Lecture et représentation d'un programme assembleur MIPS

Emel Hajjem ; Victor Lambret

2011

Abstract

Ce rapport présente la spécification, la réalisation et la procedure de recette de notre projet PSESI dont le but est de fournir une représentation d'un programme assembleur MIPS et un ensemble de fonctions de manipulation de cette représentation.

Contents

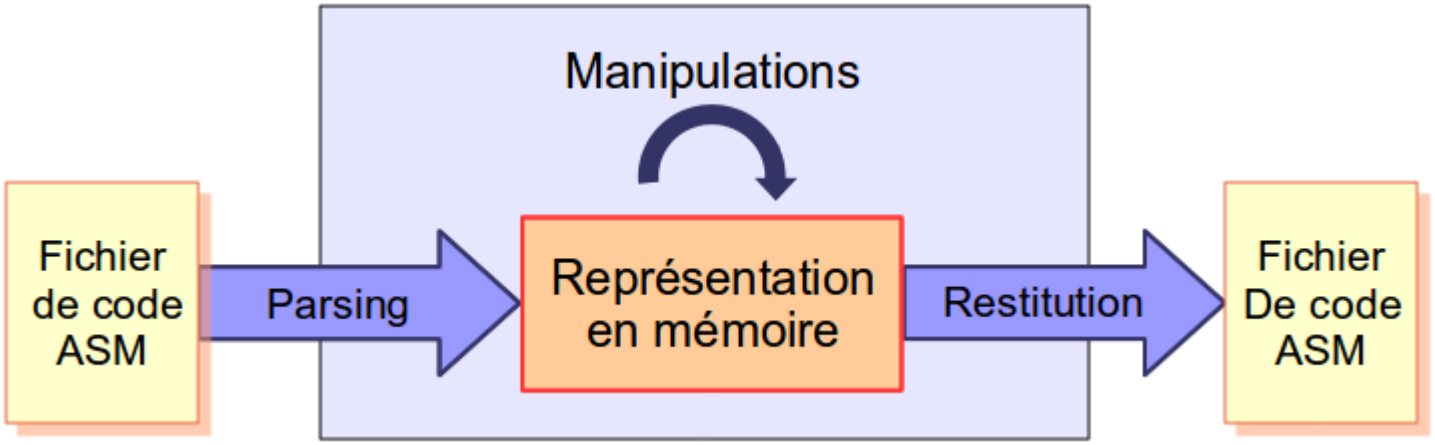

Figure 1: Schéma de fonctionement de la bibliothèque

Part I
Spécification

1 Contexte

Le but de notre projet est de fournir une représentation d'un programme assembleur MIPS. Aussi, nous lui ajouterons un ensemble de fonctionnalités de manipulation de programme. Celles-ci constitueront une brique de base qui permettra des développements ultérieurs, notamment dans le but de calculer des temps d'execution ou de mettre en oeuvre des techniques d'optimisation ou d'analyse de code. Cet outil sera étendu puis utilisé dans le cadre des enseignements de M1 d'architecture ou de compilation avancée.

Ce projet sera réalisé en binôme, Victor Lambret et Emel Hajjem et encadré par Karine Heydemann.

Il faut savoir que ce projet demande ou developpera des connaissances autour des notions suivantes.

- Le jeu d'instruction MIPS étant donné que l'on va manipuler du code assembleur MIPS.
- Un langage de programmmation qui permettra l'implémentation de la solution retenue pour ce projet.
- L'anglais technique pour la rédaction des commentaires et de la documentation.
- Des techniques et outils de parsing, qui permettront de lire un fichier assembleur.
- La structure d'un programme assembleur et des notions relatives à la compilation.

2 Objectifs

L'objectif du projet est de récupérer un code assembleur MIPS écrit à la main ou produit par un compilateur gcc, via un parseur, et de le représenter en mémoire grâce à des structures de données. Les manipulations de code suivantes sont aussi souhaitées.

- Modification d'un opérande.
- Ajout et suppression d'instructions.
- Calcul de dépendance entre deux instructions.
- Calcul des blocs de base d'une fonction.

Enfin l'outil réalisé doit être capable de reproduire le code initial ou modifié dans un fichier.

C'est dans cette perspective que nous allons réaliser une bibliothèque, afin qu'un developpeur puisse inclure facilement les fonctionnalités que nous proposons dans son programme.

Il faut donc :

- créer un parseur qui lira le code assembleur initial dans un fichier,
- définir et implanter des structures de données appropriées pour la représentation du programme,
- définir et implanter les opérations de manipulation,
- fournir une bibliothèque permettant de récupérer la représentation de la manipuler et d'en générer un fichier.

3 Organisation

3.1 Méthode de travail

Notre projet consistant à développer un même outil à deux avec une interdépendance entre les tâches à réaliser, nous allons utiliser un système de gestion de versions pour une collaboration plus aisée. De plus, nous allons adopter des méthodes de travail inspirées de l'Extreme Programming^[http://fr.wikipedia.org/wiki/EXtreme_Programming], consistant à :

- privilégier les solutions les plus simples pendant le développement et optimiser à la fin du projet en fonction du besoin,
- découper le travail en cycles de développement courts de manière à intégrer au plus vite les changements dans l'archivage,
- valider le bon fonctionement des différentes parties de la librairie par des tests unitaires,
- définir et utiliser un ensemble de règles communes de codage (nommage, commentaire...).

Toute réalisation de code suivra le schéma suivant :

1. Écriture de tests.
2. Écriture du code et de la documentation de la classe associée.
3. Validation par un passage réussi des tests.

En particulier nous nous répartirons le travail entre l'écriture des tests et du code.

