreur) Vérifie le type des arguments de read - (Erreur) Vérifie la cohérence des types sur un return - (Warning) Vérifie que les variables définies sont utilisées
Guillaume Garcia	- (Warning) Variable non initialisée alors qu'elle est accédée. - (Erreur) Vérifie l'existence d'une variable : référence indéfinie. (Variable inexistante à l'accès dans les TDS parentes). - (Erreur) Vérifie qu'une classe n'hérite pas d'elle-même. - (Erreur) Vérifie qu'une classe parente existe lors d'un héritage. - (Warning) Vérifie qu'une classe déclarée n'est pas vide. - (Erreur) Vérifie que les bornes et l'indice d'une boucle FOR sont des entiers. - (Erreur) Erreur de concordance des types dans une expression arithmétique - (Erreur) Vérifie la déclaration préalable de l'indice d'une boucle FOR, et des éventuelles variables utilisées dans les expressions des bornes - (Warning) Vérifie qu'un bloc anonyme déclaré n'est pas vide. - (Erreur) Vérifie qu'une classe instanciée est définie.
Laure Hinsberger	- (Erreur) 'this' et 'super' doivent être appelés dans des classes et pas ailleurs ; 'super' doit être appelé dans une classe fille. - (Erreur) Dans une classe, un symbole ne peut désigner à la fois un argument ou un attribut, et une méthode. Les arguments d'une méthode sont différents. - (Erreur) Pas de redéclaration d'une variable globale ou locale. - (Erreur) Pas de redéclaration d'une méthode dans le même contexte (même TDS). - (Warning) Vérifie qu'une méthode déclarée n'est pas vide. - (Erreur) Contrôle de type dans les conditions (entier ou NIL), et les variables utilisées sont déclarées. - (Erreur) Cohérence des types sur des affectations.
Gauthier Zambaux	- (Erreur) Vérifie qu'il existe au moins un 'return' dans une méthode typée, et/ou qu'il est toujours possible de retourner dans une méthode - (Erreur) Vérifie que s'il existe un ou plusieurs 'return' dans une fonction non-typée, ceux-ci ne possèdent pas de membre droit (pas de valeur retournée). - (Erreur) Pas de redéclaration d'une classe. - (Erreur) Vérifie qu'une méthode appelée existe dans l'arborescence des TDS parentes. (implicit declaration of function) - (Erreur) Vérifie le nombre d'arguments passés à une méthode.

TABLE 1 – Contrôles sémantiques implémentés par chacun

3 Traduction en assembleur, tests unitaires, améliorations

3.1 Rappels sur l'assembleur microPIUP/ASM

Afin de pouvoir mettre en oeuvre cette partie du projet, nous avons eu recours au cours du module PFSI de première année. Les PDF de Monsieur Parodi récupérés lors de l'élaboration de cette étape nous ont permis de nous apporter des rappels sur l'assembleur. De plus, nous avons eu la possibilité de lui poser des questions plus précises auxquelles il a généreusement pris le temps de répondre. Ses connaissances transmises nous ont apporté ce dont nous avions besoin pour réaliser la traduction en assembleur.

3.2 Schémas de traduction

La phase de traduction de code est la toute dernière phase de la compilation. Pour Clooc, elle survient juste après l'analyse sémantique et la construction de la TDS. Si l'analyse sémantique a révélé **1 ou plus** erreur, alors la traduction de code n'a pas lieu et le compilateur termine.

Sans option, notre compilateur effectue (si l'analyse sémantique s'est bien passée) la traduction du code en assembleur microPIUP/ASM (*.src* ou *.asm*) puis appelle le programme externe **java -jar** pour compiler ce fichier au format *.iup* directement exécutable par microPIUP4.jar. Si vous ne souhaitez que la version source (*.asm*), il faut utiliser l'option **-s** du compilateur :

```
java -jar clooc.jar -s prog.looc
```

Les programmes assembleurs générés par Clooc commencent tous pas un ensemble de définition de variables. Notamment, de manière à rendre le code plus lisible en évitant d'utiliser directement les numéros de trappes dans le programme, on choisit de les définir comme des variables. De même, les registres indiquant la pile, le registre de travail et le pointeur de base sont définis pour être utilisables plus facilement qu'avec les registres de base R13, R14 et R15.

