Rapport de projet Compilation

Pascal Ferraris, Thibaut Lausecker, Colin Luciani,
Apolline Masson, Marie Peter, Paul-Louis Renard

7 juin 2019

Table des matières

1 Introduction

Lors de notre formation en deuxième année en Informatique à l'ENSSAT, pendant le module de Théorie du Langage et Compilation, nous avons été amenés à étudier le fonctionnement d'un compilateur simple.

Le but de ce projet de compilation est de produire un compilateur pour le langage *NilNoviObjet*. Pour cela, nous disposions d'un interpréteur et d'un désassembleur pour ce langage. Nous sommes partis d'un projet comportant un analyseur lexical, la tables des identificateurs (TDI), et une partie de l'analyseur syntaxique.

Notre objectif était donc de finaliser l'analyseur syntaxique, d'implémenter l'analyseur sémantique et de remplir la TDI, de manière à ce que notre projet compile les langages *NilNoviAlgorithmique* (NNA) et *NilNoviObjet* (NNO).

2 Organisation du projet

2.1 Répartition de l'équipe et rôle

L'équipe était composée de Pascal FERRARIS, Thibaut LAUSECKER, Colin LUCIANI, Apolline MASSON, Marie PETER et Paul-Louis RENARD.

- Pascal FERRARIS s'est occupé de la TDI.
- Thibaut LAUSECKER s'est occupé de NNA.
- Colin LUCIANI s'est occupé de de la TDI.
- Apolline MASSON s'est occupée de compiler à la main correct1 de NNO.
- Marie PETER s'est occupée de la gestion de certaines erreurs, ainsi que de l'organisation au sein de l'équipe.
- Paul-Louis RENARD s'est occupé de faire fonctionner le code NNO.

2.2 Répartition des tâches

Ainsi, en particulier, Pascal FERRARIS et Colin LUCIANI se sont occupés de la partie consistant à réaliser les appels à la TDI lorsque cela était nécessaire pour avoir toutes les informations sur la classe en cours de définition.

Marie PETER s'est occupée du bon fonctionnement de certains messages d'erreur dans NNA, et s'est attachée au bon fonctionnement du travail en équipe, en essayant de faciliter la communication et le partage de connaissances entre les différents "pôles de travail" de l'équipe.

Thibaut LAUSECKER s'est occupé de la gestion de la boucle *while* et de l'alternative if...then...else.

Paul-Louis RENARD a rédigé une partie du rapport, a fait fonctionner le code sur NNO sur tout le jeu d'essai, a géré les erreurs sur NNO.

2.3 Calendrier prévisionnels et effectif

Le projet a débuté le 13 mars 2019 et s'est achevé le 7 juin 2019.

Tâche	Temps prévu initialement	Temps moyen passé
Lecture du sujet	1h	1h
Prise en main des outils et de la fourniture	2h	8h
Répartition du travail entre les membres du groupe	1-2h	4h
Conception dont :	12h30	35h
Identification des diagrammes syntaxiques	2h	5h
Aménagement de l'analyseur	2h30	5h
Gestion théorique des points de génération	3h	4h
Code des points de génération	2h	20h
Insertion des points de génération dans l'analyseur	3h	1h
Tests et corrections ultimes	2h	15h
Documentation	3h	3h

3 Description du processus de compilation NN

3.1 Description générale

Le processus de compilation NN prend en entrée un programme source NN et produit du code objet. Pour cela, on découpe déjà le programme source en unité lexicale grâce à l'analyseur lexical. La syntaxe du programme est ensuite vérifiée par l'analyseur syntaxique. Enfin, l'analyseur sémantique analyse la sémantique et prépare la production de code objet.

3.2 Analyseur lexical

L'analyseur lexical prend en entrée le flot de caractères du programme source et retourne des unités lexicales. Son objectif est de découper le programme en unités significatives.

Il permet de reconnaître les différentes composantes du programme : identificateurs, mots-clé, caractères, entier... De plus, il permet d'avoir uniquement les éléments nécessaires. Il ignorera, par exemple, les commentaires laissés par les programmeurs.