3.2 Liste des taches

Nous pouvons découper notre travail en trois grandes tâches pour réaliser nos objectifs.

3.2.1 Structures de donnée

Pour écrire ces structures de données nous avons choisi de les implémenter grace au langage C++.

Mais avant de passer à la programmation, il faut dans un premier temps définir ces structures. Pour mieux les visualiser, nous allons les représenter dans un diagramme UML.

Comme nous voulons offrir la possibilité de reproduire le code, il faut conserver toutes les informations. Nous avons donc décider de voir un programme comme un ensemble de lignes. Parmis ces lignes, seules celles correspondant aux instructions nous interessent. Toutefois, pour des developpements ultérieurs, il est utile de pouvoir accéder à toutes les informations (toutes les lignes) et donc de conserver celles-ci. Dans le but d'analyse de code ultérieure, on souhaite connaitre pour chaque instruction son code opérateur et pouvoir le modifier, sa nature (opération arithmétique ou logique, opération mémoire ou branchement) ainsi que ses opérateurs. Les opérandes sont soit des registres, des immédiats des expressions ou des labels. Les labels sont aussi importants, ils indiquent notamment le début des fonctions et des certains blocs de base. Ainsi, nous avons identifié les entités suivantes pour lesquelles nous developperons des structures de données.

- Programme : entité correspondant à un ensemble de lignes, parmis lesquelles nous identifirons celles correspondant aux début de fonction en les conservant afin d'avoir un accès direct aux fonctions.
- Ligne : entité correspondant à une ligne du code asm d'origine, pouvant être une directive, une déclaration, un label ou une instruction.

- Label : entité composée d'un nom.
- Instruction : entité composée d'un code opération et d'opérandes. Nous conservons son format et déduirons son type du code opération.
- Opérande : entité correspondant à un registre (son numéro), un label (chaîne ou identifiant), un immediat (valeur numérique) ou une expression.
- Découpage en sous-tâches :
- Diagramme : réalisation du diagramme UML précis des classes utilisées.
- Classes de base : réalisation des classes Programme, Ligne, Instruction, Label et Opérande.

3.2.2 Implémentation des fonctionnalités de manipulation

Après avoir défini nos structures de données, nous pourrons passer à la définition et à l'implémentation des différentes fonctions qui vont permettre de manipuler le code initial.

Découpage en sous-tâches :

- Calcul de dépendance : réalisation de la fonctionnalité de calcul de dépendance entre deux instructions.
- Index : réalisation de la fonctionnalité de calcul des blocs de base et de fonctions d'accès indexées aux lignes du programme par les labels, blocs de base ou fonctions.
- Restitution : réalisation de la fonctionnalité permettant d'écrire la totalité ou une partie du programme sous forme d'un fichier d'instructions assembleur.

3.2.3 Parsing

Notre projet nécessite un parseur qui fera l'analyse lexicale et syntaxique du code assembleur MIPS donné en entrée. L'écriture d'un parseur étant une tâche qui prendrait beaucoup de temps, nous allons réutiliser un parseur écrit par Mr Pirouz acceptant en entrée des fichiers assembleur MIPS produit par gcc. Nous aurons à comprendre, puis nettoyer celui-ci avant de modifier les actions afin de produire la représentation du programme avec nos structures de données.

Découpage en sous-tâches :

- Nettoyage : nettoyage du parseur existant pour en conserver les règles d'analyse.
- Parsing : construction de l'entité programme à partir du parsing d'un fichier assembleur.

3.3 Outils

Notre projet doit pouvoir être utilisé et adapté facilement, nous allons donc le développer grâce à des outils standards et libres :

- g++ pour compiler la librairie et le code de test(C++),
- gccpour mips installé via crosstools pour compiler les programmes d'exemples écrits en C vers de l'assembleur MIPS,
- make pour la gestion du projet,
- un gestionnaire de version comme git,
- Doxygen pour générer la documentation en anglais à partir de commentaires bien écrits dans le code du projet.

3.4 Planning

Nos tâches étant assez dépendantes entre elles, nous nous sommes répartis le travail pour le démarrage du projet. Il nous est assez difficile de donner une estimation précise du temps que prendra chaque tâche, c'est pourquoi les tâches suivantes seront attribuées en fonction de l'avancement.

L'ordre de réalisation des tâches sera dicté par les dépendances que nous avons identifiées dans ce schéma :

