Projet de Fin d'Année 30 mai 2008

Rapport Final

Développement logiciel dans le cadre de la génération de processeurs très hautes-performances

Client : Bertrand LE GAL

Encadrant pédagogique : Philippe DUCHON

Adnane BENITTO
Rémi BUISSON
Mohamed Amine EL AFRIT
Stéphanie GATTI
Benoît PORTRAT
Thomas RATOUIT
Ludovic STORDEUR

Table des matières

Table des figures

1 Introduction

1.1 Contexte

1.1.1 P.F.A. et choix du sujet

Le P.F.A. est un projet qui rentre dans le cadre de notre deuxième année d'informatique à l'ENSEIRB. Ce projet avait pour but de nous confronter à une demande concrète de réalisation au niveau du développement informatique. L'outil produit et finalisé trouverait toute son utilité dans l'équipe qui a souhaité le voir conçu et produit. En effet, les différents sujets des P.F.A. ont été proposés par des clients, qui souhaitaient voir développés certains outils ou fonctionnalités, dans le but de les utiliser dans le futur.

Le projet que nous avons choisi s'intitule "Développement logiciel dans le cadre de la génération de processeurs très haute-performance". Il a été proposé par monsieur Bertrand LE GAL, qui voulait qu'une équipe apporte de nouvelles fonctionnalités à son outil, nommé *GraphLab*. Lors de la conception et du développement, nous avons été encadrés par monsieur Philippe DUCHON, qui nous a apporté de précieux conseils quant à la rédaction d'un cahier des charges et à la réflexion sur les différents points envisageables à traiter.

1.1.2 Outil GraphLab et fonctionnalités à apporter

Dans le laboratoire IMS, l'équipe "Conception de systèmes numériques" dans laquelle travaille le client, monsieur Bertrand Le Gal, développe actuellement un outil s'insérant dans un processus de création de processeurs très haute performance. Un cas d'utilisation de l'outil est le suivant : en entrée, l'utilisateur vient écrire du code source MatLab (dans un Fichier MatLab), qui sera traduit via un parser Java en un graphe. Le graphe produit décrira un enchaînement d'opérations sur des variables, et sera biparti, c'est-à-dire qu'il y aura alternance d'opérations et de variables, une *variable* pouvant être une entrée d'une fonction ou une variable temporaire, utilisée pour stocker un résultat. Ce graphe sera ensuite utilisé dans la conception du processeur que l'on souhaite produire. Le passage du graphe au processeur est un processus automatisé. Il s'effectue par le biais de la génération d'un fichier V.H.D.L., ou Very high speed integrated circuit Hardware Description Language décrivant la structure matérielle, qui servira ensuite à créer des portes logiques câblées contenues dans un F.P.G.A., ou Field-Programmable Gate Array afin de relier les différents éléments de la mémoire du processeur.

L'outil GraphLab offre des étapes de conception identiques aux phases de compilation logicielle, la différence étant que la finalité est d'obtenir un processeur avec des performances optimisées. En effet, l'équipe de développement de cet outil est plutôt orientée vers la fabrication de processeurs selon les besoins et les demandes, c'est-à-dire avec des fonctionnalités pouvant varier à chaque conception. Par conséquent, le processeur obtenu après les différentes phases suivantes :

- écriture du code source MatLab,
- analyse de ce code via un parser Java,
- traduction en graphe,

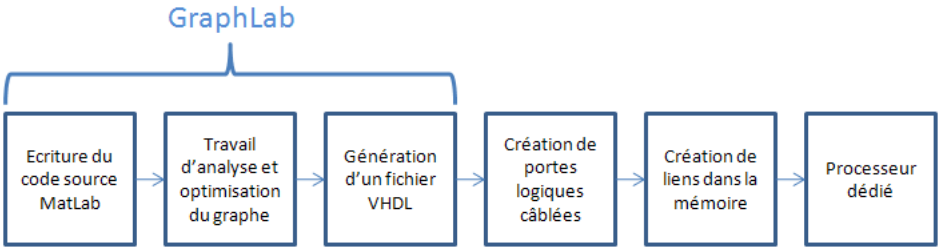


Fig. 1.1 – Les différentes étapes de développement dans lesquelles s'insère l'outil GraphLab

- génération d'un fichier VHDL,
- création de portes logiques câblées, et
- création de liens entre les différents éléments de la mémoire pour parvenir à l'obtention d'un processeur,

doit être un processeur très haute-performance, c'est-à-dire dont les caractéristiques sont optimisées et adaptées à une fonction / fonctionnalité spécifique.

La figure 1.1 représente l'enchaînement de ces étapes.

Au départ, l'utilisateur dispose d'un code MatLab décrivant des expressions arithmétiques. Un tel code présent en entrée représente les instructions que devra réaliser le processeur dédié final. Par exemple, si l'expression arithmétique **a+b+c** est passée en entrée, cela signifiera que le processeur dédié final possèdera, après les phases de traitement, une ressource de calcul de type additionneur.

De plus, il faut dissocier l'obtention d'un graphe de sa visualisation, qui sont deux étapes distinctes.

Après analyse du code MatLab, il y a obtention d'un modèle formel de calcul qui n'est pas directement visualisable. Pour le rendre visible à l'écran, il faut faire appel à un outil de visualisation de graphe, comme par exemple, la fonction *plot* sous MatLab.

Pour réaliser la traduction du code MatLab en architecture matérielle, deux fonctionnalités différentes :
- un parser de code source MatLab sommaire,
- une possibilité de mise à jour automatique pour l'utilisateur,

ont été développées par des élèves de deuxième année informatique dans le cadre d'un P.F.A. l'année dernière.

Notre travail a consisté à faire évoluer l'outil en ajoutant de nouvelles propriétés et en optimisant certains points, tout en respectant la nature et la cohérence des éléments déjà développés.

Il s'est situé à deux niveaux :
- au niveau du parser MatLab,
- au niveau du graphe obtenu après analyse.

Le développement des nouvelles fonctionnalités a été réalisé, sur ces deux niveaux, en langage Java, conformément aux souhaits du client et au respect de l'existant, à savoir un outil développé en Java.

Plus précisément, il s'agissait de reprendre le parser MatLab en analysant toute la sémantique, et également en ajoutant la possibilité d'optimiser le graphe obtenu, par détection de simplification via les propriétés des opérateurs usuels. Ces propriétés peuvent être, par exemple, l'élément neutre pour un opérateur, ce qui conduira à différentes optimisations. Par exemple, en considérant une variable a et l'opération d'addition, l'élément neutre étant le 0, $a+0$ sera optimisé en a, et il en est de même pour l'opération de multiplication où l'élément neutre est 1 : l'expression $a*1$ sera optimisée en a. En considérant par exemple l'expression arithmétique **y=a*0+b*1+0+c**, nous pouvons visualiser le graphe correspondant, ainsi que sa forme optimisée, sur la figure 1.2.

Ces deux graphes respectent les conventions des graphes produits à l'aide de GraphLab. En effet, il y a une alternance de variables et d'opérateurs, ce qui rend le graphe biparti. De plus, le graphe possède deux noeuds spéciaux : un noeud source auquel sont reliées les variables d'entrée et les constantes, et un noeud puits auquel sont reliées les sorties.

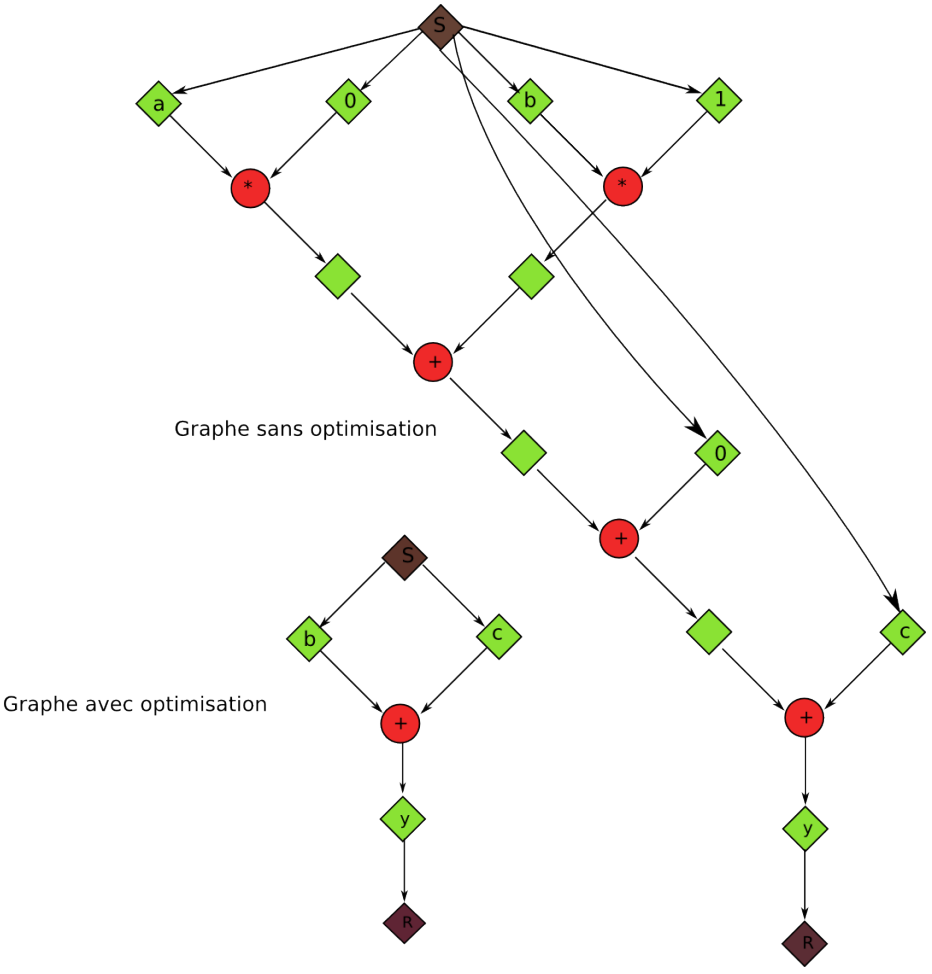


Fig. 1.2 – Graphe correspondant à l'équation **y=a*0+b*1+0+c** ainsi que le graphe optimisé, correspondant à l'expression arithmétique optimisée **y=b+c**

Lors de l'optimisation du graphe passé en entrée, il y a recherche de motifs préalablement définis, et remplacement par les motifs optimisés correspondants.

Lors du parcours du graphe, la multiplication de *a* par 0 va être détectée et remplacée par le motif associé, à savoir 0. La multiplication de *b* par 1 va également être détectée et remplacée par *b*. Ainsi, après optimisation avec la précédente donnée (0) et après remplacement du motif représentant l'addition par 0, le graphe obtenu contient un motif qui ne peut plus être optimisé, à savoir l'addition des variables *b* et *c*.

Nous voyons sur ces figures que le fait d'effectuer de telles optimisations sur le graphe diminue de façon conséquente le nombre de ressources utilisées. En effet, dans le graphe non optimisé, sont utilisées deux ressources de type multiplicateur et trois ressources de type additionneur, alors que dans le graphe optimisé, une seule ressource de type additionneur est utilisée.

Nous avons également dû considérer le problème de l'ajout ou l'appel de nouveaux opérateurs au langage MatLab ainsi que de propriétés le caractérisant. Pour cela, nous avons été amenés à définir un nouveau langage utilisé dans le fichier de définition des nouveaux opérateurs.

Notre intervention a également consisté en l'optimisation d'un graphe, en passant de sa forme parallélisée à sa forme linéarisée et inversement, grâce à l'écriture d'un fichier de définition de linéarisations et d'un fichier de définition de différentes parallélisations.

