TELECOM Nancy deuxième année (2018-2019)

Deuxième rapport du projet de compilation du langage *Tiger*

Auteurs
Alexis DIEU,
David FORLEN,
Philippe GRAFF,
Tristan LE GODAIS

Enseignants encadrants
Suzanne COLLIN,
Sébastien DA SILVA,
Pierre MONNIN

La version originale du programme est disponible à l'adresse :
gitlab.telecomnancy.univ-lorraine.fr/Philippe.Graff/graff24u

Table des matières

1 Introduction

Ce rapport est le produit de la deuxième partie du projet de compilation du langage *Tiger* dispensé à *Télécom Nancy*. Il fait suite à la première partie, qui, pour rappel, consistait à réaliser les analyses lexicales et syntaxique du langage. Les objectifs de cette deuxième partie sont :

— finalisation de la construction de la table des symboles ;
— analyse sémantique du code à partir de l'arbre syntaxique ;
— génération du code assembleur.

Par conséquent, nous allons construire ce rapport dans l'ordre chronologique des différentes étapes de la réalisation du compilateur. Ainsi, en premier lieu, nous présenterons notre table des symboles finalisée. Nous donnerons d'abord sa forme générale (diagramme de classes et quelques exemples), puis nous entrerons plus en détails dans nos choix. En outre, nous expliquerons comment nous avons dessiné cette table des symboles en utilisant *Graphviz*. Ensuite, nous expliquerons comment nous avons mené à bien la réalisation des contrôles sémantiques. Nous détaillerons quelques méthodes de la classe `SymbolTable` utile à ces contrôles. Nous présenterons enfin la génération de code, en expliquant les classes que nous avons introduites et leur rôle.

Avant d'évoquer la gestion de projet, nous présenterons rapidement notre système de tests qui a été largement amélioré depuis décembre. En effet, lors de la réalisation des tests sémantiques, nous nous sommes rendus compte qu'une organisation systématique et plus détaillée des différents types de tests (lexicaux, syntaxiques, sémantiques et à l'exécution) était nécessaire pour mener à bien notre projet.

Pour finir, nous en viendrons à notre gestion de projet. Un diagramme de *Gantt* sera présenté ainsi que la durée de travail de chacun. Nous vous souhaitons une agréable lecture !

2 Table des Symboles

2.1 Conception

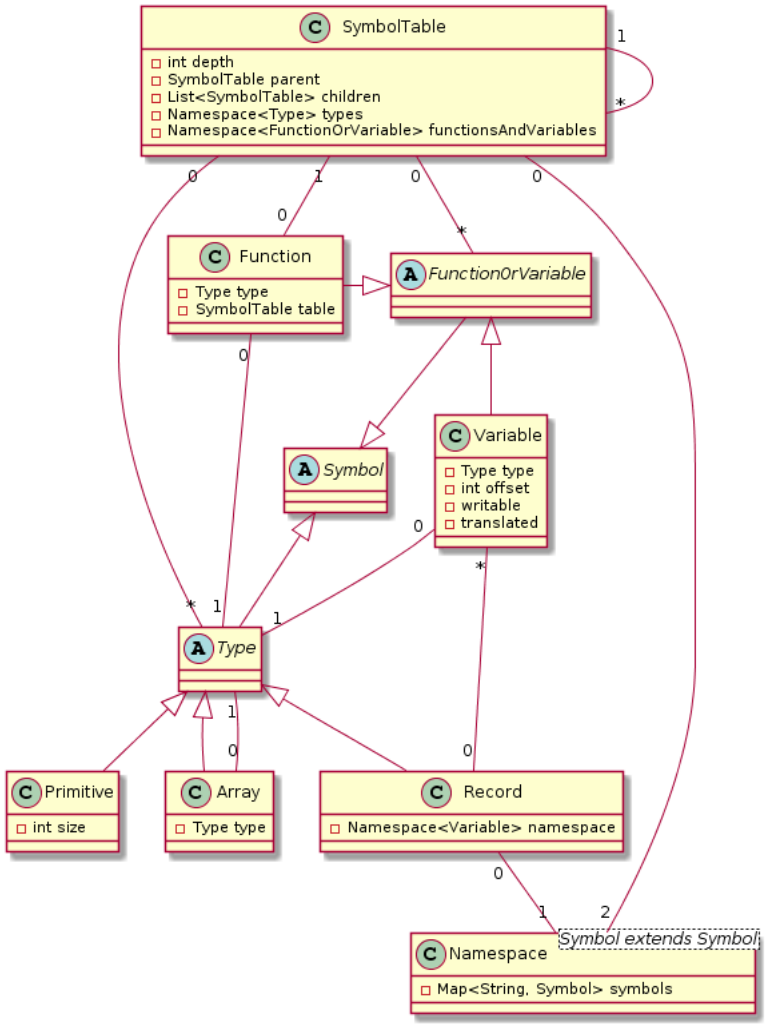

FIGURE 1 – Diagramme de classes de la table des symboles

En haut de la figure 1 se trouve la classe `SymbolTable`. Le niveau d'imbrication de la table est donné par l'attribut `depth`. Elle est également constituée d'un lien menant à la table des symboles de la région englobante, la table des symboles *parent*. La table des symboles racine (de niveau d'imbrication `0`) est constituée des types de valeurs primitives et des fonctions de la bibliothèque standard. Nous y retrouvons par exemple les fonctions `print`, `size`...

Une table des symboles doit également connaître tous ses enfants. Ils sont stockés dans une liste nommée `children`. Pour finir, deux instances de la classe `Namespace` figurent parmi les attributs de `SymbolTable`. Explicitons le rôle de ces deux éléments.

La classe `Namespace` correspond à un espace de noms. Rappelons qu'un identifiant peut être déclaré plusieurs fois dans le cas où il appartient à des espaces de noms différents. La classe en question associe donc un identifiant à un `Symbol` qui est une classe abstraite étendue par deux classes. Elles correspondent

aux deux espaces de noms définis dans la spécification du langage *Tiger* :

— la classe `Type` qui regroupe les entiers et les chaînes de caractères de classe `Primitive`, les tableaux via la classe `Array`, ainsi que les structures par le biais de `Record` ;
— la classe `FunctionOrVariable` qui regroupe des fonctions auxquelles on associe un type de retour et des variables auxquelles on associe un type et un déplacement `offset` [1].

2.2 Visualisation de l'arborescence des TDS

La visualisation des TDS d'un programme *Tiger* se fait grâce à *Graphviz*. Il convient donc de générer un fichier `.dot` contenant l'arborescence des TDS avec les liens les reliants.

La classe `TigerIllustrator` est responsable de cette fonctionnalité. Pour cela, un parcours en profondeur des TDS est effectué depuis la TDS racine. Les variables, fonctions et types déclarés dans chaque TDS sont ajoutés au graphe avec comme informations : leur nature (`type` / `function` / `var`), leur nom, leur déplacement (pour les variables uniquement) et un pointeur vers leur type (ou type de retour, pour les fonctions).

Les types déclarés sont tous répertoriés dans un tableau ne représentant pas une TDS : ce tableau est nommé *Types déclarés*. Les types composites sont ainsi plus facilement visibles : un tableau d'entier est par exemple représenté par une case `Array` dans *Types déclarés* avec un pointeur vers le type primitif représentant `int`.

La représentation fidèle de tous les liens existants entre les différentes TDS et type rend néanmoins très rapidement difficilement déchiffrable le graphe produit. Il reste cependant utile lors de la génération de code, pour mieux visualiser l'arborescence des TDS, lorsque l'on a besoin d'une vision précise de la situation. Les figures 2 et 3 présentent deux exemples de graphes générés.

