Projet de compilation

Modules PCL1

Deuxième partie

Janvier 2019 - Avril 2019

Membres :

Nathan BARLOY
Quentin MÉRITET
Lucas MINÉ
Guillaume VANNESSON

Superviseur :

Sébastien DA SILVA
Pierre MONNIN

Table des matières

Introduction

Le projet de compilation présenté dans ce rapport s'inscrit dans la continuité du module de Traduction dont l'enseignement est prodigué par Madame Suzanne Collin et dont les travaux dirigés sont encadrés par Madame Suzanne Collin et Monsieur Sébastien Da Silva. Ce projet a pour objectif de développer un compilateur CLIFF (Compilateur d'un Langage Informatique au Fonctionnement Fonctionnel) du langage Tiger, langage décrit par Andrew Appel.

L'équipe ayant collaboré sur ce projet est constituée de quatre élèves de deuxième année à TÉLÉCOM NANCY en les personnes de Nathan BARLOY, Quentin MÉRITET, Lucas MINÉ et Guillaume VANNESSON.

Ce rapport présente la démarche mise en oeuvre et les tâches réalisées dans cette deuxième et dernière partie du projet afin d'obtenir un compilateur CLIFF du langage Tiger dont les contrôles sémantiques et la génération de code.

1 Tables des symboles

1.1 Organisation

Pour gérer les tables des symboles, deux packages on étés créés :

— Un package contenant toutes les classes relatives aux tables des symboles.

— Un package regroupant les classes relatives aux identificateurs (type, paramètres, variables, etc). Il inclut un package s'occupant de la gestion des fonctions.

1.1.1 Le package de tables des symboles

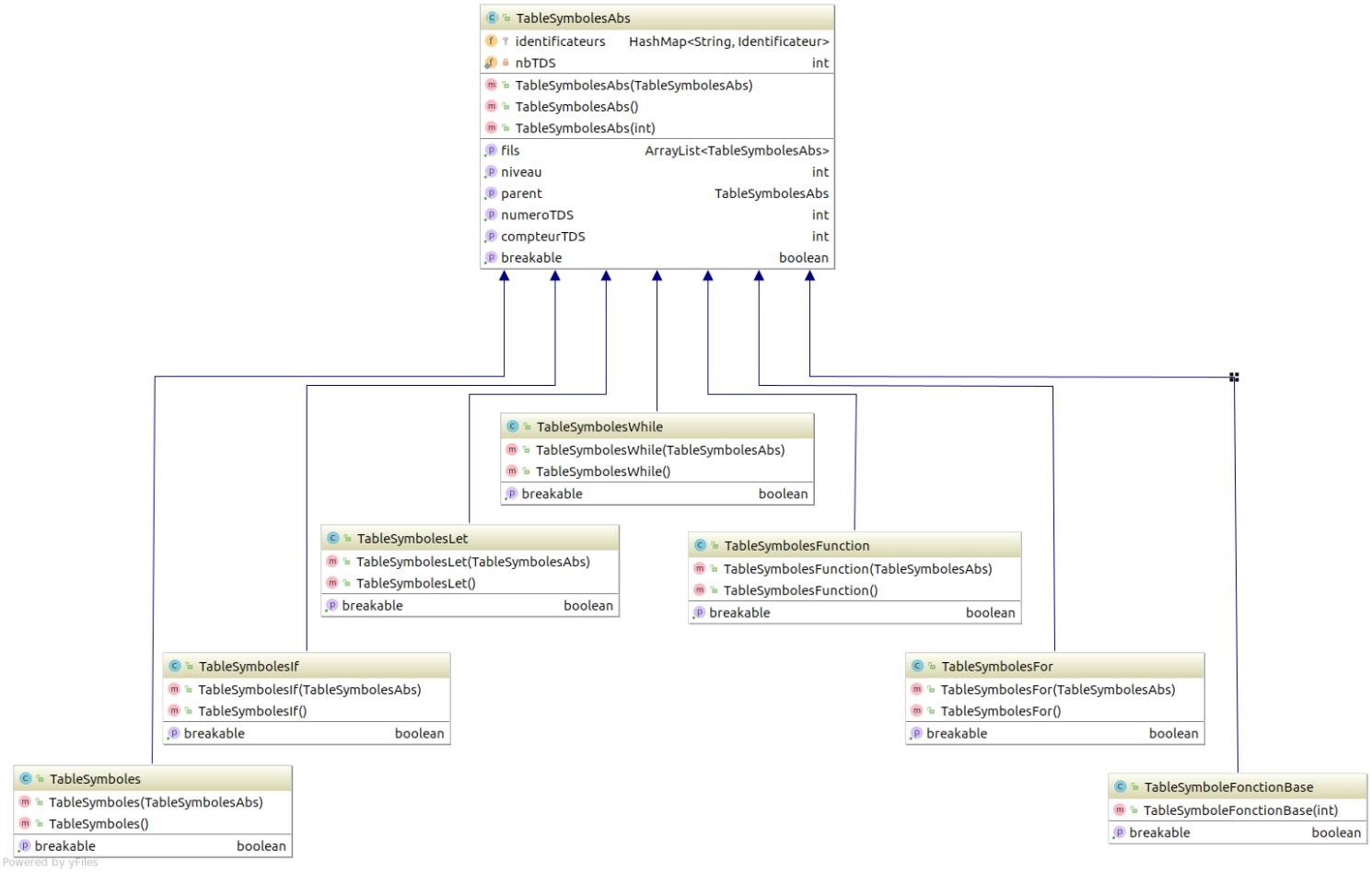

FIGURE 1 – Package tableSymbole

Ce package suit l'implémentation ci-dessus (les méthodes ont été omises pour des raisons de lisibilité). L'idée est d'avoir une classe spécifique par type de bloc, chaque classe héritant de la classe abstraite $tableSymbolesAbs$. Ceci nous servira lors de la génération de code.