Par exemple, si l'on considère l'expression arithmétique **y=(((a+b)+c)+d)**, le graphe correspondant a une forme séquentielle ou *linéarisée*, c'est-à-dire une forme de peigne. Ce graphe peut être parallélisé pour

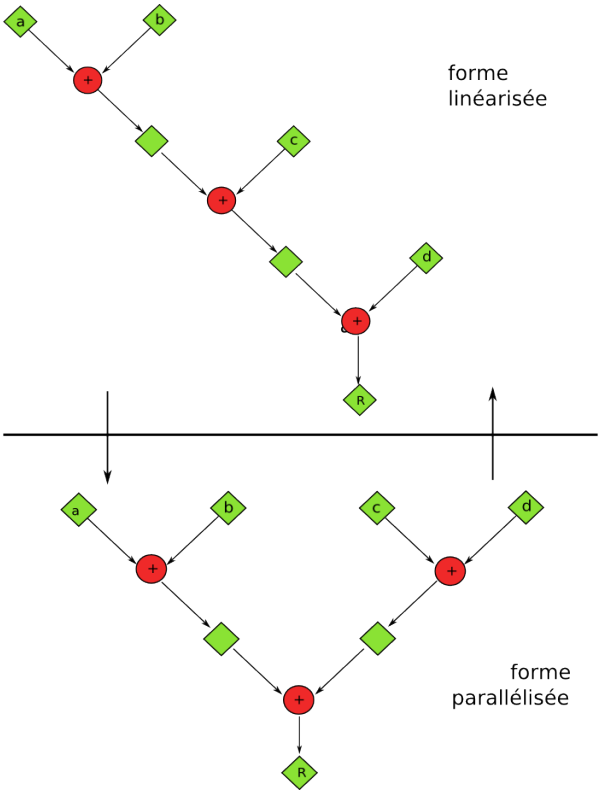


Fig. 1.3 – Graphe linéarisé correspondant à l'équation **y=(((a+b)+c)+d)** ainsi que le graphe parallélisé

donner le résultat visible sur la figure 1.3.

Pour passer d'une forme de graphe à une autre, il a fallu faire de la reconnaissance, puis du remplacement de motifs. Dans le but d'être généraliste, nous avons implanté un unique algorithme de parcours du graphe, valide que ce soit pour la parallélisation ou la linéarisation.

1.2 Organisation du projet

Le projet de travail sur le thème "Développement logiciel dans le cadre de la génération de processeurs très haute-performance" s'est articulé autour de la maîtrise d'ouvrage et de la maîtrise d'oeuvre.

La maîtrise d'ouvrage correspond au client, à savoir monsieur Bertrand LE GAL. Le client avait émis des besoins concernant l'outil GraphLab et souhaité voir s'articuler un projet afin que soient développés différents modules pour son outil, comme l'analyse syntaxique du parser MatLab, et que des améliorations y soient également apportées, comme l'optimisation du graphe généré en sortie.

La maîtrise d'oeuvre est constituée par les différents membres de notre groupe, à savoir :

Adnane BENITTO
Rémi BUISSON
Mohamed Amine EL AFRIT
Stéphanie GATTI
Benoît PORTRAT
Thomas RATOUIT
Ludovic STORDEUR

Notre tâche a été de développer, parmi les différents besoins émis par le client, les fonctionnalités princi-

pales retenues selon le temps et les moyens dont nous disposions et qui ont été décidées en accord avec lui. Une série de tests a également été effectuée lors du développement des différents modules.

Notre travail fut effectué sous la supervision d'un encadrant technique : monsieur Philippe DUCHON.

Selon le diagramme de Gantt, il apparaissait trois grandes fonctionnalités à traiter : deux au niveau du parser MatLab et une au niveau du graphe généré en sortie. Notre groupe s'est donc tout naturellement scindé en deux binômes et un trinôme, et les fonctionnalités à développer se sont réparties de la façon suivante :

- le premier binôme, Adnane et Mohamed Amine, a travaillé au niveau du parser MatLab, pour régler le problème de la collision tableau/fonction et pour gérer les fonctions à plusieurs sorties,
- le second binôme, Ludovic et Thomas, est également intervenu au niveau du parser MatLab, notamment pour régler la gestion variables qui tendent à être modifiées au cours du temps, et a travaillé en collaboration avec le premier binôme,
- le trinôme, Benoît, Rémi et Stéphanie, avait pour travail l'ajout d'une fonctionnalité d'optimisation de noeuds dans un graphe, ainsi que la parallélisation et la linéarisation de motifs dans un graphe.

Chacun des petits groupes s'est tenu informé de l'avancement des autres groupes tout au long du projet, par l'envoi de courrier électronique et de réunion hebdomadaire.

2

Description du sujet

2.1 Description générale

Ce projet nous a amené à mettre en application de nombreuses compétences en conception logicielle, telles que celles liées à la compilation ainsi qu'à l'algorithmique de graphe.

L'U.M.L. nous fournit une méthode de conception d'architectures logicielles objets.

Ce langage de modélisation nous fournit des diagrammes qui informent de manière claire et précise sur la structuration d'une application. Ainsi, nous nous proposons de décrire, afin de se mettre dans le contexte du projet, les diagrammes de Paquetage, de cas d'utilisation et de séquence associés à ce projet et déjà présentés dans le cahier des charges.

En figure 2.1 est présenté un diagramme de paquetages non exhaustif. Il présente les dépendances entre les paquetages. Le symbole $\oplus$ signifie "contient". Ici, le paquetage **GraphLab** contient les classes **Optimiseur de graphes**, **Parser MatLab**, **Console GraphLab** et **Graphe**. L'utilisateur n'interagit qu'avec la console de GraphLab. Chacune de ces classes est représentée par un ensemble de trois champs. Le premier indique le nom de la classe. Dans le deuxième sont listés les attributs de la classe (ici, ce champs est laissé vide pour simplifier la lecture). Le troisième liste les méthodes de la classe. Le + devant ces méthodes est un opérateur de portée qui indique qu'elles sont publiques, c'est-à-dire accessibles depuis toutes les autres classes.

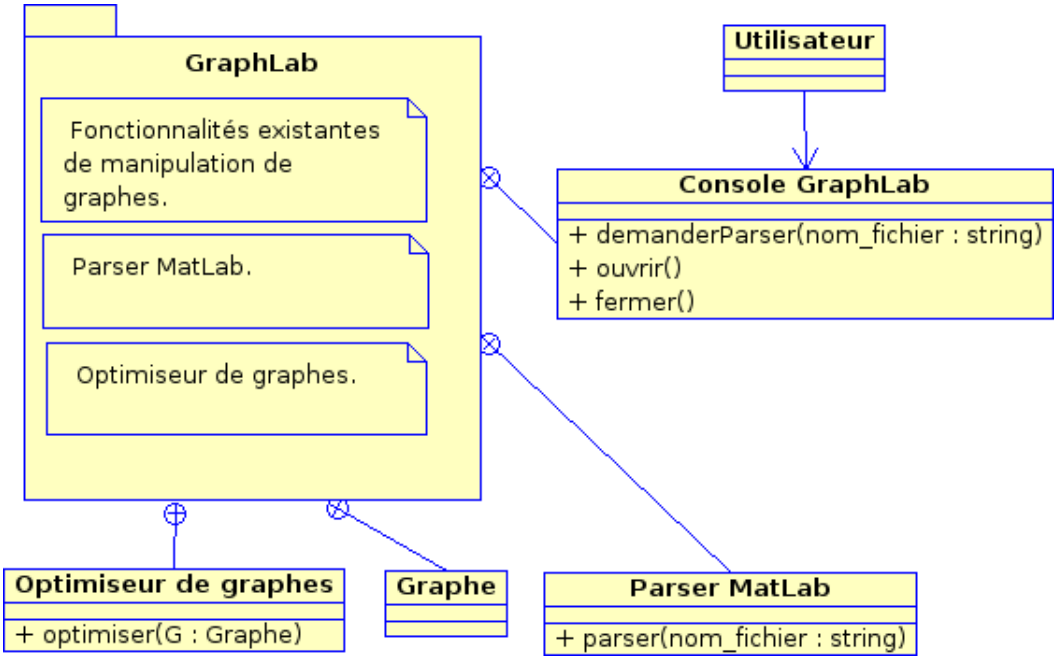


Fig. 2.1 – Diagramme de périmètre (ou de paquetage)

La classe correspondant à la console GraphLab possède les méthodes ouvrir(), fermer() et demandeParser(). La première permet d'ouvrir une console, c'est-à-dire une fenêtre dans laquelle s'affiche un invite de commande. La deuxième a pour fonction de fermer la console ouverte précédemment. La troisième prend en paramètre un nom de fichier et demande à parser le code MatLab contenu dans ce fichier. En sortie de cette dernière fonction, un graphe est obtenu. Cette analyse du code MatLab est effectuée par le biais de la méthode parser() contenu dans la classe **Parser MatLab**.

Les deux fonctionnalités contenues dans le paquetage **GraphLab** qui concernent l'analyse du code MatLab déjà présente dans l'outil avant notre valeur ajoutée, à savoir l'utilitaire de manipulation de graphes et le parser MatLab, sont décrites au travers des classes **Console GraphLab**, **Parser MatLab** et **Graphe**.

La troisième fonctionnalité, qui est l'optimisation de graphes, est réalisée via des appels à une instance de la classe **Optimiseur de graphes**, et plus précisément, via l'appel à la méthode optimiser() qui prend un graphe en entrée sur lequel elle applique des techniques d'optimisation.

En figure 2.2 est présenté un diagramme de cas d'utilisation. Il permet d'identifier les possibilités d'interaction entre le système et les acteurs (intervenants extérieurs au système), c'est-à-dire toutes les fonctionnalités que doit fournir le système. Ici, nous voyons que l'utilisateur peut interagir avec le système via trois fonctionnalités importantes. Il peut en effet demander l'analyse de son code source MatLab. A partir d'un code source analysé, il peut ensuite générer un graphe optimisé, à l'aide d'un ou plusieurs opérateurs définis préalablement, visualisable par un utilitaire intégré au logiciel GraphLab.

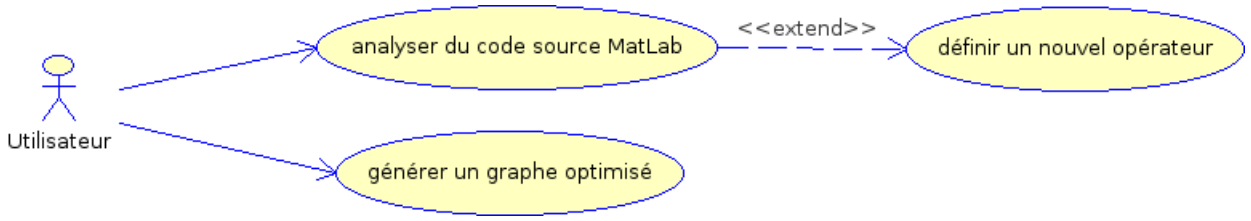


Fig. 2.2 – Diagramme de cas d'utilisation

En figure 2.3 est présenté un diagramme de séquence. Il représente de manière séquentielle le déroulement des traitements et des interactions entre les éléments du système et/ou de ses acteurs. Ici, une utilisation normale du logiciel pour l'utilisateur est, premièrement, d'ouvrir la console du logiciel GraphLab. Ensuite, il demande à ce que son fichier source MatLab soit analysé. L'analyseur ayant effectué son travail, traite le graphe obtenu afin de l'optimiser (développement/factorisation, simplifications arithmétiques ...). Enfin, l'utilisateur ferme la console ouverte.

Les flèches en pointillés constituent des retours de fonctions, c'est-à-dire que la méthode appelée informe la méthode appelante sur l'état du traitement à la sortie de cette fonction.