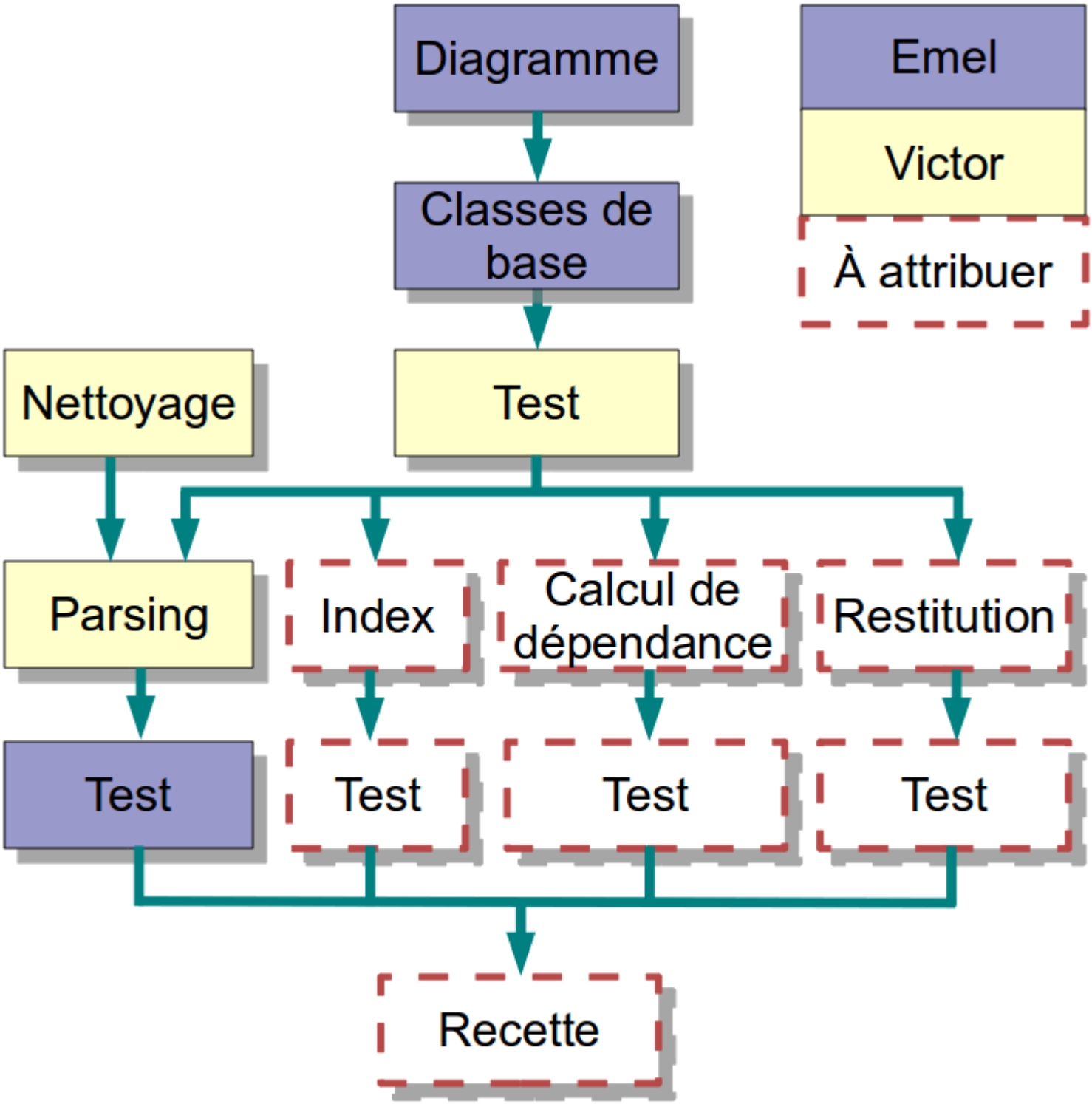

Figure 2: Planning

4 Recette

4.1 Documents fournis

A la fin de notre projet, nous produirons les documents suivants.

- Le diagramme de classes UML du projet.
- Le code de la bibliothèque de base.
- Le code des classes de tests unitaires et de l'ensemble des outils de test.
- La documentation de la bibliothèque générée par Doxygen.

4.2 Procédure de recette

La librairie produite sera validée par deux procédures de recette. Tout d'abord une recette continue composée de test unitaires, de nos revues de code et de la validation des étapes par notre encadrante.

La seconde partie de la recette sera basée sur des scénarios testant chacun un ensemble de fonctionnalités.

4.2.1 Acquisition d'un programme avec le parseur et restitution

Nous allons écrire trois petits programmes en langage C, typiquement de parcours de tableau, de multiplication de matrices ou de tri rapide. Ces programmes seront compilés vers de l'assembleur avec gcc puis représentés en mémoire en utilisant le parseur avant d'être restitués en assembleur. Nous vérifierons que les programmes restitués sont identiques aux programmes d'origine, en particulier qu'ils peuvent toujours être compilés et produire le même code exécutable.

4.2.2 Manipulation d'un programme

A partir de ces trois même programmes d'exemple, nous effectuerons les opérations suivantes.

- Affichage des informations générales du programme comme le nombre de lignes et d'instructions.
- Calcul du découpage en blocs de base et en fonctions.
- Écriture dans un fichier des séquences de lignes de ces blocs de base et fonctions dans des fichiers différents.

4.2.3 Manipulation d'instructions

Nous allons démontrer les possibilités de manipulation, insertion et suppression de lignes en transformant la séquence d'instructions suivante :

```
# Code avec des Nor # ligne 1
add $5, $0, $0 # ligne 2
or $6, $5, $0 # ligne 3
nor $4, $5, $6 # ligne 4
```

Pour obtenir le programme suivant :

```
# Equivalence en plusieurs instructions # ligne a
# ligne b
lui $4, 0xFFFF # ligne c
ori $4, $4, 0XFFFF # ligne d
xor $5, $4, $0 # ligne e
xor $6, $4, $0 # ligne f
and $4, $5, $6 # ligne g
```

En particulier :

- le contenu du commentaire de la ligne 1 sera modifié,
- la ligne 2 sera supprimée,

- les opérandes et l'opérateur de la ligne 3 seront modifiés pour obtenir la ligne c,
- l'opérateur de la ligne 4 sera modifié pour obtenir la ligne g,
- les autres lignes seront ensuite insérées dans le programme.

4.2.4 Calcul de dépendance entre instructions

En partant de cette séquence d'instructions :

```
# Exemple de programme utilisé pour la recette
1 lw $4, 0($0)
2 lw $4, 0($8) # Write after Write avec 1
3 addi $5, $4, 10 # Read after Write avec 2
4 sw $4, 0($8)
5 sub $4, $10, $5 # Write after Read avec 4 et 3
6 addi $31, $31, 4 # instruction indépendante
```

Nous effectuerons les opérations suivantes :

- Calcul et affichage des dépendances entre instructions.
- Inversion de deux instructions indépendantes.
- Production du fichier assembleur résultat.

