Projet de Compilation

Projet de compilation du langage BLOOD
-2e Rapport-

MASK
moi
MASK MASK
MASK

Année 2020-2021

Encadrants :
Suzanne COLLIN
Sébastien DA SILVA
Gérald OSTER

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : MASK

Élève-ingénieur(e) régulièrement inscrit(e) en 2e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 66666666

Année universitaire : 2020-2021

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Projet de compilation du langage BLOOD

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 10 novembre 2021

Signature :

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : moi

Élève-ingénieur(e) régulièrement inscrit(e) en 2e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 66666666

Année universitaire : 2020-2021

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Projet de compilation du langage BLOOD

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 10 novembre 2021

Signature :

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : MASK MASK

Élève-ingénieur(e) régulièrement inscrit(e) en 2e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 66666666

Année universitaire : 2020-2021

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Projet de compilation du langage BLOOD

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 10 novembre 2021

Signature :

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : MASK

Élève-ingénieur(e) régulièrement inscrit(e) en 2e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 66666666

Année universitaire : 2020-2021

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Projet de compilation du langage BLOOD

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 10 novembre 2021

Signature :

Projet de Compilation

Projet de compilation du langage BLOOD
-2e Rapport-

MASK
moi
MASK MASK
MASK

Année 2020-2021

MASK
moi
MASK MASK
MASK
mask@telecomnancy.eu
mask@telecomnancy.eu
mask
mask@telecomnancy.eu

TELECOM Nancy
193 avenue Paul Muller,
CS 90172, VILLERS-LÈS-NANCY
+33 (0)3 83 68 26 00
contact@telecomnancy.eu

TELECOM Nancy
193, avenue Paul Muller,
CS 90172, VILLERS-LÈS-NANCY
+33 (0)3 83 68 26 00

Encadrant :
Suzanne COLLIN
Sébastien DA SILVA
Gérald OSTER

Table des matières

1 Fiche projet

1.1 Auteurs

Noms	Qualité
MASK	Membre
moi	Membre
MASK MASK	Chef de projet
MASK	Membre
Professeurs	Project Owners

1.2 Cadrage

1.2.1 Finalités et importance du projet dans le cursus

Ce projet a pour but de nous confronter à la conception d'un compilateur en mobilisant les connaissances et compétences acquises dans les modules TRAD1-2 de seconde année à TELECOM Nancy. La mise en œuvre des connaissances acquises en MI1-2 (connaissances théoriques sur les compilateurs descendants, notion de grammaire LL(1), grammaire ambigüe...) nous est également demandée, ceci afin de créer une grammaire correcte et non ambiguë. Par ailleurs, la maîtrise du module de Gestion de Projet de projet est sous-entendue.

1.2.2 Objectifs et résultats opérationnels

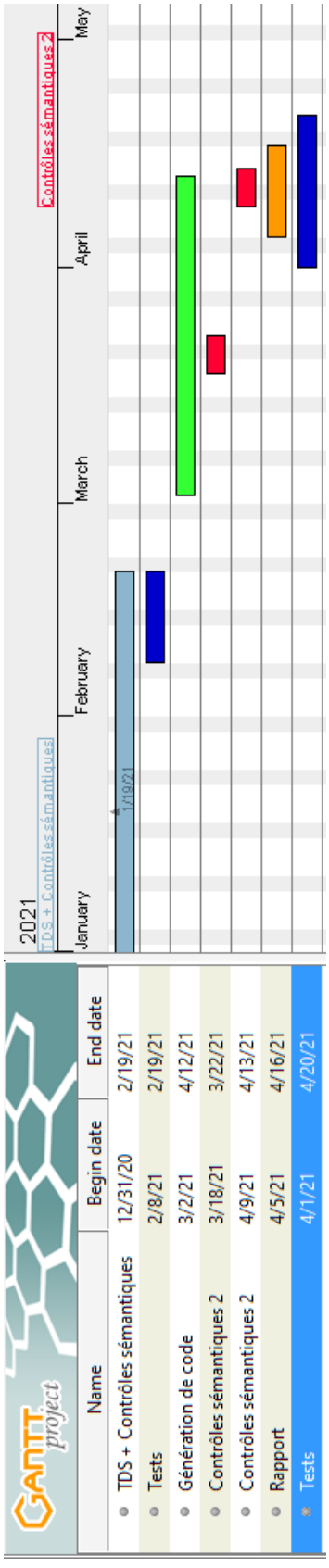
Livrable	Contraintes
Code Source	-Respecter les jalons. -Fournir une documentation (conception, notes de développement), installation, compilation, exécution, validation de vos réalisations.
Tests	-Créer et expérimenter un jeu de test varié sur la grammaire obtenue. -Le jeu de tests devra vérifier : - expressions arithmétiques (illustrant priorité et associativité des opérateurs), affectations, déclarations de variables. - if, while, avec des instructions et if / while imbriqués. - définitions de classe, constructeur, méthodes, et paramètres. - appels de méthodes.
Rapport synthétique	Éléments de Gestion de projet (Compte-Rendus de réunions, GANTT, fiche d'évaluation de la répartition du travail, répartition des tâches au sein du groupe).

Table 1.1 – Critères issus du sujet

1.3 Déroulement du projet

Numéro	Description
1 TDS	La TDS devra comporter les numéros de région et d'imbrication ainsi que le nom, le type et le déplacement de chaque variable.
2 Contrôles sémantiques	Les contrôles sémantiques devront être similaires à ce qui se fait dans un language objet (précisions sur le numéro de ligne et la tabulation, sur l'erreur en tant que telle et la variable qui l'a causée.
Génération de code	
3 Mise en place de l'environnement (tests, tasks, pipeline, passes)	L'environnement de développement tiendra compte des contraintes de développement de chacun. Nous utiliserons donc **Gradle** pour sa simplicité d'utilisation. Les passes constituront chacune une classe **Java**. Les tests seront ajoutés à chaque nouvelle fonctionnalité. Des pipeline seront mis en place pour vérifier la compilation, l'assemblage et le résultat de l'exécution.
4 Opérations arithmétiques	Les opérations arithmétiques devront être compatibles entre elles.
5 Partie traditionnelle (if/ while, print, toString)	Les fonctions **print** et **toString** seront intégrées aux objets natifs **String et Integer**. **If/While** fonctionneront avec des variables et des opérations arithmétiques.
6 Partie Objet	Fonctionnement tel que spécifié dans le sujet : constructeur, instanciation, variables static...

Diagramme de GANTT



2 Introduction

3 Intégration continue

Nous avons noté la nécessité dans la première partie du projet de changer notre manière de push sur le dépôt distant.
L'approche CI/CD permet d'augmenter la fréquence de distribution des applications grâce à l'introduction de l'automatisation au niveau des étapes de développement des applications. Les principaux concepts liés à l'approche CI/CD sont l'intégration continue, la distribution continue et le déploiement continu.[1]
Ainsi, nous inspirant de nos TP de première année et de la pratique sur Internet, nous avons développé un pipeline s'exécutant à chaque push.
Ce pipeline exécute 3 groupes de tâches :

- Les tests présentés lors de la première et seconde soutenance représentés par le script **pipeline_build_normal_tasks.sh**
- Les tests présentés lors de la première et seconde soutenance avec l'option debug représentés par le script **pipeline_debug_tasks.sh**.
- L'assemblage et l'exécution des tests présentés lors de la soutenance finale représentés par le script **assembly_and_launch.sh Cela signifie qu'à chaque push, chaque fichier .blood du répertoire /assembly est compilé, assemblé et exécuté par l'assembleur de M. Parodi.**
- En complément de la tâche précédente, nous avons défini sur la fin du projet un stage qui permet de : compiler, assembler, exécuter et vérifier le résultat de l'exécution.
  Cette tâche à un fonctionnement un peu complexe :
  - Compilation des fichiers **.blood** avec une tâche **Graddle** : on obtient les fichiers **.src**.
  - On récupère ces fichiers **.src** qu'on assemble et qu'on exécute avec l'assembleur de M. Parodi.
  - On sauvegarde le résultat de l'exécution dans un fichier nommé **<nom_du_test>_2**.
  - On lance une task **Graddle** qui vérifie si le contenu du fichier **<nom_du_test>_2** est conforme au contenu du fichier **<nom_du_test>_3**. Si les contenus sont différents, on termine avec un code non nul et le pipeline tombe, sinon on passe au prochain fichier à tester... jusqu'à arriver au bout des tests du fichier **pipeline_check_permanent_tests_quality.sh**.

Tous ces scripts sont actionnés par le fichier de configuration de **pipeline** : **.gitlab-ci.yml** .

CI / CD Analytics

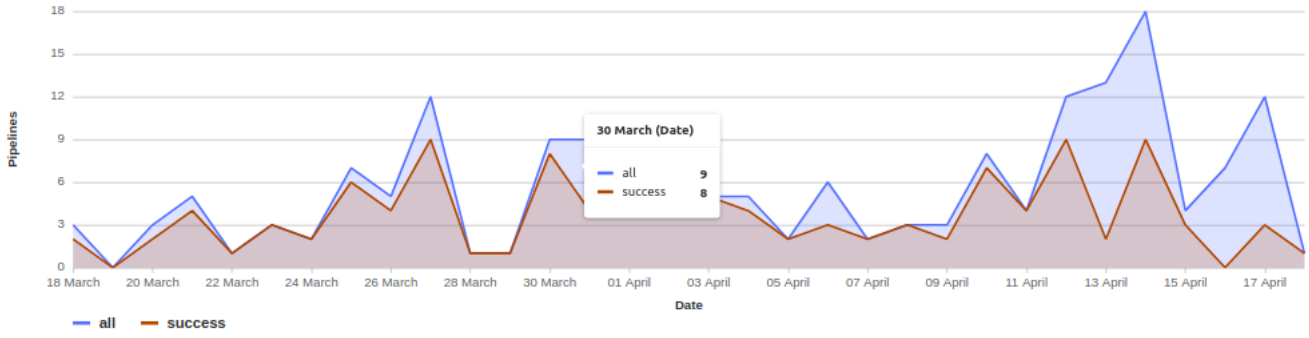
Overall statistics

- Total: **216 pipelines**
- Successful: **148 pipelines**
- Failed: **59 pipelines**
- Success ratio: **71.50%**