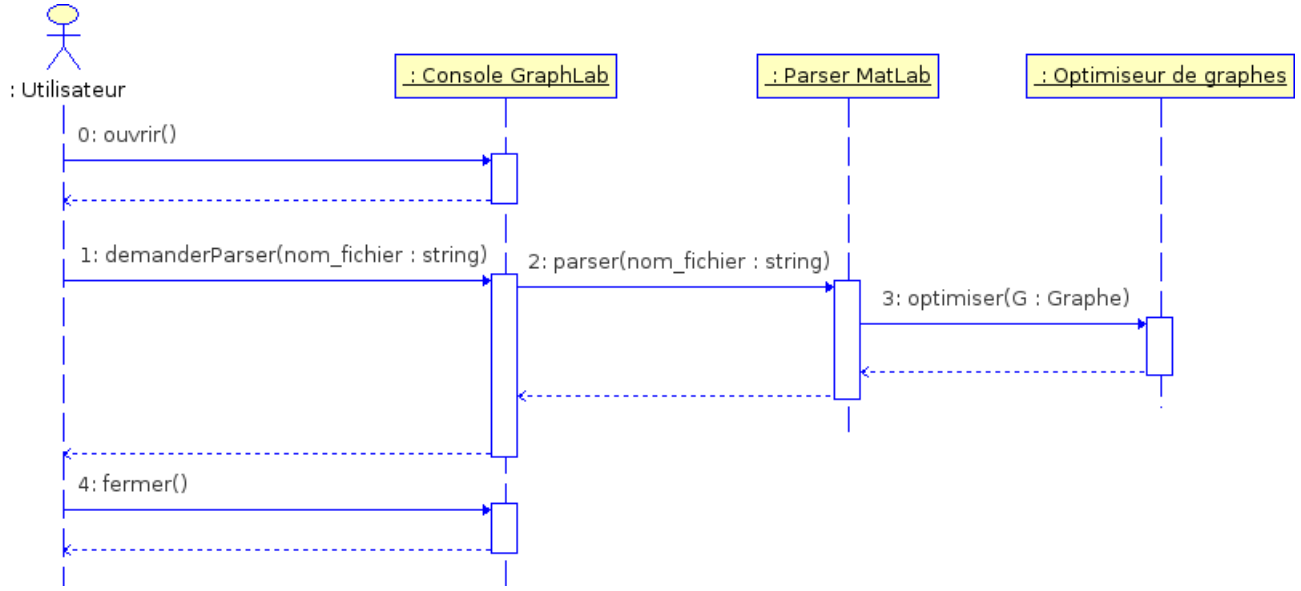


Fig. 2.3 – Diagramme de séquence

2.2 Exigences

Nous nous sommes engagés vis-à-vis de notre client à modifier ou à ajouter plusieurs fonctionnalités. Il nous a été demandé d'une part d'améliorer l'état basique du parser existant, et d'une autre part d'implanter les optimisations et transformations requises sur le (les) graphe(s) résultant(s).

2.2.1 Le parser MatLab

Les fonctionnalités que nous devions améliorer sur le parser comprennent aussi bien des implantations au niveau de :

L'analyse syntaxique

Nous nous sommes tenus à effectuer les points qui suivent :
- Améliorer le support des appels de fonctions et des accès aux tableaux. En particulier, il nous a fallu distinguer sans ambiguïté un appel de fonction d'un accès à une case d'un tableau, ces deux opérations ayant une syntaxe identique. Cependant, une variable de type 'tableau' devrait obligatoirement être déclarée dans le corps de la fonction.
- Améliorer la gestion des boucles (boucles disjointes et dont le pas est différent de 1).
- Les fonctions retournant plusieurs variables. En effet, l'une des particularités du langage MatLab est qu'il peut gérer ce genre de fonctions. Notre client a demandé à ce que l'utilisateur puisse employer ces fonctions dans son code.

L'analyse sémantique

Au niveau de cette analyse, nous devions réaliser ce qui suit :
- Gérer les variables globales même si celles-ci n'ont pas été explicitement déclarées dans le fichier MatLab à l'aide du mot clé **export**.

- Détecter une dépendance de données de type "lecture avant écriture" sur une variable globale, conduisant à l'apparition d'un noeud spécifique noté **Z-1** dans le graphe.
- Détecter des appels de fonctions définies par l'utilisateur dans des fichiers MatLab annexes mais situées dans le même répertoire.

2.2.2 La transformation des graphes

La transformation de graphes consiste en l'optimisation à divers niveaux du graphe obtenu. Cette partie du projet ne se basait pas sur un existant fonctionnel.

Fonctionnalités à ajouter

L'analyse des besoins de client à ce niveau conclut à :

- Détecter des motifs locaux traduisant l'évaluation d'expressions arithmétiques sous forme factorisée ou développée et pouvoir modifier localement le graphe afin de passer d'une forme à l'autre et réciproquement. La détection de ces motifs est limitée aux expressions ne faisant intervenir que les quatre opérateurs arithmétiques usuels que sont l'addition, la soustraction, la multiplication et la division.
  Remarque : La réalisation de ces optimisations pouvait être prise en compte au niveau du parser MatLab, nous sommes cependant obligés d'implanter celles-ci dans le module d'optimisation des graphes pour des raisons de cas d'utilisation du logiciel offrant la possibilité de visualiser des graphes importés d'une source externe.
- Détecter et simplifier des motifs locaux traduisant l'évaluation d'expressions arithmétiques dont tout ou une partie du résultat peut être connu à la compilation.
  Remarque : Pour les mêmes raisons que celles évoquées précédemment, nous sommes obligés d'implanter ces optimisations à ce niveau et non dans le parser MatLab.

2.3 Organisation du groupe

L'organisation du groupe de projet peut être considérée à plusieurs niveaux. En effet, les réunions avec tantôt notre client et tantôt notre encadrant pédagogique se tenaient d'une manière tout à fait régulière et périodique, ce qui nous permettait d'éviter tout malentendu ou divergence de compréhension. Les compte-rendus des réunions étaient soigneusement réalisés d'une façon équitable entre les membres du groupe afin de garder des traces utiles des conclusion partielles des réunions tenues.

Par ailleurs, le groupe du projet a dû se scinder en un trinôme et deux binômes pour des raisons d'indépendance générale de caractère entre des parties à développer de la M.O.E.. Aussi a-t-il été convenu d'associer la partie du développement de parser MatLab aux deux binômes et de même, d'affecter la partie sur les transformations des graphes au trinôme.

Afin de gagner du temps, il a été décidé après le développement du module d'optimisations sur les graphes, de scinder le trinôme en un binôme qui se chargeait du développement du module PArallélisation/Linéarisation et un monôme qui se chargeait quant à lui du débogage.

D'un point de vue général, les différentes entités "dissociées" du groupe étaient amenées à travailler sur des parties complémentaires, chose qui exigeait la bonne circulation de l'information au sein du groupe garantie par les réunions hebdomadaires.

3 Parser MatLab

À l'image des prestations attendues au niveau du parser MatLab, et en dépit des difficultés rencontrées lors de son développement, les améliorations devant être apportées ont été implantées en leur intégralité. Pour ce, l'analyse se fait en trois passes. C'est un existant que nous avons convenu de garder. La première passe induit la vérification et le dépliage des boucles. La seconde permet la simplification des expressions arithmétiques, et la dernière génère enfin le graphe.

Nous détaillerons ci-dessous le diagramme des classes dans un premier lieu, ensuite nous expliquerons les trois passes du parser.

3.1 Diagramme des Classes

Voir les diagrammes 3.1 et 3.2

3.2 Première phase : dépliage des boucles

3.2.1 Description de la fonctionnalité

La première phase qu'effectue le parser MatLab consiste en une opération qu'on appelle *dépliage des boucles* qu'il convient dans un premier temps d'expliquer :

Le langage MatLab dispose d'un certain nombre de structures de contrôle parmi lesquelles la boucle **for**. Celle-ci utilise la syntaxe suivante :

```
for iterator = start:step:end
    instructions
    ....
end
```

- **iterator** est le nom de la variable itératrice. Elle est nécessairement de type entier. Sa valeur changera à chaque itération de la boucle.
- **start** est la valeur initiale de l'itérateur.
- **step** est le pas d'incrémentation ou de décrémentation de l'itérateur à chaque itération de la boucle. Sa valeur peut en effet être positive ou négative.
- **end** désigne la borne de fin d'itération. Lorsque la valeur de l'itérateur dépasse cette borne, l'exécution continue à la suite de la boucle **for**.