L'analyseur lexical permet également de repérer des mots qui ne seraient pas grammaticalement corrects. Les erreurs provenant de combinaison d'unités lexicales sont quant à elles du ressort de l'analyseur syntaxique.

3.3 Analyseur syntaxique

L'analyseur syntaxique prend en entrée les unités lexicales "découpées" par l'analyseur lexical et réalise une analyse syntaxique sur ces éléments. Cela signifie qu'il vérifie que la structure du programme correspond bien à ce qui est défini par la grammaire du programme. Il peut ainsi déclencher des erreurs s'il ne trouve pas un élément après une séquence qui implique cet élément.

En particulier, dans le processus de compilation NN, il reprend identiquement les éléments de la grammaire en commençant par l'axiome "programme". Il vérifie ensuite tous les éléments qui constituent un tel programme.

3.4 Table des identificateurs

La Table des Identificateurs (TdI) est une structure de données, son rôle est de retenir les informations relatives aux identificateurs (variables globale, variables locale, paramètres, classes, méthodes/fonctions/procédures, constructeur).

- Pour les variables locales, on retient l'identificateur (le nom), le type et l'adresse statique pour pouvoir y accéder.
- Pour les paramètres, on retient l'identificateur (le nom), le type, le mode (in ou in out) et l'adresse statique pour pouvoir y accéder.
- Pour les classes, on retient l'identificateur (le nom), la classe mère (si elle existe), le constructeur de la classe et son adresse, les méthodes, les attributs et leurs adresses et l'adresse de la classe.
- Pour les méthodes, on retient l'identificateur (le nom),leurs paramètres, leur statuts (Propre , Hérité ou Redéfini), leur adresse statique pour y accéder ainsi que leur adresse d'implémentation dans le code compilé.

Du fait de l'organisation du code Nil Novi , nous somme obligés de créer des représentations temporaire pour les classes ainsi que les méthodes car on ne possède pas toute les informations lors de la création et on remplira ces dernière au fur et à mesure du parcours des unités lexicales. De même pour les variables, on ne leur attribut un type qu'après leur création.

4 Modifications/Ajouts apportés au projet

4.1 NNA

Pour rappel, un programme NNA est constitué d'éventuelles variables globales, puis du programme principal.

Nous avons donc dû mettre en place un système pour réserver le bon nombre de variables globales. Pour cela, nous avons ajouté un attribut dans la classe $analyseurLL1$ appelé $nbrevariable$ qui s'incrémente dans la procédure $listeIdent()$. Il est réinitialisé à 0 à chaque début de nouvelle liste de variables à réserver dans la procédure $listeDeclaVar()$.

Pour vérifier que deux variables n'ont pas le même nom, nous récupérons leur nom dans une liste située dans $listeIdent()$. Ensuite, dans la procédure $listeDeclaVar()$, nous vidons la liste en vérifiant l'absence doublons. S'il y en a, nous déclenchons une erreur.

Les structures conditionnelles et la boucle du langage NNA requièrent de connaître les adresses de redirection en cas d'échec ou de fin de condition. Pour cela, il doit être possible de revenir dans le code pour compléter cette adresse de redirection dès qu'elle est connue. Nous utilisons dans ce but une structure de pile qui sauvegarde l'adresse à laquelle il faudra revenir au moment où l'on aura l'adresse de saut. Lorsqu'on connaît l'adresse de redirection, on dépile pour connaître l'endroit à modifier et on affecte l'adresse actuelle comme adresse de redirection. On procède ainsi en distinguant les sauts inconditionnels (tza) et les sauts conditionnels (tze).

Pour gérer les erreurs de typage, il est nécessaire de connaître le type des expressions manipulées. Une variable *typage* stocke le type des expressions lorsqu'il est connu. Ensuite, il faut vérifier la valeur de cette variable lorsque c'est nécessaire, c'est-à-dire lorsqu'on a affaire à une condition, un *get*, un *put*, une affectation, une opération ou un opérateur booléen.

Enfin, des points de génération ont été mis dans différentes procédures au moment où nous avions tout les renseignements nécessaires pour les produire.

Remarque : La version de NNA décrite est présente sur la branche NNA dans le projet git.