Pipelines charts

Last week | Last month | Last year

Date range: 18 Mar - 18 Apr



4 Structure générale de notre compilateur

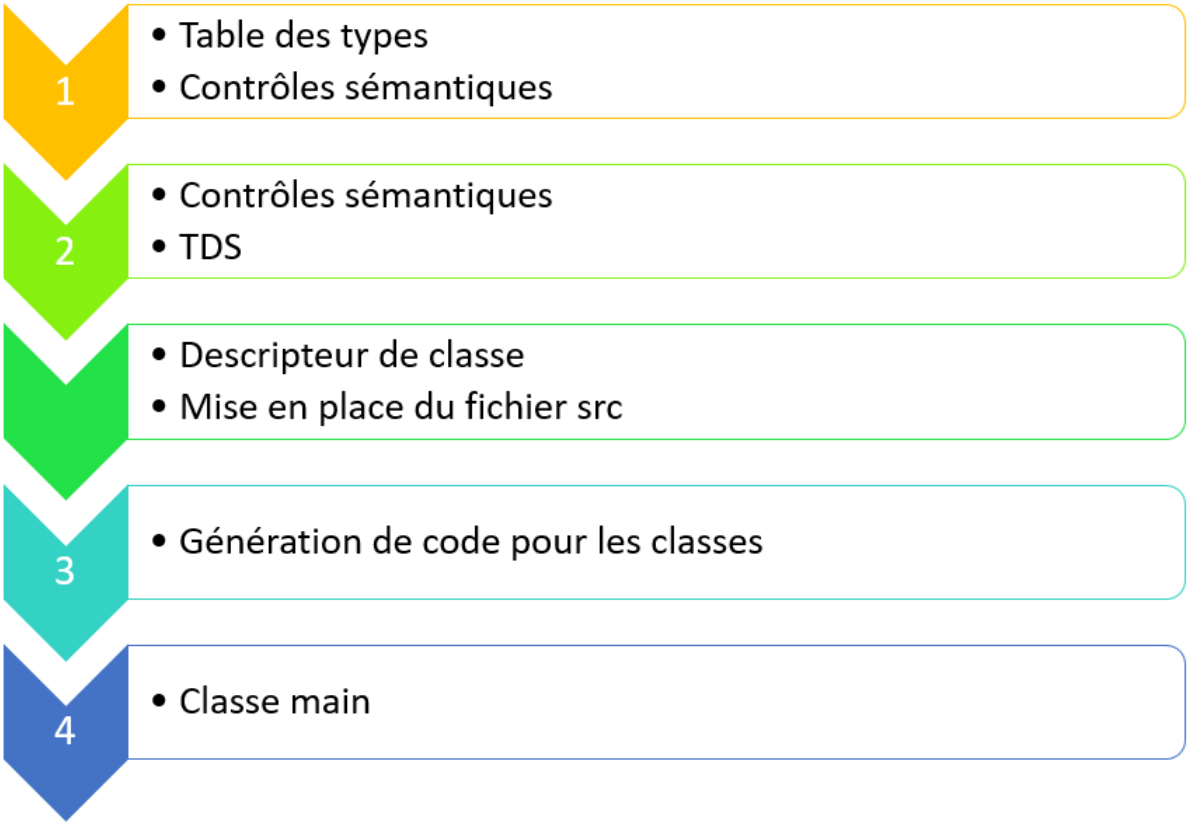


FIGURE 4.1 – Les numéros sur le côté gauche correspondent au numéro de passe. L'absence de numéro signifie une action entre 2 passes

5 TDS

Nous avons procédé en deux parties pour la mise en place de la **TDS**. D'abord, nous avons obtenu une première version pour la 2e soutenance, et ensuite, nous avons corrigé certains points qui étaient restés en suspend jusqu'à la génération de code.
Notre **TDS** est élaborée seulement à la 2e passe.
Elle est constituée de sous-**TDS** correspondant chacune au contenu d'un bloc. Ainsi, chaque entrée dans un bloc correspond à la création d'une nouvelle **TDS**. La **TDS** est remplie à chaque noeud d'**AST** nécessitant un ajout (paramètre, déclaration de variable, déclaration de méthode...). Les déplacement sont calculés au fur et à mesure, à chaque ajout et sont toujours égaux à 2 (voir après pourquoi 2 a été choisi).
Chaque nouvelle **TDS** est reliée à sa mère et possède une fonction d'affichage. Chaque élément de la **TDS** est identifié comme étant soit une méthode soit un attribut.

```
{
    x : Integer;
    y : Integer;
    z : Integer;
    s : String;
    is
    x := 3;
    y := 10;
    z := 7;
    s := x.toString();
    s.println();
}
```

```
-----------------------------
Display of symbol table
-----------------------------------------------------------
TABLE 1 Imbrication number : 1
Elements of the table
line 2:1 | deplacement: 2 | x:Integer
line 3:1 | deplacement: 4 | y:Integer
line 4:1 | deplacement: 6 | z:Integer
line 5:1 | deplacement: 8 | s:String
-----------------------------------------------------------
```

FIGURE 5.1 – La TDS correspondante

6 Les 28 Contrôles sémantiques développés

Nous avons effectué deux phases de développement de contrôles sémantiques. La première correspondait aux contrôles demandés pour la 2e soutenance. Ensuite, nous nous sommes rendus compte pendant l'étape de génération de code que certains contrôles étaient imparfaits ou manquaient.
Nous avons ainsi implémenté :

Les contrôles sémantiques sont implémentés grâce à une classe **Control**. Cette classe comporte une méthode pour chaque type de contrôle à effectuer. Chaque méthode de contrôle est appelée lors de la seconde passe.
Comme précisé dans le sujet, les contrôles sémantiques ne doivent pas être bloquants.
Nous avons fait le choix de standardiser l'affichage des messages d'erreur grâce à deux choses :

- En établissant une convention au sein du groupe.
- En utilisant une fonction d'affichage des erreurs unique.

Ces contrôles sont inspirés de notre pratique du développement et intègrent aussi bien la notion d'objet que des concepts avancés de vérifications (usages ambigus etc.).

Intitulé	Responsable
Redéclaration	
Redéclaration dans le même bloc	MASK
Redéclaration de classe	moi
Redéclaration de méthode dans le même bloc	MASK MASK
Mots Clefs du langage	
Utilisation de mots clefs réservés	MASK
Verification de l'usage de this et super	MASK
L'usage de **result** est interdit dans une méthode qui ne renvoie pas de résultat	MASK MASK
Cohérence	
Cohérence de types (affectation)	MASK MASK
Cohérence de type entre les paramètres formels et effectifs	MASK MASK
Cohérence de type dans les expressions arithmétiques	MASK MASK
Cohérence entre type de retour attendu (si attendu) et type de retour effectif	MASK MASK
Nombre de paramètre formel = nombre de paramètre effectif	moi
Variables déclarées avant utilisation	MASK
if(integer) / while(integer)	moi
Le type d'une variable/fonction existe	MASK
Lors d'un message, la destination existe-t-elle ?	MASK
Division par 0	moi
Division par/de choses inapropriées (String)	moi
Lors d'un override vérifier que la fonction qu'on override existe	MASK
Lors d'un extends vérifier que la classe de laquelle on hérite existe	moi
En cas de cast, vérifier que le cast est fait par une superclasse	MASK
Constructeur	
Pas d'usage de result dans les constructeurs	MASK
Pas de double déclaration de constructeur	moi
Constructeur conforme à la définition de classe Nom du constructeur Nombre de paramètres Type de paramètres	moi
Autres	
Toute branche d'exécution contient un retour quand précisé Un avertissement est émis en cas d'usage dangereux de **result** — Présence de **result** dans un **while** uniquement (la condition du while pourrait être fausse et **result** ne serait jamais affecté). — Présence de **result** dans une branche seulement d'un **if** (il y aurait une possibilité pour que **result** ne soit jamais vérifié si le **result** est dans le **then** alors que le **else** est vérifié et vice-versa).	moi

7 L'organisation de la mémoire dans notre projet

7.1 Explication générale

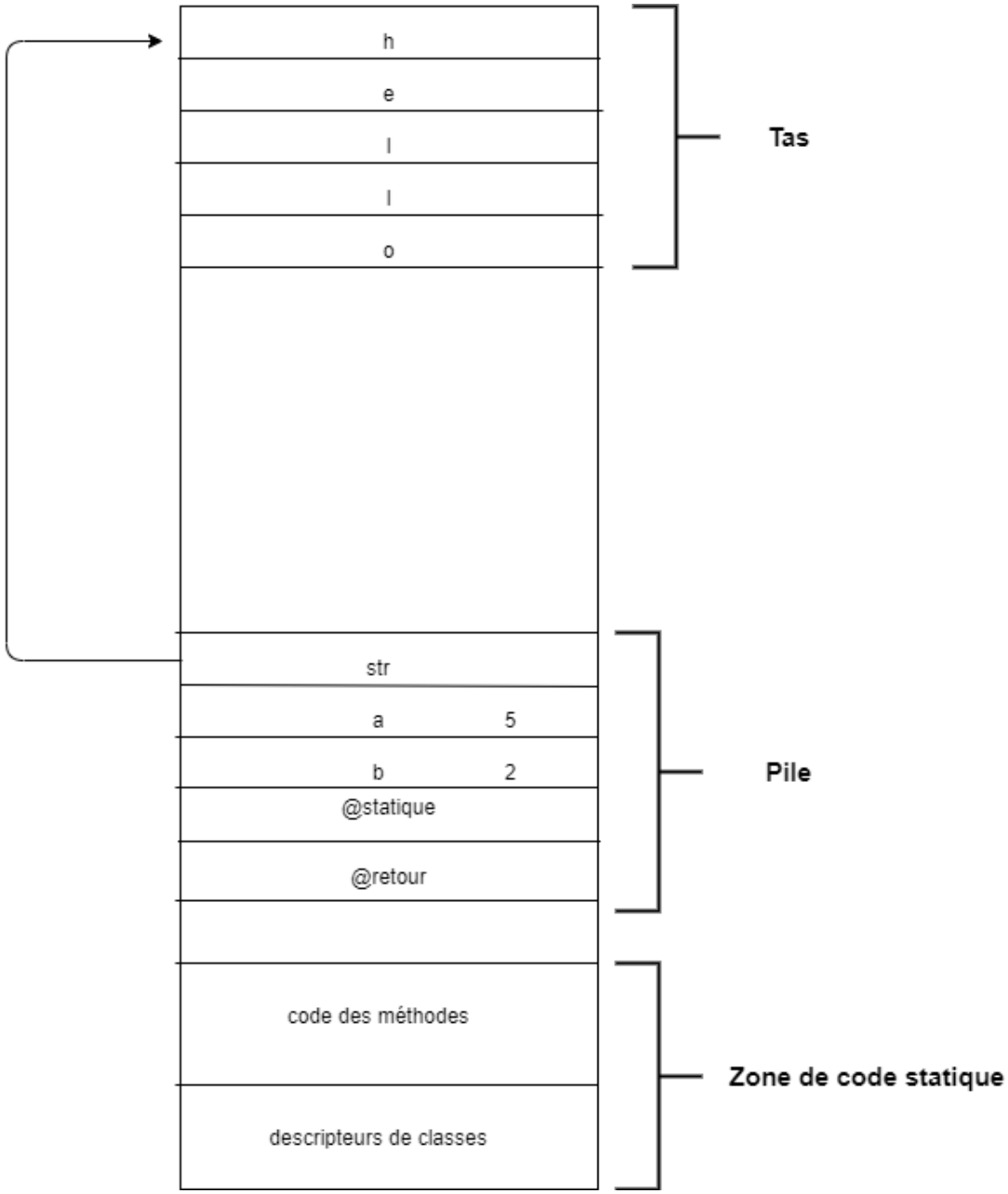
La mémoire est divisée en 4 grandes parties

- **La zone de texte (Text section)** stocke les instructions du programme. Nous ne la représenterons pas dans les schémas car elle n'est pas importante pour le travail à faire.