Les fonctions *write* et *read* sont définies au début de tous les programmes assembleurs générés par le compilateur, qu'elles soient ou non utilisées dans le programme à compiler. Lorsque write est appelée, la chaîne correspondant à ce qui doit être affiché est stockée dans une variable de type *string*, qui est

elle même chargée dans le registre R1. La chaîne est alors affichée en utilisant l'interruption (la trappe) 66. Chaque appel de la fonction write termine par un retour à la ligne et est donc équivalent à l'utilisation de *println()* en Java. Lorsque la fonction read est appelée, le registre R1 est lié à la variable qui doit stocker le résultat. L'interruption 65 est alors utilisée pour récupérer l'entrée clavier.

3.3 Jeux d'essais et limites de Clooc

Nous nous sommes efforcé, au cours de ce projet, de mettre au point des programmes exemples pour tester toutes les fonctionnalités que nous développions. Notamment, le programme en annexe C rassemble les erreurs sémantiques que nous avons implémentées et la figure 4 est le résultat de son exécution.

```
ligne 12 : Erreur : Deux arguments ont le même nom dans la méthode myMeth2.
ligne 13 : Erreur : Référence indéfinie vers la variable testnop
ligne 12 : Erreur : La méthode myMeth2 est de type void, elle n'est pas censée retourner quoique ce soit.
ligne 17 : Erreur : Le nom de méthode myMeth3 redéfinit un symbole
ligne 18 : Erreur : super n'est pas appelé dans une classe fille.
ligne 18 : Erreur : Une variable dans l'expression du membre droit de l'affectation n'est pas déclarée.
ligne 30 : Erreur : La méthode myMeth6 prend 0 arguments (1 donné(s)).
ligne 40 : Erreur : une classe ne peut pas heriter d'elle-même : Animal inherit Animal
ligne 40 : Erreur : référence indéfinie à la classe mère Animal dans la déclaration de la classe Animal
ligne 49 : Erreur : Une variable dans l'expression du membre droit de l'affectation n'est pas déclarée.
ligne 52 : Erreur : Le nom de méthode myMethHerit redéfinit un symbole
ligne 57 : Erreur : redéfinition de la classe MyClass
ligne 67 : Erreur : Une variable dans l'expression du membre droit de l'affectation n'est pas déclarée.
ligne 69 : Erreur : this n'est pas utilisé dans une classe.
ligne 69 : Erreur : Une variable dans l'expression du membre droit de l'affectation n'est pas déclarée.
ligne 70 : Erreur : super n'est pas utilisé dans une classe.
ligne 70 : Erreur : Une variable dans l'expression du membre droit de l'affectation n'est pas déclarée.
ligne 72 : Erreur : La méthode getNb n'est pas définie.
ligne 77 : Erreur : Le nom de variable redéfinit une méthode ou une variable.
ligne 84 : Erreur : L'indice ind de la boucle for n'est pas un entier.
ligne 93 : Erreur : Référence indéfinie vers la variable i (indice de la boucle for).
ligne 100 : Erreur : Référence indéfinie vers la variable test
22 erreur(s) dans le fichier prog_test/sem_test.looc.
```

FIGURE 4 – Exécution de l'exemple de l'annexe C

Pour ce qui est de la traduction en assembleur, au stade où nous écrivons ce rapport, les commandes que nous avons implémentées sont présentes sur le fichier exemple de l'annexe D et l'annexe E présente le résultat de l'exécution.