1. Sommer le déplacement et la base de l'environnement courant permet de trouver l'adresse mémoire de la variable.

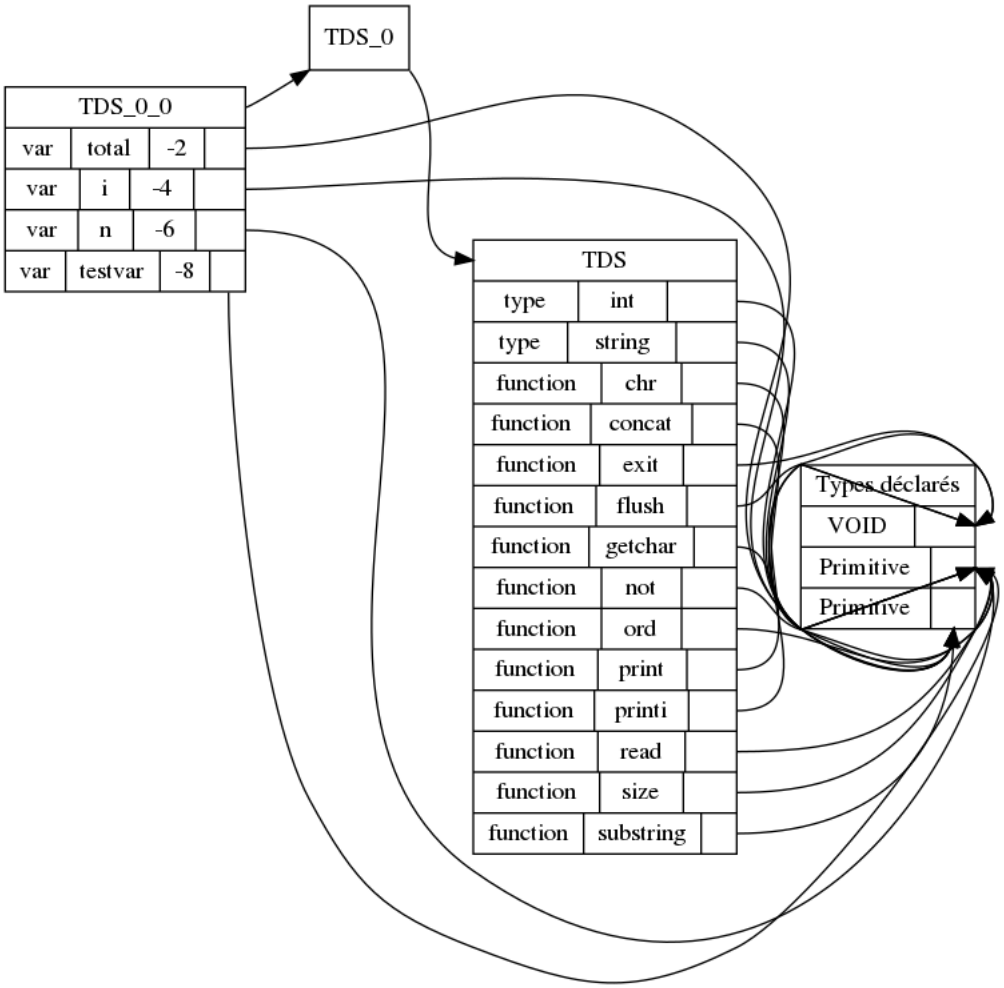

FIGURE 2 – Visualisation de l'arborescence des TDS du programme `level-1.tiger`

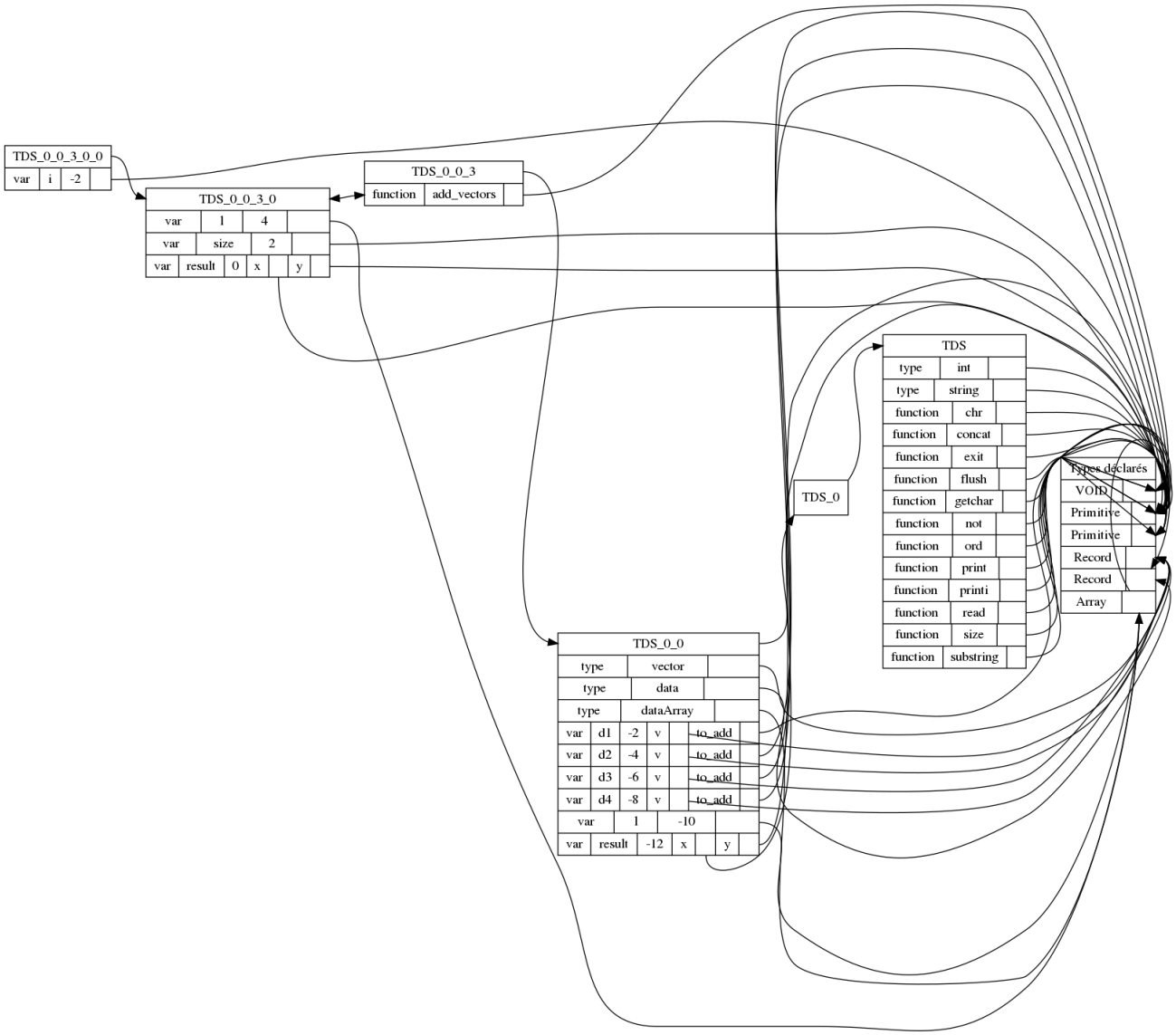

FIGURE 3 – Visualisation de l'arborescence des TDS du programme `level-4.tiger`

3 Contrôles sémantiques

Nous allons aborder ici la façon avec laquelle nous procédons aux contrôles sémantiques. L'analyseur sémantique `TigerChecker` parcoure l'arbre abstrait généré par *ANTLR* et procède à la fois au remplissage des tables des symboles et aux contrôles sémantiques. Il appelle plus particulièrement sa méthode `check`, sur chaque nœud correspondant à une expression qu'il rencontre. La sorte de nœud rencontrée conditionne le comportement et les contrôles à effectuer. Nous pouvons trouver plusieurs types de nœuds d'expressions.

3.1 Séquence d'expressions

Dans une séquence `SEQ`, aucun contrôle sémantique n'est nécessaire (outre les contrôles sémantiques enfants). Dans la méthode `checkSEQ` correspondante, il s'agit de renvoyer le type de la dernière instruction et d'appeler la méthode `check` pour chaque enfant.

3.2 Instanciation de tableau

`ARR` correspond à la création d'une valeur de type tableau. Lors de cette instanciation, il est nécessaire de vérifier plusieurs points :

— le type utilisé est défini, visible et correspond bien à un type de tableau ;
— le type de la taille du tableau doit être un entier ;
— la valeur initiale des éléments du tableau doit avoir un type cohérent avec celui donné lors de la définition du type de tableau.

3.3 Instanciation de structure

Le nœud `REC` correspond à la création d'une valeur de type structure. Les contrôles sémantiques suivants sont effectués :

— le type utilisé est défini, visible et correspond bien à un type de structure ;
— tous et seulement les champs attendus sont donnés, avec les bons noms, dans l'ordre et avec un type cohérent avec ceux donnés lors de la définition du type de structure.

3.4 Appel de fonction

Le nœud `CALL` correspond à un appel de fonction. Il convient de contrôler les points suivants :

— l'identifiant de fonction utilisé est défini, visible et correspond bien à un nom de fonction et non de variable ;
— le bon nombre de paramètres effectifs est donné ; leur type coïncide avec ceux des paramètres formels donnés lors de la définition de la fonction.

3.5 Accès à un élément de tableau

Le nœud `ITEM` correspond à l'accès à l'élément d'un tableau situé à un indice donné. Il convient de vérifier que :

— le type de la cible de l'accès est bien un tableau ;
— le type de l'indice d'accès est bien entier.

3.6 Accès à un champ de structure

Le nœud `FIELD` correspond à l'accès à un champ donné d'une structure. Il est contrôlé que :

— le type de la cible de l'accès est bien une structure ;
— le nom du champ figure bien dans l'espace de noms de ce type de structure.

3.7 Identifiant de variable

Le nœud `ID` est censé correspondre systématiquement à un identifiant de variable dans le cadre de l'analyse sémantique (étant donné que ni un identifiant de fonction ni un identifiant de type ne constitue une expression). Il convient simplement de vérifier que cet identifiant est bien défini, visible et correspond effectivement à un nom de variable et non de fonction.

3.8 Chaîne de caractères littérale et entier littéral

`STR` et `INT` sont respectivement les nœud des chaînes de caractères littérales et des entiers littéraux. Tout ce qu'il faut faire ici, c'est garder l'information du type de nœud pour le reste des contrôles sémantiques et la génération de code. Il n'y a pas de test sémantique à effectuer.

3.9 Affectation

Le nœud `:=` correspond à une affectation. En plus du contrôle sémantique de cohérence des types des opérandes, il est nécessaire de vérifier que l'opérande gauche constitue bien une *lValue* (un nœud `ID`, `ITEM` ou `FIELD`), puisque que notre grammaire avait autorisé syntaxiquement tout type d'expression à gauche de l'opérateur d'affectation.

3.10 break

Le nœud `break` correspond à l'expression du même nom. Il convient de vérifier que cette expression apparaît bien dans une boucle, avec aucune fonction entre les deux. Cependant, la spécification ne définissait pas avec suffisamment de précision où l'expression `break` était valide. Pour une boucle `while`, nous avons choisi de l'autoriser à la fois dans le test et le corps de la boucle, tandis que pour une boucle `for`, nous avons choisi de ne l'autoriser que dans le corps de la boucle et non pas dans le calcul des bornes.

3.11 Autres contrôles

La liste des contrôles sémantiques énumérés ici n'est pas exhaustive. Pour avoir un aperçu plus complet des contrôles sémantiques réalisés, lancer la commande `make torture` (éventuellement précédée d'un `make init build` si vous venez de cloner le dépôt) qui recense toutes les erreurs sémantiques du fichier `torture.tiger` alors généré.

4 Génération de code

4.1 Généralités

Dans notre implémentation, une table de symboles ne correspond pas systématiquement à une fonction *Tiger*. En effet, l'analyseur sémantique génère également une table pour chaque boucle `for` et une ou plusieurs tables pour chaque bloc `let`. Par contre dans le code assembleur généré, une routine est associée à chaque table. Cela permet de standardiser la remontée des chaînages statiques de tables de symboles en assembleur.