4.2 NNO

Un programme NNO permet de gérer un code comportant des classes ayant obligatoirement un constructeur et éventuellement des méthodes-fonctions ou des méthodes-procédures ou des attributs. Il est composé d'une partie constitué de la déclaration des classes, suivi par la déclaration des variables globales et enfin le programme principal pouvant appelé les classes définies précédemment.

Afin de déterminer le type d'un objet lors d'un appel à une fonction ou à une procédure, nous utilisons une variable $nomClasse$ qui en premier lieu contiendra le nom de l'objet. Ensuite, nous déterminons si nous avons affaire à un paramètre d'une fonction, d'une procédure ou d'un constructeur ou à une variable locale ou à une variable globale ou à un attribut d'une classe ou au mot *this* désignant la classe en cours. Cette connaissance supplémentaire nous permet ensuite de réclamer le type (donc la classe) de l'objet.

La variable *type* lors d'un appel à une fonction ou à une procédure permet de déterminer si la variable est un paramètre, un attribut, une variable locale ou une variable globale. Nous avons considéré que le mot *this* est une variable locale puisque sa compilation nécessite d'utiliser $empilerAd$ comme une variable locale.

La variable $adresse\_type$ stocke l'adresse statique de la variable dont on a déterminé la portée qu'on a stocké dans la variable *type*.

Connaissant désormais la classe d'un objet stocké dans la variable $nomClasse$, nous pouvons chercher si le nom de la méthode appelée correspond à une

procédure ou à une fonction de la classe. Si ce n'est pas le cas, c'est peut-être le constructeur.

Dans le cas où il s'agit d'une fonction ou d'une procédure, on vérifie qu'on a bien défini la portée de la variable et qu'elle n'a pas déjà été prise en compte grâce au booléen *dejaEmpiler* dans la partie *elemPrim* ou que la variable *adresse_type* n'a pas pour valeur -3 (Cette partie signifie qu'on a rencontré le mot clé *this* dans *elemePrim* et qu'on ne l'a pas empiler dans *elemPrim*). Désormais, en fonction de la valeur de la variable *type*, on empile correctement l'objet sur lequel on applique la fonction ou la procédure.

Dans le cas où il s'agit d'un constructeur, on empile l'adresse statique de la classe.

Enfin, à la suite de cette distinction, on place l'instruction *reserverBloc*. Ceci traduit l'algorithme pour appeler une fonction, une procédure ou un constructeur dans le cas où le programme est correct et ne contient pas d'erreur. Des éléments pour faire de la gestion d'erreur ont été rajouté.

Le booléen *mauvaisUtilisation* permet de déterminer si on utilise un constructeur comme une procédure. Cela est détecté par le fait qu'on ne fait pas d'affectation lors d'un appel à un constructeur. Pour cela, le booléen passe à vrai si on appelle un fonction/procédure/constructeur sans l'affecter dans la partie *instr*.

La variable *nomClassAvant* permet de stocker la valeur de la variable *nomClass* avant qu'on ne cherche le type de la variable. Si les noms correspondent, c'est qu'on a bien mis un nom de classe pour appeler un constructeur. Sinon, on distingue deux type d'erreurs : soit l'utilisateur a mis le bon nom de constructeur mais n'a pas mis le bon nom de classe lors de l'appel, soit l'utilisateur s'est aussi trompé sur le nom du constructeur.

Toujours pour l'appel à un constructeur, on vérifie s'il y a confusion entre un nom de classe et le nom d'une variable locale, un paramètre, un attribut ou une variable globale. Si c'est le cas, nous déclenchons une erreur. Il se peut aussi que l'utilisateur ne s'est pas trompé sur le nom de la classe mais s'est trompé sur le nom du constructeur.

Dans le cas d'un appel à une fonction, on sauvegarde le type de retour dans la variable *typage* dont l'objectif principal est de connaître le type d'une expression.

Dans le cas d'un appel à une procédure, si elle contient des paramètres, on stocke l'adresse statique de la procédure dans la variable *adresseProcAppelle* et on instancie le booléen $isEffectif$ à vrai. Ces deux éléments permettent de vérifier si les paramètres d'une procédure sont corrects. Cela signifie que si la procédure a des paramètres IN/OUT, elle reçoit en entrée des paramètres IN/OUT et pas uniquement des paramètres IN.