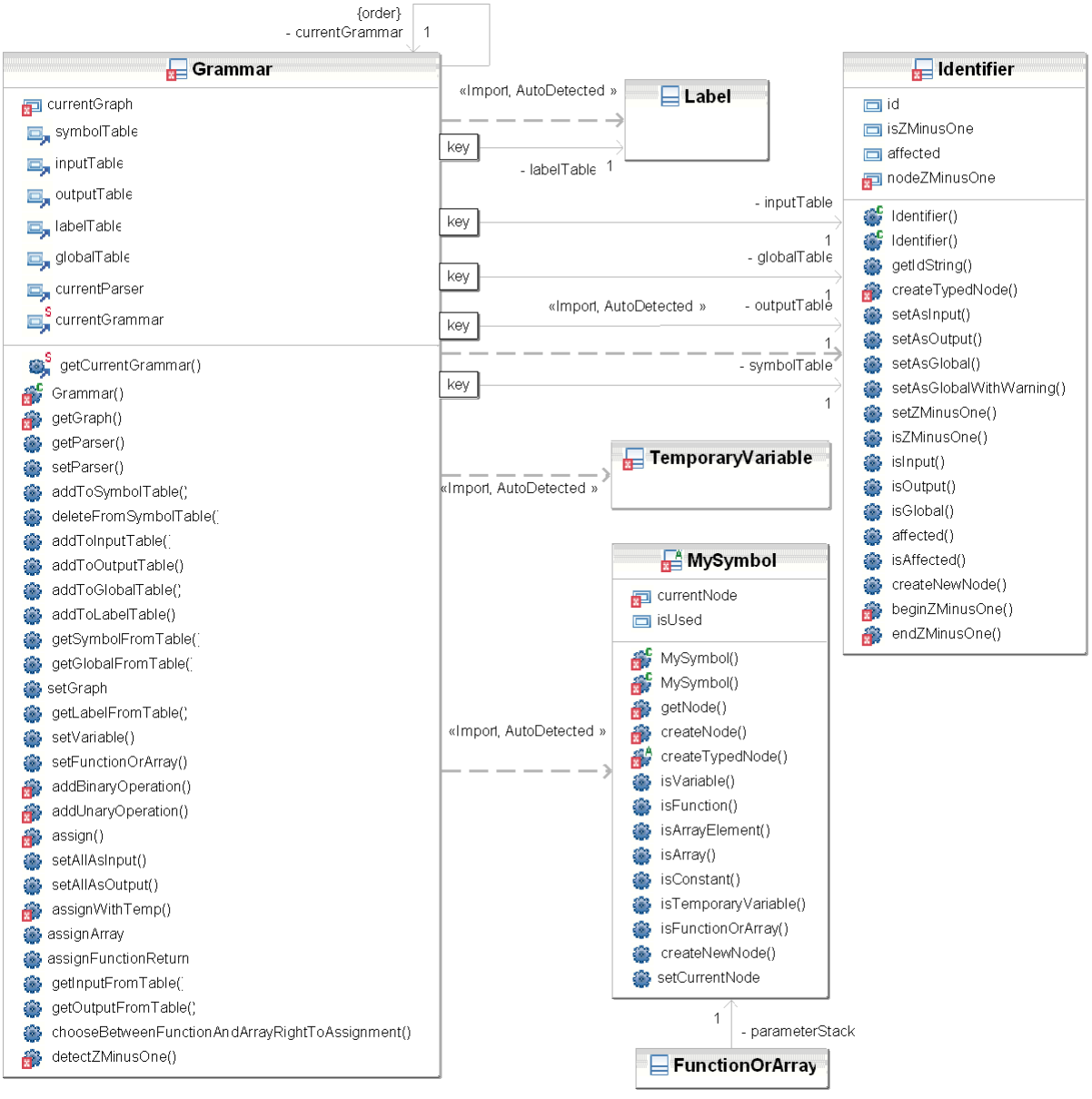


Fig. 3.1 – Diagramme des Classes du Parser MatLab

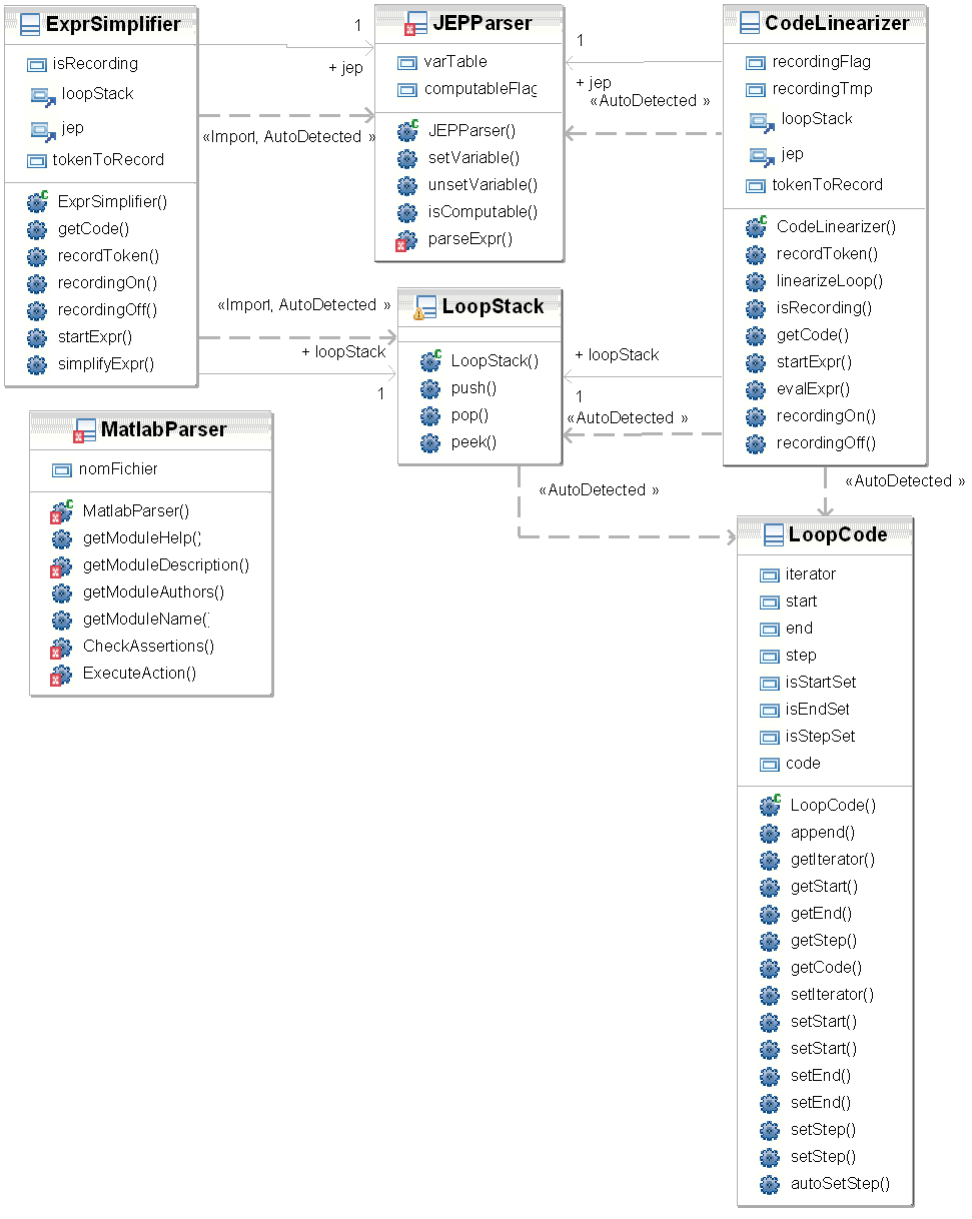


Fig. 3.2 – Diagramme des Classes du Parser MatLab (suite)

Déplier une boucle consiste à transformer le code lié à celle-ci afin de faire disparaître la structure de boucle à proprement parler sans changer le résultat de l'exécution lié à ce code. Concrètement, on effectue ce traitement en écrivant le code à l'intérieur de la boucle autant de fois qu'il y a d'itérations de boucle, en prenant soin de substituer la valeur de l'itérateur à chaque fois.

Voici un exemple, avant dépliage d'une boucle :

```
N = 5;

for i = -1:2:(2*N)
    for j = 2:1
a = a + i * j;
    end
end
```

Voici le code résultant après dépliage de la boucle :

```
N = 5;

a = a - 1 * 2;
a = a - 1 * 1;

a = a + 1 * 2;
a = a + 1 * 1;

a = a + 3 * 2;
a = a + 3 * 1;

a = a + 5 * 2;
a = a + 5 * 1;

a = a + 7 * 2;
a = a + 7 * 1;

a = a + 9 * 2;
a = a + 9 * 1;
```

Cet exemple met en évidence plusieurs aspects que notre parser MatLab doit gérer :

- La gestion de boucles imbriquées,
- La non nécessité de spécifier le pas d'itération. En effet, dans la boucle imbriquée, il est fixé implicitement à -1,
- Le signe de la valeur de départ est différent de celui de la valeur d'arrivée,
- Le pas d'itération est différent de 1,
- L'itérateur ne prend pas exactement la valeur de fin d'itération : il s'arrête à 9,
- La valeur de fin d'itération n'est pas connue à la compilation mais dépend d'une variable qui est affectée préalablement.

3.2.2 Reprise de l'existant

Lorsque nous avons repris le projet en main, celui-ci disposait déjà d'un système de dépliage des boucles. Malheureusement, il manquait de robustesse dans certains cas. Or le code tel que nous l'avons récupéré n'était absolument pas maintenable. Il n'y avait pas ou très peu de commentaires et les choix architecturaux qui avaient été fait limitaient considérablement l'évolutivité du code.

Pour cette raison, nous avons décidé de ne conserver que les éléments suivants de l'existant :

- La grammaire (décrite dans le fichier **FirstPass.cup**) qui permet de générer l'analyseur syntaxique,
- Le principe d'une étape spécifique au dépliage des boucles,
- L'algorithme général de dépliage.

3.2.3 Implantation

Le code réalisant ce traitement est localisé dans la classe **FirstPass**, qui implante l'analyseur syntaxique spécifique à cette phase, ainsi que dans la classe **CodeLinearizer** qui contient les méthodes de haut niveau relatives à l'algorithme de dépliage.

L'analyseur syntaxique évoqué ci-dessus est construit grâce à un générateur d'analyseur syntaxique. Celui utilisé dans ce projet s'appelle **CUP**. De plus, cet analyseur syntaxique s'appuie sur un analyseur lexical qui est également créé au travers d'un générateur. Celui que nous avons utilisé s'appelle **JFlex**.

Production du code

Pour cette étape de traitement ainsi que pour la seconde qui sera décrite plus loin, le parser MatLab prend en entrée le code MatLab sous forme d'une chaîne de caractères et donne en sortie le code modifié également sous forme d'une chaîne de caractères.

De ce fait, nous avons deux éléments à mettre en place : le premier est une structure de données permettant de stocker du code MatLab tandis que le deuxième est un mécanisme permettant d'appeler une fonction d'enregistrement à chaque fois que l'on rencontre une unité lexicale du langage MatLab, unité que nous appellerons par la suite un **terminal**.

La structure de donnée utilisée est implantée au travers de la classe **LoopCode** qui contient également des données nécessaires dans le cas du dépliage des boucles.

Le mécanisme d'enregistrement est fourni par **CUP** qui est le générateur d'analyseur syntaxique que nous utilisons pour notre parser. En effet, celui-ci définit dans les analyseurs syntaxiques qu'il génère une méthode **scan()** qui est appelée à chaque fois que le parser rencontre un terminal.

Algorithme de dépliage

A chaque étape de l'algorithme figure les règles de grammaire, décrites dans le fichier **FirstPass.cup**, qui indiquent à quels endroits s'effectuent les différents traitements.

Le principe général est le suivant :

On utilise un objet principal de type **LoopCode**, évoqué précédemment pour stocker le code final, c'est-à-dire après dépliement des boucles. A chaque fois que l'on rencontre une nouvelle boucle,

```
for : := FOR
```

on crée un nouvel objet **LoopCode** et on stocke le code à l'intérieur de la boucle dans cet objet. On y sauvegarde également les informations relatives à la boucle elle-même,

```
for_entete ::= for_var_control colon_expr
colon_expr ::= expr DPOINTS expr | colon_expr DPOINTS expr
```

à savoir :

- le nom de la variable itératrice,
- la valeur de départ de l'itérateur,
- le pas d'itération,
- la valeur de fin d'itération.

Lorsque l'on arrive à la fin d'une boucle,

`for_cmd_list : := delimited_input END`

on considère l'objet dans lequel est stocké le code relatif à la boucle et on l'introduit autant de fois qu'il y a eu d'itération de boucles dans l'objet **LoopCode** parent. On substitue au passage les occurrences de la variable d'itération par la valeur qu'elle prend à chaque itération.

L'algorithme met en évidence une propriété intéressante qui est la suivante : on déplie toujours en premier la dernière boucle que l'on rencontre. Ceci nous a incité à utiliser une structure de pile pour gérer les différents éléments de code MatLab que l'on stocke à un moment donné. Cette structure est implantée dans la classe **LoopStack** qui elle-même hérite de la classe **java.util.Stack** ce qui nous a permis d'écrire une classe très légère.

Boucles ayant un nombre d'itération inconnu à la compilation

En parallèle du dépliement des boucles, notre parser doit pouvoir évaluer les expressions arithmétiques qui apparaissent au niveau des instructions d'affectation à des variables :

`assignement : := reference EQ expr`

Nous disons qu'une expression arithmétique a pu être évaluée lorsqu'il est possible d'en fournir un résultat numérique.

Avec ce mécanisme, il nous est possible de connaître la valeur d'un attribut de boucle lorsque cette dernière n'est pas connue à la compilation, comme c'était le cas dans l'exemple exhibé dans l'introduction.

L'idée est d'enregistrer le code relatif à l'expression arithmétique dans un élément de notre pile **LoopStack**. Puis, lorsqu'on a fini de lire cette expression, on peut tenter d'évaluer son résultat. Le procédé d'évaluation en lui-même est laissé à une bibliothèque Java qui s'appelle JEP, ou Java Expression Parser. Celle-ci sera également réutilisée pour la deuxième étape du parser.

Nous exploitons essentiellement une fonctionnalité de **Jep** pour réaliser l'évaluation des expressions. Il s'agit de la possibilité de lui fournir, avant l'évaluation, une table qui décrit une liste des variables pour lesquelles leur valeur est connue à cet instant donné. Ainsi, lorsque **Jep** rencontre des variables dans l'expression, il les remplace par leur valeur si celle-ci est connue.

Cette table de variables que nous maintenons tout le long de l'exécution du parser est implantée dans la classe **JEPParser**. Implantée au travers d'une table de hachage, elle contient en permanence exclusivement les variables pour lesquelles la valeur est connue. Ainsi, lorsque **Jep** réussit à évaluer une expression arithmétique, il inscrit le résultat associé à la variable recevant le résultat de l'expression dans la table. Dans le cas contraire, il supprime de la table l'entrée correspondant à la variable si celle-ci était présente.

3.2.4 Limitations

Bien qu'ayant réalisé un mécanisme de dépliage des boucles fonctionnel, il subsiste une limitation concernant l'évaluation des expressions arithmétiques. En effet, il est impossible d'évaluer une expression si celle-ci contient des appels à des fonctions ou à des valeurs de tableaux. Ces deux éléments du langage MatLab possédant strictement la même syntaxe, ils sont vus du côté de **Jep** de la même manière, à savoir comme un appel de fonction, ce qui explique qu'ils posent tous les deux le même problème.