4.2 Conception de la génération de code

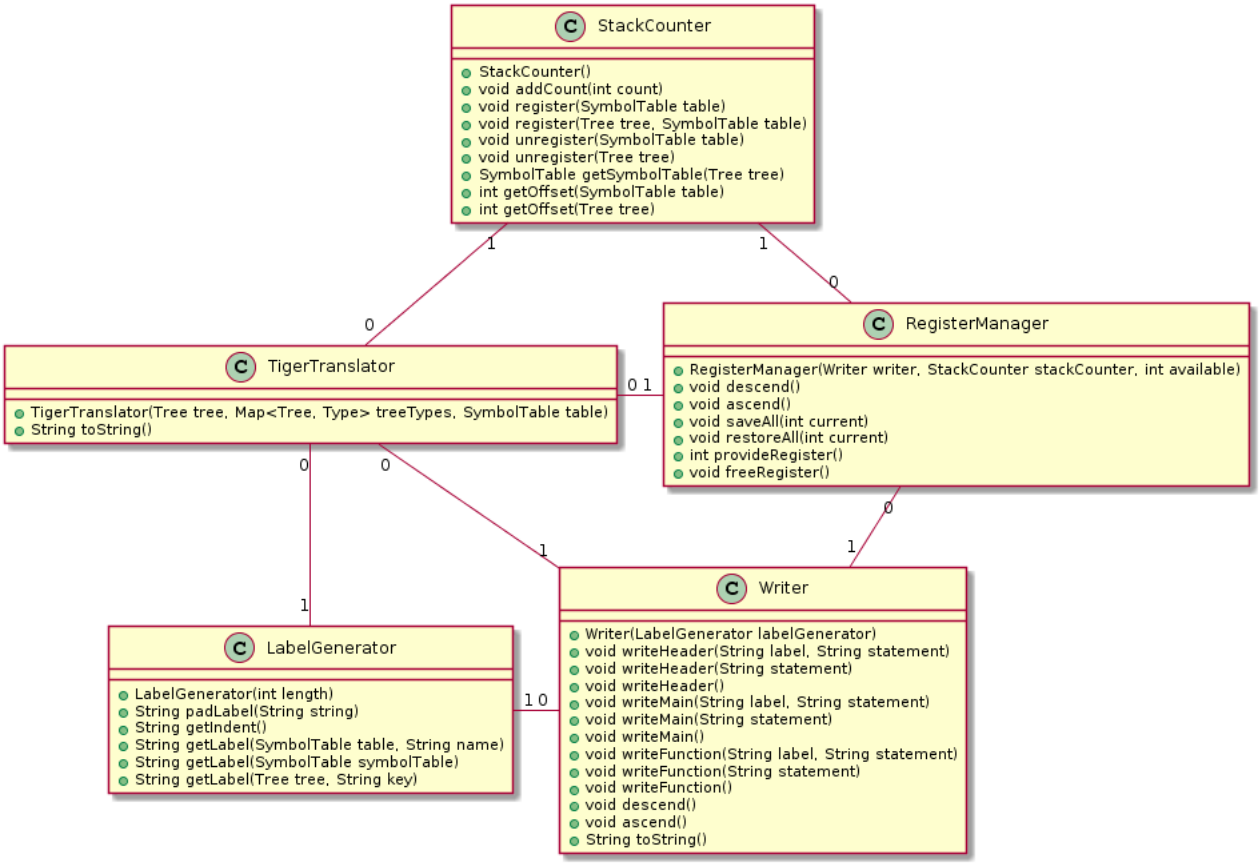

FIGURE 4 – Diagramme de classes de la partie génération de code assembleur

La figure 4 présente le diagramme de classes de la conception de la génération de code assembleur. Celle-ci repose sur la construction au préalable des tables de symboles du programme à compiler. Chaque table des symboles est représentée par une instance de la classe `SymbolTable`.

Les responsabilités des nouvelles classes présentes dans ce diagramme de classe sont les suivantes :

— `TigerTranslator` est responsable du parcours des nœuds de l'AST, synchronisé avec les descentes et remontées de tables de symboles pour toujours savoir quelle table est concernée par le nœud en cours de traitement. Cette classe génère également le code assembleur propre à chaque nœud de l'AST durant le parcours.

— `LabelGenerator` est responsable de la génération d'étiquettes uniques annotant les instructions assembleur, notamment pour faciliter les appels de routines.

— `Writer` est responsable de l'écriture du code généré par `TigerTranslator` au « bon endroit » dans le code assembleur. Cette classe permet notamment d'écrire le code d'une routine en plusieurs fois, et donc d'interrompre l'écriture du code assembleur d'une fonction *Tiger* pour écrire celui

d'une fonction qui serait définie à l'intérieur par exemple. Tout le code assembleur généré par `TigerTranslator` est écrit grâce à cette classe.
— `StackCounter` est responsable de la mesure de la taille de la pile au sein d'une fonction *Tiger*. Cette classe sert uniquement à implémenter l'expression `break` correctement.
— `RegisterManager` est responsable de la gestion des registres : elle attribue à la demande les registres qui ne sont pas utilisés, et qui peuvent ensuite être libérés à la demande également. Cette classe gère la sauvegarde temporaire des plus anciens registres dans la pile en cas de pénurie de registres et aussi la sauvegarde dans la pile de tous les registres utilisés lors de l'appel d'une routine par exemple, pour les restaurer à la fin de cet appel.

4.3 Parcours de l'AST pour la génération de code

`TigerTranslator` est conçu sur le même principe que `TigerChecker` pour son parcours de l'AST, sauf que ce parcours sert à générer du code assembleur en s'aidant des tables de symboles créées par le premier parcours de `TigerChecker`.

Le parcours débute à partir de la table racine, à l'instar du premier parcours fait par `TigerChecker`. À chaque fois que l'on descendait d'une table dans le premier parcours, on descend également d'une table ici. On garde l'état du parcours des différents fils d'une table dans une pile, afin de descendre dans les tables dans l'ordre exact établi par le premier parcours par `TigerChecker`.

La descente et la remontée de table de symboles est communiquée aux instances de `Writer` et de `RegisterManager` pour que le code généré soit écrit au bon endroit et que les registres soient empilés et dépilés correctement, en gardant en mémoire dans une pile *Java* le nombre de registres utilisés par chaque table au moment de la descente dans une table fille.

4.4 Fonctions natives

La bibliothèque standard *Tiger* compte au total 10 fonctions, auxquelles il faut ajouter `printi`, `read`. À chacune de ces fonctions correspond une routine, dont le code assembleur est écrit dans l'entête du fichier généré grâce à la méthode `writeHeader` de la classe `TigerTranslator`. Une routine supplémentaire, `spaceship`[2], a été introduite. Elle est utilisée par les opérateurs de comparaisons de chaînes de caractères, ce qui permet de factoriser grandement leur code assembleur.

4.5 Gestion du tas

Avant d'entrer dans le détail de la génération de code, nous allons expliquer la représentation en mémoire des différentes valeurs.

Le tas est une zone de la mémoire dont le sens de remplissage est contraire à celui de la pile (lorsqu'on « empile » dans le tas, le pointeur de sommet de tas augmente au lieu de diminuer). Il a été introduit pour stocker toutes les valeurs qui ne sont pas de type `int`, c'est-à-dire les chaînes de caractères, les tableaux et les structures. La pile ne contient donc que des pointeurs vers ces valeurs situées dans le tas.

2. `spaceship(a, b)` vaut `0` si et seulement si `a = b`, `1` si et seulement si `a > b` et `-1` sinon.

4.5.1 Chaîne de caractères

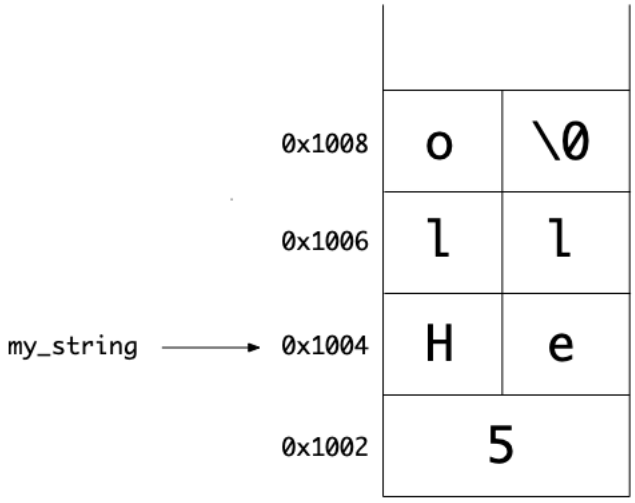

FIGURE 5 – Chaîne `"Hello"` dans le tas

La taille des chaînes de caractères est calculée une seule fois et insérée juste avant le début de la chaîne. À titre d'exemple, la figure 5 montre comment la chaîne `"Hello"` est représentée en mémoire.

Le fait de mémoriser la taille de la chaîne de caractère permet d'éviter de la recalculer lorsque l'on veut les concaténer par exemple. À noter également que si la chaîne avait été de taille paire (sans compter le caractère nul final), par exemple la chaîne de caractères `"hi"`, alors elle aurait été ponctuée de deux caractères nuls au lieu d'un seul.

Les chaînes de caractère littérales sont allouées au début du programme, mais sans utiliser la directive `STRING` afin de gérer les caractères spéciaux. En théorie, on peut donc considérer qu'elles sont allouées statiquement, mais en pratique les caractères d'une chaîne sont effectivement ajoutés deux par deux dans le tas. Dans le cas où plusieurs littéraux surviennent dans le code *Tiger* et correspondent à des chaînes de caractères équivalentes, une unique chaîne est allouée afin de limiter le nombre d'instructions générées ainsi que la place en mémoire occupée.