Enfin, dans les deux cas (constructeur ou méthode-fonction/méthode-procédure), on vérifie si le nombre de paramètres entrés est cohérent avec la définition du constructeur ou de la méthode-fonction/méthode-procédure.

Afin de connaître l'avancement du programme et savoir exactement si on est en train de déclarer les paramètres d'un constructeur, d'une fonction, d'une procédure ou si on est en train de déclarer une fonction, une procédure, un constructeur ou si on est dans une classe ou dans le programme principal, nous avons déclaré une énumération $declaVarType$ qui permet de faciliter la déclaration des variables dans la TDI mais également l'appel de méthode ou l'utilisation de variable. De plus, nous avons ajouté un compteur de classe nommé $nombreClass$ qui permet par la suite de distinguer les variables globales des classes en fonction de leur adresse. En effet, la première variable globale a pour adresse statique $nombreClass$.

Pour gérer l'héritage de classe, nous utilisons une méthode de la TDI $creerMere$ qui permet d'avoir les méthodes et attributs de la classe mère dans la classe fille. Dans la classe PointsGeneration, nous avons ajouté deux attributs : $adressProc$ et $adressFonct$ qui permet de sauvegarder l'adresse statique d'une procédure ou d'une fonction qui est redéfinie. Ces variables sont réutilisées respectivement dans les méthodes $finProcedure$ et $finFonction$ pour passer le statut de la méthode en tant que redéfinie puisqu'on l'a vu dans la partie interface de la classe fille.

Dans la méthode *heritage* de l'analyseurLL1, nous récupérons le nom de la classe mère en la stockant dans la variable $classHerite$. Cela permettra par la suite de faire de la gestion d'erreur en vérifiant si les attributs propres de la classe fille n'ont pas le même nom que ceux de la classe mère. Cette vérification est faite grâce à la variable $listeAttributClasse$ qui est une HashMap stockant l'ensemble des nom des attributs d'une classe. On peut accéder aux attributs d'une classe en connaissant son adresse statique. Pour connaître l'ensemble des attributs d'une classe avant de les stocker dans $listeattributClasse$, on utilise une liste appelé $listeAttributEnCour$ qui permet de stocker l'ensemble des attributs de la classe en cours de définition pour ensuite les transférer dans la HashMap.

Lorsque nous déclarons une méthode-fonction, une méthode-procédure ou un constructeur, nous stockons le nom des paramètres dans la liste $listeParamCours$. Ensuite, nous associons son adresse statique dans la classe ou -1 (dans le cas où il s'agit du constructeur) dans la Hashmap $listeParamMethodClassCour$. Cela sert par la suite dans la méthode $listeIdent$ de l'analyseurLL1 pour repérer s'il y a des doublons entre le nom des paramètres et le nom des variables locales des méthodes ou du constructeur.

En particulier dans l'interface, la variable $nomMethodeClasseIT$ stocke le nom des méthodes et du constructeur pour repérer dès qu'il y a un méthode qui porte le même nom qu'une autre. Elle ne stocke que les méthodes de la classe

courante. Il n'y a pas de problème avec l'héritage puisque celui-ci est gérée dans la classe *PointsGeneration* comme expliqué plus haut. Nous avons également réalisé la même opération pour vérifier s'il y avait des doublons dans la partie implémentation grâce à la variable $nomMethodeClasseIM$.

Les procédures peuvent avoir des paramètres de type IN/OUT. Cela est géré par des appels à la TDI pour indiquer un mode aux paramètres énoncés dans la partie *mode* de l'analyseurLL1.