Le problème vient du fait que **Jep** impose à l'utilisateur de déclarer explicitement les fonctions qui pourraient apparaître dans l'expression avant que celui-ci n'en fasse l'analyse. En plus de déclarer le nom de la fonction, cette restriction nous pose essentiellement deux problèmes :

- Il est impossible de savoir à l'avance quels appels de fonction apparaîtront dans une expression. Il faudrait procéder avant chaque évaluation d'expression à une analyse de cette dernière pour détecter ces appels et déclarer à **Jep** les fonctions correspondantes.
- Le fait que les tableaux et les fonctions MatLab soient vus comme des fonctions par **Jep** pose des problèmes dans la mesure où le traitement à opérer n'est absolument pas le même.

3.3 Deuxième phase : simplification des expressions arithmétiques

3.3.1 Introduction

A l'issue de la première phase du parser, nous avons récupéré un code dans lequel les structures de boucle de type **for** avaient été dépliées. Cette deuxième phase se propose de simplifier le plus possible les expressions arithmétiques qui apparaissent dans le code lors des instructions d'affectation. De cette manière, le graphe qui sera produit au final sera plus simple, ce qui a un impact direct sur la complexité du processeur qui sera créé.

Bien que la première phase du parser était nécessaire du fait que l'outil GraphLab n'a pas été conçu pour représenter les structures de boucles sur les graphes, cette deuxième phase est facultative. En effet, on obtient un graphe tout à fait correct sans celle-ci. De plus, l'un des objectifs de ce projet était de développer un module d'optimisation des graphes qui a pour but d'opérer des simplifications mais en agissant directement sur les graphes. Ce module justifierait encore moins l'utilité de cette seconde étape du parser MatLab. Cependant, il était possible d'effectuer des simplifications à ce stade pour un coût minime dans la mesure où nous réutilisions plusieurs modules de l'étape précédente. Qui plus est, certaines optimisations n'étaient réalisables qu'au niveau du parser MatLab.

3.3.2 Reprise de l'existant

Le parser, dans l'état où nous l'avons repris, possédait déjà un traitement dédié à la simplification des expressions arithmétiques. Cependant, celui-ci souffrait de nombreux défauts. En particulier, lorsque une expression faisait référence à une fonction ou à une valeur d'un tableau, l'expression arithmétique disparaissait purement et simplement dans le code final.

Tout comme pour la partie précédente, cette partie du code n'était pas véritablement maintenable. De plus, comme nous allons le voir plus loin, l'essentiel de notre implantation se base sur des classes que nous avons déjà écrites pour la première étape du parser, ce qui nous a permis de réduire significativement le développement.

3.3.3 Implantation

Le code réalisant cette phase est localisé dans la classe **SecondPass** qui implante l'analyseur syntaxique pour cette seconde étape ainsi que dans la classe **ExprSimplifier** qui contient les méthodes de haut niveau relatives à la simplification des expressions arithmétiques.

Ici encore, l'essentiel du traitement des expressions arithmétiques est laissé à la bibliothèque **Jep**. Si nous l'avions utilisé précédemment pour ses possibilités d'évaluation d'expressions, nous utilisons en plus ici ses capacités à effectuer des simplifications.

Voici une liste non exhaustive des simplifications que **Jep** peut opérer :

```
a + a + a + 3 =>  3 * a + 3
a * -1 =>  -a
a + 7 * 3 + 4 =>  a + 25
```

Ces simplifications sont opérées en amont du procédé d'évaluation. Nous avons gardé ce dernier mécanisme dans cette seconde phase car il peut nous permettre de simplifier encore plus les expressions, à nouveau en tenant compte des variables ayant une valeur connue au moment de l'évaluation des expressions. De ce fait, nous réutilisons la table des variables implantée dans la classe **JEPParser**.

Il convient de noter que ces deux étapes de traitement fonctionnent de manière complètement dissociée l'une de l'autre. Ainsi, le fait qu'il soit impossible d'évaluer une expression ne veut pas nécessairement dire qu'elle ne soit pas simplifiable. Dans ce cas, le parser retournera le meilleur résultat qu'il puisse fournir, à savoir l'expression uniquement simplifiée.

3.3.4 Limitations

S'appuyant à nouveau sur **Jep**, cette phase souffre du même défaut que la précédente, à savoir qu'il est impossible de traiter les expressions arithmétiques faisant des appels à des fonctions ou à des éléments de tableaux. C'est une limitation encore plus contraignante à ce niveau dans la mesure où cela empêche également de simplifier les expressions. Ainsi, l'expression suivante :

```
f(a) + f(a) + f(a) + 3
```

en plus de ne pouvoir être évaluée, ne sera même pas simplifiée en :

```
3 * f(a) + 3
```

3.4 Troisième phase : génération du graphe

3.4.1 Description du problême

Le but de cette passe est de transformer le code MatLab généré par les deux premières phases en un graphe. Le code MatLab fourni est limité dans sa sémantique par rapport aux possibilités du langage. En effet, on n'interprète pas les règles de grammaire suivantes :

- les boucles **for** (celles-ci sont déjà dépliées dans la première phase)
- les boucles **while**
- les branchements conditionnels (**if** ... **then** ... **else** ... **end**)
- la commande **return**

Le code à parser est donc assez simplifié.
Pour expliciter un peu mieux le problème, nous allons présenter un exemple :

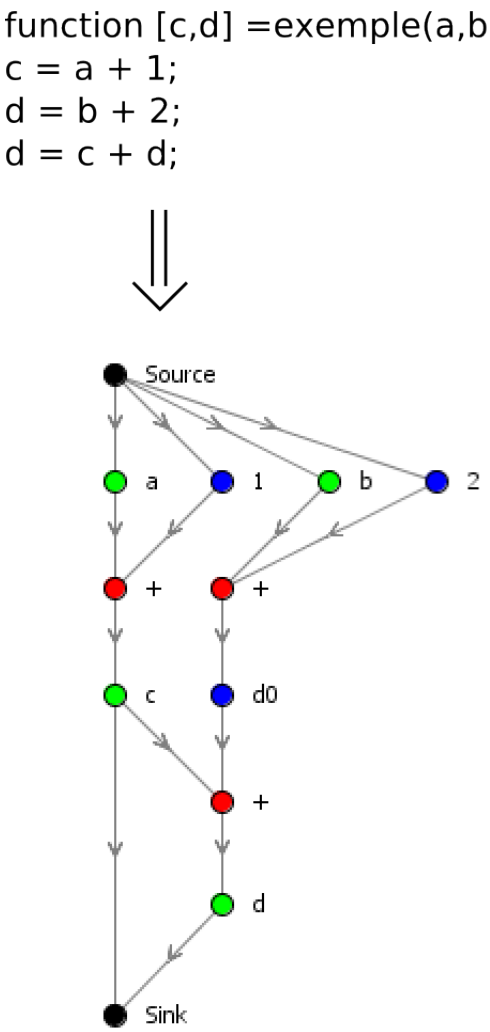


Fig. 3.3 – Exemple simple de graphe généré par MatlabParser

3.4.2 Reprise de l'existant

Ce PFA étant une suite de la version de l'année dernière, il nous a fallu reprendre l'existant de cette 3^{me} phase. Dans un premier temps, nous avons donc essayé de comprendre le code, les outils et les concepts réalisés l'année dernière.

Concernant les générateurs d'analyse syntaxique et sémantique, nous avons bien entendu conservé *Cup* et *JFlex*, ceux-ci étant des générateurs reconnus dans le monde Java et d'utilisation assez simple. L'un des gros avantages par rapport à *Flex* et *Yacc* du langage C est de pouvoir typer les terminaux et non-terminaux utilisés (par exemple Object, String, ...).

Nous avons également gardé la grammaire MatLab qui nous a été fournie.

Par contre, nous avons dû supprimer la quasi-totalité du code existant sur cette phase. Ceci est le résultat de deux causes principales :

1. L'absence de commentaire rendant le code très difficile à comprendre. Ceci était accentué par le fait

que le nom de chaque méthode de la classe principale était le même que celui de la règle, rendant l'interprétation des mécanismes difficile à comprendre.

2. Le type *symbole* (un type symbole bien conçu étant le principal facteur d'efficacité et d'évolutivité d'un compilateur) est un type "fourre-tout", sans réelle distinction entre variables, tableaux et fonctions et, surtout, sans utilisation des mécanismes de bases du langage Java (notamment de l'héritage).

Ainsi, avant de répondre aux demandes exprimées dans le cahier des charges, il a fallu refaire la plupart des fonctionnalités créées l'année dernière. Ceci passa par une nouvelle représentation des données et un code plus évolutif et mieux commenté.

3.4.3 Représentation des données

Nous allons détailler dans cette partie l'implantation du type *symbole* comme nous l'avons déjà expliqué, ainsi que les structures de données permettant de les manipuler (tables de hâchage, fils, ...)

Le type symbole

Pour implémenter un bon type symbole, il faut avant tout utiliser les avantages que nous procure le langage dans lequel est écrit le compilateur. Ainsi, pour décrire les différents symboles, nous allons utiliser le mécanisme d'héritage Java.

La première classe implémentée est donc la classe **MySymbol**, qui sera la classe Mère de toutes les autres classes pouvant être un symbole. Le nom **MySymbol** a été choisi pour éviter les collisions de type avec les symboles utilisés par *Cup* et *JFlex* (**symbol** et **Symbol**). Pour simplifier la démarche de création du graphe, chaque symbole (à quelques exceptions près que nous verrons plus tard) possède une référence vers le noeud du graphe correspondant. Le noeud est par défaut ajouté directement au graphe.

Pour décider ensuite des classes Filles à implanter, il faut étudier plus en détail les spécificités du langage MatLab. Le langage propose quatre principes que nous représenterons sous forme de symboles :

1. les constantes,
2. les variables,
3. les tableaux (unidimensionnels, bidimensionnels ou plus),
4. les fonctions

Les Constantes

Les constantes sont les cas de symbole les plus simples. Comme il faut créer une variable à chaque fois que l'on en rencontre une dans le code, les constantes n'ont pas besoin d'être référencées dans une table de symboles.

Les Variables

Les variables (commes les tableaux) ne sont pas déclarées comme en C avec un type donné. Typiquement, nous avons le genre de code suivant pour initialiser une variable *c* :

```
c = 1;
```

ou encore

```
c = a + b - 42;
```

Les variables devront être placées dans une table des symboles pour pouvoir être réutilisées ultérieurement. En MatLab, il n'y a pas de notion de portée des variables (autre qu'entre les fonctions). Ainsi, nous ne mettrons dans la table des symboles qu'une seule et unique instance de cette variable. Nous changerons alors le noeud courant du symbole suivant les besoins. Les variables sont représentées par la classe **Variable**.

Les Tableaux et Eléments de Tableau

Les tableaux se présentent sous deux formes en MatLab. Le premier point est l'initialisation de tableaux et le passage en paramètre de fonction. Par exemple, voici une initialisation d'un tableau bidimensionnel :

```
m = [ 1 , 2 , 3 ; 4 , 5 , 6 ];
a = f(m);
```

Nous remarquons qu'il peut y avoir dans ce cas collision entre variables et tableaux. Or, selon les souhaits de notre client, un tableau ne doit être initialisé qu'une seule fois. L'initialisation d'un tableau est donc assez rare en comparaison avec l'initialisation et l'utilisation de variables. Dans le but avoué de gain de performances, nous créerons par défaut un objet de type **Variable**, que nous détruirons dans les cas d'initialisation de tableaux.

L'autre utilisation des tableaux, et la plus courante, est l'affectation et l'utilisation d'un élément du tableau. Par exemple :

```
m(1,1) = 2;
m(1,1) = m(1,1)*42;
```

La tableau n'est pas nécessairement initialisé pour pouvoir accéder à un de ses éléments. Dans l'exemple, nous avons un tableau bidimensionnel, et nous utilisons l'élément à l'index $(1, 1)$.

Pour représenter un tableau, nous allons donc créer deux classes :

1. la classe **ArrayElement**, qui va représenter un élément du tableau avec un index précis.
2. la classe **Array**, qui va représenter le tableau lui-même.