3.4 Fonctionnalités et améliorations apportées

Comme complément de ce qui était demandé, nous avons apporté quelques améliorations au projet. Nous avons notamment introduit un ensemble d'options possibles pour l'exécution du *jar* du compilateur. Par exemple, l'option

$-d$ permet de spécifier un dossier de destination pour le programme assembleur généré à l'issue de l'exécution du programme.

```
java -jar clooc.jar -d exemple.looc
```

Nous avons aussi développé un script pour *atom* mettant en forme un programme LOOC. Celui ci peut être trouvé aux liens suivants :

— https ://atom.io/packages/language-looc
— https ://github.com/AOSauron/language-looc

Nous avons par ailleurs au cours du projet utilisé un script shell pour compiler rapidement le code java. Il est sur le dépôt de la Forge de l'école.

Conclusion

À l'issue de ce projet de compilation des langages, nos connaissances des chaînes de compilation de codes se sont nettement amélioré et nous avons maintenant une vision plus large du fonctionnement du processus. Ce projet ayant duré plusieurs mois, nous avons eu le temps de prendre complètement en main le sujet.

Tout au long du projet, nous avons dû faire face à des difficultés liées au sujet même mais également aux dates limites imposées régulièrement. Ce sont ces difficultés qui nous ont réellement permis de nous améliorer et d'aller chercher plus loin les solutions aux problèmes. La résolution des difficultés ont parfois été très chronophages, mais ont généralement abouti à des solutions mettant en oeuvre les différentes ressources de connaissances à disposition (cours, conseils de professeurs, ressources en ligne, etc).

Malgré un projet qui répond à de nombreuses exigences du sujet, certaines améliorations au code auraient été possibles, comme par exemple l'ajout de contrôles sémantiques que nous n'avons pas poursuivi par faute de temps et par priorité des autres tâches.

Finalement, le projet rendu fait fonctionner les contrôles sémantiques implémentés et génère le code efficacement.