- **La zone de code statique (Data section)** stocke des informations qui ne changent pas tout au long du programme, dans notre cas nous y avons mis le descripteur de classe ainsi que le code des méthodes de chaque classe traduit en assembleur.

- **La pile (Stack, le pointeur de pile SP est utilisé)** stocke les valeurs des variables (entiers ou pointeur vers la donnée). Elle garde aussi en mémoire les différents appels de méthodes et par extension l'environnement d'exécution actuel.

- **Le tas (Heap, en général l'instruction BRK est utilisée dans les assembleurs)** contient les instances des objets (et donc aussi les string). Lorsqu'une variable est un objet, la pile contient une adresse qui pointe vers le tas.



Dans l'exemple précédent, nous avons une représentation des 3 parties de la mémoire qui nous intéressent.
On constate que la zone de mémoire statique se situe en dessous de la pile. On est ainsi certains qu'elle ne sera pas modifiée.
La pile juste au-dessus monte dans les adresses tandis que le tas descend. En théorie, il est possible que le tas et la pile se rencontrent notamment avec un grand nombre d'appels récursif. On peut voir dans l'exemple que la pile contient 3 variables, 2 entiers dont la valeur est explicitement écrite et une chaîne de caractères (qui contient une adresse qui pointe vers une adresse dans le tas où se trouve effectivement les caractères).

7.2 La mémoire en MicroPIUP

MicroPIUP a la particularité de n'avoir implémenté ni les segments ni le tas, nous avons donc une zone mémoire vierge sans moyen préfait de stocker dans différentes zones mémoire.

- **Zone de texte :** l'instruction "ORG #adresse" permet de définir où seront écrites les instructions du programme
- **Zone de code statique :** à une adresse donnée (par exemple 0xf000) nous écrivons le contenu de cette zone, il faut simplement la mettre suffisamment loin dans la mémoire pour ne pas avoir de collision avec les autres parties.
- **La pile :** à l'adresse suivant la fin de la zone de code statique on peut commencer la pile celle-ci s'agrandira grâce à l'instruction "ADQ -2, SP", le registre SP contient toujours le sommet de pile.
- **Le tas :** à une adresse donnée (nous utilisons 0x1000) on commence un tas à l'envers, il s'agrandira avec l'instruction "ADQ 2, HP", le registre HP (HEAP) est ajouté pour garder en mémoire la sommet du tas.

7.3 Les registres

Nous avons mis en place un fichier commun (**src\main\java\assembly\RESERVED_REGISTERS.java**) qui recense tous les registres utilisés à l'occasion de chaque opération. Cela nous a apporté deux choses :

- Une plus grande facilité de communication : chacun savait où était le résultat de l'opération faite par quelqu'un d'autre.
- Une plus grande agilité dans le code dans la mesure où une seule modification dans ce fichier permet la modification de toutes les utilisations du registre dans le code assembleur généré.

8 Génération de code d'un langage traditionnel

8.1 Affectations/Déclaration

Dans cette partie, on ne traite que des variables déclarées dans le `main`.

8.1.1 Déclaration

La déclaration des variables se faisant au même endroit dans le bloc, il suffit de parcourir les noeuds dans l'arbre et, à l'aide de la table des symboles, incrémenter le registre correspondant au *Stack Pointer* (SP) de la valeur du décalage.

8.1.2 Affectation

L'affectation se fait selon un double processus.
On s'intéresse au type de noeud du membre droit : **Opération arithmétique, Opération booléenne, Integer, String...**, et on adapte le comportement de l'affectation en fonction. Par exemple si c'est une opération booléenne (`x := 2 <> 3`), on ira seulement chercher le résultat dans **R5**, dans la mesure où l'opération de comparaison aura été faite juste avant dans le code assembleur. Pour une **String** par exemple, cela est différent. Il faut stocker la **String** dans le tas puis récupérer dans un registre son adresse.
Une fois le membre gauche droit traité, on affecte la valeur au membre gauche. De la même manière que précédemment, on distingue différents cas en fonction du noeud d'AST du membre gauche.

8.2 Opérations booléennes

On rappelle tout d'abord que les Booléens en langage Blood sont des entiers. On a false pour un entier qui vaut 0, et true sinon.

8.2.1 Récupération des fils droit et gauche

Lorsqu'une opération booléenne est détectée en parcourant l'arbre syntaxique, une fonction a été créée afin de récupérer les valeurs de ses fils afin de pouvoir plus simplement effectuer plusieurs opérations à la suite (par exemple 1+2 < x-8, où l'on a deux opérations arithmétique et une opération booléenne). Pour cela, après le parcours du fils gauche, sa valeur est placée dans le registre R3, puis après le parcours du fils droit sa valeur est placée dans R4. Si le fils est un entier, on place directement sa valeur dans le registre, si c'est une variable ou un attribut de méthode, on va chercher son déplacement dans la table des symboles. Ainsi, lors de l'appel à la fonction correspondante à l'opération booléenne, on peut manipuler les registres R3 et R4 en étant sûr qu'ils contiennent bien les valeurs des fils.

8.2.2 Opérations <,>,<> et =

Les opérations <,>,<> et = fonctionnent selon le même principe. Tout d'abord, on charge 0 ou 1, selon le cas, dans R5, puis on soustrait le contenu de R3 et R4 où nous avons auparavant chargé les valeurs des fils et en comparant la valeur obtenue avec 0, on change ou non le contenu de R5. Ainsi, le résultat d'une opération booléenne est toujours dans R5.

8.2.3 Opérations || et &&

Les opérations || et && fonctionnent quasiment de la même façon que les précédentes, si ce n'est qu'à la place de soustraire les valeurs de R3 et R4, on les multiplie pour &&, et on les additionne pour ||. Le résultat est de nouveau placé dans R5.

8.3 Opération & : la concaténation

Dans la mesure où nous avons fait le choix de stocker nos **String** dans le tas, la tâche était un peu plus complexe.

Nous avons ainsi seulement géré le cas où les string sont données par l'utilisateur et non dans des variables :

```
{
t : String;
is
t := "coucou" & " Suzanne " & "et" & "Sebastien" & "\n" & "ce projet
    merite 20!" ;
t.println();
t := "hey salut!";
t.println();
}
```

Pour se faire, nous avons parcouru l'**AST** jusqu'en bas (la première **String** : Suzanne) est le dernier noeud en bas). Puis, nous sommes remontés dans l'arbre pour aller chercher le contenu de chaque chaîne et le rajouter dans le tas.

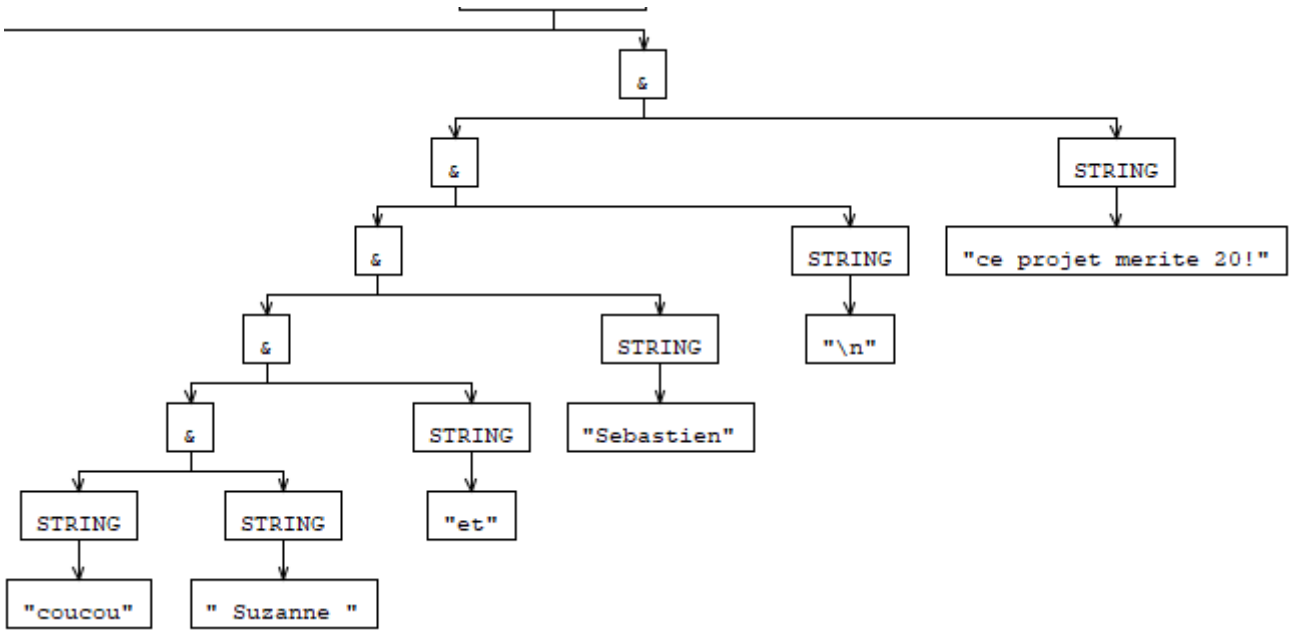


FIGURE 8.1 – Caption

Attention, toutefois, ce n'est pas si simple pour un cas : dans l'assembleur de M. Parodi, on doit écrire dans les registres systématiquement 2 octets correspondants à 2 lettres. Et nous utilisons ces registres pour ajouter le contenu de la string dans le tas 2 lettres par 2 lettres donc. Mais que se passe-t-il si on a une **String** avec un nombre de lettres impair ? Il faut stocker la lettre en trop, qu'on ne peut pas mettre dans le tas, et la garder pour la rajouter à la prochaine **String** qu'on va concaténer.
Mais, ce n'est toujours pas fini, et que se passe-t-il si on a une **String** avec un nombre pair de caractères ? Il faut tester si le registre qui contient la lettre en suspend et vide et ajouter la lettre si le registre ne l'est pas.