Part II
Réalisation

Une fois les spécifications établies, nous sommes donc passés à la réalisation de notre bibliothèque. L'objectif du projet étant de lire un code assembleur MIPS écrit à la main ou produit par un compilateur de type gcc, via un parseur, de le représenter en mémoire et d'effectuer dessus des manipulations, nous avions défini trois grandes tâches : définition et implantation des structures de données, l'implémentation des fonctionnalités de manipulation et le parsing. Nous allons donc vous présenter dans cette partie la réalisations de ces tâches.

5 Classe de bases

Les structures de données représentent le programme contenu dans un fichier assembleur (données, code). Ces structures vont être répariti en classes représentant les différents objets qui caractérisent un code assembleur. Nous avons donc construit nos classes de bases de manière hiérarchique. Tout d'abord nous avons le programme que nous avons choisi de voir comme une suite de lignes qui peuvent être de différents types : labels, instructions et directives. Ce sont les instructions qui sont, pour nous, les plus importantes, elles ont chacunes un code opérateur, des opérandes, un format de codage(R, I, J, O) et une nature (BR, MEM, ALU, OTHER).

- Le programme est vu comme une liste chaînée de lignes. La classe Node correspond aux noeuds de la liste chaînée. Dans ce programme, nous pouvons supprimer ou ajouter des lignes. Nous pouvons aussi connaitre le nombre de ligne du programme, l'afficher et chercher une ligne en particulier.
- La classe Line correspond à l'information contenue dans un noeud de la liste chaînée. C'est une classe abstraite qui permet aux sous-classes Instruction, Directive et Label d'hériter de ses caractéristiques. Chaque ligne a pour attribut une chaîne de caractère. Nous pouvons modifier cette ligne, récupérer la ligne ou son type (instruction, label ou directive).
- Les directives correspondent aux lignes de déclaration de zone (.data, .text) de données, de fonction. La directive peut être suivi d'un champs. Cette classe peut identifier le début et la fin d'une fonction(.ent, .end), ce qui est utile lors du calcul des fonctions.
- Les labels sont les étiquettes utilisées pour désigner des données ou instructions dans le code.