Annexes

A Grammaire LOOC

```
/*  GRAMMAIRE POUR LE LANGAGE LOOC POUR PROJET DE COMPILATION TELECOM NANCY 2016-2017
Authors :

DUBOIS Nicolas
GARCIA Guillaume
HINSBERGER Laure
ZAMBAUX Gauthier

Maj : 09/03/17   14:55
*/

 grammar Looc;

 options {
   k=1;  /* Pour forcer LL(?) => LL(1) */
   output=AST;
   ASTLabelType=CommonTree;
 }

/* Tokens imaginaires pour l'AST */
 tokens {
   PROGRAM;
   CLASS;
   VARDEC;
   METHODDEC;
   METHODARGS;
   AFFECT;
   IF;
   FOR;
   ANONYMOUSBLOCK;
   WRITE;
   READ;
   RETURN;
   EXPR;
   NEW;
   METHODCALLING;
   THIS;
   THEN;
   SUPER;
   INT;
   STRING;
   ARG;
   BLOCK;
   DO;
   ELSE;
   CALC;
   STRING_AFF;
   VAR;
   CSTE_INT;
}
```

```
/* Template:
regle: (^regex+ | regex?)*
       -> ^(Reecriture eventuelle en arbre) ;
*/

program:   (class_decl)* decl_ins*
           -> ^(PROGRAM (class_decl)* decl_ins*);

class_decl:   'class' IDFC ('inherit' IDFC)? '=' '(' class_item_decl ')'
              -> ^(CLASS IDFC (IDFC)? class_item_decl);

class_item_decl:   decl*
                   -> ^(BLOCK decl*);

decl:   var_decl
    |   method_decl
    ;

decl_ins:   var_decl
        |   instruction
        ;

var_decl:   'var' IDF ':' type ';'
            -> ^(VARDEC IDF type);

type:   IDFC
        -> ^(IDFC)
    |   'int'
        -> ^(INT)
    |   'string'
        -> ^(STRING)
    ;

method_decl:   'method' IDF '(' method_args? ')' (':' type)? '{' decl_ins* '}'
               -> ^(METHODDEC IDF method_args? type? ^(BLOCK decl_ins*));
/* On autorise syntaxiquement la déclaration de méthode vide, mais un warning (sémantique) sera renvoyé */

method_args:   IDF ':' type (',' IDF ':' type)*
               -> ^(METHODARGS ^(ARG IDF type) ^(ARG IDF type)*);

instruction:   IDF ':=' affectation ';'
               -> ^(AFFECT IDF affectation)
           |   'if' expression 'then' a+=instruction+ ('else' b+=instruction+)? 'fi'
               -> ^(IF expression ^(THEN $a+) ^(ELSE $b+)?)
           |   'for' IDF 'in' expression '..' expression 'do' instruction+ 'end'
               -> ^(FOR IDF expression expression ^(DO instruction+))
           |   '{' decl_ins* '}'
                -> ^(ANONYMOUSBLOCK decl_ins*)
/* On autorise syntaxiquement la déclaration de blocs vides, mais un warning (sémantique) sera renvoyé */
           |   'do' expression ';'
               -> ^( DO  expression )
           |   print
           |   read
           |   returnstate
           ;

affectation:   expression
           |   'nil'
           ;

print:   'write' expression ';'
```

```
         -> ^(WRITE expression);

read:   'read' IDF ';'
        -> ^(READ IDF);

returnstate:   'return' '(' expression? ')' ';'
               -> ^(RETURN expression?);

expression:   'this' expressionbis
              -> ^(THIS expressionbis?)
          |   'super' expressionbis
              -> ^(SUPER expressionbis?)
          |   STRING_CST expressionbis
              -> ^(STRING_AFF STRING_CST expressionbis?)
          |   'new' IDFC expressionbis
              -> ^(NEW IDFC)
          |   exprio1 expressionbis
              -> exprio1 expressionbis?
          ;

exprio1 : exprio2 ( '+'^ exprio2 | '-'^ exprio2)* ;

exprio2 : exprio4 ( '*'^ exprio4)* ;

exprio4 : exprio7 ( '=='^ exprio7 | '!='^ exprio7 | '<'^ exprio7 | '<='^ exprio7 | '>'^ exprio7 | '>='^ exprio7)* ;

exprio7 : ('-'^)? exprio8 ;

exprio8 : INT_CST
          -> ^(INT_CST)
        | IDF
          -> ^(IDF)
        | '(' expression ')' -> expression
        ;

expressionbis:   '.'  IDF '(' (expression)? (',' expression)* ')' expressionbis
                 -> ^(METHODCALLING IDF ^( METHODARGS (expression)*)? (expressionbis)?)
             |   /*Le mot vide*/
             ;

IDFC:   ('A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;

IDF:   ('a'..'z') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;

INT_CST:   '0'..'9'+ ;

STRING_CST:   '"' (ESC_SEQ | ~('\r'|'\n'|'"'|'\\'))+ '"' ;

WS  :   (' '|'\t'|'\n'|'\r')+ {$channel=HIDDEN;} ;

COMMENT:   '/*' (options {greedy=false;}:.)* '*/' {$channel=HIDDEN;} ;

fragment ESC_SEQ:   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
                |   UNICODE_ESC
                |   OCTAL_ESC
                ;

fragment OCTAL_ESC:   '\\' ('0'..'3')('0'..'7')('0'..'7')
                  |   '\\' ('0'..'7')('0'..'7')
                  |   '\\' ('0'..'7')
                  ;
```

```
fragment UNICODE_ESC:   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT ;

fragment HEX_DIGIT:   ('0'..'9'|'a'..'f'|'A'..'F') ;
```

B Contributions des membres

	Création de la grammaire	**Analyse sémantique**	**Génération de code assembleur**
Nicolas Dubois	traduction de la grammaire donnée suivant la syntaxe ANTLR **10h**	contrôles sémantiques **8h**	
Guillaume Garcia	traduction de la grammaire donnée suivant la syntaxe ANTLR **20h**	contrôles sémantiques, TDS **45h**	squelette du code java, traduction de code **30h**
Laure Hinsberger	mise en place de la priorité des opérateurs **10h**	contrôles sémantiques, TDS **10h**	traduction de code **10h**
Gauthier Zambaux	réécriture des règles de la grammaire ANTLR pour générer l'arbre **16h**	contrôles sémantiques, TDS **30h**	traduction de code **30h**

Suite à son abandon de la formation, Nicolas Dubois n'a pas participé à la dernière partie du projet.