Enfin, une fois cela fait, il faut terminer la **String** avec un ajout dans le tas de **0x0000**, pour signifier la fin de **String**.

8.4 Opérations arithmétiques

L'algorithme permettant les opérations arithmétiques (+, ×, /, -) se base sur une fonction récursive et l'utilisation de la pile.

8.4.1 Initialisation

Pour initialiser l'algorithme, on doit définir quels sont les trois registres à utiliser (décrits ci-après), à quel endroit dans le code on se situe (c'est-à-dire le nom de la classe où se trouve l'opération à effectuer) ainsi que le noeud de l'arbre syntaxique indiquant la racine de l'opération.
Les registres utilisés sont :

- **R1** : utilisé pour stocker la valeur du fils gauche de l'opérateur arithmétique.
- **R2** : utilisé pour stocker la valeur du fils droit de l'opérateur arithmétique.
- **R6** : utilisé pour stocker le résultat des opérations et pour charger les valeurs avant qu'elles soient empilées ou pour les stocker lorsqu'elles sont dépilées.

8.4.2 La partie récursive de l'algorithme

La partie récursive de l'algorithme a pour objectif de parcourir le sous-arbre de l'opération. On définit une opération arithmétique dite "chaînée" lorsqu'elle possède plus d'un opérateur.
Exemple :
`5 + (y + point.get(x)) * 2` est une opération chaînée
`x + 8` n'est pas une opération chaînée.

La récursion s'arrête lorsqu'une feuille de l'arbre est atteinte (entier, variable ou appel de méthode), et se poursuit lorsqu'un des éléments suivants est atteint :

- **Une autre opération arithmétique** : Si on est dans le cas d'une opération chaînée
- **Des parenthèses**
- **Un moins unaire ou un plus unaire**

Cas des feuilles de l'arbre

Lorsqu'une feuille est atteinte, il faut placer la valeur portée par cette feuille dans la pile. Si cette valeur est un **entier**, alors on la charge dans le registre (**R6**) et on l'empile.
Si la feuille correspond à une **variable**, alors on va d'abord récupérer dans la pile sa valeur, que l'on place dans **R0**. Ensuite, on la charge dans **R6**, puis on l'ajoute au sommet de la pile. Cette méthode est assez peu optimale mais permet de simplifier la récupération des valeurs en remontant la récursion.
Si la feuille correspond à un appel de message, alors on va récupérer la valeur de retour de ce message et l'empiler.

Cas des opérations chaînées

Lorsqu'une nouvelle opération est détectée, alors la fonction récursive est appelée sur les deux fils de l'opérateur. Le résultat de ces appels étant placé au sommet de la pile, il suffit, lors de la remontée, de dépiler deux fois et de placer la première valeur dépilée dans **R1** (il s'agit du résultat de l'opérande de gauche) et la seconde dans **R2** (résultat de l'opérande de droite). Ainsi, il suffit d'effectuer l'opération avec **ADD**, **SUB**, **MUL** ou **DIV** et d'empiler le résultat, qui pourra par la suite être dépilé pour être utilisé lors de la remontée.

Cas des parenthèses

Ce cas est assez simple ; en effet, dans l'AST, les parenthèses ne sont rien de plus que la création d'un nouveau sous-arbre, avec un opérateur arithmétique (ou une feuille) en racine. La fonction est alors appelée sur ce noeud, et le résultat des opérations effectuées dans les parenthèses pourra être dépilé par la suite.

Cas des moins unaires

Ce cas est également assez simple ; la fonction est appelée sur ce qui suit le moins unaire, empilant ainsi le résultat. Il est ensuite dépilé, multiplié par -1 puis réempilé pour qu'il puisse être exploité par la suite.

Cas des plus unaire

Ce cas aussi est très simple : le plus unaire n'ayant pas d'impact sur le résultat, la fonction est appelée sur ce qui suit l'opérateur et le résultat est tout simplement empilé pour une utilisation future.

8.4.3 Fin de l'algorithme

Une fois la récursion terminée, on peut trouver le résultat soit dans la pile, soit dans **R6**. Il est ainsi exploité par le reste de l'expression, comme par exemple lors d'une affectation où il remplacera l'ancienne valeur d'une variable ou d'un attribut, ou bien il sera comparé à une autre valeur dans la condition d'un **if** ou d'un **while**.

8.5 Instructions de base

8.5.1 If/While

Les instructions **If/While** fonctionnent par le test d'une condition avec l'instruction assembleur **JEQ** ou **JNE** en fonction des circonstances.

```
1  if3 LDW R9, #0
2      SUB R1, R9, R9
3      JEQ #else3-$-2
```

Ici on vérifie que R1 est positif, on aura donc au préalable chargé dans R1 la valeur contenue dans la condition du if.

On peut remarquer que le saut se fait par des étiquettes pour chaque if/while : **ifn, thenn, elsen**, **whilen, endwhilen**. Cela est assuré par un générateur d'étiquette. A chaque fois que l'on tombe sur l'étiquette **IF/WHILE** dans l'**AST**, on incrémente un compteur global (une variable **static** en fait) qui identifie chaque **if/while** et permet ainsi les imbrications :

```
1  {
2    x:String;
3    y : Integer := 0;
4    z : Integer := 1;
5    t : Integer := 1;
6    is
7    while t do {
8      "1".println();
9      x := t.toString();
10     t := 0;
11     while z do{
12         "2".println();
13         z := 0;
14     }
15   }
16   x.println();
17   z.toString().println();
18 }
```

Nous avons choisi que 0 correspondait à Faux, et tout ce qui est différent de 0 est Vrai. Toutefois, if ou while peuvent prendre beaucoup des choses variées en argument :

- un entier : `if(12)`
- une/des opérations arithmétiques : `if(12 + 3 <> 1)`
- — une variable : `if(x)`
  - une variable d'un objet : `if(o.x)`

L'objectif est d'aller chercher dans le bon registre la valeur souhaitée pour pouvoir tester la condition dessus. Il n'a pas été possible d'uniformiser ces cas de manière simple. En effet, comment faire que lorsqu'on a un entier dans la condition, il faille aller chercher cet entier dans le même registre que dans le cas d'une suite d'opérations arithmétiques ? Ainsi, **If/While** adaptent leur comportement en fonction du contenu de la condition.

Nous avons également pris en compte le cas où un utilisateur souhaiterait faire :

```
1 if/while( (((((((n))))))) )
```

Cela est géré au moment de la génération de code et non à l'assemblage.

Prise en compte du saut de bloc Le passage dans un bloc suite à la vérification de la condition de **if/while** n'est pas simple. Quand on change de bloc, il faut aussi changer de **TDS** et empiler le chaînage, de même quand on sort du bloc.

Particularité de while La condition de While doit être systématiquement retestée à la fin du bloc **do**. Pour cela il faut sauter à l'étiquette de début de while. Mais, cela n'est pas si simple.

```
1 x := 12;
2 while (x) do{
3     x := x - 1;
4 }
```

Dans ce cas, la valeur de **x** est amenée à changer à chaque tour de boucle. Il faut donc à la fin du bloc **do** sauter non pas au test d'égalité de x avec 0 mais plus haut encore. Il faut en effet charger la nouvelle valeur de x, puis tester si cette valeur est égale à 0, puis éventuellement retourner dans le bloc **do**.

8.6 Fonctions de base du langage

Dans le sujet, il est précisé que la classe **Integer** possède une méthode **toString** et que la classe **String** possède deux méthodes : **print** et **println**.

8.6.1 Fonction print/ln

Nous avons pour l'essentiel réutilisé le code des fonctions de M. Parodi. Nous avons dû néanmoins l'adapter légèrement pour prendre en compte le chaînage.
Le **println** s'effectue par un appel à **print** suivi d'un autre avec pour argument un saut de ligne.
*L'appel à **print** se fait par un saut à la classe **String** puis par un appel à la méthode de cette classe.*

8.6.2 Fonction toString

Les string étant stockées dans le tas, nous n'avons malheureusement pas pu utiliser la fonction de M.Parodi pour convertir un entier en String. Par ailleurs, la fonction de M.Parodi était légèrement erronée sur un point : elle ajoutait dans la string systématiquement le signe de l'entier converti (même en cas d'entier positif).
Ainsi, nous avons choisi de la refaire nous-même.

Tout d'abord, il y a 2 cas à distinguer.

- `(12345).toString();`
  Dans ce cas, nous avons décidé de stocker **12345** dans **R10**.
- `x.toString();`
  Dans celui-ci, nous avons décidé d'aller chercher la valeur de x et la stocker dans **R10**.

R10 contient donc systématiquement la valeur à convertir.
La fonction récupère cette valeur et la découpe chiffre à chiffre : **1, 2, 3, 4, 5**. Cette opération se fait par division successive de R10 et récupération des restes dans la division euclidienne.
Ensuite, ces chiffres sont convertis en **ASCCI** en ajoutant **0x0030** (0 en **ASCII**) et stockés dans un registre.
Enfin, chaque chiffre est ajouté par pair à son précédent. Le but pour **1, 2** est de former **12** en **ASCII**.
Pour cela :

- On décale 1 pour former 10. On multiplie la conversion **ASCII** de 1 par 256.
  La conversion de 1 en ASCII : **0x0031**.
  Le décalage donne : **0x3100**
- On rajoute la conversion ASCII de 2

— On obtient **0x3132**, c'est-à-dire **12**.

On procède de la même manière pour chaque paire de chiffres. On stocke chaque paire dans un registre puis on stocke le registre dans le tas. On prend soin d'ajouter à la fin **0x0000** pour indiquer à notre fonction `print` que la chaîne à afficher est terminée.

Si le nombre à convertir est négatif, on rajoute au premier registre ajouté dans le tas le signe - en ASCII : **0x2d00**.

Optimisations de la fonction

— Nous avons introduit la notion d'optimisation des registres utilisés.

— De même, notre fonction prend en compte le cas de la conversion des petits nombres. En effet, **-5** ne sera pas converti par **-00005** mais bien par **-5**.
Cela a été rendu possible par une grande utilisation d'étiquettes et par des tests d'égalité. On vérifie en effet si les 2 premiers chiffres sont nuls, puis les 2 suivants et on s'arrête dès qu'on ne trouve pas un couple nul.
Toutefois, cela n'a pas été aussi simple... que faire quand dans un couple de 2 chiffres, le chiffre de gauche est un 0 inutile et le chiffre de droite différent de 0 ?
Il faut dans ce cas-là décaler les contenus des registres pour faire partir le 0 inutile en faisant remonter le contenu des registres contenant les chiffres suivants.