Dans le cas de l'implémentation d'une procédure, nous vérifions que la procédure a bien été déclarée comme étant une procédure dans l'interface. Si c'est la cas, nous stockons dans la TDI l'adresse d'implémentation de la méthode. Nous la récupérerons plus tard à la fin de la classe puisque la compilation d'un programme NNO exige de connaître ces adresses. Dans *corpsProc* de l'analyseurLL1, nous passons le booléen $impossibiliteglobaleP$ à vrai. Cela permet par la suite de savoir que nous sommes dans une procédure (ou un constructeur puisque l'implémentation d'un constructeur passe aussi par *corpsProc*) et ainsi, appeler les bonnes méthodes de la TDI si nécessaire. *corpsProc* fait appel à *instr*. Cette partie de l'analyseurLL1 permet de lancer toutes les instructions possibles dans le langage $NilNovi$. Dans le cas où on effectue une affectation, il est important de connaître la portée d'une variable pour utiliser la bonne variante de l'instruction empiler. On utilise la TDI dans ce cas. Mais avant de faire appel à la TDI, nous distinguons la procédure du constructeur grâce à la variable *range* présente dans PointsGeneration qui retourne une valeur de l'énumération $declaVarType$. Nous avons déjà évoqué son utilité et il s'agit ici d'un cas concret de son utilisation. Dans le cas de l'affectation, nous sauvegardons le type de la variable. Nous avons déclenché une erreur lorsqu'on affecte la variable *this*. Nous vérifions ensuite que l'affectation est cohérente dans le sens où l'élément à gauche est soit une classe mère, soit les types des éléments à gauche et à droite de l'affectation sont identiques. Les boucles, structures conditionnelles, ne changent pas par rapport à NNA. Le *get* doit désormais faire appel à la TDI pour avoir le type de l'élément et vérifier qu'il est bien de type *integer*. Après avoir parcouru l'ensemble du code de la procédure, nous validons le fait que nous avons définie la procédure. Enfin, nous plaçons l'instruction $retourProc()$.

Dans le cas de l'implémentation d'un constructeur, cela se passe de manière similaire à la différence que la TDI possède des méthodes spécifiques pour accéder aux variables et pour valider la définition.

Dans le cas de l'implémentation d'une fonction, il faut vérifier que l'utilisateur n'a pas oublié de placer un *return*. Notre booléen *presenceRetour* permet de vérifier si le programme passe par *retour* de l'analyseurLL1 et donc de déclencher une erreur si la fonction ne possède pas de *return*. De même que pour la procédure, il y a un booléen $impossibiliteglobaleF$ qui nous indique que nous sommes en train d'implémenter une fonction. Il se passe ensuite les mêmes étapes que pour une procédure mais adapté avec une fonction. Les fonctions sont

les seuls éléments possédant le mot clé *return* dans leur programme. Ainsi, si on voit un *return* alors qu'on n'est pas dans une fonction, une erreur est déclenchée.

La partie *expression* est une partie importante du programme. Elle est appelée pour l'affectation, pour le put, pour les paramètres d'appel d'une méthode ou constructeur, pour les boucles, la structure if ou encore les retour pour les fonctions. Il était important de mettre à jour la variable *typage* du programme NNA pour qu'elle puisse récupérer le type d'une variable quelle que soit sa portée. Il faut aussi que nous prenions en compte que les fonctions retournent un élément d'un type donnée. C'est ici qu'intervient le booléen *empecheChangement* qui permet de ne pas récupérer le type des paramètres d'une fonction mais de ne prendre uniquement que le type de la valeur de retour. Ce booléen est aussi appelé dans *opRel* pour gérer le fait qu'on ne s'intéresse pas aux éléments autour d'un élément présent autour de l'opérateur mais juste du type final.

5 Résultats et Discussions

5.1 Jeu d'essai NNA

5.1.1 Description du jeu d'essai NNA

Nous avons testé les fonctionnalités au fur et à mesure grâce aux jeux d'essais fournis. Ceux-ci sont composés de 10 programmes NNA dont 4 programmes corrects et 6 programmes incorrects dans lesquels des messages d'erreurs sont attendus.