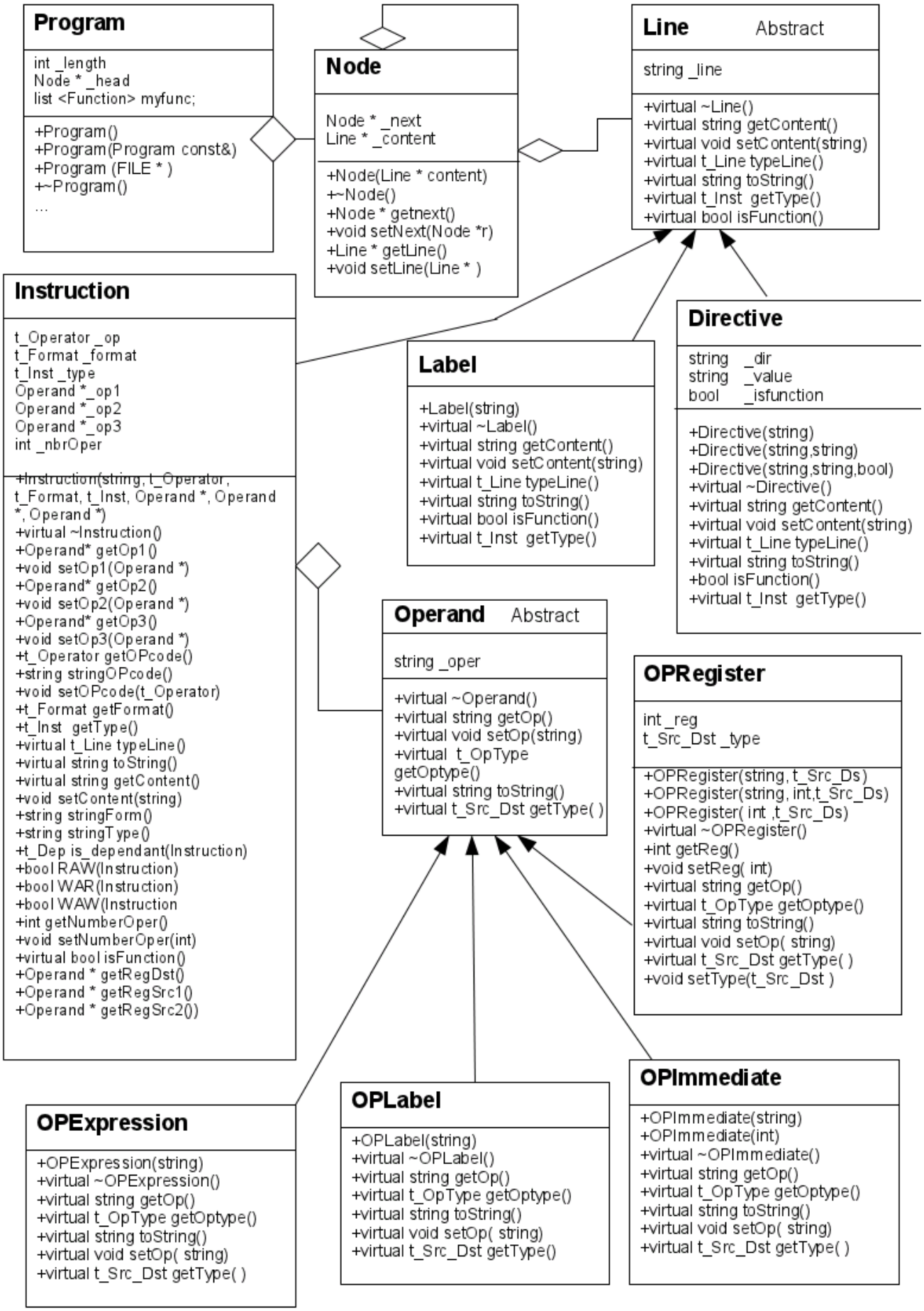

Figure 3: Diagramme UML

- Les instructions correspondent au code proprement dit. Une instruction contient un code opérateur, des opérandes, son type et son format. Dans cette classe nous pouvons récupérer ses caractéristiques mais nous ne pouvont modifier que son code opérateur et ses opérandes car le type et le format sont déduit de l'opérateur et des opérandes.
- Le code opérateur est représenté par un type énuméré, qui lui même représente toutes les mémoniques asm.
- Les opérandes sont des objets dérivant de la classe abstraite Operand. Il y a une classe par type d'opérandes possibles : registre, immediat, expression ou label. Pour toutes ces classes nous pouvons modifier l'opérande et la récupérer, elle et son type(registre, label...).

6 Index

La fonctionnalité d'indexation permet d'accéder aux fonctions d'un programme et d'accéder aux blocs de bases d'une fonction.

6.1 Fonction

Un programme peut contenir plusieurs fonctions. Nous avons donc implémenté une fonctionnalité permettant de délimiter ces fonctions. Le calcul des fonctions se fait dans la classe Program qui a une vue d'ensemble de tout le code assembleur, donc de toutes les fonctions définies. On conserve ces fonctions dans une liste stockée dans le programme correspondant. La classe Function représente une fonction du programme. Elle contient un pointeur sur la ligne de début et la ligne de fin de la fonction. La classe offre la possibilité d'afficher le contenu de la fonction, de restituer le code lui correspondant dans un fichier et de calculer sa taille.

Voici l'algorithme en pseudo code du calcul des fonctions d'un programme :

```
compute_function()
beginFunction function
 if (program.is_not_Empty()) then
        Node current_node<-program.head while (current_node) then
                if (current_node = directive_of_end_function) then
                        function.end <-current_node
                        program.add_list_function ( function)
                end if
                if (current_node = directive_of_begining_function) then
                        function.head <- current_node
                end if
                if (current_node.next_node = program.end) then
                        break
                else current_node <- current_node.next_node
                end if
        end while
end
fin
```

6.2 Bloc de base

Dans chacune des fonctions d'un programme il y a plusieurs blocs de bases. Ces blocs de bases sont des séquences d'instructions contenant un seul point d'entré et un seul point de sortie. La classe Basic_block représente un bloc de base. Elle contient un pointeur sur la ligne de début (une instruction ou un label) et la ligne de fin du bloc. Cette classe offre la possibilité d'afficher le contenu du bloc, le restituer dans un fichier et calculer sa taille. Le calcul des blocs de bases se fait dans la classe Function, qui va conserver ces blocs dans une liste.