*L'appel à **toString** se fait par un saut à la classe **Integer** puis par un appel à la méthode de cette classe.*

9 Les objets

9.1 Les instances

Lors de l'instanciation d'un objet, nous utilisons le registre SP pour faire les opérations dans le tas (ce n'est pas très optimal, l'idéal aurait été d'utiliser un registre réservé). La première étape consiste à affecter l'adresse du descripteur de classe du type dynamique dans la première adresse du tas disponible. Ensuite la place pour chaque attribut est allouée, la valeur du sommet du tas est sauvegardée afin d'être affectée à la variable qui instancie l'objet. La valeur du sommet du tas peut maintenant être mise à jour et prendre celle de SP. Enfin l'adresse de cet objet est affectée à la variable qui a appelé le constructeur qui est maintenant appelé via un JUMP.

```
1  public static void allocationObjet(String type) {
2          DescripteurDeClasse ddc = get(type);
3          String bloc = "//debut instanciation----------\n"+ToolsASM.
       link()+
4                  "    LDW SP, HP  \n"+//passage au tas
5                  "    LDW R1, #STACK_ADRS\n"+
6                  "    ADQ -"+(2+getDeplacement(type))+", R1\n"+
7                  "    STW R1, (SP)\n"+
8                  "    ADQ "+ddc.taille+", SP\n"+
9                  "    LDW R1, HP\n"+//debut de l'instance pour plus tard
10                 "    LDW HP, SP\n"+//actualisation sommet du tas
11                 ToolsASM.unlink();
12
13
14         try {
15             writer.write(bloc);
16         } catch (IOException e) {
17             // TODO Auto-generated catch block
18             e.printStackTrace();
19         }
20
21     }
```

9.2 Appel de méthode

Lors d'un appel de méthode nous devons empiler le registre R11 qui contient l'objet de l'instance actuelle, empiler tous les paramètres de la fonction, empiler la valeur du PC pour pouvoir revenir au programme et faire un JEA à l'adresse du code de la méthode.

```
for(ElementObjet param: fonction.getListeAttributOuParam()) {
            if(param.getNature().contentEquals("Integer"))
                writer.write("  LDW R0, #"+param.getNom()+"\n");
            else if(param.getNature().contentEquals("String")) {
                allocationString(param.getNom());
                writer.write("  LDW R0, R1\n"
                        + "//fin stockage string----------\n");

            }
            else {
                //2 cas, si c'est une variable on va simplement la chercher
                //si c'est un objet il faut aller a la valeur pointe par l'attribut qui contient la String Integer
                //sinon on aurait une variable qui pointe sur un pointeur de String/Integer
                if(param.getNature().contentEquals("Objet")) {
                    TasASM.attributDansR0(tds, param, -1, false);
                        writer.write("  LDW R0, (R0)\n");
                }
                else
                    TasASM.attributDansR0(tds, param, -1, false);

            }
            writer.write("  ADQ -2, SP\n"
                    + " STW R0, (SP)\n");
        }
    }
```

Le code ci-dessous se charge d'empiler tous les paramètres, on distingue 5 cas :

- Un entier, on ne fait que l'empiler.
- Une String, on doit allouer la String dans le tas puis empiler l'adresse qui pointe vers son allocation.
- Une variable, on ne fait que l'empiler, il faut éventuellement remonter des chaînages statiques pour obtenir sa valeur.
- Un objet, dans ce cas une fonction attributDansR0 permet d'aller chercher l'adresse (String/Objet) ou la valeur(Integer) d'une suite d'attributs/fonctions.
- Un calcul, par exemple 5+9 qui est de type Integer, on doit alors simplement faire le calcul.

```
writer.write("  LDW R0, (R11)\n"
                + " ADQ -"+ddc.getDeplacementMethode(fonction.getNom())
    +", R0\n"
```

```
3                   + " LDW R0, (R0)\n"
4                   + " MPC WR\n"// charge le contenu du PC dans WR
5                   + " ADQ 8, WR\n"//adresse de retour en enlevant les
      parametres
6                   + " ADQ -2, SP\n" // decremente le pointeur de pile SP
7                   + " STW WR, (SP)\n" // sauvegarde l'adresse de retour
      sur le sommet de pile
8                   + " JEA (R0)\n"
9                   + " ADQ "+(fonction.getListeAttributOuParam().size()*2)
      +", SP\n"
```

Le code ci-dessus correspond enfaite à l'instruction JSR cependant elle ne peut pas être utilisé avec une adresse, seulement avec des étiquettes. On va chercher l'adresse de la méthode dans le descripteur de classe (Adresse de R11 qui pointe le début de l'instance de l'objet), ensuite on empile le PC que l'on empile enfin on peut se rendre à l'adresse de la méthode.

9.3 Static

Un attribut ou une méthode statique se situe en dehors de toute instance mais reste cependant dans la classe. Les attributs static ont été stockés directement dans le descripteur de classe de chaque type, l'accès à celui-ci se fait avec "TypeClasse.attribut". Pour les méthodes static on utilise exactement la même façon de faire qu'une méthode classique seule l'appel diffère. En effet la fonction initFonction permet d'appeler une méthode en fonction d'un type et d'un nom de méthode. Elle se charge d'empiler tous les paramètres ainsi que de lancer la méthode avec un JEA. Si c'est une méthode static on va chercher le descripteur de classe pour le mettre dans R11 plutôt que d'utiliser celui de l'instance.

9.4 Super

Le mot clé super fonctionne de la même façon que static, on a un super type obtenu grâce à la table des types en java et un nom de méthode, grâce à la fonction initFonction un tel appel est trivial. Pour le cas d'un attribut, il suffit de trouver le déplacement de l'attribut dans l'objet courant comme si on avait fait un "this.attribut" au lieu de "super.attribut".

9.5 Cast

Lors d'un cast on doit changer de descripteur de classe à utiliser pour faire le prochain appel de méthode (pour un attribut un cast n'a pas d'effet ou super classe a simplement moins d'attribut disponible mais y accède de la même façon). R11 contient toujours l'instance actuel ce qui veut dire que (R11) contient le descripteur de classe pour faire un cast

on a fait une disjonction de cas qui va chercher le descripteur de classe du type que l'on cast et on utilise celui là pour aller chercher l'adresse de la méthode.

9.6 Enchaînement d'opérations

Dans un langage objet tel que **Java**, on s'attend à ce que les opérations puissent être faites "à la chaîne". Par ce terme on peut désigner :

```
{
  test : String;
  test2 : Integer;
  livre1 : Livre;
  prixCher : Integer := 161;
  inflation : Integer := 20;
  nombrePage : Integer := 1009;
  nombrePageInutiles : Integer := 1000;
  nombrePageStr : String;
  is
  livre1 := new Livre(prixCher * 20, "Dragon Book", nombrePage -
    nombrePageInutiles);
  livre1.nbPages.toString().println();
  livre1.prix.toString().println();
  livre1.nom.println();
  "\n\n".println();
  livre1.toString().println();
}
```

Ici on voit que

- l'assemblage de plusieurs fonctions à la suite comme **toString** et **print/ln** se fait.
- des opérations arithmétiques peuvent être données comme paramètres d'une fonction.

10 Limites du projet

— Les entiers supérieurs à **65535** ne sont pas gérés correctement.

11 Conclusion et Bilan

11.1 Conclusion

11.1.1 Niveau atteint

Lors de la génération de code, nous avons implémenté toutes les fonctionnalités nécessaires à la validation des niveaux 1 à 4. En plus de ces fonctionnalités, nous avons implémenté les optimisations de registres fixes.

11.1.2 Apport du projet

Ce projet a été enrichissant à toutes les étapes. Il nous a permis à chacun d'appréhender ce qu'il se passe derrière les commandes que nous utilisons tous les jours, **gcc** par exemple. Nous avons saisi l'enjeu d'avoir un compilateur efficace et à quel point une petite erreur dans le programme assembleur peut causer des difficultés à la machine pour obtenir l'exécution souhaitée.

11.1.3 Perspectives

Avec plus de temps, nous aurions pu implémenter des librairies pour notre language. Ces librairies auraient pu être faites soit sous forme de fonctions type **toString, print/ln** et donc présentes en code assembleur, soit sous forme de fichiers .blood à inclure.

11.2 Bilan des membres de l'équipe projet

11.2.1 Bilan individuel MASK

Points positifs	— La communication au sein du groupe s'est améliorée depuis la fin de la première partie du projet, où ce problème avait été mis en avant. — Grâce à cette dernière partie de génération de code, j'ai pu mieux comprendre comment fonctionne un compilateur au niveau du code assembleur généré.
Expérience personnelle/ressenti	— Bien que j'ai trouvé cette partie de génération de code intéressante, je l'ai moins appréciée que la première partie du projet. — J'ai trouvé parfois frustrant, lorsqu'un test ne donnait pas le code assembleur attendu, de ne pas savoir quelle partie du code était responsable. Ainsi, beaucoup de temps a été passé à enlever des problèmes qui se situaient parfois ailleurs. De plus, les problèmes étaient parfois constaté lors de l'ajout d'une nouvelle fonctionnalité et tout devait être refait.

11.2.2 Bilan individuel moi

Points positifs	— Cette 2e partie de projet était très différente de la première. Nous avons su tenir compte de nos remarques faites lors du **post-mortem** de la 1ère partie — La mise en place d'un **pipeline** nous a grandement aidé et nous a permis de gagner en agilité. Nous avons su anticiper plusieurs erreurs et gagner du temps.
Expérience personnelle/ressenti	— Je crois que je n'avais pas saisi au début l'ampleur de ce que représentait ce projet et je me réjouis de l'avoir compris assez tôt pour être en mesure de proposer un rendu correspondant au niveau 4. — Je regrette parfois un manque de suivi des encadrants. En effet, au-delà de questions assez simples répondues pendant la séance de TP consacré, nous avons été un peu laissés seuls.

11.2.3 Bilan individuel de MASK MASK

Points positifs	— La communication dans le groupe était meilleure que pendant la première partie — Les remarques faites lors de la soutenance de la première partie ont été bien prises en compte par les différents membres du groupe.
Expérience personnelle/ressenti	— J'ai préféré cette partie de manière générale. Elle m'a permis de m'améliorer en algorithmique, en plus de me faire comprendre un peu mieux les dessous d'un langage objet. — Il était parfois difficile d'avoir une vue d'ensemble sur ce qui était fait (surtout comment c'était fait) au cours du projet, ce qui était aggravé par la taille du code et sa complexité. Cela m'a beaucoup agacé parce que la définition de cas de test devenait rapidement un casse-tête.
Axes d'amélioration	— Le code produit est, à part à quelques endroits, de plutôt mauvaise qualité. Par endroit, la base avait pu être propre à un moment mais les corrections successives font effet de rustines qui rendent le tout assez peu lisible et complexe. De plus, il est dans l'ensemble peu documenté.