Le type **Array** ne contiendra pas de référence à un noeud du graphe. Seules les instances de type **ArrayElement** possèderont les noeuds du tableaux. Pour ce qui est du passage en paramètre d'une fonction d'un tableau entier, il faut que chacun des éléments du tableau soit passé à la fonction. Ainsi, la classe **Array** contiendra un tableau de références à chacune des instances de type **ArrayElement** appartenant au tableau.

Les Fonctions

Les fonctions apparaissent sous une seule forme :

```
[a,b] = f(c,d);
e = g(h(2));
```

Sur cet exemple, f, g et h sont des tableaux. Comme nous pouvons le voir, les fonctions peuvent retourner plusieurs valeurs sous la forme d'un scalaire (ici $[a, b]$). Le principal problème est de pouvoir différencier les fonctions des éléments de tableaux. Il s'agit d'ailleurs du besoin exprimé dans le cahier des charges sous le nom de *Collision Tableau-Fonction*. Pour résoudre ce problème, nous ajoutons un type indifférencié appelé **FunctionOrArray** et qui sera remplacé par une instance de type **Function** ou **ArrayElement** lorsque nous pourrons les différencier. Les instances de type **FunctionOrArray** ne seront pas référencées dans la table des symboles car le symbole doit être différencié à la fin de chaque instruction de code MatLab pour pouvoir

compiler de façon "on-line". Les instances de **Function** seront référencées dans la table des symboles même si, à chaque appel de fonction dans le code MatLab, nous devons créer un nouveau noeud pour la fonction dans le graphe. Cet ajout permet de faire une bonne gestion d'erreur et de différencier plus vite élément de tableau et fonction.

Les variables temporaires

Les variables temporaires sont les seuls éléments n'apparaissant pas dans le code MatLab mais qui doivent être présents en tant que symboles. Les variables temporaires permettent de stocker le résultat d'une opération. Nous retrouvons ces variables temporaires dans le graphe. Elles ne doivent pas apparaître dans la tableau des symboles. La classe représentant les variables temporaires est **TemporaryVariable**.

Les structures de données

Comme nous l'avons déjà évoqué précédemment, il faut créer une table des symboles. Etant donné qu'il n'existe pas de système de portée de variables à l'intérieur d'une même fonction, nous imposons le fait qu'un symbole ne peut être ajouté qu'une seule fois dans la table. Pour vérifier que ce principe est respecté, en phase de développement nous avons imposé un retour dans le cas d'ajout multiple.

Il nous faut en plus ajouter d'autres tables :

1. Une table des entrées (Input Table). Celle-ci référence tous les tableaux et variables passés en paramètre de la fonction.
2. Une table des sorties (OutPut Table). Celle-ci référence toutes les variables retournées par la fonction (nous ne retournons pas de tableau).
3. Une table des globaux (Global Table). Celle-ci référence toutes les variables et tableaux globaux à la fonction.

Hiérarchisation des classes de symbole

Organiser les types symboles uniquement par l'héritage direct de la classe **MySymbol** ne permettrait pas de révéler les spécificités communes entre certains symboles.

Pour organiser de manière plus fine les symboles, nous avons étudié la grammaire MatLab. Il en est ressorti qu'il était possible de regrouper d'un coté les identifiants (**Variable**, **Function**, **Array**, **ArrayElement** et **FunctionOrArray**) et de l'autre ceux qui ne le sont pas (**TemporaryVariable** et **Constant**). Ainsi, nous avons créé les classes **Identifier** et **UnreferencedSymbol**, héritant de **MySymbol**, pour représenter respectivement les identifiants et les autres. De ce fait, les classes **Variable**, **Function**, **Array**, **ArrayElement** et **FunctionOrArray** héritent de **Identifier**, et **TemporaryVariable** et **Constant** de **UnreferencedSymbol**.

De plus, nous avons remarqué que les types/sous-types de **UnreferencedSymbol** ne sont pas référençables (comme son nom l'indique) dans la table des symboles. Ainsi tous les éléments héritants de **Identifier**, à l'exception de **FunctionOrArray** seront référencés dans la table des symboles.

Pour les trois autres tables, nous n'ajouterons que des éléments de type **Identifier** strictement (pas d'instance de classes filles). Ceci s'explique à la fois par la grammaire MatLab et par l'impossibilité de distinguer le symbole pour les trois mécanismes mis en oeuvre.

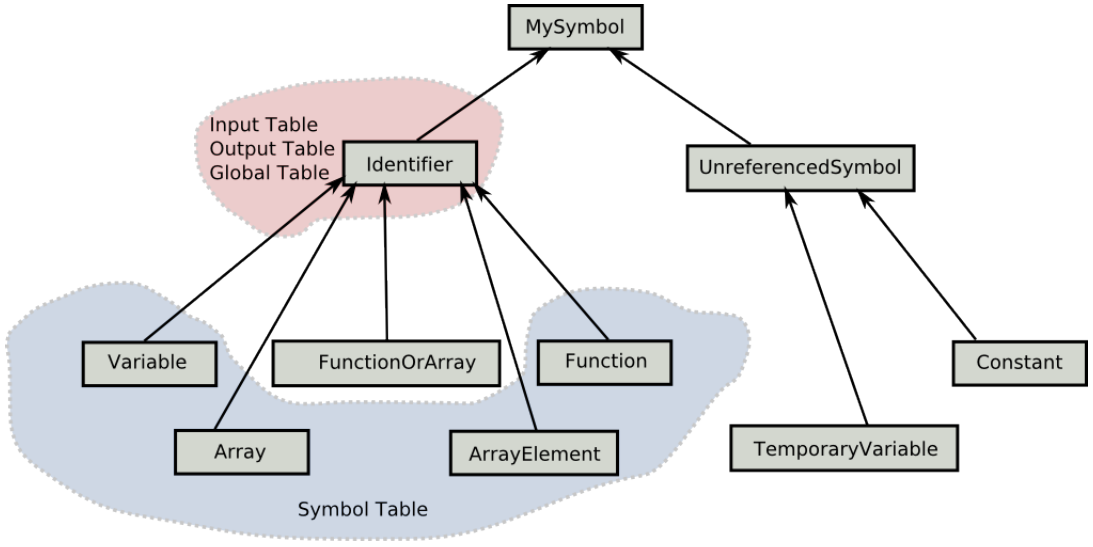


Fig. 3.4 – le type symbole / 3^{me} phase

3.4.4 Expression des Règles de Grammaires

Maintenant que les données ont été spécifiées et ordonnées, il faut les utiliser pour rendre le compilateur fonctionnel. Ainsi, nous allons détailler dans cette partie la façon dont sont exprimées les règles de grammaire pour générer le graphe.

Références et Identifiants

Il s'agit des règles de génération des variables, tableaux et fonctions. Il en existe deux :

1. une *référence* est un *identifiant*

2. une *référence* est un *identifiant* suivi de parenthèses ouvrante et fermante à l'intérieur desquelles il y a une liste de paramètres.

La première est la plus simple : il s'agit de celle servant à générer les variables et, plus rarement, les initialisations de tableaux. Par contre, la seconde peut générer soit un élément de tableau, soit une fonction. Nous comprenons mieux ici l'utilité de créer le type **FunctionOrArray**.

Les variables sont représentées dans le graphe par un unique noeud de type variable. Les éléments de tableau sont représentés par un unique noeud de type variable. La différence par rapport aux variables est que, dans le nom, est spécifié l'index dans le tableau. Les tableaux ne sont pas représentés dans le graphe. Les fonction sont représentées sous la forme suivante.

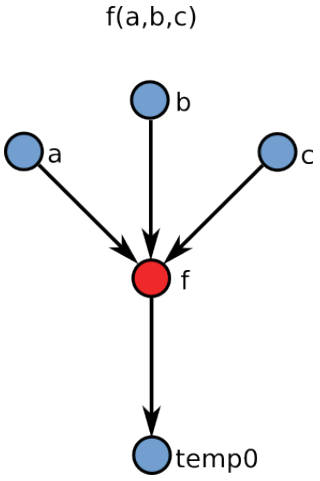


Fig. 3.5 – représentation de base de la fonction

De base, nous considèrerons que les fonctions ne peuvent retourner qu'une seule valeur (qui sera de base dans une variable temporaire).

Les règles Expression

Dans la grammaire MatLab, une expression est :

- une constante, ou
- une opération unaire (un opérateur suivi d'une expression), ou
- une opération binaire (une expression suivie d'un opérateur suivi d'une expression).

La première règle consiste en la création de l'instance de type **Constant**. Cette instance crée le noeud de la constante, l'ajoute au graphe et ajoute un arc de la source vers le noeud créé.

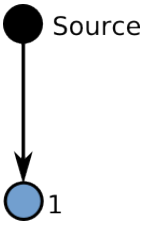


Fig. 3.6 – représentation d'une constante

La seconde reprend le noeud courant de l'expression, crée un noeud opérateur, ajoute un arc entre le noeud courant et le noeud opérateur, crée un noeud temporaire et ajoute un arc entre le noeud opérateur et le noeud temporaire.

Il y a néanmoins deux exceptions à cette règle dans la génération du graphe :

1. le *plus* unaire. Dans le cas du *plus* unaire, aucune modification n'est faite sur le graphe.
2. le *moins* unaire avec une constante. Dans ce cas, nous reprenons le noeud de la constants et changeons son nom. Dans le cas où le nom contient un "moins", nous l'enlevons, sinon nous l'ajoutons.

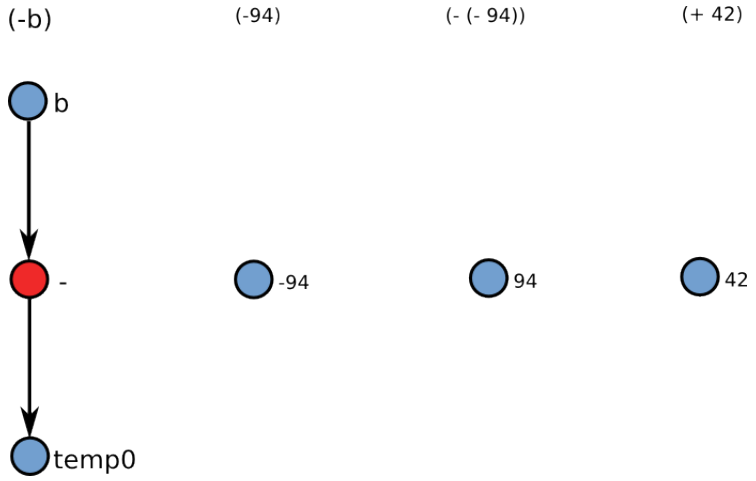


Fig. 3.7 – représentation d'un opérateur unaire

La troisième règle reprend les deux noeuds expressions, crée un noeud opérateur, ajoute un arc de chacun des noeuds expression vers le noeud opérateur, crée un noeud temporaire et ajoute un arc du noeud opérateur vers le noeud temporaire.

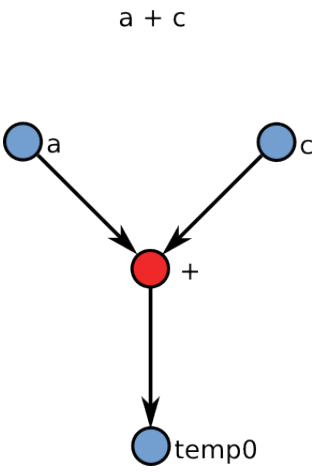


Fig. 3.8 – représentation d'un opérateur binaire

Matrices

Nous allons décrire ici les règles dites de matrices. Ces règles ne servent pas uniquement à construire des tableaux, elles servent aussi dans les fonctions retournant plusieurs variables. Ces règles s'expriment (de façon simplifiée) sous la forme suivante :

- Une matrice est une suite de lignes (les lignes étant séparées par des points virgules ou des sauts de ligne)
- Une ligne est une liste d'expressions séparées par des virgules.