Abstract

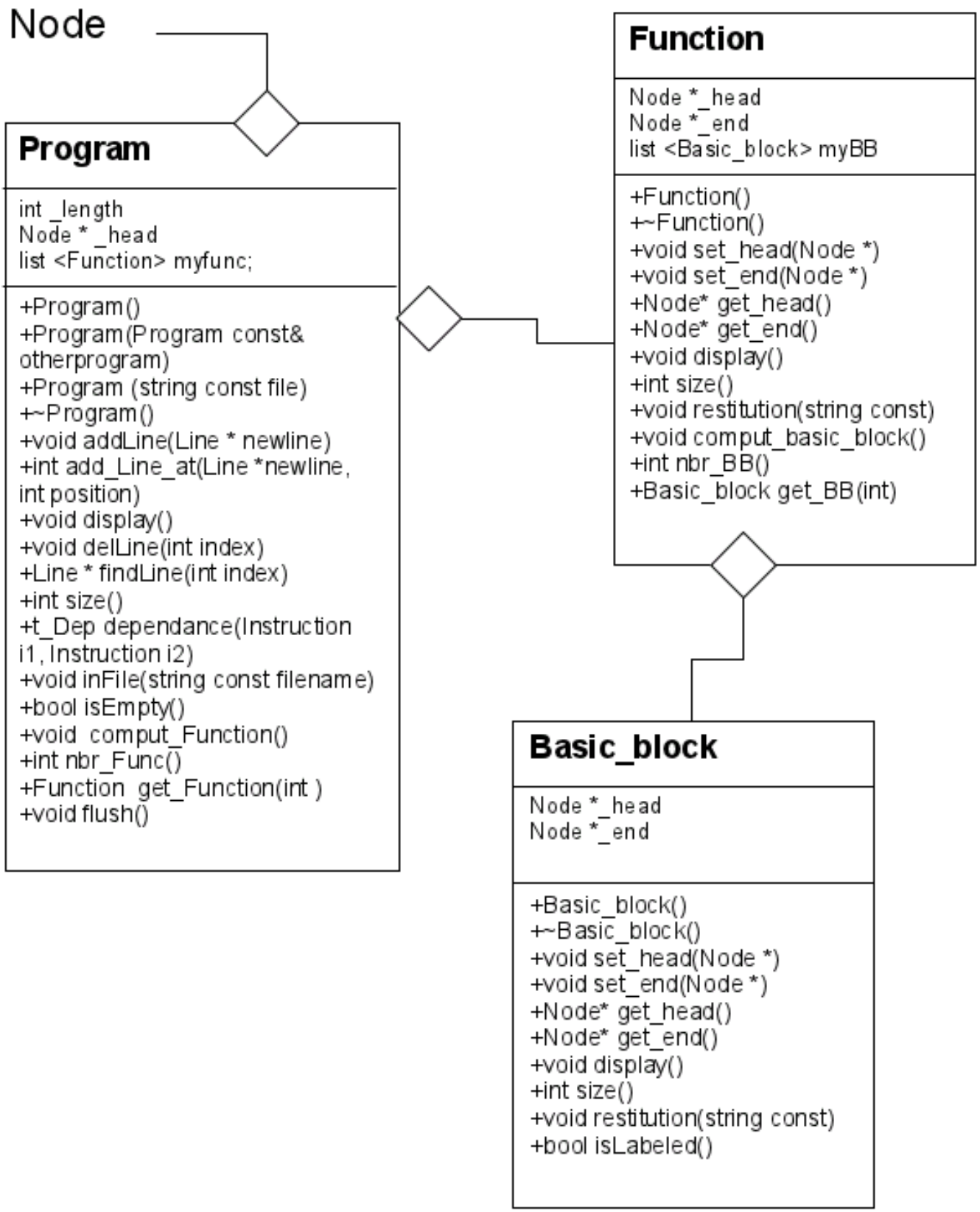

Figure 4: Diagramme UML index

Voici l'algorithme en pseudo code du calcule des blocs de bases dans une fonction:

```
compute_Basic_block()
begin
        Node current_node <- function.head
        block.head <- function.first_node_label_or_instruction
        while (current_node != function.end) then
                if (current_node = instruction)
                        if (current_node = branch) then
                                block.end <- current_node.Nem_next_node(N delayed slot)
                                function.add_list_block ( block)
                                if (current_node.next_node != function.end &&
                                 current_node.next_node._next_node != function.end)
                                                block.head <- current_node.next_Nem_node
                                                (the node next the Nth delayed slot)
                                        current_node <-  current_node.next(block.head)
                                                else break
                                end if
                        end if
                end if

                if (current_node.next_node = label) then
                        block.end <- current_node
                        function.add_list_block ( block )
                        block.head <- current_node.next_node
                end if

                if (current_node.next_node = directive) then
                        block.end <- current_node
                        function.add_list_block ( block )
                        break
                end if

                if (current_node.next_node = function.end) then
                        block.end <- current_node
                        function.add_list_block ( block)
                        break
                else current_node <- current_node.next_node
                end if
        end while
end
```

7 Calcul de dépendance

Nous avons réalisé la fonctionnalité qui permet de calculer la dépendance entre deux instructions. Pour cela nous avons implémenté des méthodes correspondant à chacunes des dépendances possibles dans la classe Instruction:

- RAW: détéction des dépendances lecture après écriture entre deux instructions i1 et i2, c'est à dire la lecture d'un registre par i2 après son écriture par i1.
- WAR: détéction des dépendances écriture après lecture entre deux instructions i1 et i2, c'est à dire l'écriture dans un registre par i2 après sa lecture par i1.
- WAW: détéction des dépendances écriture après écriture entre deux instructions i1 et i2, c'est à dire l'écriture d'un registre par i2 après l'écriture par i1.

Ces méthodes testent si la dépendance est verifiée entre l'instruction courante et celle donnée en paramètre. Nous avons aussi implémenté une méthode permettant de calculer la dépendance entre deux instructions particulières dans un programme.

Voici les algorithmes pour chacun des types de calcul des dépendances :

7.1 Read after Write

Cette fonction renvoie vrai si il y a une dépendance de type WAR entre l'instruction courante et i2 passé en paramètre, sinon retourne faux.

```
begin
        if (self.RegDest && i2.RegSrc1 && self.RegDest==i2.RegSrc1)      return true
        if (self.RegDest && i2.RegSrc2 && self.RegDest==i2.RegSrc2)      return true
        return false
end
```

7.2 Write after Read

Cette fonction renvoie vrai si il y a une dépendance de type WAR entre l'instruction courante et i2 passé en paramètre, sinon retourne faux.

```
WAR(Instruction i2)
begin
        if (i2.RegDest && self.RegSrc1 && i2.RegDest==self.RegSrc1)      return true
        if (i2.RegDest && self.RegSrc2 && i2.RegDest==self.RegSrc2)      return true
        return false
end
```

7.3 Write after Write

Cette fonction renvoie vrai si il y a une dépendance de type WAW entre l'instruction courante et i2 passé en paramètre, sinon retourne faux.

```
WAW(Instruction i2)
begin
        if (self.RegDest && i2.RegDest && self.RegDest==i2.RegDest)      return true
        return false
end
```

8 Restitution

Nous avons réalisé la fonctionnalité qui permet de restituer un programme, une fonction ou un bloc de base sous forme d'un fichier d'instruction asssembleur. La restitution se fait dans une méthode de chacune des classes associée. Ces méthodes prennent en parametre le nom du fichier dans lequel on veux écrire. On parcour la structure, récupère la chaîne de caractère de chaque ligne et on l'écrit dans le fichier.

9 Parseur

9.1 Départ d'un parseur existant

Nous sommes partis sur un simulateur déjà existant écrit par Mr Pirouz utilisant un parseur pour interprêter un code assembleur mips. La partie parsing était basée sur lex comme analyseur lexical et yacc comme analyseur syntaxique. Nous souhaitions récupérer les règles de découpage en mots clés ainsi que la grammaire et remplacer tous les actions du parseur par des appels aux constructeurs de notre bibliothèque de base.