11.2.4 Bilan individuel MASK MASK

Points positifs	sujet très intéressant
Expérience personnelle/ressenti	— J'ai tout particulièrement aimé cette deuxième partie du projet PCL, cela m'a permis de comprendre comment fonctionne vraiment un langage objet ainsi que certaines notions comme "static" que je ne comprenais pas avant. — Projet parfois irritant dû au fait que du code non testé est sans arrêt push sur le GitLab.
Axes d'amélioration	— Comme j'en ai souvent parlé, je trouve que l'assembleur utilisé n'est pas adapté à la tâche. Par exemple, la concaténation de string devient affreuse à réaliser là où elle aurait été triviale avec un autre assembleur. — Java chaotique car aucune harmonisation n'a été faite, on se retrouve à avoir de la duplication de code énorme et donc une complexité assez grande pour debugguer le code. Parfois, le code assembleur est généré dans les feuilles de l'AST, parfois c'est une fonction qui écrit tout un noeud.

11.3 Travail réalisé

Étapes	MASK	moi	MASK MASK	MASK
Jalons				
TDS	1h	6h	2h	1h
Contrôles sémantiques	13h	15h	15h	4h
Génération de code				
Mise en place de l'environnement (tests, tasks, pipeline, passes)	2h	6h	2h	3h
Opérations arithmétiques et booléennes	29h	1h	24h	0h
Partie traditionnelle (if/while, print, toString)	2h	18h	3h	6h
Partie Objet	2h	4h	3h	40h
Travail de secrétariat				
Compte-Rendus de réunion	0h	5h	0h	0h
Rapport	6h	6h	6h	5h
Total	55h	61h	55h	59h

Bibliographie / Webographie

[1] Redhat. Qu'est-ce que l'approche CI/CD ?, 1997. https ://www.redhat.com/fr/topics/devops/what-is-ci-cd : :text=L'approche%20CI%2FCD%20permet,continue%20et%20le%20d%C3%A9ploiement%20continu. xiv

Annexes : Compte-Rendus de réunion

A CR Réunion 22/01/2021

Présents :

- MASK
- moi
- MASK MASK
- MASK

Heure de début : 14H

Durée : 1h20

Ordre du jour :

- exploitation des données produites par **Antlr**
- la structure de la table des symboles,
- les erreurs/contrôles sémantiques traitées par le compilateur,

I Exploitation des données produites par Antlr

La partie sur **Antlr** est maintenant terminée. Après analyse d'une grammaire, **Antlr** produit deux fichiers, `<Grammaire>Lexer` et `<Grammaire>Parser`,
Ce sont ces deux fichiers **Java** qu'il faut que l'on exploite pour la suite du projet (production de la TDS, contrôles sémantiques).
En effet, **Antlr** ne produit pas l'AST depuis un fichier sous une forme directement exploitable pour construire la TDS. Il s'agit donc, depuis ces fichiers **Java** de reconstruire de manière récursive l'AST pour le rendre exploitable pour construire la suite du projet.

MASK MASKsuggère de construire une base de travail sur le répertoire git avec **Graddle**, et de se resservir de ce qui avait était fait dans le TP1 pour l'exploitation du lexer et parser produits par **Antlr**.

```
1  import org.antlr.runtime.*;
2  public class Test {
3      public static void main(String[] args) throws Exception {
4          ANTLRInputStream input = new ANTLRInputStream(System.in);
5          ExprLexer lexer = new ExprLexer(input);
6          CommonTokenStream tokens = new CommonTokenStream(lexer);
7          ExprParser parser = new ExprParser(tokens);
8          parser.prog();
9      }
10 }
```

II La structure de la table des symboles

Rappels *La table des symboles [est] créée au début de l'analyse sémantique, et remplie au fur et à mesure de cette analyse. On crée généralement une table des symboles par région*[1] *; à la fin de l'analyse, on rassemble le tout dans une table des symboles globale.*

Une table des symboles contient une entrée par identificateur. Cette entrée contient les informations suivantes :

- *le type, la place mémoire occupée, et les bornes s'il s'agit d'un tableau.*
- *le déplacement (différence d'adresses mémoires) nécessaire pour accéder à la variable depuis la base de la pile ou de la fonction.*
- *la visibilité et la portée de cet identifiant (numéro de bloc, nombre d'imbrications)*
- *s'il s'agit d'une fonction, les arguments et leur type, ainsi que le type de retour, sont également précisés.*

Nous nous mettons d'accord sur le fait qu'il faut commencer à développer une ou plusieurs classes **Java** correspondant à la structure de la TDS.
La TDS sera récursive et sous forme d'arbre (relations frère, fils, père). Notre implémentation devra contenir :

- les colonnes de la TDS listées ci-dessus.
- une fonction d'affichage composée d'appels à la fonction `System.out.println`.

III Les erreurs/contrôles sémantiques traités par le compilateur

Madame Collin suggère que chaque membre peut facilement développer 5-6 contrôles différents, ce qui en fait facilement une vingtaine. Par ailleurs, elle précise que contrôler les types d'une expression avec une addition ne compte pas pour 1 contrôle, mais est à inclure dans le contrôle de typage des expressions arithmétiques en général.

Lors de cette réunion nous avons recensé les contrôles sémantiques suivants à implémenter :

- Variables déclarées avant utilisation
  A chaque utilisation, trouver la déclaration associée
- Redéclaration dans le même bloc
- Utilisation de mots clefs réservés

1. par fonction, procédure, bloc sans nom, ...

— Cohérence de types (affectation)
— Cohérence de type entre les paramètres formels et effectifs
— Cohérence de type dans les expressions arithmétiques
— Cohérence entre type de retour attendu (si attendu) et type de retour effectif
— Nombre de paramètre formel = nombre de paramètre effectif
— if(integer) / while(integer)
— Le type d'une variable/fonction existe
— Lors d'un message, la destination existe-t-elle ?
— Division par 0
— Lors d'un override vérifier que la fonction qu'on override existe
— Lors d'un extends vérifier que la classe de laquelle on hérite existe
— L'usage de **result** est interdit dans une méthode qui ne renvoit pas de résultat
— Pas d'usage de result dans les constructeurs
— En cas de cast, vérifier que le cast est fait par une superclasse

Nous nous mettons d'accord sur le fait qu'il faut qu'une personne se charge de la recherche des contrôles sémantiques manquants pour arriver au nombre demandé. Cette personne sera également chargée d'un début d'implémentation de ces contrôles (étudier comment ces contrôles pourraient être faits et mettre en place ce qui peut déjà être fait avant la fin de la TDS.

Prochaine réunion : Pour la prochaine fois :

MASK MASK	moi	MASK	MASK
Mise en place de l'environnement de travail **Java** doté de **Graddle**	Implémentation de l'AST en **Java**	Début d'implémentation de la TDS	Début d'implémentation des contrôles sémantiques
			Recherche des contrôles sémantiques manquants

B CR Réunion 26/01/2021

Présents :
- MASK
- moi
- MASK MASK
- MASK

Heure de début : 16H

Durée : 1h30

Ordre du jour :
- Structure de la TDS
- Remplissage de la TDS
- Table des types
- Contrôles sémantiques
- Résumé des passes du compilateur
- Discussions diverses

I Structure de la TDS

MASKa commencé à implémenter la structure de TDS.
Nous décidons que cette TDS contiendra une **Table de Hashage** remplie avec des **Element**. Les **Element** peuvent être de type fonction ou variable, il conviendra de créer ces sous-classes. moiremarque que des conflits peuvent exister si l'on ajoute une

variable et une fonction à la table des symboles dans la **Table de Hashage**. Il y a 2 solutions possibles :
- MASK MASKsuggère de choisir un identifiant unique comme clef.
- moipropose de créer 2 tables de hachage dans la TDS, une pour les variables, une pour les fonctions
  Cette création s'accompagnait de la modification de la fonction d'insertion d'**Element** dans la TDS. Il faudrait que la fonction distingue avant insertion dans la TDS, la table de hashage de destination.

II Remplissage de la TDS

Le remplissage de la TDS sera assuré par MASK MASKet moi.
moiremarque que des blocs inattendus sont concernés par la construction d'une TDS. Il cite notamment les blocs **Then, Else, Do (While)**.

MASKinciste sur le fait que les numéros de région et d'imbrication doivent être changés à chaque nouvelle table. MASK MASKet moidevront donc veiller lors de la création de nouvelles tables à la prise en compte de ce phénomène. Nous nous mettons d'accord sur le fait que **la classe TDS contiendra un integer *static* pour permettre l'incrémentation des numéros de région.**

Néanmoins, moidevra modifier la fonction de parcours d'arbre pour y intégrer un nouvel argument : la TDS courante.
Lorsqu'un noeud représentera un changement de bloc (exemple : méthode, classe, bloc...), il faudra remplacer la TDS pour les appels récursifs suivants par une nouvelle, préalablement définie comme fille de la courante.

III Table des types

M. Oster nous a suggéré de créer une **table des types**. Celle-ci contiendrait l'ensemble des types Objets définis dans un programme.
MASKsuggère de peupler initialement cette table avec les types primitifs du langage Blood : String et Integer.
Cette table sera ensuite peuplée au cours de la 1ère passe du compilateur.

IV Contrôles sémantiques

Au terme de cette réunion, nous concluons que certains contrôles sémantiques peuvent être effectués pendant la 1ère passe du compilateur, alors que d'autres ne sont pas exécutables à cette étape. En effet, certains contrôles requièrent de posséder l'entièreté de la TDS avant de pouvoir vérifier si un contrôle est conforme.
Il faudra veiller à ne pas utiliser la TDS avant validation de tous les contrôles sémantiques.

Néanmoins, nous nous sommes répartis les contrôles déjà listés dans le CR précédent.

MASKexplique également que, pour réaliser les contrôles sémantiques, il serait pertinent d'utiliser le **design pattern Visiteur**.