1.1.2 Le package identificateur

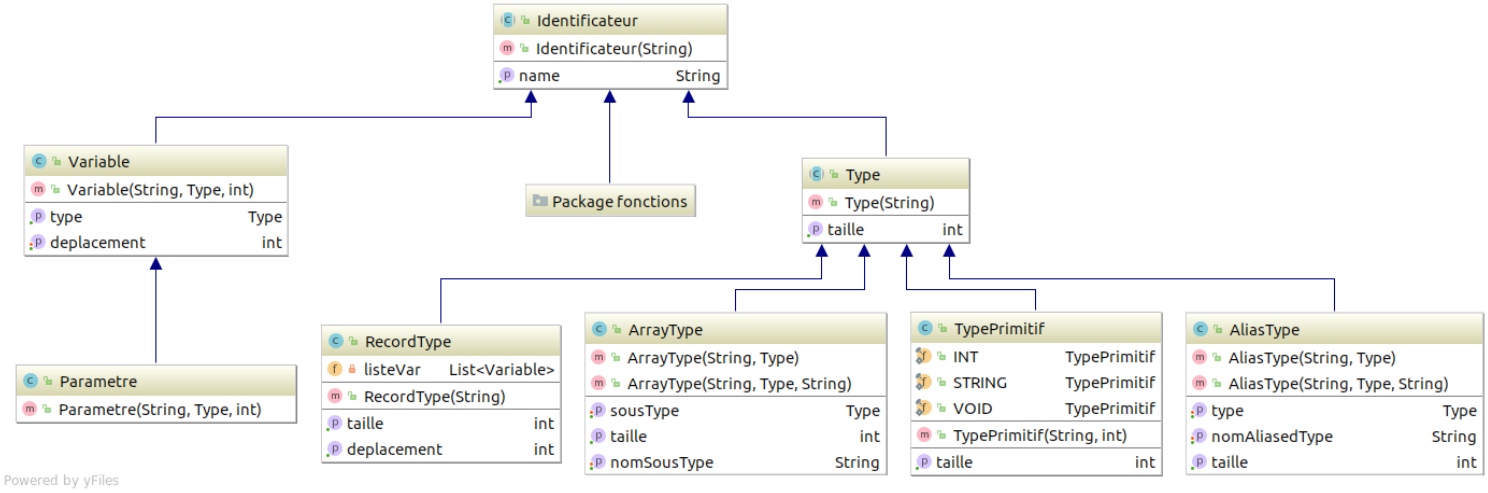

FIGURE 2 – Package identificateur

Ce package nous permet de gérer les différentes déclarations (type, variables, fonctions, ect) et d'effectuer efficacement les contrôles sémantiques associés.
Le package $fonctions$ contient une classe $fonction$ dont héritent une classe gérant les fonctions définis par l'utilisateur et une classe par fonction pré-définie.

1.2 Création de la table des symboles

Pour établir la tables des symboles d'un programme, nous parcourons récursivement l'arbre syntaxique de ce dernier précédemment établit. La fonction $parcoursArbre$ correspond essentiellement à un $switch$ où chaque $case$ correspond à un token.

À l'initialisation, la fonction est lancée avec en paramètre un objet $TableSymboles$ qui correspond à la TDS de niveau 0. Puis lorsqu'un token correspondant à un bloc (let, fonction, bloc anonyme) est rencontré, un nouvel objet $tableSymboles$ est instancié avec pour parent la table des symboles actuelle. Le parcours continue ensuite avec la nouvelle table des symboles.

Lors de ce parcours, chaque noeud implique un traitement spécifique. Par exemple, lorsqu'un noeud $VARDEC$ (correspondant à une déclaration de variable) est rencontré, un objet $Variable$ va être ajouté à la TDS. Cet objet contiendra le nom, le type, et le déplacement de la variable. Un raisonnement similaire est appliqué lors de la déclaration de fonction, type et autres identificateurs.

Dans un soucis d'optimisation, les contrôles sémantiques sont réalisés parallèlement à la création de la TDS lors du parcours de l'AST.

2 Contrôles sémantiques

Pour effectuer les contrôles sémantiques on récupère l'AST du fichier spécifié au compilateur et on ajoute les types et les fonctions de base à la TDS de niveau 0. Une fois initialisé, nous parcourons l'arbre noeud par noeud et nous les traitons.

2.1 Erreurs sémantiques traitées

Voici les contrôles sémantiques détaillés mis en place :

2.1.1 Déclaration

Nous avons effectué les contrôles suivants sur les déclaration :

- Déclaration d'une variable avec type et initialisation
- Déclaration d'un type déjà déclaré
- Déclaration d'un identificateur qui existe déjà dans la TDS actuelle
- Existence du type lors de l'ajout d'une variable
- Existence du type de retour lors de la déclaration d'une fonction

2.1.1.a Déclaration des types

Dans une première version de nos contrôles, On vérifiait seulement que la déclaration d'un type n'entrait pas en contradiction avec les autres types déjà existant : le nom du nouveau type ne devait pas déjà être utilisé, et les types utilisés dans la définition du nouveau type devaient exister. Cependant, cela empêchait de créer des types récursifs, puisqu'il apparaît dans sa propre définition : le type n'existe pas encore lorsqu'on le crée. Nous avons donc vérifié ce qui était marqué dans la spécification de Tiger.
Selon la spécification, il faut introduire une notion de *bloc de déclaration de type*, dans lequel plusieurs types sont déclarés en même temps. Ces blocs sont simplement constitués de déclarations de type qui sont faites directement l'une à la suite de l'autre. Ainsi, il faut considérer que tous les types définis dans un seul bloc existent au moment de leur définition, et qu'on peut les utiliser dans ces définitions. Cela permet aussi d'introduire des structures mutuellement récursives (le typeA est défini a partir du typeB, et le typeB est défini à partir du typeA). Cependant, on peut aussi voir un problème apparaître : il est maintenant possible de définir des types qui ne peuvent pas exister car ils ne sont jamais définis à partir de types existant, par exemple typeA = typeB puis typeB = typeA. Pour éviter cela, il faut donc vérifier qu'un type déjà existant est bien utilisé au bout d'un moment lors de la déclaration d'un nouveau type.

2.1.2 Cohérence des types

Nous avons effectué la cohérence des types pour les cas suivants :

- Affectation d'une variable avec une autre variable ou constante
- Affectation d'une variable avec un retour de fonction
- Cohérence pour les opérations (+, -, *, /) : que des entiers
- Cohérence pour les opérations logiques (&, |) : que des entiers
- Cohérence pour les comparateurs (=,<>) : même type pour chaque opérande
- Cohérence pour les comparateurs (<, <=, >, >=) : entiers ou strings de chaque côtés

2.1.3 IDBEG

Pour le cas d'IDBEG, nous vérifions le nombre de ses fils ainsi que la nature de son fils droit.
Nous pouvons donc séparer en plusieurs cas : soit IDBEG a un seul fils auquel cas c'est une variable, il faut donc vérifier qu'elle soit bien déclarée, soit IDBEG a deux fils auquel cas nous faisons un *switch case* sur son fils droit :