Dans un premier temps nous avons cherché à isoler le code de ce parseur du reste du simulateur afin de le compiler dans un premier temps, puis l'adapter à notre besoin. Cette étape nous a fait conserver uniquement 3 élements :

- Les règles dans deux fichiers lex et yacc,
- une bibliothèque asm contenant les fonctions utiles pour les actions,
- une biblothèque utl de fonctions de manipulations de données pour le passage de données lex à et yacc, ou entre actions.

Dans un premier temps nous avons commenté les actions existantes pour n'avoir qu'un simple affichage du nom de la règle parcourue afin de tester la seule compilation du code. Cette étape s'étant bien passée, nous avons pensé

9.2 Le passage au C++

Pour que notre parseur fasse appel à des constructeur C++ dans les actions nous avons besoin de compiler le fichier yacc avec g++. A priori g++ pouvant compiler du C nous avons d'abord cherché à lui faire compiler le code passant avec gcc. Avant d'y arriver nous avons rencontré deux types de problèmes :

- g++ est plus exigeant sur les types et n'accepte pas l'ancienne syntaxe C pour déclarer les paramètres des fonctions. Nous avons donc mis à jour les headers et les corps des fonctions des bibliothèques et corrigé les incohérences de type dans le code.
- L'usage de C++ dans le fichier yacc a posé un problème long à résoudre : le mécanisme de dialogue entre Lex et Yacc ne marchait plus. Les deux logiciels étant des projets différents, et le support du C++ évoluant suivant les versions, l'information ne fut pas facile à trouver. Lors de cette étape git fut très utile pour créer une branche par solution essayée.

Une fois ces deux points résolus, nous avons put utiliser C++ librement.

9.3 Intégration à la bibliothèque

9.3.1 Adaptations

Le parseur étant initialement prévu pour de la simulation, son comportement n'était pas adapté pour notre besoin de représenter le code d'un programme :

- Les règles initiales transformaient les données au plus tôt pour les mettre dans des structures de données dédiées à la simulation, souvent sous forme binaire. Nous avons cherché à faire remonter les données jusqu'aux règles appelants les constructeurs. Pour préserver les données comme du texte autant que possible, nous avons placé les chaines de caractères (par exemple les labels) dans un tableau associatif dynamique puis fait remonter un identifiant numérique, la valeur étant ensuite récupérée par le constructeur via cette clé.
- Les expressions étant évaluées, il aurait été long de chercher à les conserver sous forme de chaine de caractère, nous les avons donc conservé sous leur forme numérique, perdant en lisibilité sur le code final.
- L'analyseur lexical gérait plusieurs architectures différentes. Nous avons décidé de tout accepter dans notre parseur afin de simplifier le test des règles et de reconnaître le plus de codes possibles.

Comme les règles utilisent tous les constructeurs, nous en avons profité pour nous questionner sur les interfaces de nos fonctions. Nous avons ainsi put trouver des améliorations pour que notre bibliothèque gagne en sûreté et en simplicité.

9.3.2 Tests

Afin de vérifier le bon fonctionnement du programme, nous avons créé un code complete.s contenant pour chaque règle le code permettant de tester tous les cas qu'elle couvre (par exemple, toutes les instructions d'un format). Ceci nous a permis de tester la bonne restitution et le respect de l'ordre des arguments pour chaque cas.

9.3.3 Fonctionnement final

Au final, le parseur est appelé par un constructeur de la classe Program prenant le nom du fichier source en paramètre. Dans le parseur, les règles associées aux instructions, aux labels et aux directives instancient les Classes appropriées et les ajouter à un Program local au parseur.

Les macros sont expansées dans le programme par leur suite d'instructions. Ceci nous simplifie le reste de la bibliothèque qui n'a pas à gérer les pseudos instructions, par exemple pour le calcul de dépendances.

Une fois le code parsé, on copie les instructions du Program local vers le Program appelant.

Part III
Recette

10 Planning

Nos tâches sont assez dépendantes entre elles, nous nous sommes donc répartis le travail pour le démarrage du projet. Ensuite, les tâches ont été attribué en fonction de l'avancement de chacun. Voici le schéma représentant la répartion des tâches.

11 Documents fournis

A la fin de notre projet, nous avons produit les documents suivants.

- Le diagramme de classes UML du projet.
- Le code de la bibliothèque de base.
- La documentation de la bibliothèque générée par Doxygen.

12 Procédure de recette

La librairie produite a été validée par deux procédures de recette. Tout d'abord une recette continue composée de nos revues de code et de la validation des étapes par notre encadrante.

La seconde partie de la recette est basée sur des scénarios testant chacun un ensemble de fonctionnalités.

12.1 Acquisition d'un programme avec le parseur et restitution

Nous avons écrit trois petits programmes en langage C, typiquement de parcours de tableau, de multiplication de matrices ou de tri rapide. Ces programmes ont été compilé vers de l'assembleur avec gcc puis représenté en mémoire en utilisant le parseur avant d'être restitués en assembleur. Nous avons vérifié que les programmes restitués sont identiques aux programmes d'origine, en particulier qu'ils peuvent toujours être compilés et produire le même code exécutable.

13 Manipulation d'un programme

A partir de ces trois même programmes d'exemple, nous avons executé les opérations suivantes dans test1.cpp:

- Affichage du programme et du nombre de lignes que contient celui-ci.
- Calcul du découpage en blocs de base et en fonctions.
- Écriture des séquences de lignes de ces blocs de base et fonctions dans les fichiers test1.txt et test2.txt.