Intitulé	Responsable	Etat (OK si implémenté + dans `pres2`)
Redéclaration		
Redéclaration dans le même bloc	MASK	OK
Redéclaration de classe	moi	OK
Redéclaration de méthode dans le même bloc	MASK MASK	OK
Mots Clefs du langage		
Utilisation de mots clefs réservés	MASK	OK
Verification de l'usage de this et super	MASK	OK
L'usage de **result** est interdit dans une méthode qui ne renvoie pas de résultat	MASK MASK	OK
Cohérence		
Cohérence de types (affectation)	MASK MASK	OK
Cohérence de type entre les paramètres formels et effectifs	MASK MASK	OK
Cohérence de type dans les expressions arithmétiques	MASK MASK	OK
Cohérence entre type de retour attendu (si attendu) et type de retour effectif	MASK MASK	OK
Nombre de paramètre formel = nombre de paramètre effectif	moi	OK
Variables déclarées avant utilisation	MASK	OK
if(integer) / while(integer)	moi	OK
Le type d'une variable/fonction existe	MASK	OK
Lors d'un message, la destination existe-t-elle ?	MASK	OK
Division par 0	moi	OK
Division par/de choses inapropriées (String)	moi	OK
Lors d'un override vérifier que la fonction qu'on override existe	MASK	OK
Lors d'un extends vérifier que la classe de laquelle on hérite existe	moi	OK
En cas de cast, vérifier que le cast est fait par une superclasse	MASK	OK
Constructeur		
Pas d'usage de result dans les constructeurs	MASK	OK
Pas de double déclaration de constructeur	moi	OK
Constructeur conforme à la définition de classe Nom du constructeur Nombre de paramètres Type de paramètres	moi	 OK OK OK
Autres		
Toute branche d'exécution contient un retour		
On ne peut ajouter de méthode ou de sous classe aux classes prédéfinies		

V Résumé des passes du compilateur

Nous nous mettons d'accord sur le fait qu'il faut qu'une personne se charge de la recherche des contrôles sémantiques manquants pour arriver au nombre demandé. Cette personne sera également chargée d'un début d'implémentation de ces contrôles (étudier comment ces contrôles pourraient être faits et mettre en place ce qui peut déjà être fait avant la fin de la TDS.

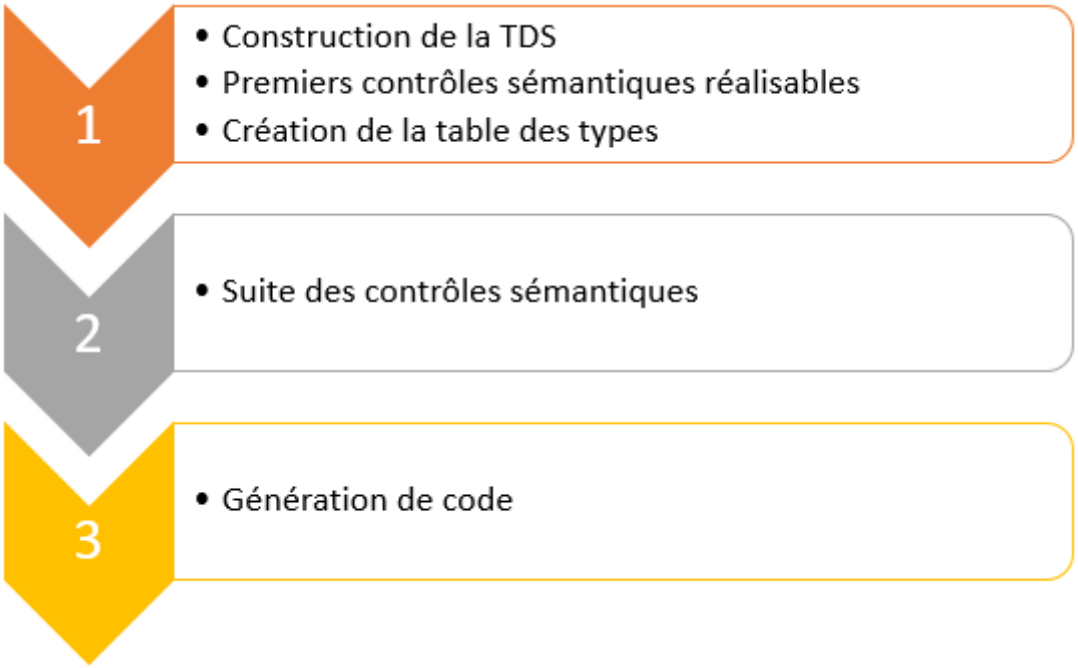


FIGURE B.1 – Passes de notre compilateur

VI Discussions diverses

Package incorrect dans les Lexer et Parser MASKremarque que les Lexer et Parser comportent des imports incorrects, moiva les modifier.

Prochaine réunion : Pour la prochaine fois :

MASK MASK	moi	MASK	MASK
Remplissage de la TDS (19 derniers noeuds)	Remplissage de la TDS (19 premiers noeuds)	Modification de la classe **Element** (suppression des champs inutiles et création des sous-classes pour un Element de type variable et un Element de type fonction)	Création de la classe responsable de la table des types
	Modification de la fonction de parcours de l'AST pour intégrer une TDS en argument	Réflexion sur les conflits possibles lorsqu'une fonction et une variable sont ajoutées dans la TDS	Remplissage de la table des types
Commencer à réfléchir/implémenter les contrôles sémantiques choisis			

C CR Réunion 01/02/2021

Présents :

- MASK
- moi
- MASK MASK
- MASK

Heure de début : 17H

Durée : 1h30

Ordre du jour :

- Présentation du travail effectué depuis la dernière fois
  - TDT
  - TDS
  - Vérification du remplissage de la TDS
  - Nouveaux contrôles sémantiques
- Contrôles sémantiques (dp visiteur ou dans la première passe de manière libre ?)

I TDT

MASKrevient sur la classe TDT qu'il a créée et dont il a assuré le remplissage. A chaque nouveau type créé (nouvelle classe), une nouvelle entrée est faite dans la TDT et ensuite toutes les variables et fonctions afférentes.
MASKprécise que sa classe TDT dispose d'une fonction de recherche :

```
1             public Element get(String type, String attributOuMethode)
2
```

Cette fonction peut notamment être utilisée pour les contrôles sémantiques nécessitant une TDT/TDS complète.

II TDS

MASKet moidétaillent le fonctionnement et le remplissage de la TDS.
MASK MASKsuggère que la TDS devrait comporter une fonction de recherche qui permettrait de savoir si une variable a été déclarée dans la table courante ou dans les

tables parentes.
Cette fonction aurait pour signature :

```
1                public boolean get(String variable, TDS tds)
2
```

III Vérification du remplissage de la TDS

MASK MASKexplique que les `case` dont il avait la charge pour le remplissage de la TDS n'impliquent pas l'ajout d'information à la TDS.

IV Contrôles sémantiques déjà implémentés

moiprésente les contrôles sémantiques qu'il a mis en place.
Il a créé un contrôle sémantique vérifiant que la division par 0 n'est pas pratiquée.

- MASK MASKne juge pas opportun de vérifier que des expressions du type $\frac{numerateur}{12-6-6}$ soient vérifiées
- MASK MASKet MASKrelèvent que pour le moment les expression du type $\frac{1}{str}$ avec str un string (par exemple) sont acceptées, ce qui ne devrait pas être le cas

moia également créé un contrôle sémantique dont le but est de vérifier que if(Integer) est écrit

- MASK MASKne juge pas opportun de vérifier que des expressions du type $12 - 6 - 6$ soient vérifiées
- MASK MASKet MASKrelèvent que pour le moment les expression du type `if(str)` avec str un string (par exemple) sont acceptées, ce qui ne devrait pas être le cas

MASKa créé une fonction de contrôle des types

V Poursuite des contrôles sémantiques : 2e passe

Pour la 2e passe nous avons à décider si nous poursuivons sur le modèle déjà utilisé de parcours de l'AST fourni par **Antlr** ou si nous générons une structure d'AST que nous parcourrons en respectant le design pattern **Visiteur**.
Dans la mesure où nous sommes contraints par le temps et où l'implémentation actuelle est plus simple, nous développerons une fonction de seconde passe adoptant un fonctionnement similaire à celle de première passe.

Prochaine réunion : Pour la prochaine fois :

MASK MASK	moi	MASK	MASK
Implémenter les contrôles sémantiques choisis			
	Corriger les contrôles sémantiques déjà effectués		Création d'une fonction de recherche dans la TDS (en tenant compte des liens de parenté)
	Créer la structure de la fonction de 2e passe		

D CR Réunion 04/03/2021

Présents :

- MASK
- moi
- MASK MASK
- MASK

Heure de début : 15H30

Durée : 1h30

Ordre du jour :

- Présentation des recherches effectuées depuis la dernière fois
  - Documents fourni par Mme Collin
  - Document fourni par Clément Crouzet
  - Autres recherches
- Tâches à effectuer
- Manière de procéder

I Présentation des recherches effectuées depuis la dernière fois

Nous avons pu expérimenter le code assembleur et son exécution grâce aux différents documents qui nous sont parvenus.
Nous notons la nécessité de réimplémenter les fonctions de base utilitaires pour n'importe quel morceau de code `.blood` à assembler : `print, println...`.

II Tâches à effectuer

Il ressort de nos échanges que les premières tâches à effectuer sont :

- Créer un générateur de fichier `.src`. Le générateur devra initialiser le fichier avec les registres d'usage et les fonctions de base nécessaires à tout programme `.blood`.
- Créer une fonction de 3^{e} passe.
- Générer le code assembleur correspondant à la **déclaration , aux affectations et aux opérations arithmétiques.**
- Créer le descripteur de classe.
- Générer le code assembleur correspondant aux `if` et `while`.