- **EXPBEG** : l'identificateur doit être de type *array* et l'index de type *int*. Nous faisons alors un *switch case* sur le fils droit de EXPBEG s'il existe. Nous avons les quatre choix possibles suivants :
  - **BRACBEG** : le fils de BRACBEG doit être de même type que les éléments du tableau
  - **EXPSTOR** : l'expression doit être du type *int*
  - **IDSTOR** : L'*id* doit avoir le nom d'un champ du record
  - **assignment** : les types doivent correspondre
- **FIELDEXP** : l'expression de base doit être de type *record* et l'*id* doit nommer un champ du record. Nous faisons alors un *switch case* sur le fils droit de FIELDEXP s'il existe. Nous avons trois des quatre cas précèdent : EXPSTOR, IDSTOR et assignment. Nous faisons les mêmes contrôles que dans le cas avec EXPBEG.
- **RECCREATE** : l'*id* doit être de type *record* et l'ordre, le nom et le type des champs doivent correspondre
- **ASSIGNMENT** : les types doivent correspondre
- **CALLEXP** : L'*id* doit être une fonction, le nombre et le type des paramètres doivent correspondre à la définition

2.1.4 IFTHEN

On doit vérifier que le type de la condition est de type *int*.
Il y a deux cas possible : soit on a un *else*, soit on en a pas. S'il y a présence d'un *else*, il doit y avoir une concordance des types entre le corps du *then* et le corps du *else*. Dans l'autre cas, il faut vérifier que le type est *void*.

2.1.5 NEGATION

On doit vérifier que l'opérande est de type *int*.

2.1.6 BREAK

On doit vérifier que l'on est bien dans une boucle *for* ou dans un *while*. Pour cela, la méthode *isBreakable*() est appelée sur chaque table des symboles parentes jusqu'à en avoir une *breakable*. Si aucune ne respecte cette condition, on a une erreur sémantique.

2.1.7 nil

Il est possible d'affecter un record avec la valeur nil.
Si l'un des paramètres d'une fonction est de type record, il est possible d'utiliser nil lors de l'appel.

2.2 Implémentation des tests

Afin de s'assurer que l'analyse sémantique fonctionne, nous avons écrit des tests fonctionnels et des tests non fonctionnels.
Nous avons choisis d'écrire un fichier par cas testé afin de nous simplifier la tâche. Ces fichiers sont classés suivant le contrôle à effectuer puis s'il est fonctionnel ou non fonctionnel.

2.3 Problèmes rencontrés

Les cas qui nous ont posé le plus de difficultés sont les cas des tableaux et structures. En effet, les règles de réécritures dans ces cas là ne sont pas optimales et rendent le code plus compliqué et nécessite de récupérer des informations sur des noeuds plus éloignés dans l'arbre.

3 Génération de Code

La génération de code est réalisée en parcourant l'AST et en s'appuyant sur les TDS construites lors des contrôles sémantiques. Le générateur de code utilise deux StringBuilder entre lesquels il commute en fonction de s'il génère du code pour le code principal ou pour le code des fonctions. Le builder dans lequel le générateur écrit à un instant T sera décrit comme le "builder actuel".

3.1 Factorisation du traitement des conditions et des expressions

Cette section décrit le code qui est à générer à plusieurs endroits de façon identique et qui a donc donné lieu à des méthodes dédiées à générer ce dernier.

3.1.1 comparaison()

Cette méthode permet de traiter les conditions lors des if et des $while$. Son but est de charger les registre R1 et R3 avec les valeurs souhaitées. Le contenu du registre R1 sera ensuite comparé à celui de R3 dans la suite du code.
La méthode utilise deux booléens. Un pour indiquer si un opérateur de comparaison est présent. L'autre pour indiquer si l'on doit charger dans R1 ou dans R3.

Le fonctionnement est basé sur un $switch$ gérant les cas où l'expression est un entier, une variable, un appel de fonction ou une comparaison.

Si aucun opérateur de comparaison est présent, on chargera R1 avec la valeur de l'expression et R3 avec 0. Dans le cas échéant, on appellera la méthode $comparaison()$ sur les fils gauche et droit avec les bonnes valeurs pour les booléens.

Dans le cas d'un opérateur logique, les registres R1 et R3 seront chargés avec 1 ou 0 après évaluation des expressions.

3.1.2 traiterCondition()

La méthode traiterCondition() utilise une structure de contrôle de type $switch$ afin d'ajouter au builder actuel le branchement conditionnel relatif court correspondant au texte porté par le noeud. Par exemple, pour un noeud passé en paramètre portant le texte "=", la méthode ajoutera "BEQ" au builder actuel.

3.1.3 debutBloc()

Cette méthode met en place un nouveau bloc en calculant le nombre de chaînage à remonter dans la TDS. Elle génère ensuite le code calculant le chaînage statique à l'aide du nombre de remontées à réaliser calculé précédemment, puis génère le code sauvegardant les registres, le code qui empile le chaînage dynamique, le chaînage statique et met à jour le base pointer.

3.1.4 finBloc()

Cette méthode génère le code qui permet de dépiler le chaînage statique, puis de restaurer la valeur du base pointer du bloc imbriquant et de restaurer les registres à leur valeur avant l'entrée dans le bloc dont on est en train de sortir.

3.1.5 recupererAdresseVariable()

Cette méthode prenant en paramètre une variable (correspondant à une variable définie dans une TDS) génère dans le builder actuel le code nécessaire à récupérer l'adresse de la variable dans le registre R2. Tout d'abord, on détermine le nombre de chaînage à remonter selon la formule $n_x - n_y$ ou n_x correspond au niveau d'imbrication de la TDS dans laquelle on se trouve et n_y correspond au niveau d'imbrication de la TDS de déclaration de la variable. On utilise la méthode calculerChainageStatique() pour calculer le nombre de chaînages statiques à remonter s'il faut en remonter. Une fois le chainage calculé, que l'on ai remonté des chaînages ou pas, on sait que WR contient l'adresse de la base du bloc de déclaration de

la variable passée en paramètre. On va ensuite ajouter à l'adresse contenue dans WR le déplacement correspondant à la variable que l'on veut récupérer. Si le déplacement est positif (variable locale), on ajoute $-(deplacement + 4)$, $+4$ permettant de passer le chaînage statique et de se placer sur le bit de poids fort de la variable. Si le déplacement est négatif (cas d'un paramètre), on ajoute $deplacement + 2$, $+2$ permettant de passer le chaînage dynamique.

3.1.6 calculerChainageStatique()

Cette méthode commence par générer le code qui charge le contenu du base pointer dans le working register. Si le nombre de chaînage est supérieur à 0, on génère le code qui met le nombre de chaînage à remonter dans R10. On met ensuite en place une boucle dans laquelle on va se déplacer depuis l'adresse contenue dans le working register pour se placer sur le chaînage statique, puis on va charger dans le working register l'adresse pointée par le working register. On soustrait ensuite 1 à R10 et on recommence la boucle s'il reste des chaînages à remonter. Que des chaînages statiques aient étés remontés ou non, cette méthode place la valeur du base pointer recherché dans le working register.