C Programme de test des contrôles sémantiques

```
class MyClass = (

  var test : int;
  var myMeth3 : int;

  method myMeth() {
    var testnop : int;
    test := 3;
    return();
  }

  method myMeth2(a : int, a : int) {
    testnop := 4;
    return(9);
  }

  method myMeth3(a : int, b:int) {
    test := super.getNb();
    return();
  }

  method myMeth6() : MyClass {
    var a : MyClass;
    a := new MyClass;
    return(a);
  }

  method myMeth4 (a : int){
    do this.myMeth6().myMeth3(a,b);
    do this.myMeth6(8).myMeth3(a,9);
  }

  method retourneRien() : int {
    var a : int;
  }

)


class Animal inherit Animal = (

)


class MyClassic inherit MyClass =
(
  method myMethHerit() {
    test := 4;
    test := super.getNb();
  }

  method myMethHerit() {

  }
)

class MyClass = (

)
```

```
{
/*bloc vide*/
}

var variable : MyClass;

variable := new Classe;

variable := this.getNb();
variable := super.getNb();

do variable.getNb();

variable := "gdfgdf";
variable := (32+1)*variable;

var variable : int;
var ent : int;
ent := 4;
var entier : int;
entier := 1;
var ind : string;

for ind in ent..entier do
  variable := new MyClass;
  {
    var mytest : int;
    mytest := 4;
    write mytest;
  }
end

for i in ent..entier do
  variable := new MyClass;
end

write "Salut";
write variable;

test := 3;

read variable;
```

D Programme de test de traduction en assembleur microPIUP

```
var entier : int;
var sum : int;

entier := 1;
sum := 7;

sum := sum + entier;

write "Testing...";
read sum;
write sum;
write "Terminated";
```

E Résultat de l'exécution du compilateur avec le programme de l'annexe D

```
EXIT_EXC EQU 64
READ_EXC EQU 65
WRITE_EXC EQU 66
LOAD_ADRS EQU 0xF000
NIL EQU 0
STACK_ADRS EQU 0x1000

SP EQU R15
WR EQU R14
BP EQU R13
INT_SIZE EQU 2

ORG LOAD_ADRS
START main_

write_        LDQ 0, R1
        ADQ -2, SP
        STW BP, (SP)
        LDW BP, SP
        SUB SP, R1, SP

        LDW R0, BP
        ADQ 4, R0
        LDW R0, (R0)

        LDW WR, #WRITE_EXC
        TRP WR

        LDW R0, #NEWLINE
        LDW WR, #WRITE_EXC
        TRP WR

        LDW SP, BP
        LDW BP, (SP)
        ADQ 2, SP
        LDW WR, (SP)
        ADQ 2, SP
        JEA (WR)

read_        LDQ 0, R1
        ADQ -2, SP
        STW BP, (SP)
        LDW BP, SP
        SUB SP, R1, SP

        LDW R0, BP
        ADQ 4, R0
        LDW R0, (R0)

        LDW WR, #READ_EXC
        TRP WR

        LDW SP, BP
        LDW BP, (SP)
        ADQ 2, SP
        LDW WR, (SP)
        ADQ 2, SP
```

```
        JEA (WR)

main_        LDW SP, #STACK_ADRS
        LDQ NIL, BP
        LDQ 0, R1
        ADQ -2, SP
         STW BP, (SP)
         LDW BP, SP
        SUB SP, R1, SP

NEWLINE RSW 1
        LDW R3, #0x0a00
        STW R3, @NEWLINE

        ADQ -INT_SIZE, SP
        ADQ -INT_SIZE, SP
STRING0 string "Testing..."
        LDW R1, #STRING0
        STW R1, -(SP)
        JSR @write_

READINT0 RSB 4
        LDW R1, #READINT0
        STW R1, -(SP)
        JSR @read_

STRING1 string "Terminated"
        LDW R1, #STRING1
        STW R1, -(SP)
        JSR @write_

        LDW WR, #EXIT_EXC
        TRP WR
```