III Manière de procéder

Nous convenons que chacun devra effectuer ses tests dans des fichiers de tests `.blood` séparés et correspondant donc à des **tasks graddle** distinctes.
De plus, moise chargera de créer un sous ensemble de taks pour ne pas empiéter sur les builds des fichiers issus des soutenances précédentes.
Par ailleurs, nous notons la nécessité de procéder de manière organisée dans les tâches sus-citées (nécessité de respecter l'ordre établi). **Enfin, nous nous rappelons que nous avons pris pour convention 0 = Faux et != 0 -> Vrai pour les vérifications booléennes des if et while à suivre.**

Prochaine réunion : Pour la prochaine fois :

MASK MASK	moi	MASK	MASK
Générer le code assembleur pour — Déclarations — Affectations — Opérations + / - *	— Créer la génération du fichier `.src` correspondant — Initialiser le fichier avec les fonctions essentielles et les registres — Créer les taks graddle afférentes	— Créer la fonction de 3e passe — Générer le code assembleur pour les opérations &&, <>, =, <, >, ||, &	Descripteurs de classe

E CR Réunion 11/03/2021

Présents :

- MASK
- moi
- MASK MASK
- MASK

Heure de début : 17H15

Durée : 1h

Ordre du jour :

- Présentation des développements effectués depuis la dernière fois
- Tâches à effectuer
- Manière de procéder

I Présentation des développements effectués depuis la dernière fois

- MASK, en charge du descripteur de classe, nous présente le fonctionnement du descripteur de classe qu'il envisage d'implémenter.
  Il n'a pas encore pu mettre en oeuvre ce qu'il souhaite faire car il aurait besoin d'accéder au tas, mais ne possède pas la documentation de l'assembleur de M. Parodi. Il a dû créer une quatrième passe (ici c'est la 3ème qu'il utilise) pour créer les descripteurs de classe.
  Il nous apparaît nécessaire de découper le fichier contenant l'ensemble des passes, celui-ci devient en effet trop volumineux.
- MASK MASKnous présente le fonctionnement des déclarations et affectations de variable, actuellement les déclarations et affectations de **Integer** et **String** sont reconnues.
- MASK, en charge des opérations arithmétiques de comparaison et de concaténation a d'abord commencé par implémenter la concaténation. Cette opération étant particulièrement difficile, elle préfère pour le moment s'occuper des opérations de comparaison.
  Par ailleurs, nous avions défini à la précédente réunion que la structure du **if** était a implémenter rapidement ; ces opérateurs de comparaison nous seront particulièrement utiles à cette étape.
  Nous convenons qu'il faut mettre en place un fichier recensant les registres utilisés, cela permettant d'en affecter un pour le résultat des opérations de comparaison.

— moiprésenter le remplissage des fichiers **.src**. A chaque task Gradle d'assemblage, un fichier .src est généré. Ce fichier contient un ensemble de fonctions de base (print, ntoa...).
Il s'est ensuite chargé de la mise en place du pipeline. Celui-ci vérifie à chaque push sur le dépôt que les fichiers commités compilent, s'assemblent et s'exécutent bien.
Il a également pris en charge les fonctions **print/ln**.

II Tâches à effectuer pour la prochaine fois

Il ressort de nos échanges que les tâches à effectuer sont :

— Nous notons la nécessité d'implémenter rapidement la fonction **toString** pour vérifier le résultat des opérations arithmétiques. Nous remarquons qu'une telle fonction est déjà implémentée dans les fichiers fournis par Mme Collin. Il s'agira de l'intégrer et adapter au projet.

— Nous poursuivons nos recherches sur le descripteur de classe.

— Nous développerons la structure de **if**.

Prochaine réunion : Pour la prochaine fois :

MASK MASK	moi	MASK	MASK
— Découper le fichier contenant toutes les passes — Opérations arithmétiques	— Fonction **toString** — Structure de if	Opérations arithmétiques	Descripteur de classes

Éléments post réunion : Nous avons appris que l'assembleur de M. Parodi ne possède pas de zone spéciale pour le tas. Il faut donc stocker en mémoire ce tas fictif de la même manière que la pile à la fin de la zone d'adressage.

F CR Réunion 25/03/2021

Présents :

- MASK
- moi
- MASK MASK
- MASK

Heure de début : 14H30

Durée : 1h

Ordre du jour :

- Tâches effectuées depuis la dernière réunion
  - Descripteur de classes
  - Gestion des objets
  - Rajouts dans le rapport
  - Opérations de comparaison
  - **If/While**
  - Correction du print
- Tâches à effectuer
  - Reste des opérations
  - **toString**
- Manière de procéder

F.1 Tâches effectuées depuis la dernière réunion

I Descripteur de classes

MASKnous présente le fonctionnement du descripteur de classes.
Il permet actuellement les manipulations simples sur les objets : leurs valeurs (les appels de fonctions ne sont pas gérés actuellement).

```
1  class Nb2() extends Test is{
2      var x:Integer;
3      var y:String;
4      var z:Nb2;
5      def Nb2() is{}
6      def a() is{}
7   }
8  class Test() is {
9      var attribut: Integer;
10     def test() is{
11    }
12       def a() is{}
```

```
13 }
14 {
15       a: Nb2;
16       is
17       a:= new Nb2();
18       a.x := 15;
19       a.z:=new Nb2();
20       a.z.z := new Nb2();
21       a.z.z.z := new Nb2();
22       a.z.z.z.x := 15;
23 }
```

II Rajouts dans le rapport

MASKa rajouté dans le rapport les parties qu'il a développées : Descripteur de classes et Tas.
moia rajouté un paragraphe sur l'intégration continue dans le projet.

III Opérations de comparaison

Les opérateurs de comparaison ont été implémentés pour 4 cas :

- nombre operateur nombre
- nombre operateur variable
- variable operateur nombre
- variable operateur variable

IV If/While

Le **If** et **While** ont été implémentés.
Il faut tester pour le **While** si la condition d'arrêt est bien respectée.
MASKprécise qu'actuellement il est difficile de le tester dans la mesure où la prise en compte du changement de bloc pour les variables n'est pas encore faite.

V Correction du print

MASKa identifié une erreur dans l'appel à la fonction print et la corrigée.

F.2 Tâches à effectuer pour la prochaine fois

I Reste des opérations

Actuellement, les opérations **+, -, *, /** sont en cours d'implémentation.
Les opérations de comparaison (les opérations simples devront aussi gérer ces cas) sont à perfectionner sur les points suivants :

- Gestion des variables de type objet

```
1       ...
2       if( x.y.z <> a.b)
3
```

- Gestion des cascades d'opérations

```
1       3 + 5 - 8 <> 4 * 22
2
```

MASKet MASK MASKse mettent d'accord sur le fait qu'il faut que le résultat des opérations précédentes soit systématiquement mis dans R1 de manière à pouvoir utiliser le résultat sur une séquence d'opérations.

II toString

La fonction **toString** est à implémenter. On pourra se resservir du travail de MASKsur les String.

Prochaine réunion : Pour la prochaine fois :

MASK MASK	moi	MASK	MASK
— Finir les opérations — Mettre en application la norme convenue avec MASKpour les opérations séquentilles — Gérer le cas des variables issues d'objets — Rajouter dans le rapport ses développements	— Fonction **toString** — Tester **if/while** — Nettoyer le fichier **src** des fonctions non utilisées — Résoudre le problème de **println** — Rajouter dans le rapport ses développements	— Mettre en application la norme convenue avec MASK MASKpour les opérations séquentielles — Gérer le cas des variables issues d'objets — Rajouter dans le rapport ses développements	— Créer une fonction qui stocke dans un registre l'adresse d'une variable de type objet — Créer le passage d'un bloc à l'autre (gestion au niveau du chaînage statique et des TDS) — S'informer sur le polymorphisme — Rajouter dans le rapport ses développements

Éléments post réunion : MASKa mis en place une fonction pour accéder aux éléments objets, elle sera utile pour tous les types d'opérations.

G CR Réunion 01/04/2021

Présents :

- MASK
- moi
- MASK MASK
- MASK

Heure de début : 14H30

Durée : 1h

Ordre du jour :

- Tâches effectuées depuis la dernière réunion
  - Descripteur de classes
  - Gestion des objets
  - Rajouts dans le rapport
  - Opérations de comparaison
  - **If/While**
  - Correction du print
- Tâches à effectuer
  - Reste des opérations
  - **toString**
- Manière de procéder

G.1 Tâches effectuées depuis la dernière réunion

I MASK

- Correction du **print**.
- Création d'une fonction pour récupérer l'adresse d'un objet.
- Correction du passage d'un bloc à l'autre.
- A rajouté dans le rapport les explications de ses développements.
- A modifié le descripteur de classe : fonctionne correctement maintenant (est laissé vide puis est rempli avec les adresses des méthodes).

II MASK MASK

- Opérations arithmétiques quasiment terminées : l'addition est finie mais pas les autres

III MASK

Opérateurs de comparaison en cours et pas encore push.

IV moi

La fonction **toString** fonctionne pour les entiers inférieurs à 65537.
Pour le moment son fonctionnement n'est pas optimisé car elle utilise une l'ensemble des registres. De plus, elle affiche des 0 en trop.

Prochaine réunion : Pour la prochaine fois :

MASK MASK	moi	MASK	MASK
— Finir les opérations — Prendre en compte le cas du **PLUS_UNAIRE** et le **MOINS_UNAIRE** — Mettre en application la norme convenue avec MASKpour les opérations séquentielles — Gérer le cas des variables issues d'objets — Rajouter, une fois fini le code dans la 3e passe — Rajouter dans le rapport ses développements	— Fonction **toString** à optimiser — Ne plus utiliser autant de registres (**R1, R2, R3, R4, R5** à éliminer). — Résoudre le problème des 0 en trop. — Tester **if/while** — Tester **println** — Affectation du **PLUS_UNAIRE** — Rajouter dans le rapport ses développements	— Finir les opérations. — Mettre en application la norme convenue avec MASK MASKpour les opérations séquentielles — Gérer le cas des variables issues d'objets — Prendre en compte le cas du **PLUS_UNAIRE** et le **MOINS_UNAIRE** — Rajouter le code une fois fini dans la troisième passe — Rajouter dans le rapport ses développements	— Résoudre le bug découvert pendant la réunion — Appel de fonction dans les paramètres de fonction **f1(f2(arg), f3(arg1))** — Polymorphisme (liaison dynamique et surcharge) — super()

H CR Réunion 15/04/2021

Présents :

- MASK
- moi
- MASK MASK
- MASK

Heure de début : 13H30

Durée : 1h

Ordre du jour :

- A faire
  - Code
    - 3e passe
  - Rajouts dans le rapport
  - Tests
  - Soutenance
    - Test
    - Rapport

Prochaine réunion : Pour la prochaine fois :

MASK MASK	moi	MASK	MASK
— Rajouter le code des opérations dans la 3e passe — Gérer le cas des variables issues d'objets et les inclure dans les tests. — Compléter le rapport sur le code, le temps de travail et le bilan du projet	— Vérifier si le code de la concaténation est présent dans la 3e passe — Vérifier si le contrôle sémantique d'appel des méthodes **static** est correct. — Identifier d'où provient l'erreur qu'a exposé MASKpendant la réunion — Faire un exécutable pour la soutenance	— Identifier d'où provient l'erreur qu'a exposé MASKpendant la réunion — Faire des tests sur les opérateurs — Vérifier si le code des opérations booléennes est présent dans la 3e passe. — Compléter le temps de travail et le bilan du projet	— Vérifier si les méthodes **static** sont gérées. — Inclure **super** et **static** dans les tests — Identifier d'où provient l'erreur qu'a exposé MASKpendant la réunion — Faire une documentation technique, notamment sur la liaison entre les différentes classes et sur attributDansR0. — Compléter le rapport sur le code, le temps de travail et le bilan du projet. — Faire le programme de tests