3.2 Noeuds de l'AST produisant du code

La production de code est réalisée à l'aide d'une fonction récursive appelée parcourirArbre(). Elle repose sur une structure de contrôle $switch$ qui implémente un "case" pour chaque noeud générant du code. Les noeuds générant du code sont les suivants :

3.2.1 FUNDEC

Dans le cas d'une déclaration de fonction, on commence par enregistrer le builder actuel, puis on définit un nouveau StringBuilder actuel, dans lequel on ajoute le nom de la fonction, le code qui empile les chaînages dynamique et statique et le code qui sauvegarde les registres (c'est l'instruction JSR, ajoutée lors de l'appel correspondant au noeud CALLEXP décrit dans la section 3.2.6.e qui s'occupe d'empiler l'adresse de retour), puis on réalise un parcours récursif (à l'aide de parcourirArbre()) sur le noeud qui contient le code de la fonction pour générer ce dernier.

Une fois ce parcours récursif terminé, on ajoute dans le builder actuel le code de fin de fonction, c'est à dire le code qui restaure les registres, qui supprime le chaînage statique, restaure le base pointer et l'instruction RTS qui dépile l'adresse de retour et replace le PC à cette adresse. Finalement, on ajoute tout le code contenu dans le builder actuel au builder qui contient le code des fonctions et on redéfinit le builder actuel comme le builder enregistré au début.

3.2.2 LET / SEQEXP

Les cas LET et $SEQEXP$ ont le même comportement excepté que, lors d'un LET, la table des symboles courante est mise à jour.

Le fonctionnement est le suivant :
On lance la méthode $parcourirArbre()$ sur l'ensemble des fils du noeud. De plus, lorsque l'on est dans le cas du dernier fils, le parcours de l'arbre aura chargé l'évaluation de l'expression dans R1. Afin de gérer la valeur de retour de ces blocs, on génère le code qui recopie le registre R1 dans R0.

3.2.3 WHILE

Dans le cas d'une boucle while, on définit un nouveau bloc (en utilisant la méthode nouveauBloc() décrite dans la section 3.1.3). On ajoute ensuite dans le builder actuel le nom du bloc (qui contient surtout le numéro de bloc, nécessaire pour s'assurer que les sauts absolus reviendront bien à cet endroit dans le code). On ajoute ensuite tout le code qui va réaliser la vérification de la condition à l'aide de comparaison() (section 3.1.1) et traiterCondition() (section 3.1.2). Si la condition est respectée, on effectue un branchement relatif court qui va brancher après le saut absolu et on va donc effectuer le code de la boucle. Si la condition

n'est pas respectée, le branchement relatif court n'est pas réalisé et on réalise un saut absolu directement à la fin du code de la boucle (au label "fin" auquel est concaténé le nom du bloc).

Le code à l'intérieur de la boucle est généré à l'aide de parcoursArbre() sur le noeud contenant le code de la boucle. Si le code de la boucle n'est pas contenu dans un LET ou un SEQEXP, on stocke en plus la valeur de R1 dans R0 car on est dans le cas d'un retour de fonction).

A la fin du code de la boucle se trouve un saut absolu qui revient au début du code de la boucle pour vérifier la condition. Après cette ligne est ajouté une ligne contenant "fin" auquel est concaténé le nom du bloc : c'est ici que l'on sautera si la condition au début du bloc n'est pas respectée. On restaure enfin le bloc précédent à l'aide de la méthode finBloc() (section 3.1.4).

3.2.4 FOR

Dans le cas d'une boucle for, on définit un nouveau bloc (en utilisant la méthode nouveauBloc() décrite dans la section 3.1.3). On évalue ensuite la borne inférieure de la boucle et on stocke sa valeur dans un registre de boucle, puis on évalue la borne supérieure et on stocke sa valeur dans un registre maximum de boucle. On ajoute ensuite dans le builder actuel le nom du bloc (qui contient surtout le numéro de bloc, nécessaire pour s'assurer que les sauts absolus reviendront bien à cet endroit dans le code). On ajoute enfin le chargement de la valeur de la variable qui porte la boucle dans le registre de boucle. S'en suit la génération du code à l'intérieur de la boucle à l'aide de parcourirAbre().

Une fois tout le code à l'intérieur de la boucle généré, on ajoute le code qui incrémente le registre de boucle, puis le code qui évalue la condition du for. Si la condition n'est pas vérifiée, le saut conditionnel relatif n'est pas réalisé et on passe alors sur un saut absolu qui nous ramène au début de la boucle. Si la condition est vérifiée, alors le branchement conditionnel relatif est réalisé, le saut absolu n'est pas réalisé et on sort de la boucle. On ajoute alors le code qui dépile la variable portant la boucle puis on ajoute un label de fin de bloc (voir 3.2.9) et enfin le code qui permet de sortir du bloc à l'aide de la méthode finBloc() (3.1.4).

3.2.5 VARDEC

Dans le cas d'une déclaration de variable, on commence par évaluer la valeur de l'expression à laquelle on veut assigner la variable en utilisant parcoursArbre(), sur le deuxième noeud dans le cas ou le type n'est pas précisé et sur le troisième noeud si le type est précisé. Enfin, on récupère dans la TDS le type de la variable que l'on déclare et on génère le code qui va décaler le stack pointer de la taille du type. Enfin, on empile le contenu du registre qui contient l'évaluation de l'expression réalisée dans un premier temps.

3.2.6 IDBEG

3.2.6.a Cas d'une variable

Dans le cas d'une variable, on récupère la variable dans la TDS et on utilise la fonction recupererAdresseVariable() (section 3.1.5). Enfin, on charge la valeur dans un registre.

3.2.6.b Cas EXPBEG

Ce cas correspond à un tableau. Les opérations à effectuer dépendent des fils de ce noeud.

— **Un seul fils :** On souhaite accéder à une valeur du tableau. Le fils de $EXPBEG$ est l'indice auquel on souhaite accéder. On génère donc le code pour récupérer l'adresse du tableau dans la pile. La valeur de cette adresse contient en fait un pointeur vers le tas. On ajoute ensuite l'index multiplié par deux à cette valeur pour avoir l'adresse mémoire qui nous intéresse. Finalement, on génère le code qui charge le contenu de l'adresse que l'on vient d'obtenir dans R1.

— **Deux fils :**

— **BRACBEG :** On est en train de déclarer un tableau. On récupère donc sa taille, on sauvegarde l'adresse actuelle du tas dans R1 (qui sera utilisée par le noeud $VARDEC$ section 3.2.5) et on augmente l'adresse du tas de la taille du tableau.