4.5.2 Tableau

À l'instar des chaînes de caractères, les tableaux sont précédés de leur taille dans le but de faciliter la gestion des erreurs lors de l'accès à l'élément d'un tableau. Les différents éléments se succèdent ensuite. S'il s'agit d'entiers, alors la valeur se trouve directement dans la liste. Dans le cas contraire, on trouvera un pointeur vers une autre zone du tas.

4.5.3 Structure

Les structures sont composées d'une succession de champs dont les valeurs se trouvent dans l'ordre dans lequel ils ont été déclarés par commodité. Il s'agit à nouveau soit d'entiers, soit de pointeurs.

Dans le cas d'un type de structure vide, un mot est tout de même réservé pour éviter que deux variables différentes puissent avoir la même adresse. La structure `nil` n'a quant à elle pas de représentation en mémoire dans le tas. Il s'agit d'un pointeur nul, qui pointe ainsi vers une case qui ne peut être allouée pour une autre structure, puisque n'étant pas dans le tas dans le cadre de notre implémentation.

4.6 Génération de code durant le parcours

Le parcours est effectué au travers de la classe `TigerTranslator`, qui implémente pour chaque type de nœud de l'arbre abstrait correspondant à une expression une méthode chargée de le convertir en code assembleur *MicroPIUP*.

Tout nœud d'expression passe donc par la méthode `translate` qui sélectionne via une structure `switch` la bonne méthode chargée de traduire ce nœud. Nous n'expliciterons pas le fonctionnement de toutes ces méthodes, bien que chaque nœud ait ses spécificités. Nous allons seulement détailler certaines méthodes, notamment celles avec lesquelles nous avons rencontré des difficultés.

4.6.1 Opérateurs de base

Les opérateurs de base sont `=`, `<>`, `<`, `>=`, `>`, `<=`, `|`, `&`, `+`, `-`, `*` et `/`. Ils se traduisent presque tous de manière transparente dans la mesure où des instructions *MicroPIUP* font exactement ce que nous attendons. Par exemple, on peut utiliser `ADD` pour traduire l'addition `+`.

En revanche, d'autres nécessitent un peu plus de réflexion, par exemple `&`. En effet, il n'existe pas de fonction *MicroPIUP* capable de gérer directement la conjonction (et cela sans rappeler que l'évaluation s'arrête au premier nœud renvoyant `0`). Il a donc été nécessaire de recourir à l'instruction `TST` permettant de tester si la valeur d'un registre est nulle ou non, et une étiquette de fin de conjonction à laquelle se déplacer si cette valeur est effectivement nulle.

En outre, le langage *MicroPIUP* est plutôt limité concernant la gestion des chaînes de caractères. Il n'existe pas d'instruction préconçue capable de comparer deux chaînes dans l'ordre lexicographique ; nous avons donc dû l'écrire nous même. C'est ce que fait justement la routine `spaceship`.

Par ailleurs, la spécification ne définit pas la valeur de retour d'une conjonction ou d'une disjonction. Afin de perdre le minimum d'information, nous avons choisi de renvoyer la valeur du dernier opérande évalué.

4.6.2 Accès à une variable

L'accès à une variable consiste à retrouver son adresse dans la pile. Ainsi, on compte le nombre de tales de symboles (donc de routines) à remonter pour accéder au bloc où a été déclarée la variable. On génère le code assembleur permettant de remonter ce nombre de tables pour stocker l'adresse du pointeur de base de l'environnement de déclaration de la variable recherchée dans un registre. À ce registre, on ajoute le déplacement de la variable pour obtenir l'adresse de la variable.

Pour obtenir la valeur de cette variable, il suffit de prendre la valeur de la case de l'adresse qu'on vient de calculer.

4.6.3 Accès à un élément de tableau ou un champ de structure

Dans un premier temps, on génère le code pour accéder au pointeur du tableau ou de la structure : ils sont considérés comme des variables dont la valeur est l'adresse de leur stockage dans le tas (voir partie *Accès à une variable*).

Une fois l'adresse calculée et stockée dans un registre, on ajoute à cette adresse :

— pour un champ de structure : son déplacement, en vérifiant à l'exécution que la structure n'est pas nil et en interrompant le programme à l'aide de la trappe logicielle `EXIT_EXC` dans le cas contraire ;
— pour un élément de tableau : l'indice de l'élément multiplié par la taille des éléments du tableau. Mais on vérifie avant cela que l'indice est positif et inférieur à la taille du tableau, taille stockée

dans le mot précédant le tableau.

À partir de ce calcul d'adresse, obtenir la valeur se fait en prenant la valeur contenue dans la case situé à cette adresse.

4.6.4 Appel de routine

À l'appel d'une routine, il faut préparer son environnement :
— sauvegarder dans la pile les registres en cours d'utilisation,
— réserver un registre pour l'empilement des paramètres effectifs de la routine appelée si elle en a ;
— empiler ces paramètres effectifs ;
— déterminer le chaînage statique de la routine.
Le chaînage statique de la routine appelée est le pointeur de base de l'environnement dans lequel elle a été définie. Pour le calculer à l'exécution, il faut à la compilation générer le code pour remonter `niveauImbricationFonctionAppelante - niveauImbricationFonctionAppelée` table de symboles et mettre dans le registre `R0` l'adresse du pointeur de base de la table à laquelle on est arrivé après cette remontée.

On peut ensuite appeler la routine à l'aide d'une instruction `JSR` et de l'étiquette de la routine appelée.

Il faut ensuite nettoyer l'environnement de la routine appelée :
— dépiler les paramètres effectifs ;
— restaurer les registres sauvegardés en début d'appel ;
— transférer la valeur du registre `R0` dans le registre dans lequel le retour de la fonction est attendu.
De manière générale, lorsqu'on sauvegarde ou restaure les registres utilisés, il n'est pas nécessaire d'appliquer ce traitement au registre dans lequel on compte mettre la valeur de retour, puisque sa valeur ne nous importe pas encore à ce moment-là.

En cas d'appel à une fonction native, on ajoute sa déclaration au code assembleur généré si elle n'a pas déjà été appelée. De cette manière, seules les fonctions natives présentes dans le code sont écrites dans le code généré. Ceci s'applique aussi pour la routine `spaceship`.

4.6.5 Affectation

L'affectation se fait en trois temps : le calcul de l'adresse de `lValue`, la variable cible de l'affectation, et le calcul de la valeur de l'expression à droite de l'opérateur `:=`, et enfin l'affectation de cette valeur à l'adresse de `lValue`.

Le calcul de l'adresse de `lValue` se fait soit par accès à une variable, à un élément de tableau ou à un champ de structure, comme décrit précédemment, sans prendre la valeur située à l'adresse calculée. Cette adresse est stockée dans un registre.

Le calcul de la valeur de retour de l'expression à droite du `:=` se fait par génération de code à l'aide de la méthode `translate`, cette valeur de retour étant stockée dans un deuxième registre.

Il ne reste qu'à affecter la valeur du deuxième registre dans la case dont l'adresse est stockée dans le premier registre.

4.6.6 Branchement et boucles

Les branchements `if` et les boucles `while` sont similaires dans la mesure où elles fonctionnent avec des sauts. Leurs codes assembleurs sont donc quasiment identiques.

Pour prendre l'exemple du `while`, il s'agit de générer deux étiquettes : une au niveau du test (que l'on a simplement nommée `test`), et une en fin de boucle (là où l'on saute si le test renvoie `0`) nommée

`end`.

En ce qui concerne les boucles `for`, elles sont gérées comme des routines appelées immédiatement avec deux paramètres effectifs : les bornes de la boucle. Elle disposent également d'une unique variable locale : l'indice de boucle. Là aussi deux étiquettes sont nécessaires : une en début de tour de boucle à laquelle remonter à la fin de chaque tour de boucle et une en fin de boucle à laquelle sauter lorsque la boucle est terminée.

4.6.7 break

L'évaluation d'une expression `break` se fait en trois étapes : restaurer la pile, restaurer la base de l'environnement et restaurer les registres tels qu'ils étaient avant la boucle.

Pour restaurer la pile, il faut donc dépiler toutes les variables, tous les paramètres effectifs, tous les chaînages, toutes les adresses de retours, ainsi que tous les registres sauvegardés temporairement dans la pile en cas de pénurie de registres. Cette information est déterminée à la compilation en utilisant la classe `StackCounter` introduite à cet effet. Elle comptabilise le nombre de mots empilés au sein d'une fonction *Tiger* et mémorise la taille de la pile au début d'une boucle, ce qui permet d'en déduire la taille à dépiler au moment de l'évaluation d'une expression `break`. D'une manière similaire, on peut retrouver la base de l'environnement initial.

La restauration des registres s'effectue à partir d'une simple sauvegarde préventive des registres utilisés, effectuée avant chaque boucle, comme lors de l'appel d'une routine.

5 Tests

Au fur et à mesure de l'avancement du projet, le système de tests a évolué, et a finalement dissocié chaque étape de la compilation ainsi que l'exécution. Ainsi 365 fichiers de tests lexicaux, syntaxiques, sémantiques et à l'exécution ont été écrits pour vérifier le bon comportement de notre compilateur *Java*. On compte parmi ceux-ci les fichiers envoyés par nos encadrants.

Notre dossier comportant les tests, nommé `tst`, est présent à la racine de notre dépôt. Il est composé de quatre dossiers : `lexical`, `syntactic`, `semantic` et `runtime`, chacun composés de deux sous-dossiers `fail` et `pass` qui contiennent finalement des fichiers d'extension `.tiger`.