- Le programme correct1 permet de vérifier que des boucles imbriquées fonctionnent en affichant 4 fois la suite : 1 2 3 4.
- Le programme correct2 permet de tester les expressions relationnelles avec des booléens avec une simple condition if...then...else.
- Le programme correct3 permet de tester l'évaluation de conditions complexes en évaluant une condition présentant un "and", suivie d'une condition après une affection d'un booléen.
- Le programme correct4 permet d'afficher le nombre de valeurs paires lues et de sortir dès que l'utilisateur entre la valeur 0 et lui donne le nombre de valeurs paires lues.
- Le programme error1 permet de vérifier qu'on ne peut pas utiliser la fonction *get* avec un booléen.
- Le programme error2 permet de vérifier qu'on ne peut pas utiliser la fonction *put* avec un booléen.
- Le programme error3 permet de vérifier qu'on ne peut pas affecter une valeur d'un autre type que le type défini lors de la déclaration d'une variable.
- Le programme error4 permet de vérifier que l'expression contenue dans la condition d'un if doit être de type booléen.

— Le programme error5 permet de vérifier qu'une expression dans la condition d'un while est bien de type booléen.
— Le programme error6 permet de vérifier qu'il est impossible de définir 2 variables avec le même nom.

5.1.2 Résultats

Test	Succès ?	Remarques
correct1	OUI	
correct2	OUI	
correct3	OUI	
correct4	OUI	
error1	OUI	Vérification du type de la variable en utilisant la TDI
error2	OUI	Sauvegarde du type grâce à une variable dans expression()
error3	OUI	Vérification du type à gauche et à droite d'une affectation
error4	OUI	Vérification que le type est booléen
error5	OUI	Idem que error4 mais pour une boucle
error6	OUI	Stockage des noms puis vérification s'il y a des doublons

5.1.3 Discussions sur les résultats et critique des choix effectués

L'ensemble du code NNA fonctionne, la gestion d'erreurs est faite. On peut critiquer le fait que les points de génération ne sont pas écrits dans la classe approprié mais directement écrit dans l'analyseurLL1.

5.2 Jeu d'essai NNO

5.2.1 Description du jeu d'essai NNO

Le jeu d'essai NNO est composée de 54 programmes NNO dont 11 programmes corrects et 43 programmes incorrects dans lesquels des messages d'erreurs sont attendus.

— Le programme correct1 permet de tester la redéfinition d'un identificateur d'attribut par un identificateur de constructeur.
— Le programme correct2 permet de tester la redéfinition d'un identificateur d'attribut par un identificateur de variable locale.
— Le programme correct3 permet de tester la fonction "error".
— Le programme correct4 permet d'essayer le polymorphisme et les liaisons dynamiques.
— Le programme correct5 permet de manipuler des listes d'objets.
— Le programme correct6 permet de tester l'imbrication d'appels de méthodes-fonctions.
— Le programme correct7 permet de tester si l'exécution de méthodes identifiées par integer et nil provoque une erreur.
— Le programme correct8 permet d'essayer le passage de paramètres et l'utilisation du pronom *this* .

— Le programme correct9 permet de tester la définition de méthodes dans un ordre différent de leur déclaration.
— Le programme correct10 permet d'observer l'évaluation d'expressions arithmétiques en NNO.
— Le programme correct11 permet d'essayer l'expression d'un objet complexe grâce à l'appel d'une fonction qui délivre un objet.
— Le programme correct12 effectue un appel récursif d'une méthode-procédure et utilise le pronom *this*.
— Le programme error1 permet de tester si la redéfinition d'une méthode-procédure par une méthode-procédure avec un profil différent, un nombre de paramètres différents provoque une erreur.
— Le programme error2 permet de tester si la redéfinition d'une méthode-procédure par une méthode-procédure avec un profil différent, un type de paramètres différents provoque une erreur.
— Le programme error3 permet de tester si la redéfinition d'une méthode-procédure par une méthode-procédure avec un profil différent, un mode de passage différent provoque une erreur.
— Le programme error4 permet de tester si la redéfinition d'une méthode-procédure par une méthode-procédure avec un profil différent, un nom de paramètres différents provoque une erreur.
— Le programme error5 permet de tester si la redéfinition d'une méthode-fonction par une méthode-procédure provoque une erreur.
— Le programme error6 permet de prendre en compte le cas d'erreur lorsque *this* est déclaré comme identificateur de paramètre formel.
— Le programme error7 permet de prendre en compte le cas d'erreur lorsque *this* est utilisé comme identificateur d'attribut.
— Le programme error8 permet de prendre en compte le cas d'erreur lorsque *this* est utilisé comme identificateur de classe.
— Le programme error9 permet de vérifier qu'une erreur se déclenche lorsqu'il y a une redéfinition d'un attribut dans une même hiérarchie.
— Le programme error10 permet de vérifier qu'une erreur se déclenche lorsque *integer* est utilisé comme identificateur d'attribut.
— Le programme error11 permet de vérifier qu'une erreur se déclenche lorsque *integer* est utilisé comme identificateur de classe.
— Le programme error12 permet de vérifier qu'une erreur se déclenche lorsqu'une classe ne possède pas de constructeur.
— Le programme error13 permet de vérifier qu'une erreur se déclenche lorsqu'un paramètre formel est déclaré plusieurs fois dans une méthode.
— Le programme error14 permet de vérifier qu'une erreur se déclenche lorsqu'un identificateur d'attribut est déclaré plusieurs fois.
— Le programme error15 permet de vérifier qu'une erreur se déclenche lorsqu'un identificateur de classe est déclaré plusieurs fois.
— Le programme error16 permet de vérifier qu'une erreur se déclenche lorsqu'on redéfinit un paramètre par une variable locale d'une méthode.
— Le programme error17 permet de vérifier qu'une erreur se déclenche lorsqu'on redéfinit une opération dans la même classe.