— **ASSIGNMENT :** On récupère l'adresse de l'élément du tableau que l'on souhaite affecter et on la stocke dans R3. On évalue ensuite l'expression correspondant à la valeur affectée en lançant $parcourirArbre$ sur le fil du noeud $ASSIGNMENT$. On a alors cette valeur chargé dans R1. Enfin, on affecte la valeur contenue dans R1 à l'adresse contenue dans R3.

3.2.6.c Cas FIELDEXP

Dans ce cas, on réalise une opération d'accès à un champ ou une affectation d'un champ d'une structure.
L'opération effectuée dépend du nombre de fils :

— **Un seul fils :** On souhaite accéder au champ. On récupère dans la pile l'adresse de la variable correspondant à la structure. Le contenu de cette adresse est un pointeur vers le tas. A partir de la TDS, on récupère la position du champ auquel on veux accéder que l'on ajoute à l'adresse précédente. Puis, on charge la valeur du champ dans R1.

— **Deux fils :**On souhaite affecter une valeur à un champ. On récupère cette valeur dans R1 en appelant $parcourirArbre$ sur l'expression correspondante. On récupère ensuite l'adresse du champ dans le tas que l'on stocke dans R2. Enfin, on charge la valeur de R1 à l'adresse contenu dans R2.

3.2.6.d Cas RECCREATE

On est dans le cas d'une déclaration de structure. On initialise chaque champ dans le tas puis on met l'adresse initiale du tas dans R1 (que $VARDEC$ utilisera, section 3.2.5).

3.2.6.e Cas d'un appel de fonction

Dans le cas d'un appel de fonction, on commence par générer le code qui va empiler les paramètres à l'aide de parcourirArbre() qui va prendre en paramètre les noeuds correspondants aux expressions à évaluer comme paramètre. Une fois les paramètres empilés, on utilise la fonction calculerChainageStatique() (section 3.1.6) qui va calculer le chaînage statique pour la fonction que l'on va appeler et on le stocke dans le WR (qui sera empilé au début du code de la fonction). Dans ce cas, il est nécessaire de remonter $n_x - n_y + 1$ où n_x correspond au niveau d'imbrication de l'appelant et n_y au niveau d'imbrication de la fonction appelée.

Une fois le calcul du chaînage statique réalisé, on ajoute au code généré l'instruction JSR @nomFonction qui va s'occuper d'empiler l'adresse de retour et déplacer le PC à l'adresse de la fonction dans le code assembleur.

On ajoute ensuite le code qui correspond à la suite de l'appel de la fonction, c'est à dire une fois que la fonction est terminée, en dépilant les paramètres. Pour finir, on déplace le contenu du registre de retour de fonction R0 dans le registre d'évaluation d'expression.

3.2.6.f Cas d'une assignation

Dans le cas d'un assignation, on commence par générer le code qui va évaluer l'expression à assigner et stocker le résultat dans un registre. Ensuite, on génère le code qui va récupérer l'adresse de la variable à laquelle on veut assigner le résultat de l'évaluation de l'expression dans un registre, puis on génère le code qui va stocker le contenu du registre qui contient le résultat de l'évaluation de l'expression à l'adresse contenue dans le registre où l'on a récupéré l'adresse de la variable.

3.2.7 NEGATION

Dans le cas d'une négation, on génère simplement le code qui va évaluer l'expression et stocker sa valeur dans un registre, puis on va générer le code qui va réaliser la différence entre 0 et l'évaluation de l'expression afin d'obtenir son opposé.

3.2.8 IFTHEN

Dans le cas d'un if, on définit un nouveau bloc (en utilisant la méthode nouveauBloc() décrite dans la section 3.1.3). On génère ensuite tout le code qui va réaliser la vérification de la condition à l'aide de comparaison(section 3.1.1) et traiterCondition(section 3.1.2). A partir de là, il est nécessaire de distinguer deux cas : le cas où un else est présent et le cas où il n'y a pas de else :

— Cas où un else est présent : Si la condition n'est pas respectée, le branchement relatif court n'est pas réalisé et on réalise un saut absolu directement à la partie else (au label "else" auquel est concaténé le nom du bloc). Si la condition est respectée, on effectue un branchement relatif court qui va brancher après le saut absolu et on va donc effectuer le code du if (code se trouvant au label "then" auquel est concaténé le nom du bloc).

— Cas où il n'y a pas de else : Si la condition n'est pas respectée, le branchement relatif court n'est pas réalisé et on réalise un saut absolu directement à la partie fin (au label "fin" auquel est concaténé le nom du bloc). Si la condition est respectée, on effectue un branchement relatif court qui va brancher après le saut absolu et on va donc effectuer le code du if (code se trouvant au label "then" auquel est concaténé le nom du bloc).

A partir d'ici, peu importe qu'un else soit présent ou non, le code généré est le même puisqu'il correspond au then : on ajoute le label "then" auquel est concaténé le nom du bloc, puis on génère le code correspondant au then à l'aide de parcourirArbre(). Si le code correspondant au then n'est pas contenu dans un LET ou un SEQEXP, on stocke en plus la valeur de R1 dans R0 car on est dans le cas d'un retour de fonction). A la fin du bloc then, on ajoute enfin un saut absolu vers la fin du if.
On génère ensuite le code du bloc else s'il est présent : on ajoute au code généré le label "else" auquel est concaténé le nom du bloc. On génère ensuite récursivement le code contenu dans le else à l'aide de parcourirArbre(). Si le code correspondant au else n'est pas contenu dans un LET ou un SEQEXP, on stocke en plus la valeur de R1 dans R0 car on est dans le cas d'un retour de fonction).
Enfin, on ajoute le label "fin" auquel on concatène le nom du bloc, puis on gènere le code qui permet de sortir du bloc à l'aide de la méthode finBloc() (section 3.1.4).

3.2.9 break

Dans le cas où l'on rencontre un noeud portant le texte break, il est nécessaire de déterminer le nombre de niveau d'imbrication du bloc actuel à l'intérieur de la boucle que l'on veut break. On remonte donc les TDS mères jusqu'à en obtenir une *breakable*. On génère ensuite le code de fin de bloc autant de fois que de bloc qu'il est nécessaire de finir (soit autant de fois que d'imbrications). Finalement, on génère le code qui permet de sauter à la fin du bloc que l'on "break" en ajoutant au code généré l'instruction JEA @labelFinBloc. En effet, comme expliqué dans les sections 3.2.4 et 3.2.3, un label $finwhile$ et $finFor$ est généré à la fin du code correspondant au for et au while.

3.2.10 Opérateurs numérique(+, -, *, /)

Dans le cas d'un opérateur, on va générer le code qui correspond à l'évaluation du fils gauche à l'aide de parcourirArbre() et empiler le résultat de l'évaluation, puis on va évaluer le fils droit de la même façon et empiler le résultat. Ensuite on va dépiler dans deux registres différents les deux évaluations et réaliser l'opération correspondant à l'opérateur porté par le noeud, puis stocker le résultat dans R1.