Alors que les tests à la compilation se contentent de vérifier les codes d'erreurs du compilateur, les tests à l'exécution portent quant à eux principalement sur la sortie standard des programmes auxquels on a soumis une certaine entrée standard. Ainsi au sein du dossier `pass`, pour chaque fichier `.tiger`, deux fichiers de même nom mais d'extension `.tiger.in` et `.tiger.out` sont fournis et correspondent respectivement à l'entrée standard *donnée* et à la sortie standard *attendue*. Concernant le dossier `fail`, il en va de même, sauf que chaque fichier `.tiger.out` correspond à la sortie standard qu'il ne faut *pas* obtenir.

Afin de lancer la suite de tests, il convient de lancer la commande `make test` (éventuellement précédée d'un `make init build`). Une étiquette associée à un code couleur est utilisée pour symboliser le niveau de réussite d'un test. La signification de ces étiquettes est définie ainsi :

- `[PASS]` : test qui a réussi ou échoué comme prévu ;
- `[FAIL]` : test qui a réussi ou échoué alors que le contraire était attendu ;
- `[EXIT]` : test produisant un plantage du compilateur ;
- `[WARN]` : test produisant une erreur à une étape de la compilation antérieure à celle testée (dans ce cas, soit le test est faux soit il est dans le mauvais dossier).

		Type d'erreur					
		Java error	Lexical error	Syntactic error	Semantic error	Runtime error	Pas d'erreur
Dossier	lexical/pass	EXIT	FAIL	PASS	PASS	N/A	PASS
	lexical/fail	EXIT	PASS	FAIL	FAIL	N/A	FAIL
	syntactic/pass	EXIT	WARN	FAIL	PASS	N/A	PASS
	syntactic/fail	EXIT	WARN	PASS	FAIL	N/A	FAIL
	semantic/pass	EXIT	WARN	WARN	FAIL	N/A	PASS
	semantic/fail	EXIT	WARN	WARN	PASS	N/A	FAIL
	runtime/pass	EXIT	WARN	WARN	WARN	FAIL	PASS
	runtime/fail	EXIT	WARN	WARN	WARN	PASS	FAIL

PASS	Test qui a réussi ou échoué comme prévu
FAIL	Test qui a réussi ou échoué alors que le contraire était attendu
EXIT	Test produisant un plantage du compilateur
WARN	Test produisant une erreur à une étape de la compilation antérieure à celle testée
N/A	Cas de figure ne pouvant pas se présenter

FIGURE 6 – Codes couleurs selon les types d'erreurs rencontrées et les dossiers de tests

Pour référence, le figure 6 fournit une description plus précise de la correspondance entre les différents codes couleurs, les types d'erreurs rencontrées et les dossiers où se trouvent les tests. À noter qu'il faut partir du principe que si plusieurs erreurs sont rencontrés, alors c'est l'erreur la plus à gauche dans le tableau qui est considérée.

6 Gestion de projet

6.1 Gantt

Le diagramme de Gantt réalisé dans la première partie du projet à été complété pour couvrir la deuxième partie du projet (figure 7).

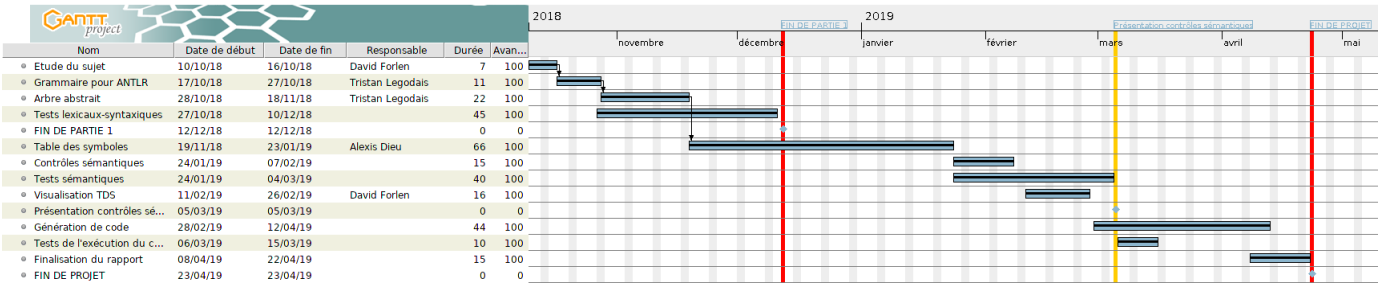

FIGURE 7 – Gantt du projet

6.2 Répartition du travail

	David	Tristan	Alexis	Philippe
Analyses lexicale, syntaxique et sémantique				
Prise de connaissance du sujet	03 : 00	00 : 30	00 : 30	02 : 30
Écriture de la grammaire	06 : 00	11 : 00	06 : 00	04 : 00
Réalisation de l'arbre syntaxique abstrait	06 : 00	05 : 00	01 : 00	01 : 00
Table des symboles	07 : 00	15 : 00	05 : 30	05 : 00
Contrôles sémantiques	06 : 00	01 : 30	10 : 30	02 : 00
Écriture des tests	02 : 00	29 : 00	05 : 30	10 : 00
Visualisation de l'arborescence des tables de symboles	14 : 00	02 : 00	00 : 00	00 : 00
Génération de code				
Général	04 : 00	23 : 00	19 : 30	05 : 00
Parcours de l'arbre syntaxique abstrait	05 : 00	04 : 00	04 : 00	02 : 00
Chaînages	06 : 00	00 : 00	00 : 00	00 : 00
Affectation de variables	08 : 00	00 : 00	00 : 00	00 : 00
Gestion d'erreurs à l'exécution	03 : 00	00 : 00	00 : 00	00 : 00
Fonctions natives	02 : 00	06 : 00	12 : 00	25 : 00
Gestion de projet				
Gantt	01 : 00	00 : 00	00 : 00	00 : 00
Réunions	19 : 00	19 : 00	19 : 00	19 : 00
Rapports				
Rapport (partie 1)	05 : 00	05 : 00	06 : 00	03 : 00
Rapport (partie 2)	03 : 00	07 : 00	06 : 00	03 : 30
Total				
Total	100 : 00	132 : 00	095 : 30	082 : 00

FIGURE 8 – Répartition du travail

6.3 Colorateur syntaxique pour *Atom*

Le colorateur syntaxique réalisé pour *Atom* en première partie de projet a légèrement évolué dans la seconde partie pour permettre une rédaction de tests et de programmes encore plus efficiente au

sein du groupe. Au cours du projet, d'autres groupes ont également été amenés à utiliser ce colorateur syntaxique.

Des fonctionnalités d'auto-complétion pour les différentes structures syntaxiques du langage *Tiger* ainsi que pour les fonctions de la bibliothèque standard ont été ajoutées, avec une gestion à la volée et automatique de l'indentation. En pratique, cela permet d'éviter les fautes de frappes ou de déceler plus facilement le mauvais parenthèsage par exemple. L'analyseur sémantique de notre compilateur permet ensuite d'éliminer les erreurs restantes.