— Le programme error18 permet de vérifier qu'une erreur se déclenche lorsqu'on utilise le même nom pour un attribut et une classe.
— Le programme error19 permet de vérifier qu'une erreur se déclenche lorsqu'on appelle une fonction dans un contexte incompatible avec sa définition.
— Le programme error20 permet de vérifier qu'une erreur se déclenche lorsqu'on affecte *this* dans une méthode.
— Le programme error21 permet de vérifier qu'une erreur se déclenche lorsqu'on appelle un constructeur incompatible avec sa déclaration dans le cas où il est appelé dans une classe différente de celle où il est défini.
— Le programme error22 permet de vérifier qu'une erreur se déclenche lorsqu'on appelle un constructeur incompatible avec sa déclaration dans la cas où il est appelé avec un objet (et non une classe).
— Le programme error23 permet de vérifier qu'une erreur se déclenche lorsqu'on implémente une procédure alors qu'on avait déclaré une fonction.
— Le programme error24 permet de vérifier qu'une erreur se déclenche lorsqu'on retourne un élément d'un type différent du type déclaré.
— Le programme error26 permet de vérifier qu'une erreur se déclenche lorsqu'on appelle une méthode spécifique sur un type dynamique. Cela se traduit par le fait que, bien que la valeur d'une variable d'une classe fille soit affectée à une variable de type la classe mère, on ne peut exécuter des méthodes de la classe fille sur la variable de la classe mère.
— Le programme error27 permet de vérifier qu'une erreur se déclenche lorsque une procédure attend un paramètre formel de mode in/out et que l'utilisateur rentre un paramètre effectif de mode in.
— Le programme error28 permet de vérifier qu'une erreur se déclenche lorsqu'une méthode déclarée n'est pas définie dans la classe.
— Le programme error29 permet de vérifier qu'une erreur se déclenche lorsqu'une fonction possède des paramètres in/out.
— Le programme error30 permet de vérifier qu'une erreur se déclenche lorsqu'un constructeur possède des paramètres in/out.
— Le programme error31 permet de vérifier qu'une erreur se déclenche lorsqu'on met un return dans un constructeur.
— Le programme error32 permet de vérifier qu'une erreur se déclenche lorsqu'on met un return dans le programme principal.
— Le programme error33 permet de vérifier qu'une erreur se déclenche lorsqu'une classe hérite d'une classe qui n'existe pas.
— Le programme error34 permet de vérifier qu'une erreur se déclenche lorsqu'il y a plusieurs constructeurs dans la même classe.
— Le programme error35 permet de vérifier qu'une erreur se déclenche lorsqu'il n'y a pas de constructeur.
— Le programme error36 permet de vérifier qu'une erreur se déclenche lorsqu'il y a un *return* dans une méthode-procédure.
— Le programme error37 permet de vérifier qu'une erreur se déclenche lorsqu'on essaye d'accéder à un attribut en dehors de la classe.
— Le programme error38 permet de vérifier qu'une erreur se déclenche lors-