3.2.11 INT

Dans le cas d'un entier seul, on va simplement générer le code qui charge dans un registre la valeur de cet entier.

3.2.12 STRING

Dans le cas d'une chaîne de caractère, on va utiliser l'instruction assembleur *string* afin de créer une chaîne de caractère. On ajoute l'instruction de création de la chaîne de caractère

avant le main dans le code. Une fois la chaîne crée, on charge l'adresse de cette chaîne dans un registre.

3.2.13 Opérateurs logique (=, <>, >, <, >=, <=)

Le cas des opérateurs logiques se rapproche du cas des opérateurs numériques : on commence par générer le code qui va évaluer le fils droit, puis on génère le code qui charge le résultat de l'évaluation contenu dans R1 dans R3. Ensuite, on génère le code qui va évaluer le fils gauche et stocker le résultat dans R1. On génère ensuite le code qui réalise la comparaison entre l'évaluation du fils gauche et l'évaluation du fils droit puis on génère un saut conditionnel correspondant à l'opérateur à l'aide de traiterCondition() (section 3.1.2). Si la condition est vérifiée, un saut conditionnel relatif permet de sauter à une instruction stockant 1 dans R1. Le cas échéant, on stocke 0 dans R1 puis on réalise un branchement relatif non conditionnel pour sauter l'instruction qui stocke 1 dans R1.

3.3 Problèmes rencontrés et solutions

3.3.1 Imbrication de boucles for

Afin de pouvoir imbriquer des boucles for (section 3.2.4), il est nécessaire d'enregistrer les valeurs des registres de début et de fin de boucle avant de réaliser une nouvelle boucle et de les restaurer après avoir réalisé la boucle. Sans cela, la boucle imbriquée va modifier les valeurs d'indices de la boucle imbriquante et perturber le bon déroulement de cette dernière.

3.3.2 Sauts conditionnels relatifs longs

Étant donné que l'assembleur utilisé ne gère pas bien les sauts conditionnels longs, nous avons du modifier la première version que nous avions réalisée afin d'utiliser des sauts absolus (à l'aide de l'instruction JEA). Le souci est résolu assez facilement et facilite même les contrôles puisqu'au lieu d'utiliser la condition inverse, on vérifie la condition et on saute simplement l'instruction de saut absolu si cette dernière est vérifiée.

4 Jeux d'essais

4.1 Cas traités

Dans cette section nous allons détaillé les différents cas que nous avons traités, séparés en deux parties : la première partie correspondra aux jeux d'essais des cas traités par l'analyse sémantique, la deuxième partie correspondra aux jeux d'essais des cas traités par la génération de code.

4.1.1 Cas de l'analyse sémantique

Voici le fichier avec tous les cas fonctionnels traités par l'analyse sémantique (fichier présenté à notre encadrant lors de l'évaluation).

```
1  let
2      type vector = { x : int, y : int }
3      type data = { v : vector, to_add : int }
4      type dataArray = array of data
5
6      var d1 := data { v = vector { x = 0 , y = 0} , to_add = 1}
7      var d2 := data { v = vector { x = 1 , y = 1} , to_add = 1}
8      var d3 := data { v = vector { x = 2 , y = 2} , to_add = 0}
9      var d4 := data { v = vector { x = 3 , y = 4} , to_add = 1}
10
11     var l := dataArray [4] of nil
12     var result := vector { x = 0 , y = 0}
13     var testVar : int := 0
14     var testVar2 : int := testVar
15     var sizeOfString : int := size("Test")
16     var testVarFois2 : int := testVar+testVar2
17     var testVarCarre : int := testVar*testVar
18     var testVarMoins := testVar-testVar
19     var testLog := testVar & testVar
20     var testLog2 := testVar | testVar
21     var testEqual := testVar = testVar
22     var testDiff := testVar <> testVar
23     var testSup := testVar > testVar-1
24     var testInf := testVar-1 < testVar
25
26     function fibonacci ( generations : int ) : int =
27     let
28         function computation ( generations : int , previous : int , pprevious : int ) : int =
29             if generations = 0 then
30                 previous
31         else
32             computation ( generations - 1 , previous + pprevious , previous )
33     in
34         if generations = 0 | generations = 1 then
35             generations
36         else
37             computation ( generations - 2 , 1 , 0)
38     end
39
40
41     function add_vectors ( l : dataArray , size : int , result : vector ) =
42         for i := 0 to size - 1 do
43             if l[i].to_add then
44             (
45                 result.x := result.x + l[i].v.x ;
46                 result.y := result.y + l[i].v.y
47             )
```

```
48      function return1 () : int =
49          1
50  in
51      l[0] := d1;
52      l[1] := d2;
53      l[2] := d3;
54      l[3] := d4;
55
56      while testVar < 1
57      do
58      testVar := testVar + 1;
59
60      if testVar = 1
61      then testVar :=2;
62
63      if testVar = 2
64      then testVar := 3
65      else testVar := testVar*2;
66
67      if testVar = 2
68      then 1
69      else 0;
70
71      testVar := -testVar;
72
73      while 1
74      do
75      (
76      print("coucou");
77      break);
78
79
80      for i := 0 to 10
81      do
82          (
83          print("coucou");
84          break);
85
86      add_vectors (1, 4, result) ;
87      print(result.x) ;
88      print(result .y);
89
90      print ( fibonacci (12) )
91  end
```

Voici le fichier avec tous les cas non fonctionnels traités par l'analyse sémantique.

```
 1  let
 2          var total : int := 0
 3          var total : int := 1
 4          var texte : string := total
 5          var oeoe : string := "oeoe"
 6          var a := nil
 7
 8          type record = { x : int }
 9
10          // mauvais type dans une structure
11          var rec := record { x = "coucou" }
12
13          type dataArray = array of int
```

```
14          var l := dataArray [4] of nil
15
16          // function type error
17          function maFonction() = 3+4
18
19          // dec function
20          function maFonction1 () : string =
21                  (
22                          total := 1
23                  )
24
25          // redef function
26          function maFonction1 () : int = 3+4
27
28
29          function add_vector ( l : dataArray , size : int ) : int = 3
30
31          // arrayType dec
32          var variable := 2
33          type type1 = array of int
34          type type2 = array of inexistant
35          type type3 = array of variable
36
37          // recordType dec
38          type type4 = { a : inexistTyp, b : int}
39
40  in
41          // sup not int
42          for i := 0 to oeoe do
43          (
44                  x := 3;
45                  i := 4
46          );
47
48          // assigmnent different type
49          oeoe := 0;
50
51          3 < "ok";
52          3 <> "string";
53          3 + "ok";
54
55          // not defined
56          char := total;
57
58          // if different type
59          if 4
60          then
61                  5
62          else
63                  "coucou";
64
65          // condition non int
66          if "coucou"
67          then
68                  print("coucou");
69
70          // corps de type non void if then
71          if 1
72          then
73                  2+3;
74
75          // nil dans if
76          if nil = nil
```

```
77          then
78                  print("ok");
79
80          // negation string
81          - 4;
82          - "String";
83          - oeoe;
84
85          // break or de for et while
86          break;
87
88          // non int dans condition
89          while "coucou"
90          do
91                  (
92                          print("ok")
93                  );
94
95          // Corps pas de type void while
96          while 1
97          do
98                  (
99                          3 + 2
100                 )
101
102 end
```