Une matrice est bien entendu une expression. Il est donc possible de créer une matrice à l'intérieur d'une matrice et ainsi créer des tableaux à n dimensions. Toutefois, nous fûmes obligés de restreindre la grammaire pour ne faire que des tableaux unidimensionnels et bidimensionnels du fait d'une trop grande complexité de la tâche au vu du temps imparti.

Affectation

L'affectation est surement le mécanisme le plus complexe à gérer dans le parser. Il existe trois règles d'affectation :

1. une référence suivie d'un signe égal suivi d'une expression.
2. une référence suivie d'un signe égal suivi d'une matrice.
3. un scalaire (tableau unidimensionnel) suivi d'un signe égal suivi d'une expression.

La première règle est la plus simple : nous affectons une référence (donc une variable ou un élément de tableau) avec une expression. Sur le graphe, cette règle peut s'exprimer de deux manières différentes :

- si le noeud courant de l'expression est une variable temporaire, nous remplaçons le noeud (en créant une nouvelle instance du symbole ou en allant chercher le symbole dans la table des symboles) par le noeud représentant la référence.
- sinon, nous créons un nouveau noeud de la référence et nous ajoutons un arc entre le noeud représentant l'expression et le noeud de la référence.

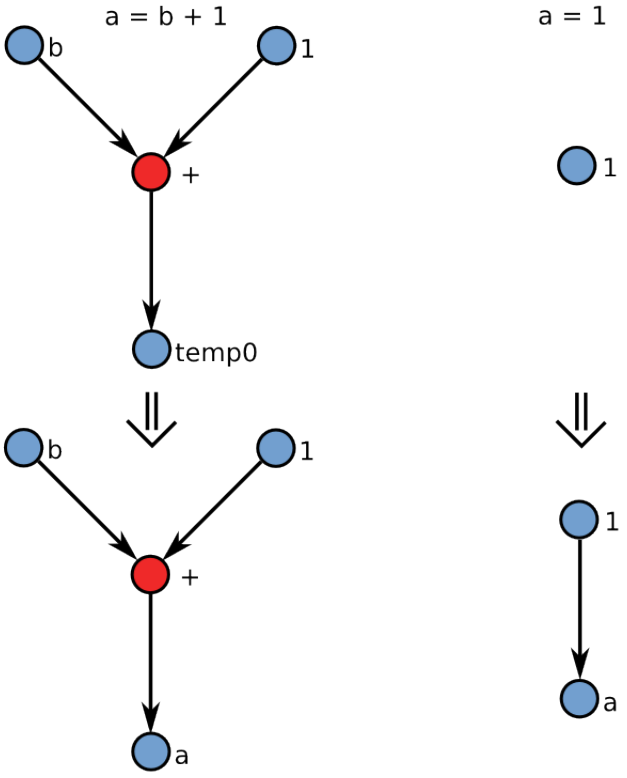


Fig. 3.9 – représentation d'une affectation

Ce mécanisme introduit un nouveau principe que nous appellerons les *Labels*. Ce système permet de générer un numéro spécifique à chaque noeud d'une même référence. Ainsi, pour une variable c, nous pourrons créer successivement les noeuds c, $c0$, $c1$, $c2$, ... Nous génèrerons un nouveau noeud à chaque affectation de la référence. La seule exception concernant la création d'un noeud sans affectation apparaît lorsque que la référence est passée en paramètre ou est globale lors de la première utilisation.

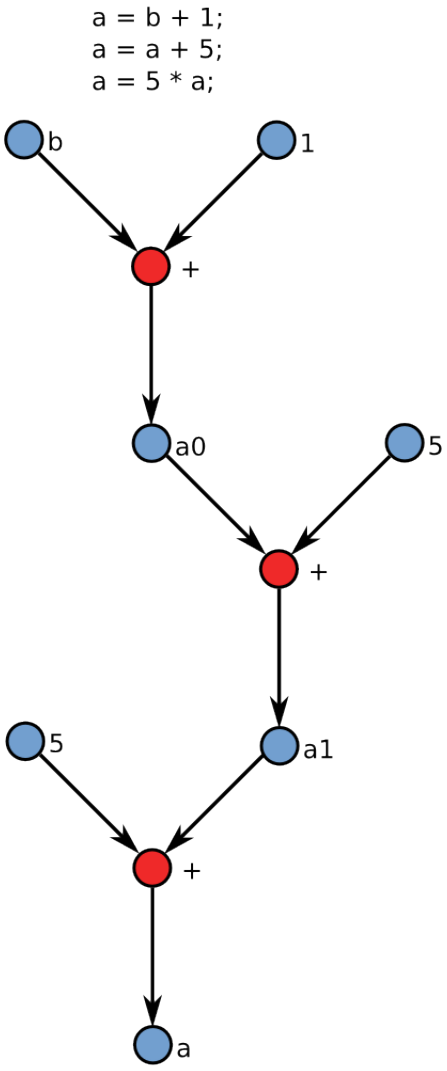


Fig. 3.10 – exemple d'utilisation des labels

Maintenant que nous avons vu cette première règle, les deux autres deviennent plus faciles à exprimer en gardant les mêmes principes.

Pour la seconde règle, il s'agit de l'initialisation d'une matrice que nous avons déjà présentée dans la section précédente. Pour réaliser cette règle, il suffit de stocker le contenu de la matrice puis, lors de la règle d'affectation, d'affecter en boucle les éléments de la matrice en utilisant les mêmes méthodes que pour la première règle.

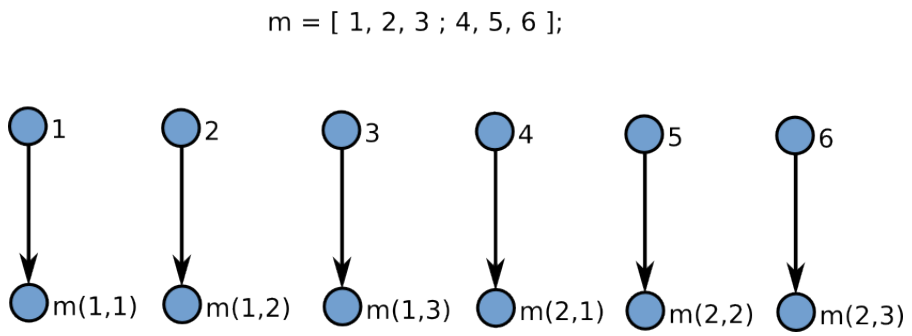


Fig. 3.11 – exemple d'affectation d'un tableau (bidimensionnel)

Pour la troisième règle, il s'agit d'un retour de fonction avec plusieurs valeurs. Nous avons typiquement le genre de code MatLab suivant :

```
[a,b] = f(c);
```

f est une fonction prenant un seul paramètre et retournant deux valeurs. Dans MatlabParser, nous autorisons uniquement en retour de fonction des variables et des éléments de tableau. Nous ne pouvons interpréter le retour d'un tableau en entier. Pour cette règle d'affectation, nous supprimons le noeud temporaire et nous ajoutons un noeud de variable ou d'élément de tableau pour chaque variable de retour, ainsi que les arcs manquants.

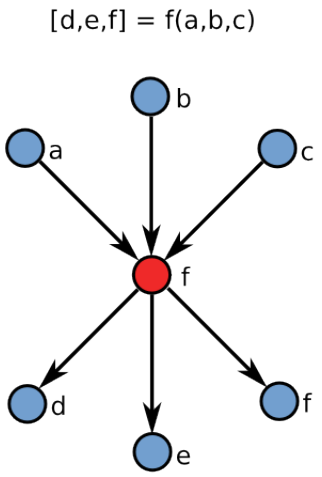


Fig. 3.12 – exemple de fonction à plusieurs retours

Déclaration de fonction

Nous allons maintenant nous intéresser aux déclarations des fonctions. Nous considérons que le nom déclaré pour la fonction est le même que celui d'un fichier MatLab dans lequel il est contenu. Les déclarations de fonctions sont de la forme suivante :

```
function [a,b,c] = nomDeFonction(d,e,f)
```

a, b et c sont des variables de retour de la fonction. d, e et f sont des variables ou des tableaux de paramètres de la fonction. Pour les paramètres d'entrée, il est impossible de savoir s'il s'agit de tableaux ou de variables. C'est pour cela que les éléments des tables d'entrée et de sortie ne peuvent prendre que des instances de **Identifier**. Nous utilisons la table des entrées pour créer une variable ou des éléments de tableau avec un arc venant de la source du graphe, et la table des sorties pour ajouter un arc du noeud de la variable vers la sortie. Dans le cas de création d'un nouveau noeud représentant une variable de sortie, nous supprimons l'arc de l'ancien noeud vers la sortie et nous ajoutons un nouvel arc entre le nouveau noeud et la sortie.

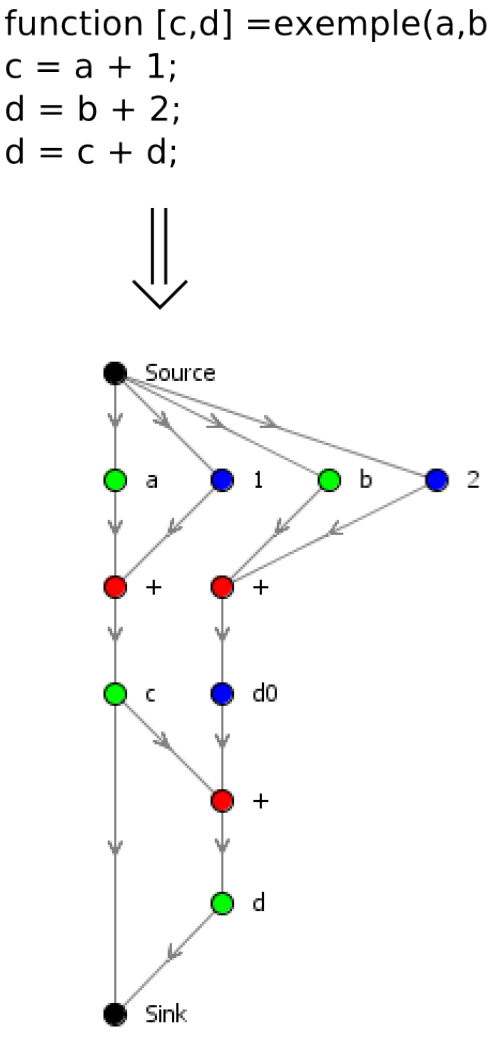


Fig. 3.13 – Entrées et sorties de fonction

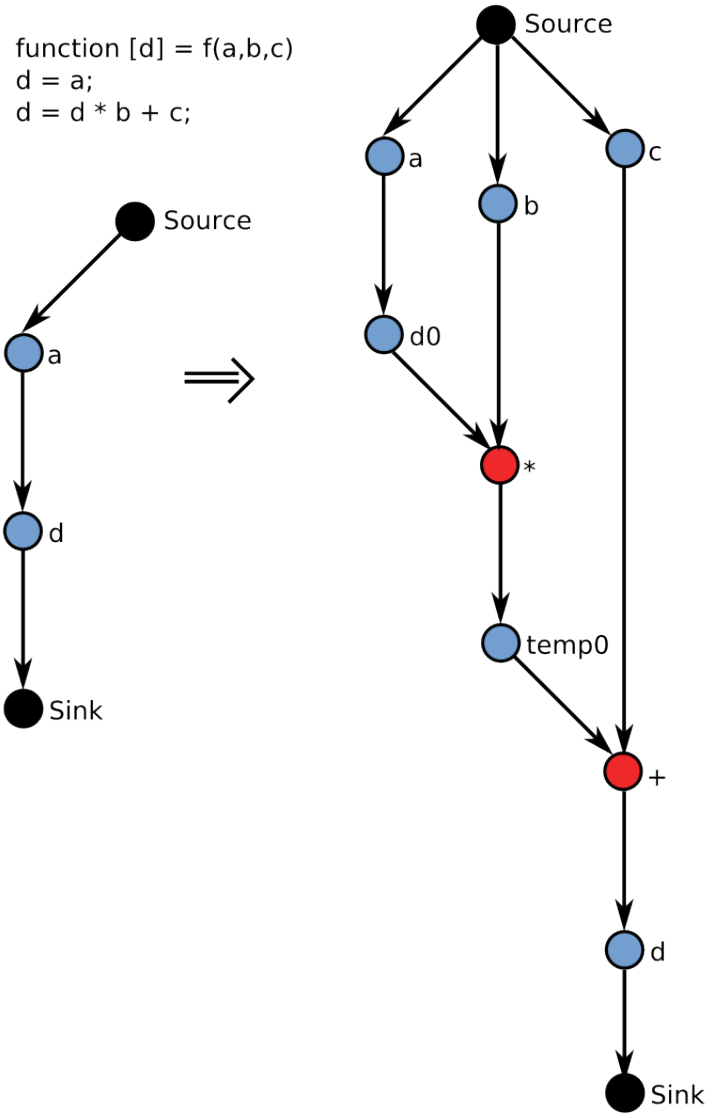


Fig. 3.14 – Suppression et ajout des arcs vers le puits (Sink)

Les noeuds de variable d'entrée et de sortie ont un attribut spécial pour les distinguer des autres noeuds.