7 Annexes

7.1 Exemple d'exécution d'un programme compilé avec notre compilateur

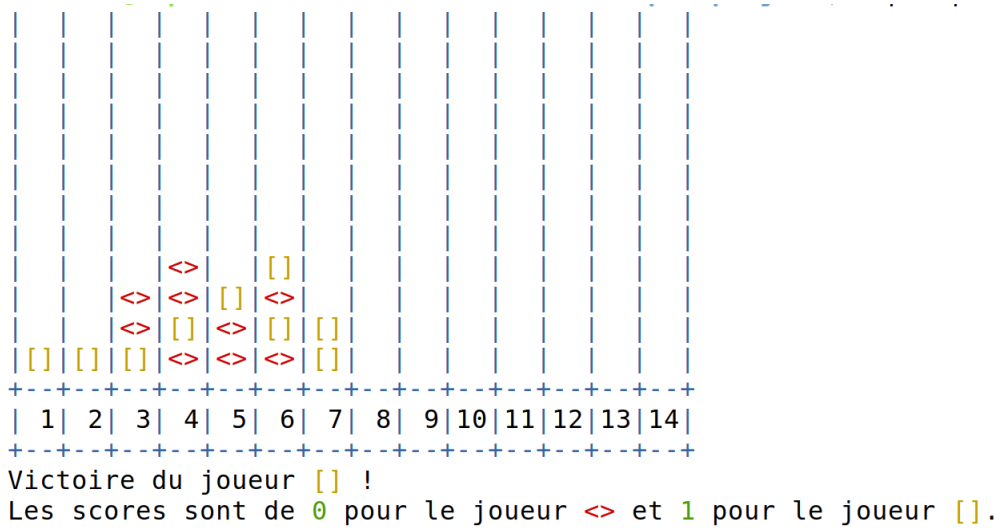

FIGURE 9 – Aperçu de l'exécution du programme `ansi-connect-four.tiger`

7.2 Diagramme de classes de la seconde partie du projet

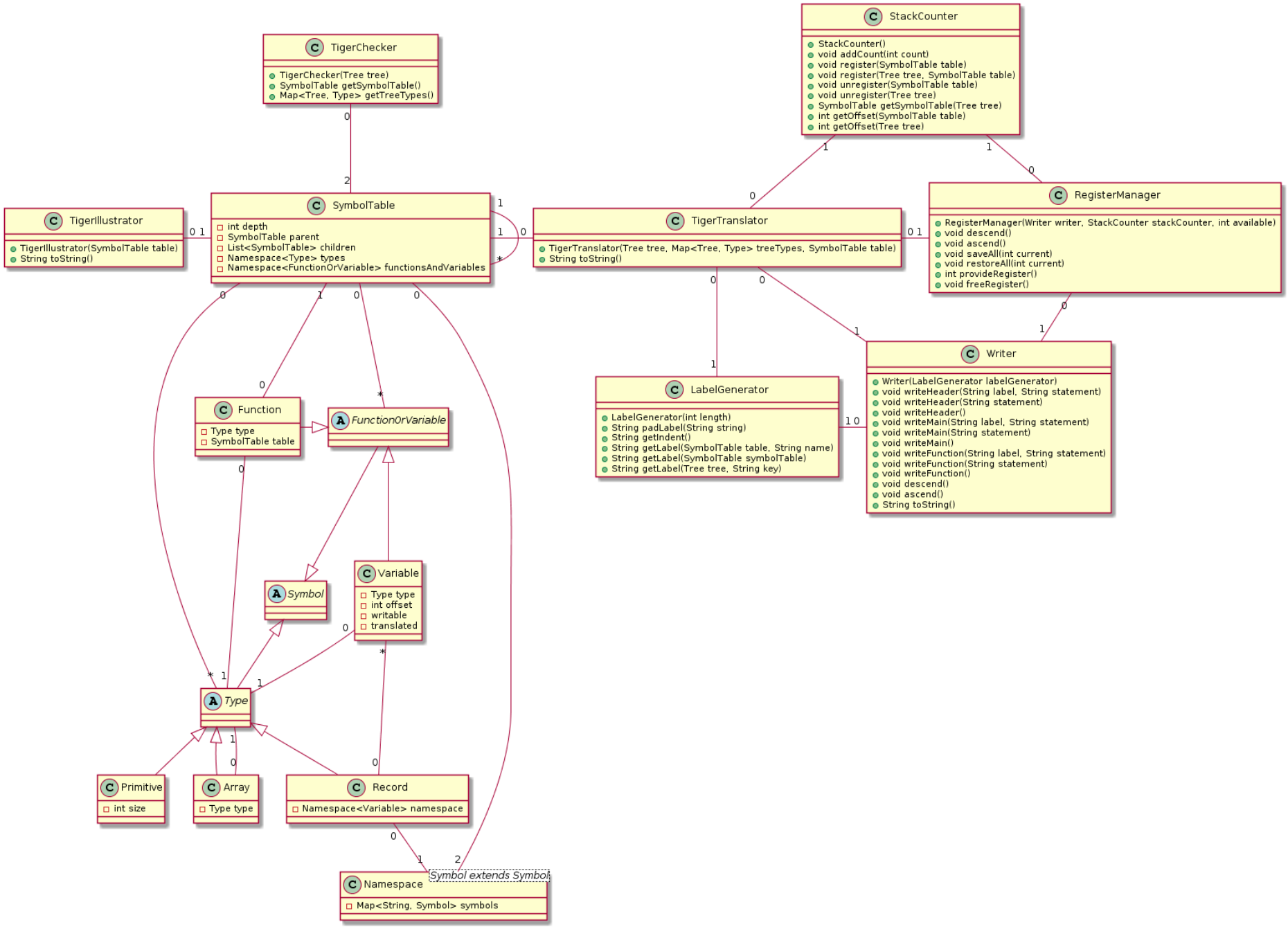

FIGURE 10 – Diagramme de classes de l'analyse sémantique, de la visualisation des TDS et de la génération de code

8 Comptes-rendus de réunions

Les pages ci-dessous exposent nos comptes-rendus de réunion (uniquement pour le deuxième semestre).

Réunion 11

Date et heure. 26 janvier 2019 à 10 heures

Prochaine réunion. 2 février 2019 à 10 heures

Ordre du jour.

1. Jalons
2. Code des contrôles sémantiques
3. Travail à faire

1. Jalons

2. Code des contrôles sémantiques

David demande à Tristan où on en est. Tristan a essayé de faire une première implémentation du remplissage de la TDS, surtout les parties `let`. Il a géré les espaces de noms (c'est bien débuté). La table n'est pas encore parfaitement remplie : il manque `var`, l'algorithme qui parcourt plusieurs fois le `let` pour gérer les fonctions et types récursives, il manque le type de toutes les variables. Pour l'instant, il n'y a que des fonctions et des types. Les TDS d'imbrication supérieure à 1 ne sont pas encore implémentées. On n'a pas non plus géré `nil`. On n'a pas non plus géré l'absence de type de retours (`void`).

David rappelle que l'indice d'une boucle est toujours de type `int`. Faut-il remonter toutes les TDS ? Comment faire si l'utilisateur déclare son propre type `int` ? Une solution serait de passer la TDS d'ordre zéro en paramètre de la fonction de remplissage.

David propose de tester qu'une variable est bien déclarée

- Type non définie
- Variable non déclaré
- Fonction non définie
- Utilisation d'une fonction au lieu d'une variable ou vice-versa
- À chaque fois qu'on rencontre `nil` dans le code, on doit pouvoir identifier son type
- Un break est bien utilisé dans une boucle

Tristan rappelle qu'il est possible de tenter d'accéder à un indice de tableau même si celui-ci est plus grand que la taille du tableau. Cela ne provoque pas d'erreur, mais le comportement n'est pas défini à l'exécution.

La division par zéro, même en clair, n'est *à priori* pas un contrôle sémantique. On peut le traiter quand même.

Tristan a trouvé une façon de trouver les lignes où les bugs sont apparus : antlr permet de le faire facilement.

3. Travail à faire

Alexis.

- Lire la spécification
- Vérifier les contrôles sémantiques déjà écrits
- Remettre la ligne du fichier `level-4.tiger`
- Écrire des tests qui réussissent sémantiquement
- Commencer les contrôles sémantiques

Philippe.

- Lire la spécification
- Écrire des tests qui réussissent sémantiquement
- Commencer les contrôles sémantiques

David.

- Finir la TDS

Tristan.

- Finir la TDS

Réunion 12

Date et heure. 2 février 2019 à 10 heures

Prochaine réunion. 8 février 2019 à 14 heures

Ordre du jour.

- TDS
- Contrôles sémantiques
- Travail à faire

1. TDS

Comme prévu, la branche `dev` a été abandonnée et on travaille sur la branche `dev2`. Dessus, on a ajouté les espaces de noms avec la classe `Namespace` qui est nouvelle. Tristan a réimplémenté la table des symboles d'ordre zéro. Il s'est aussi surtout occupé du `let`. Il s'est occupé de la définition des types et des variables, a géré la récursivité des fonctions.

Tristan a géré des tables des symboles intermédiaires dans les `let`. Il peut y avoir plusieurs tables des symboles dans un `let`. En effet, on peut déclarer plusieurs fois la même fonction dans un même bloc `let`, mais certaines variables, fonctions ou types entre ces variables

```
function f() : int = 1
let x : int = f()
function f() : int = 0
```

Il ne reste rien à faire à part l'initialisation des variables. Pour l'instant, tout ce qui est déclaré est dans la table, type et fonctions. Au niveau de la TDS, il ne reste donc que l'initialisation des variables.

Philippe propose de parler de `fillWithFor`. C'est David qui a écrit le corps de cette fonction.

2. Contrôles sémantiques

Des contrôles sémantiques ont été effectués sur `SEQ`, `CALL` et les opérateurs.

David a effectué quelques contrôles sémantiques : ils sont tous réunis au niveau du `fillWith`. Elle concerne tout ce qui est opérateur. Il propose aussi de réunir tous les contrôles sémantiques dans un documents. On n'a pas à gérer les `Alias` dans les contrôles de type.

David explique comment fonctionne le `Notifier`.

On remet à plus tard la coloration sémantique des erreurs. Tristan propose d'écrire toutes les erreurs en anglais.

Tristan remarque que plusieurs erreurs sémantiques qui n'ont pas encore été évoqués. Ils sont sur `Issues` dans Discord. David demande à Tristan de mettre ces contrôles sémantiques dans le fichier concerné.

Ce qu'il reste à faire dans les contrôles sémantiques.

- Vérifier que l'on affecte pas l'indice d'une boucle dans une boucle
- Ce qui est à gauche d'une affectation est une l-value
- Inférence des types lors de la déclaration d'une variable
- Vérifier le type d'une affectation
- Choses commentées dans le `switch`

Bien faire attention à ce qu'on ait que des expression dans les `switch`. On ne doit pas faire de `fillWith` sur la partie gauche d'une affectation (par exemple).

3. Travail à faire

Alexis.

- Revoir `SEQ` : il peut y avoir zéro expressions dans un tel nœud
- Revoir `CALL`: il y a un itérateur dans les namespace
- Contrôles sémantiques `ARR` , `REC`
- Écrire et tester ses tests

Philippe.

- Contrôles sémantiques `FIELD`, `ITEM` (faire attention aux `a.b.c.d` et aux `a[b][c][d]`
- Écrire et tester ses tests

David.

- Contrôle sémantique : inférence des types lors de la déclaration d'une variable (`case var` dans le `fillWithLet`)
- `case :=`
- `case ID`
- Écrire et tester ses tests

Tristan.

- Mettre-à-jour la suite de tests
- Corriger le compte du nombre d'erreurs
- Mettre-à-jour le fichier des contrôles sémantique
- Vérifier que l'on affecte pas l'indice d'une boucle dans une boucle
- Contrôles sémantiques sur `while` et `if else`
- `nil`
- Écrire et tester ses tests

Réunion 13

Date et heure. 8 février 2019 à 14 heures

Prochaine réunion. 18 février 2019 à 17 heures

Ordre du jour.

- Point sur les contrôles sémantiques qui ont été faits et implémentés.
- Standardisation des messages d'erreurs
- Visualisation de la TDS
- Lancement de la suite des tests
- Travail à faire

1. Point sur les contrôles sémantiques

Il ne faut pas systématiquement retourner `null` après une erreur afin d'analyser tout le code.

L'inférence de type à la déclaration des variable a été faite.

Philippe était chargé de faire les contrôles sémantiques pour `FIELD` et `ITEM`. Il ne faut pas utiliser la fonction `getSymbol()`. `ITEM` n'est pas encore tout à faire terminé.

Pour que deux types soient compatibles, il ne suffit pas qu'ils soient égaux. Par exemple, `nil` doit pouvoir être donné en champ d'un *record*.