4.1.2 Cas de la génération de code

Le fichier utilisé pour la soutenance est le suivant :

```
1  let
2    type vector = { x : int, y : int, z : int }
3    type tableau = array of int
4    var v := vector { x = 2, y = 3, z = 5}
5    var tab := tableau[3] of int
6    var entreeUtilisateur : int := 0
7    var resultat : int := 0
8    var condition : int := 0
9    var coucou : string := "Bonjour "
10
11   function fibonacci ( generations : int ) : int =
12   let
13       function computation ( generations : int , previous : int , pprevious : int ) : int =
14           if generations = 0 then
15               previous
16       else
17           computation ( generations - 1 , previous + pprevious , previous )
18   in
19       if generations = 0 | generations = 1 then
20           generations
21       else
22           computation ( generations - 2 , 1 , 0)
23   end
24
25   function puissance ( a : int , b : int ) : int =
26     let
27       var i := 0
28       var resultat := 0
29     in
30       if b = 0 then
```

```
31          1
32        else
33        (
34          i := 1;
35          resultat := a ;
36          while i < b do
37            (
38            resultat := resultat * a ;
39            i := i + 1
40            );
41            resultat
42        )
43      end
44
45
46
47  in
48   print(coucou);
49   entreeUtilisateur := read();
50   printi(fibonacci(entreeUtilisateur));
51   print(" ");
52
53   tab[0] := puissance(2,1);
54   tab[1] := puissance(2,2);
55   tab[2] := puissance(2,3);
56   printi(tab[0]);
57   print(" ");
58   printi(tab[1]);
59   print(" ");
60   printi(tab[2]);
61   for i := 0 to 10 do
62    (
63    printi(i);
64    print(" ");
65    condition := i > 5;
66    if condition
67      then
68        (
69        print("Break");
70        break
71        )
72    );
73    if entreeUtilisateur > 5 | entreeUtilisateur = 3 & entreeUtilisateur = 6
74      then
75        v.y := v.x + v.z
76      else
77       v.x := v.x*2;
78    print(" ");
79    printi(v.x);
80    print(" ");
81    printi(v.y);
82    print(" ");
83    printi(v.z);
84    print(" ");
85    print(getchar());
86    print(" ");
87    printi(chr())
88
89
90  end
```

4.2 Limites

Tout comme pour les contrôles sémantiques, les tableaux et structures nous ont posé des problèmes. En effet, dans ces cas, l'AST n'est pas optimum et rend à nouveau le code plus compliqué que nécessaire. De plus, nous n'avons pas réussi à gérer le cas des tableaux multi-dimensionnels et des structures imbriqués. Ceci nous montre l'importance de chacune des étapes du projet. Un erreur de conception dans les premières parties se répercutera sur toutes les parties suivantes.

5 Gestion de projet

5.1 Répartition des tâches

Nom de la tâche	Nathan B.	Quentin M.	Lucas M.	Guillaume V.
Suppression récursivité à gauche	4 heures	4 heures	4 heures	2 heures
Priorité des opérateurs	6 heures	6 heures	6 heures	4 heures
Mise en place de l'AST	3 heures	4 heures	4 heures	3 heures
Améliorations et corrections	2 heures	2 heures	2 heures	2 heures
Parcours de l'AST	1 heure			1 heure
Création des TDS lors du parcours	1 heure		1 heure	1 heure
Ajout des informations dans la TDS				3 heures
Écriture de code source Tiger à tester	1 heure		3 heures	3 heures
Générisation des méthodes de test JAVA				2 heures
Rédaction du rapport	2 heures	4 heures	4 heures	3 heures
Rédaction des éléments de gestion de projet		2 heures	1 heure	1 heure
Deuxième partie du projet				
Table des symboles	2 heures		3 heures	4 heures
Contrôles sémantiques	11 heures	15 heures	12 heures	6 heures
Autre génération de code	3 heures		5 heures	4 heures
FUNDEC	3 heures		1 heure	6 heures
LET / SEGEXP			0,5 heure	

WHILE	3 heures			15 minutes
FOR				5 heures
VARDEC				2 heures
IDBEG			4 heures	12 heures
NEGATION		15 minutes		
IFTHEN			4 heures	45 minutes
break		20 minutes		
Opérateurs numériques				5 heures
INT				1 heure
STRING				1 heure
Opérateurs logiques			2 heures	
Print				1 heure
Printi	1 heure		6 heures	
Read			4 heures	
Fonctions pré-définies			5 heures	2 heures
Rédaction du rapport	2 heures	5 heures	3 heures	4 heures
Rédaction des éléments de gestion de projet		4 heures		
Total	45 heures	49,5 heures	74,5 heures	79 heures

5.2 GANTT

	Nom des tâches	Durée	Début	Fin
9	Table des symboles	59 jours	05.01.19	3.02.19
12	Ajout des informations dans la TDS	59 jours	5.01.19	3.03.19
16	Contrôles sémantiques	57 jours	7.01.19	05.03.19
17	Listing des contrôles sémantiques à effectuer	3 jours	7.01.19	10.01.19
18	Implémentation des contrôles sémantiques	28 jours	10.01.19	05.03.19
19	Génération de code	49 jours	05.03.19	23.04.19
20	Listing des noeuds produisant du code assembleur	10 jours	05.03.19	15.03.19
21	Implémentation de la génération de code	39 jours	15.03.19	23.04.19
13	Tests	103 jours	10.01.19	23.04.19
14	Écriture de code source Tiger à tester	103 jours	10.01.19	23.04.19