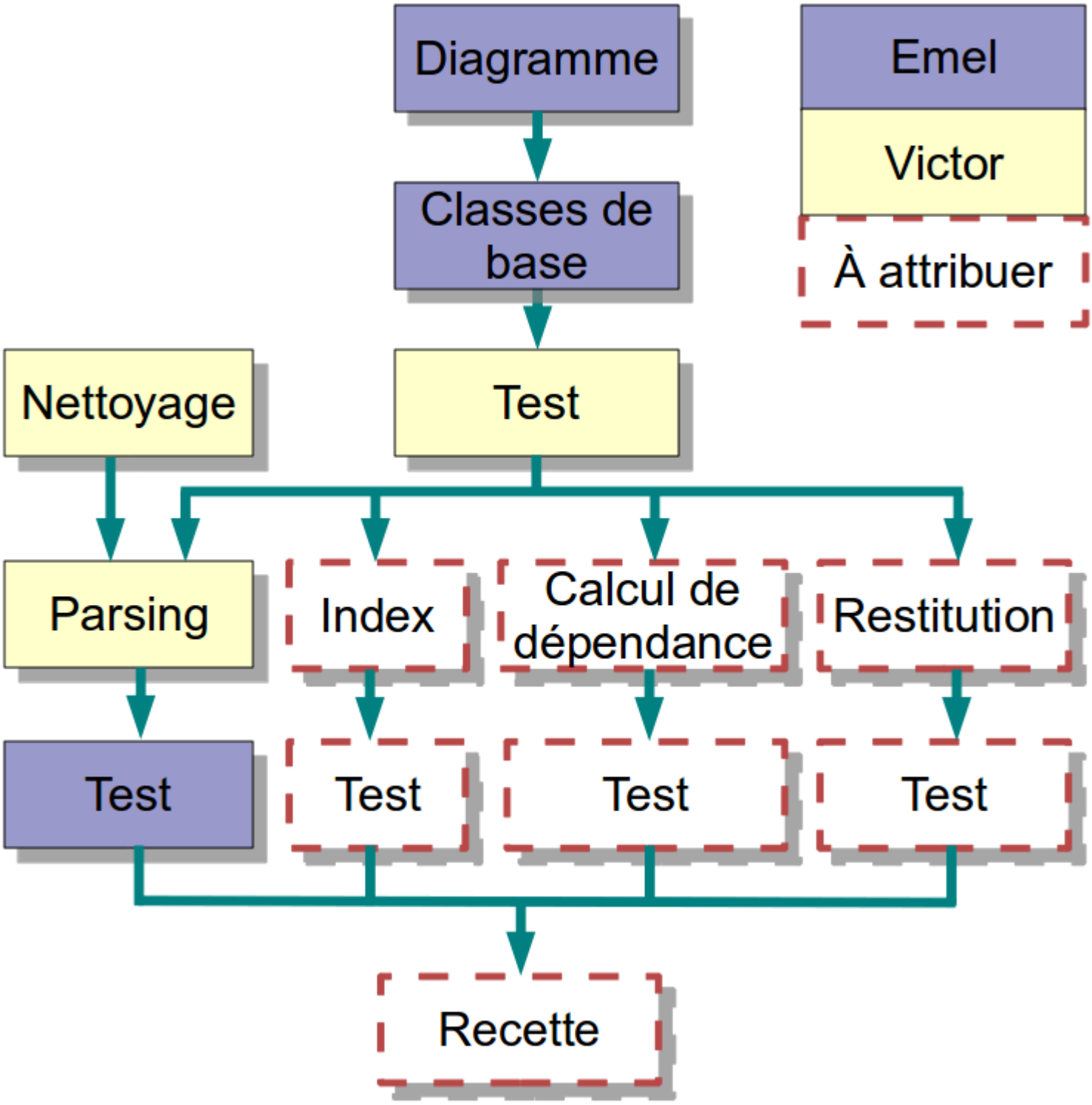

Figure 5: Planning final

14 Manipulation d'instructions

Nous démontrons les possibilités de manipulation, insertion et suppression de lignes en transformant la séquence d'instructions suivante qui est implémenté dans le fichier test2.cpp :

```
# Code avec des Nor # ligne 1
add $5, $0, $0 # ligne 2
or $6, $5, $0 # ligne 3
nor $4, $5, $6 # ligne 4
```

Pour obtenir le programme suivant :

```
# Equivalence en plusieurs instructions # ligne a
# ligne b
lui $4, 0xFFFF # ligne c
ori $4, $4, 0XFFFF # ligne d
xor $5, $4, $0 # ligne e
xor $6, $4, $0 # ligne f
and $4, $5, $6 # ligne g
```

En particulier :

- le contenu du commentaire de la ligne 1 est supprimé automatiquement par le parseur,
- la ligne 2 est supprimée,
- les opérandes et l'opérateur de la ligne 3 sont modifés pour obtenir la ligne c,
- l'opérateur de la ligne 4 est modifié pour obtenir la ligne g,
- les autres lignes sont ensuite insérées dans le programme.

14.1 Calcul de dépendance entre instructions

15 Conformité à la spécification

Malheureusement, lors de la réalisation de la bibliothèque certains problèmes ont ralenti l'implémentation et ont provoqué des changements par rapport aux sépcifications fournis.

- Fonctionnalité d'indexation: l'indexation sur les labels n'a pas été fait. Mais, cela dis, nous pouvons récupérer les labels grace au blocs de bases en testant si le bloc commence par un label ou pas.
- ####################Victor_remplir##

16 Extension possible##################Victor_remplir###

Part IV
Conclusion

Travailler avec Karine et Victor a été super! En plus nous avons fait le plus important: le projet fonctionne (vous n'y verrez que du feu) et l'écriture du rapport n'a pas du tout posé de problème! Vive les biblothèque qui lise le MIPS!!!