qu'on utilise *this* en dehors d'une classe.
— Le programme error40 permet de vérifier qu'une erreur se déclenche lorsqu'on déclare plusieurs fois une méthode-procédure.
— Le programme error41 permet de vérifier qu'une erreur se déclenche lorsqu'on affecte des variables objets incompatibles.
— Le programme error42 permet de vérifier qu'une erreur se déclenche lorsqu'on appelle un constructeur comme une procédure.
— Le programme error43 permet de vérifier qu'une erreur se déclenche lorsqu'une fonction modifie un paramètre formel.
— Le programme error44 permet de vérifier qu'une erreur se déclenche lorsqu'il n'y a pas de *return* dans le corps d'une fonction.
— Le programme error45 permet de vérifier qu'une erreur se déclenche lorsqu'une classe hérite d'elle-même.

5.2.2 Résultats

Test	Succès ?	Remarques
correct1	OUI	
correct2	OUI	
correct3	OUI	
correct4	OUI	
correct5	OUI	
correct6	OUI	
correct7	OUI	
correct8	OUI	
correct9	OUI	
correct10	OUI	
correct11	OUI	
correct12	OUI	
error1	OUI	Utilisation de egalMethVirtProc de la TDI
error2	OUI	Idem
error3	OUI	Idem
error4	OUI	Idem
error5	OUI	Idem
error6	OUI	
error7	OUI	
error8	OUI	Vérification lors de la création d'une classe
error9	OUI	Utilisation de *listeAttributClasse*
error10	OUI	
error11	OUI	
error12	OUI	Utilisation de existeConstrClasseCour de la TDI
error13	OUI	
error14	OUI	

Test	Succès ?	Remarques
error15	OUI	
error16	OUI	Utilisation de listeParamMethodClassCour pour vérifier la redéfinition
error17	OUI	Utilisation de nomMethodeClasseIM pour vérifier la redéfinition
error18	OUI	
error19	OUI	
error20	OUI	
error21	OUI	
error22	OUI	
error23	OUI	Utilisation de getSorteMethVirt de la TDI
error24	OUI	
error26	OUI	
error27	OUI	
error28	OUI	
error29	OUI	
error30	OUI	
error31	OUI	
error32	OUI	
error33	OUI	
error34	OUI	
error35	OUI	
error36	OUI	
error37	OUI	
error38	OUI	
error40	OUI	
error41	OUI	
error42	OUI	
error43	OUI	
error44	OUI	
error45	OUI	

5.2.3 Discussions sur les résultats et critique des choix effectués

Le travail effectué permet de compiler un programme en NNO. Les erreurs présentes dans le jeu d'essai ont toutes été traitées.

On peut critiquer l'abus d'attributs dans la classe analyseurLL1 dont certains sont utilisés pour résoudre une seule erreur. Certains attributs sont utilisés pour sauvegarder des valeurs qui sont déjà présent dans la TDI.

6 Perspectives et conclusion

Ce projet, d'une ampleur plutôt grande, nous a permis de consolider nos connaissances et de développer nos compétences de travail en équipe.

En effet, durant ces trois mois, nous avons été amenés à nous plonger dans

un code assez conséquent, en ayant pour objectif d'approfondir, mais surtout d'appliquer des concepts vus en cours de manière plutôt théorique. Il a ensuite été assez ardu de se répartir le travail, en particulier dans une équipe de 6 personnes ayant des méthodes de travail et de compréhension différentes.

Nous sommes plutôt satisfaits du travail que nous avons fourni, même si nous avons pris un peu de retard par rapport au planning que nous nous étions fixé en début de projet. Nous avons en effet pu rendre un code permettant de compiler des programmes algorithmiques et objets, et qui traite la plupart des erreurs contenues dans les fichiers de tests fournis.

Pour conclure, ce projet a été une expérience enrichissante, qui nous a permis d'appliquer les concepts vus en cours, mais également d'apprendre à analyser un programme conséquent qui n'est pas le notre directement. Enfin, ce projet a également développé notre capacité de travail en équipe.