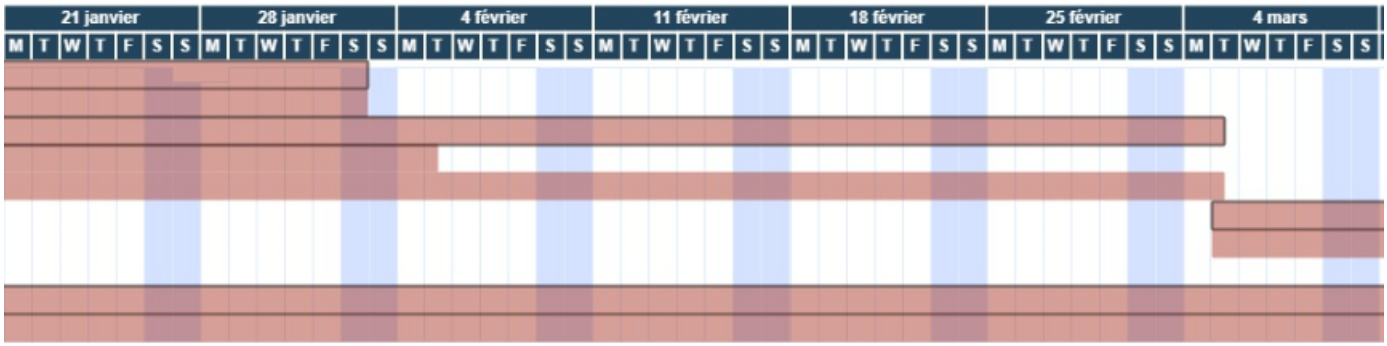

FIGURE 3 – GANTT

6 Conclusion

Dans ce rapport, nous avons donc présenté la démarche que nous avons suivie pour réaliser un compilateur fonctionnel. Au cours de ce projet, nous avons pu aborder la grosse majorité des aspects constituant un compilateur (bien que notre réalisation ne comporte pas de phase d'optimisation du code généré). Bien que ne couvrant pas la totalité des possibilités du langage (les tableaux multi-dimensionnels et les structures imbriquées n'étant pas gérés), nous avons réussi à développer un compilateur permettant d'écrire des programmes relativement complexes.
De plus, ce projet nous a permis de comprendre en détail l'ensemble des étapes d'un compilateur, leur inter-dépendance et leurs intérêts.

Compte rendu réunion

Minutes for 09/01/19

Present: Guillaume VANNESSON Nathan BARLOY Quentin MÉRITET Lucas MINÉ

Réunion tenue à TELECOM Nancy

Ordre du jour

— TDS
— Contrôles sémantiques

L'animateur de la séance est Quentin MÉRITET et le secrétaire : Lucas MINÉ. La séance a débuté à 14h.

TDS

Implémentation d'un classe $Identificateur$ pour éviter d'avoir plusieurs $hashmap$ dans les TDS. On aura donc 4 classes héritant de $Identificateur$.
La $hashmap$ aura pour clé une $String$ et pour valeur un identificateur.

Contrôles sémantiques

La liste des contrôles sémantiques à été commencée, il faut la finir pour la semaine prochaine. Il est nécessaire de définir où effectuer ces contrôles (dans la classe Main et/ou dans la classe TableSymboles). Bien penser à mettre en vert les contrôles sémantiques implémentés sur le drive.

TODO

— Finir la liste des contrôles sémantiques (deadline : 1 semaine)
— Implémenter les contrôles
— Remplir la TDS.

Fin de la réunion à 14h45.
Next Meeting: A définir en fonction des PIDR.

Compte rendu réunion

Minutes for 08/02/19

Present: Guillaume VANNESSON Nathan BARLOY Quentin MÉRITET Lucas MINÉ

Réunion tenue à TELECOM Nancy

Ordre du jour

- While
- IDBEG
- detectionTypeExp()
- Contrôles sémantiques restants

L'animateur de la séance est Quentin MÉRITET et le secrétaire : Lucas MINÉ. La séance a débuté à 10h10.

While

Il faut contrôler que le type de retour du exp de la condition est int. Le détecter avec $detectionTypeExp()$.

IDBEG

Différents cas :

- Un seul fils variable (vérifier qu'elle soit déclarée)
- Deux fils
  En fonction du fils droit :
  - EXPBEG : l'identificateur doit être de type array et l'index de type int
    Fils droit (3eme) de EXPBEG (si existe)
    - BRACBEG : Le fils de BRACBEG doit être de même type que les éléments du tableau
    - EXPSTOR : l'exp doit être de type int
    - IDSTOR : L'id doit avoir le nom d'un champ du record
  - FIELDEXP : l'expression de base doit être de type record et l'id doit nommer un champ du record
    Fils droit (3eme) de FIELDEXP (si existe)
    - EXPSTOR : l'exp doit être de type int
    - IDSTOR : L'id doit avoir le nom d'un champ du record
  - RECCREATE : l'id doit être de type record et l'ordre, le nom et type des champs doit correspondre (dans fieldCreate : id = exp)
  - CALLEXP : L'id doit être une fonction, le nombre et le type des paramètre doivent correspondre à la définition

detectionTypeExp()

Cette fonction a pour but de retourner le type d'une exp, tous les cas ne sont pas encore implémentés. Il faut la compléter.

Contrôles sémantiques restant

- IFTHEN : type de la condition : int. Si pas de else corps : type void sinon corps du then et du else de même type
- NEGATION : opérande de type int
- BREAK : ne peut être usé que dans while et for

Fin de la réunion à 12h00.
Next Meeting: Après les vacances.

Compte rendu réunion

Minutes for 21/03/19

Present: Guillaume VANNESSON Nathan BARLOY Quentin MÉRITET Lucas MINÉ

Réunion tenue à Télécom Nancy

Ordre du jour

— Génération de code

— Liste des tokens

L'animateur de la séance est Quentin MÉRITET et le secrétaire : Lucas MINÉ. La séance a débuté à 13h00.

Génération de code

Il faut refaire un parcours de l'AST avec un pointeur vers la TDS courante. A chaque token rencontré, il faut générer le code assembleur correspondant.

Il y a déjà une classe $GenerateurDeCode.java$ mais il faudra la modifier pour utiliser l'AST car elle utilise la TDS actuellement.

La logique de génération est la suivante :

Parcours de l'arbre $\rightarrow$ Rencontre d'un token $\rightarrow$ Récupération de la TDS correspondante (si besoin) $\rightarrow$ générer le code correspondant $\rightarrow$ continuer le parcours.

Liste des token

RECCREATE
FIELDEXP
IFTHEN
RECTY
ARRTY
TYDEC
ASSIGNMENT
WHILE
FOR
LET
NEGATION
VARDEC
EXPSTOR
IDSTOR
STRINGLIT
NIL
IDBEG
EXPBEG

FIELDDEC
BRACBEG
FIELDCREATE
IFTHEN
LET
SEQEXP
CALLEXP
FUNDEC
INT
STRING

Attention aux fonctions $\rightarrow$ le code ne doit pas être généré dans le $main$ mais après.

Fin de la réunion à 14h00.
Next Meeting: A définir