Travail de Tristan. les contrôles sémantiques sur *while* et *if* ont été réalisés. Pour le while, il s'agit de vérifier que la condition est un entier et que le type de la boucle est un `void`. `null` dans `checktype` signifie `void`. Pour le if, on vérifie que la condition est un entier. S'il n'y a qu'un `then`, il faut que son type soit `void`. S'il y a un `then` et un `else`, il doit y avoir compatibilité entre les types.

Le `nil` nécessite un nouveau type "record vide". C'est une valeur spéciale de type et le checkType a été adapté en conséquence. Le checktype ne prend plus le dernier argument qui était inutile. Il vérifie maintenant si un des deux types sont `nil`. Tristan a fait une quinzaine de tests sur `nil`. David a également fait des tests, mais ne passent pas pour l'instant.

Vérifier qu'on affecte pas l'indice d'une boucle dans une boucle : à faire.

2. Standardisation des messages d'erreurs

Tous les messages d'erreurs doivent être en anglais. Pas de majuscule ni de points, les phrases doivent être au présent.

3. Visualisation de la TDS

Le sujet impose une visualisation de la TDS. On pourrait utiliser GraphViz. Pour l'instant, seule la TDS est capable de connaître son contenu.

4. Lancement de la suite des tests

`make prompt` permet de tester un programme à la voler. Se référer au manuel pour voir comment tester un fichier en particulier.

5. Travail à faire

Alexis.

- Revoir `CALL`
- Tester ses contrôles sémantiques
- Revoir le code de `ARR` : on ne doit pas retourner `null` juste après un contrôle sémantique et même chose pour `REC`
- Ne pas comparer deux types avec `==`
- Mettre à jour le fichier des contrôles sémantiques
- Écrire des tests (en veillant à ce qu'il n'y ait pas de WARN)
- Chercher d'autres contrôles sémantiques éventuellements oubliés
- Commencer le rapport

Philippe.

- Terminer le test sémantique sur `ITEM`
- Mettre à jour le fichier des contrôles sémantiques
- Écrire des tests (en veillant à ce qu'il n'y ait pas de WARN)
- Chercher d'autres contrôles sémantiques éventuellements oubliés
- Commencer le rapport

David.

- Mettre à jour le fichier des contrôles sémantiques
- Écrire des tests (en veillant à ce qu'il n'y ait pas de WARN)
- Visualisation de la TDS

Tristan.

- Commenter
- Indice de boucle (contrôle sémantique)
- Mettre à jour le fichier des contrôles sémantiques
- Écrire des tests (en veillant à ce qu'il n'y ait pas de WARN)
- Corriger la sortie des couleurs
- Visualisation de la TDS

Réunion 14

Date et heure. 18 février 2019 à 17 heures

Prochaine réunion. 28 février 2019 à 17 heures

Ordre du jour.

- Point sur ce qui était à faire pour aujourd'hui
- Où nous en sommes en terme de planning
- Répartition rédaction du rapport

1. Point sur ce qui était à faire pour aujourd'hui

Alexis a corrigé certain tests sémantiques qui étaient faux sémantiquement. Après un `pull` de nombreux `EXIT` apparaissent. Tristan propose que dans nos fichiers `main.java`, on comment la ligne numéro 62. David remarque alors que tout ce passe beaucoup mieux malgré quelques fails. David finira la TDS ce soir.

Philippe a retravaillé le test sémantique sur `ITEM`.

David rappelle qu'on avait, à la base, la liste de tous les contrôles sémantiques qu'on avait identifiés. Il va falloir orgraniser ce fichier dans l'optique de classer ces contrôles. Tristan pense qu'il suffit de chercher dans la table des symboles (`SymbolTable.java`) les endroits où des erreurs sémantiques sont levés et de les reporter dans ce fichier. On va remettre ce point pour tout le monde.

David a réalisé la visualisation de la TDS, mais ce n'est pas tout à fait fini. La visualisation devrait être terminée d'ici ce soir. Cela aidera à voir la hiérarchie des programmes que nous avons écrits.

Tristan rappelle qu'il faut installer GraphViz. Il faudrait que l'on puisse faire la génération d'images depuis java. Quand on lance le main, le programme génère la TDS, mais en console. On peut le mettre dans un fichier .gv, et générer un visuel de la TDS. Mais ce n'est pas très pratique.

Tristan a réalisé le contrôle d'indice de boucle. Il n'a pas touché au fichier des contrôles sémantiques, et n'a pas écrit de tests à part pour l'indice de boucle que l'on n'a pas le droit d'affecté. Il faut encore un peu toucher aux tests. Quand on va voir dans le dossier logs, on n'a plus des caractères spéciaux qui s'affichent. On peut *lire* dans le fichier log.

2. Où nous en sommes en terme de planning

David remarque qu'il faudrait commencer la génération de code. Tout le debogage sera basé sur les prints. On aura effectivement à faire des tests unitaire sur l'affichage. Il faudra également voir comment implémenter les fonctions qui nous

ont été données. Mme Collin ne nous a donné que le print. Tristan propose qu'on fasse un nouveau dossier *runtime* pour les tests avec la même structure qu'ailleurs (fail et pass).

Pour cette génération de code, il va falloir qu'on se la répartisse. Logiquement, on ne devrait pas retoucher à la table des symboles. David propose, pour le système de tests, de mettre la sortie attendue en commentaire en début de fichier `.tiger`. Tristan propose, pour les fonctions de la bibliothèque standard, de réfléchir à un pseudocode.

3. Répartition des tâches pour le rapport

Il va surtout falloir entamer la partie sur les contrôles sémantiques. Chacun doit remplir sa partie sur les contrôles sémantiques.

4. Travail à faire

Alexis.

- Mettre à jour le fichier des contrôles sémantiques
- Lire la documentation sur MicroPIUPK
- Écrire le code assembleur des opérations de bases (arithmétiques et comparaisons)
- Remplir sa partie sur les contrôles sémantiques

Philippe.

- Mettre à jour le fichier des contrôles sémantiques
- Lire la documentation sur MicroPIUPK
- Écrire le code assembleur des opérations de bases (arithmétiques et comparaisons)
- Remplir sa partie sur les contrôles sémantiques
- Revoir le diagramme de classes de la TDS

David.

- Mettre à jour le fichier des contrôles sémantiques
- Rendre optionnel la génération de la visualisation de la TDS
- Lire la documentation sur MicroPIUPK
- Programmer le `print_i` en assembleur
- Ajouter le GANTT au rapport
- Remplir sa partie sur les contrôles sémantiques

Tristan.

- Mettre à jour le fichier des contrôles sémantiques
- Commenter son code
- Ajouter la documentation de MicroPIUPK sur le dépôt

- Mettre à jour le système de tests pour prendre en compte les tests à l'exécution
- Lire la documentation sur MicroPIUPK
- Programmer le `print_i` en assembleur
- Remplir sa partie sur les contrôles sémantiques
- Remplir la partie sur les tests

Réunion 15

Date et heure. 28 février 2019 à 17 heures

Prochaine réunion. 13 mars 2018 à 16 heures

Ordre du jour.

- Travail effectué
- Gantt
- Assembleur

1. Travail effectué

Philippe a effectué des notes concernant la documentation MicroPIUPK. Philippe a demandé sur le Discord où écrire le code assembleur. Réponse : il faut faire un deuxième parcours de la TDS.

Tristan évoque la présentation des contrôles sémantiques dans le rapport. Il compte cinq catégories d'erreurs sémantiques.

1. Élément non déclaré, redéclaration, définition cyclique d'un type
2. Nombre et ordre des éléments dans l'appel d'une fonction ou d'une structure
3. Cohérence des types (y compris tout ce qui se rapport à `nil`, aux tableaux et au structures), et inférence des types possible
4. Un indice de boucle ne peut pas être affecté, `break` ne peut pas être en dehors d'une boucle
5. Le membre gauche d'une affectation doit être un `lValue`

Maintenant, pour demander la visualisation de la TDS, il faut mettre l'option. Tout est marqué dans le README.

David remarque que `graphic` ne lance plus de EXIT. Le GANTT a été ajouté. Il est encore en évolution.

Il faut commencer par générer l'en tête du code assembleur. Ensuite, il faudra qu'on implémente le `print` ; pour cela il faudra pouvoir gérer les string dans le code. Dans notre code Java, on va gérer nos lignes d'assembleur une par une. Chaque fois qu'on va lire une chaîne de caractère dans l'AST, on devra la mettre dans l'en tête du code assembleur. Il faudra identifier les parties du code assembleur dans lesquels il est pertinent d'insérer du code. Déjà, on est sûr qu'il y aura une séparation en-tête / corps.

D'après Tristan, on pourrait d'abord lever la contrainte du nombre de registre.

Alexis.

- Finir de lire la documentation MicroPIUPK
- Écrire le code assembleur des opérations de bases (arithmétiques et comparaisons)

Philippe.

- Finir de lire la documentation MicroPIUPK
- Ajouter les cardinalités dans le diagramme de classes
- Écrire le code assembleur des opérations de bases (arithmétiques et comparaisons)

David.

- Pseudo remplissage

Tristan.

- Mettre à jour le fichier des contrôles sémantiques
- Commenter son code
- Ajouter la documentation de MicroPIUPK sur le dépôt
- Mettre à jour le système de tests pour prendre en compte les tests à l'exécution
- Programmer le `print_i` en assembleur
- Remplir sa partie sur les contrôles sémantiques
- Remplir la partie sur les tests
- Ranger les tests

Réunion 15

Date et heure. 13 mars 2019 à 16 heures

Prochaine réunion. 22 mars 2018 à 16 heures

Ordre du jour.

- Travail effectué
- Problèmes à régler
- Travail à faire

1. Travail effectué

David a travaillé sur le pseudo remplissage. Lorsque le `fillWithLet` sera remplacé, le reste sera facile à réalisé. Une classe `TigerTranslator` a été codée. Celle-ci a exactement la même structure que la TDS, mais sert à générer le code à partir de la TDS.

David propose d'utiliser plusieurs `TigerTranslator` afin de pouvoir gérer le sens des enfants.

En ce qui concerne les tests, Tristan avait rangé les tests pour la soutenance. Il a rangé l'intégralité des tests qui sont dans `semantic/fail`. Il a également réalisé le programme de concaténation, et a corrigé un bug présent dans `let`. On n'avait pas vérifié que le code d'une fonction était cohérent avec son type de retour.

Le nouveau système de test est en cours mais est compliqué car micropiupk ne donne pas de code d'erreur. L'exécution du code machine peut aboutir à une erreur, mais le code est toujours le même.

David propose que l'on se concentre sur la génération de code pour le moment. On verra le rapport un peu plus tard.

2. Problèmes à régler

Comment on gère les registres ? Ceux-ci peuvent être libres ou non. On peut utiliser un système de tableau comme évoqué en tutorat. On a aussi la possibilité d'utiliser une pile des registres libres.

Proposition de Tristan : les fonctions `fillWithXXX` doivent renvoyer le nombre de registres à utiliser.

3. Travail à faire

Alexis.

- Finir de lire la documentation MicroPIUPK
- Écrire le code assembleur des opérations de bases (arithmétiques et comparaisons)

Philippe.

- Finir de lire la documentation MicroPIUPK
- Écrire le code assembleur des opérations de bases (arithmétiques et comparaisons)

David.

- Pseudo remplissage

Tristan.

- Mettre à jour le fichier des contrôles sémantiques
- Commenter son code !important
- Mettre à jour le système de tests pour prendre en compte les tests à l'exécution
- Programmer le `print_i` en assembleur

Réunion 17

Date et heure. 22 mars 2019 à 16 heures

Prochaine réunion. 28 mars 2019 à 17 heures

Ordre du jour.

- Point sur ce qui a été fait
- Gestion du GIT
- Travail à faire

1. Point sur ce qui a été fait

La dernière fois, on avait proposé de faire une pile des registres disponibles. Il faut une fonction capable d'allouer des registres, les désallouer et d'ajouter des variables dans la pile s'il n'y a pas assez de registres.

Les fonctions d'addition et de multiplication de base vont être vraiment simple. En assembleur, il suffit de l'écrire `ADD R0, R1, R0`.

Il faut gérer une fonction capable de récupérer le décalage d'une variable (base pointer + offset). David et Tristan se chargeront de faire cela car ils ont eu des cours dessus en Traduction 2. Il faut pour cela faire une classe contenant une pile de registres identifiés par une chaîne de caractères, et une variable contenant le nombre de registres disponibles. S'il n'y a pas de registres disponibles, il faut quand même pouvoir réserver des registres : cette variable contenant le nombre de registres disponibles devient négative.

Il faut désallouer dans l'ordre inverse du sens dans lequel on a alloué, et il faut être dans le même contexte d'éxecution (ie. dans la même fonction *Java*).

La pile des registres disponibles est une pile *Java*, et non pas une pile en assembleur.

Le pseudo remplissage n'a pas beaucoup avancé, mais on n'est pas vraiment bloqués. En revanche, à terme, il faudrait que cela soit fait.

Pour l'instant, on ne fait pas de ramasse-miettes, c'est vraiment la dernière chose que l'on fera.

Tristan a mit quelques commentaires pour David.

À chaque TDS correspond une fonction en assembleur.

2. Gestion du GIT

Penser à faire un `git pull` avant de commencer à travailler.

3. Travail à faire

Alexis et Philippe.

- Écrire la fonction de gestion des registres (en Java)
- Écrire les fonctions des opérateurs de base, et gérer le cas où ce n'est pas unaire, attentions aux entiers

David et Tristan.

- Écrire la fonction permettant de récupérer le décalage d'une variable
- Coder l'accès à un champ
- Faire le tas

Réunion 18

Date et heure. 28 mars 2019 à 17 heures

Prochaine réunion. 5 avril 2019 à 14 heures

Ordre du jour.

- Point sur ce qui a été fait
- Travail à faire

1. Point sur ce qui a été fait

Philippe a écrit la classe permettant la gestion des registres. Les résultats sont concluants dans le test : on y libère une trentaines de registres pour les relibérés. Les tests ont été fait en java et on édite le fichier assembleur en même temps. Attention, R0 ne peut pas être utilisé, R15 non plus, et possiblement R11, R12, R13. Par mesure de prudence, mieux vaut n'utiliser que ceux de 1 à 10.

Tristan remarque que l'on a directement écrit dans un fichier avec `file.append`, pour des raisons que David nous décrira plus tard. Mieux, il faudrait renvoyer le code assembleur pour qu'une autre fonction écrive à un moment différent. Le code ne va pas être écrit dans l'ordre.

David explique : c'est la partie du travail qu'il a effectué. Lorsqu'on a une série d'instructions, elle est exécutée. Mais si on déclare une fonction, le code assembleur de la fonction va être mis au début du fichier `.src`. On ne peux pas à chaque fois en ajoutant à la fin du fichier. Mais il y a des cas plus délicats : si on déclare une fonction dans une fonction, son code assembleur respectif ne sera pas dans le code assembleur de la première.

La solution est d'écrire une classe qui effectue tout cela. Celle-ci possèderait une méthode *écrire* qui est à peu près la seule que l'on utilisera. Cette classe a déjà été écrite et est intitulée `Writer`, elle n'est pas encore sur la branche master.

Quand on écrit notre code (Java), il ne faut pas utiliser de labels. C'est David et Tristan qui vont s'en occuper. Les labels sont uniques. Tristan demande à David si le `Writer` prend en paramètre des chaînes de caractères multilignes. D'après David, cela devrait fonctionner. Tous les labels commencent par des underscore `_`.

D'après Tristan, il faut quelques modifications sur la fonction `print` donnée par M. Parodi. Il attire l'attention sur l'instruction `flush`. C'est cette instruction qui fait effectivement l'affichage. La différence entre `flush` et `print` n'est pas très claire. Il faudrait peut-être envoyer un mail aux enseignents pour avoir plus de détails.

Tiger autorise les backslash suivis de guillemets, il faut donc les parser et les remplacer par leurs bonnes valeurs. Ce que l'on doit gérer, c'est, à partir d'une

chaîne de caractère Tiger. En plus, Tiger autorise un backslash suivit de plusieurs espaces (c'est-à-dire au moins 1) puis d'un autre backslash. Ces espaces doivent être ignorés.

Philippe et Alexis ont travaillé sur la gestion des strings : comparaison et égalité.

David a terminé la partie principale de `TigerTranslator` qui gérait le bon parcours de l'AST et de la TDS. Les deux classes en plus, `Writer` (pour écrire au bon endroit) et celle de la gestion des labels. La structure est quasiment similaire au `SymbolTable` : on a `translateFor`, `translateWhile`, etc. David a mis des Todo un peu partout pour qu'on sache où il faut écrire le code. Tous les translates prennent en argument le registre où l'on veut que se retrouve le résultat de ce que l'on va écrire. Ce registre est sous forme d'entier. Cette fonction retourne le type de l'expression, mais on pourra éventuellement le retiré car celui-ci ne sert à rien. Tristan propose de savoir si ce qu'on a retourné est un pointeur ou non.

Les `translateRec` `translateArr` risquent de poser un peu problème. Le reste, on peu y aller et faire le code qu'il faut. Pour les boucles, il faut un label de début de boucle, mais aussi un label de fin de boucle. Mais pour le `for`, on a une TDS, donc d'après Tristan, il suffit de faire un RTS.

Tristan demande qui a accès à `Writer`. D'après David, seule `TigerTranslator` y a accès. À la création de `GestionRegister`, on donne l'instance `writer`.

David propose d'ajouter des commentaires dans le code généré.

2. Travail à faire

Alexis et Philippe.

- Programmer les opérateurs de base
- Mettre-à-jour la classe `GestionRegister` pour qu'elle prenne en paramètre l'instance `writer`
- Mettre les registres de 1 à 10
- Tester la classe `GestionRegister` dans MicroPIUPK

David et Tristan.

- Écrire la fonction permettant de récupérer le décalage d'une variable
- Coder l'accès à un champ
- Faire le tas
- print_i

Réunion 19

Date et heure. 8 avril 2019 à 9 heures

Prochaine réunion. Demain

Ordre du jour.

- Organisation de la semaine

Chaque jour, on fera une petite réunion le matin pour décider ce que l'on fera pendant la journée et l'avancement par rapport au planning.

Organisation de la semaine

Les opérateurs de bases : il faut revoir le `DIV` qui calcule le reste et le quotient. La fonction permettant de récupérer le décalage d'une variable a été fait (il faudrait peut être améliorer la syntaxe). David remarque qu'il faudra que l'on consacre un jour aux tests.

L'accès à un champ n'est pas encore fait. D'après Tristan, le tas se fait en une ligne. Par contre, pendant l'exécution du code Tiger, on n'a plus accès aux types : pas moyen de savoir les types de champs dans les structures. On ne connaît donc pas la taille des champs dans une structure.

`print_i` nécessite que l'on puisse lancer des fonctions. Tristan pense qu'il faudrait d'abord faire le `print` (reprendre le code de M. Parodi et le mettre aujourd'hui.

Dans nos tests futurs, il ne faudra pas mettre de chaînes de caractères trop longues. En effet, elles risquent de polluer notre code assembleur car il faut une instruction par caractère.

David : on a maintenant une *hashmap* qui permet d'associer à chaquue nœud de l'AST le type renvoyé par le fonction `fillWithXXX`. On peut donc maintenant faire des tests sur le type d'un nœud dans les fonctions `translate`.

Tristan est repassé sur le `Writer`. `writeFunction` n'a pas changé à part qu'il y a trois versions. On outre, `writeHeader` permet d'écrire toutes les fonctions natives (`print_i`, etc.). `writeMain` permet d'écrire juste avant le début du code de la première TDS afin d'effectuer les allocations de tests.