Déclaration globale

Une variable ou un tableau peut être déclaré comme global à partir de l'instruction :

```
global nomDeVariableOuTableau;
```

Cette déclaration n'est pourtant pas obligatoire dans le cas d'une variable. Si la variable n'est pas présente en entrée de fonction et qu'elle est utilisée avant d'être affectée, cela signifie qu'elle est globale. Elle est alors ajoutée à la table des globaux.

Dans le cas des tableaux, ceci n'est pas possible car on ne pourrait pas faire la différence avec une fonction. La déclaration avec le mot-clef *global* est donc obligatoire.

On notera que si une variable ou un tableau est déclaré global mais qu'il est affecté avant d'être utilisé, la déclaration est considérée comme fausse et la variable ou le tableau est considéré comme local.

3.4.5 Retour sur les tâches exprimées dans le cahier des charges

Dans la partie précédente, nous avons exprimé tous les principes de base concernant le développement du parser. Nous avons notamment refait le travail de l'année dernière avec une meilleure structure des données et un débogage accru, et nous avons en plus développé quelques tâches à accomplir cette année. Nous allons maintenant revenir sur les points exprimés dans le cahier des charges pour répertorier les tâches que nous avons résolues et développer celles que nous n'avons pas encore abordées dans ce rapport.

Collision Tableau-Fonction

Dans les parties précédentes, nous avons donné les structures permettant de différencier les tableaux et les fonctions. Il ne reste plus qu'à exprimer l'algorithme de différenciation.

Si l'élément indifférencié est à droite de l'affectation :

- s'il n'apparaît pas dans la table des entrées, des globaux ou qu'il n'existe pas dans la table des symboles en tant que tableau, alors c'est une fonction,
- sinon c'est un élément d'un tableau, à condition que l'élément ait été affecté au moins une fois, sinon c'est une erreur.

Si l'élément indifférencié est à gauche de l'affectation, alors c'est un élément de tableau.

Il est possible de détecter le fait d'être en partie droite d'affectation lorsque l'élément est :

- soit à l'intérieur d'une expression,
- soit en paramètre d'une fonction,
- soit directement à droite de l'affectation.

Pour que l'algorithme de différenciation soit efficace, c'est-à-dire pour que la génération du graphe soit 'on-line', il faut que la différenciation soit exprimée pour chacune des possibilités de détection exprimées ci-avant.

Gestion des variables globales et noeuds $Z-1$

La gestion des variables globales consiste principalement en leur détection. Ce point a déjà été abordé dans la section précédente.

Il reste maintenant à voir l'ajout des noeuds $Z-1$ dans le graphe.

Le principe est assez simple. D'abord, sur le premier noeud de la variable global sera ajouté comme prédécesseur le noeud $Z-1$. Le noeud et l'arc ne sont ajoutés que lors de la création d'un nouveau noeud (la création d'un nouveau noeud ne pouvant survenir qu'après une affectation). Nous gardons un pointeur sur le noeud $Z-1$. Ensuite, pour chaque nouveau noeud de la variable créée, nous supprimons les arcs prédécesseurs de $Z-1$ et nous ajoutons un nouvel arc entre le noeud courant de la variable et le noeud $Z-1$.

Il ne manque plus qu'un seul élément dans l'algorithme : la détection d'un vieillissement ou d'une adaptation entraînant le $Z-1$. Pour cela, il ne reste plus qu'à détecter l'utilisation à droite de l'affectation comme nous l'avons fait pour la partie **Collision Tableau-Fonction**, et nous démontrons ainsi qu'il y a utilisation avant affectation.

Pour finir, nous avons montré au client un exemple de code générant un noeud $Z-1$:

```
function [a] = Z()
c = b + 2;
b = c - 1;
a = b * 5;
```

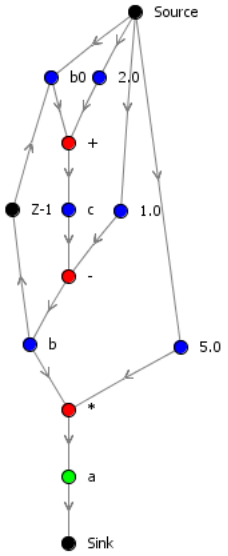


Fig. 3.15 – Exemple de graphe avec un noeud $Z-1$

Fonction renvoyant plusieurs valeurs

Cette partie a déjà été résolue dans la section précédente.

Déclaration de fonction dans les fichiers annexes

Cette partie, que nous n'avons toujours pas abordée, consiste en la génération d'un mécanisme simple de compilation séparée. Le principe est simple : pour chaque fonction rencontrée dans le fichier parsé, nous regardons dans le dossier si un fichier du même nom suivi de ".m" existe. S'il existe, nous l'analysons sinon, nous ne faisons rien.

Pour implémenter cette fonctionnalité, nous avons ajouté à MatlabParser une table de hâchage et une file, contenant toutes les deux les noms des fichiers à parser. La table de hâchage permet de ne pas parser plusieurs fois le même fichier. Le raisonnement est le suivant : à chaque 'tentative' d'ajout de la fonction dans la file, nous regardons si la fonction existe dans la table. Si tel est le cas, nous ajoutons la fonction à la table et à la file, sinon, nous ne faisons rien. Les éléments de la file sont ensuite analysés jusqu'à ce que la file soit vide.

3.4.6 Gestion d'erreur

La gestion des erreurs n'est pas une tâche explicitement énoncé dans la cahier des charges. Néanmoins, tout bon compilateur, parser ou interpréteur, doit en posséder une. Voici la liste des erreurs gérées par MatlabParser :

- Duplication des noms d'entrée (erreur lorsqu'une variable est présente deux fois dans les paramètres d'entrée et de sortie de la fonction),
- Déclaration d'une variable globale alors qu'elle est déjà présente en entrée ou en sortie,
- Incompatibilité entre symboles (exemple : élément du code utilisé comme fonction, puis comme variable),
- Retour de fonction non scalaire (une fonction doit retourner soit une variable, soit un tableau unidimensionnel composé de variables),
- Initialisation de matrices avec des éléments non constants,
- Utilisation d'un élément de tableau non initialisé (n'étant ni global, ni en entrée),

– Accès à un élément de table via un index non constant (exemple : m(x,y) où la valeur de x et/ou de y est inconnue).

Ainsi que la liste des warnings :

– Variable globale non déclarée,
– Fonction dont le fichier de code est inexistant.

Nous avons également fait en sorte de donner une liste des erreurs non gérées par le parser :

– Erreur dans le nombre de paramètres passés à une fonction,
– Erreur dans le nombre de variables de retour d'une fonction.

Il existe sûrement d'autres possibilités d'erreurs que nous n'avons pas soulevées, cependant, nous avons traité celles qu'il nous semblait indispensable de gérer..

3.4.7 Les améliorations à apporter

Il reste quelques points non développés dans le parser qui pourraient améliorer ce dernier. Il faudrait notamment :

1. Compléter la gestion des tableaux (aller au delà du bidimensionnel),
2. Compléter la gestion d'erreur,
3. Réaliser d'autres points non abordés (branchements conditionnels, tableaux en retour de fonction,...)

3.5 Modifications apportées à la 3^{me} phase

Le graphe généré après la troisième passe ne correspond pas complètement au graphe attendu en sortie. Il y a trois points qui posent problème :

1. Les noeuds liés à *Source* et *Sink* ne sont pas dans le même ordre, selon l'ordre des paramètres d'entrée et de sortie de la fonction,
2. Le graphe peut contenir des chemins n'allant jamais vers le puits (noeud *Sink*), comme par exemple, dans le cas d'une affectation de variable jamais réutilisée par la suite (et n'étant pas une variable de sortie),
3. Le graphe peut ne pas être biparti. Ceci est dû aux cas d'affectations directes tels que $a = b$; ou $a = 2$;.

Dans cette partie, nous allons expliquer comment nous avons résolu ces 3 problèmes. Les 2 premiers sont resolus par l'intermédiaire d'une classe contenant uniquement des méthodes statiques appelée **GraphCleaner**. Le troisième est résolu via la classe **RemoveUnusedVariable**, qui fut développée par Bertrand Le Gal lors de la version précédente du parser.

3.5.1 Réorganisation des noeuds d'entrée et de sortie

Pour bien comprendre le problème, nous allons donner un exemple :

```
function [c] = f(a,b)
c = 1 + b + a;
```

Après la 3^{me} phase, nous obtenons le graphe suivant :

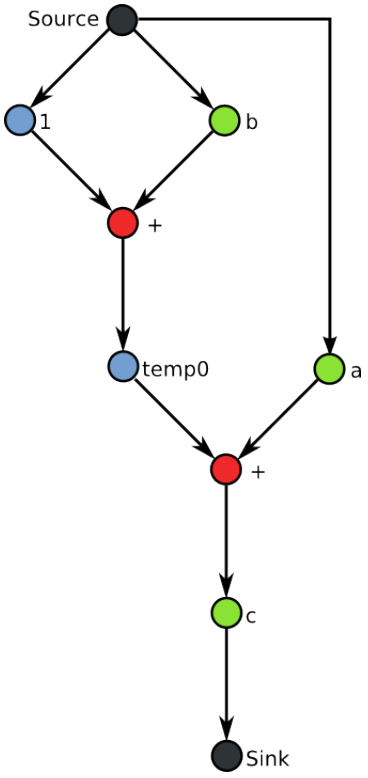


Fig. 3.16 – Ordre des entrées généré par la 3^{me} phase

le premier successeur de la source est le noeud 1, le deuxième *b* et le troisième *a*. On remarque que l'ordre des arcs du noeud vers les noeuds d'entrée n'est pas respecté : on voudrait en successeur *a*, puis *b*, puis 1.

Le même problème se pose pour la sortie, même si c'est plus simple car les noeuds prédécesseurs du noeud Sink sont toutes des sorties.

L'algorithme reste néanmoins assez simple et ce fait en 2 étapes :

1. Pendant la 3^{me} phase, on stocke l'ordre des entrées et des sorties dans un vecteur de manière à garder l'ordre des paramêtres.
2. Une fois la troisième passe terminée, on cherche dans la table des symboles les variables et tableaux correspondant aux noms contenus dans le vecteur ci-dessus de manière à garder l'ordre. Dans le cas de l'entrée, on supprime tous les arcs provenant de la *Source*, on recrée les arcs dans l'ordre donné par le vecteur. Puis on recrée les arcs entre la *Source* et les noeuds ne correspondant pas aux variables d'entrées (variables globales et constantes). Dans le cas de la sortie, il s'agit du même algorithme avec la *Sink* et ses noeuds prédécesseurs. La seule différence est qu'on n'a pas à ajouter d'autres arcs que pour les noeuds de sorties.

3.5.2 Suppression des chemins inutiles

Pour bien comprendre le problème, nous allons de suite donner un exemple :

```
function [c] = f(a)
c = a + 1;
c = a + 2;
```

On obtient les résultats suivants :

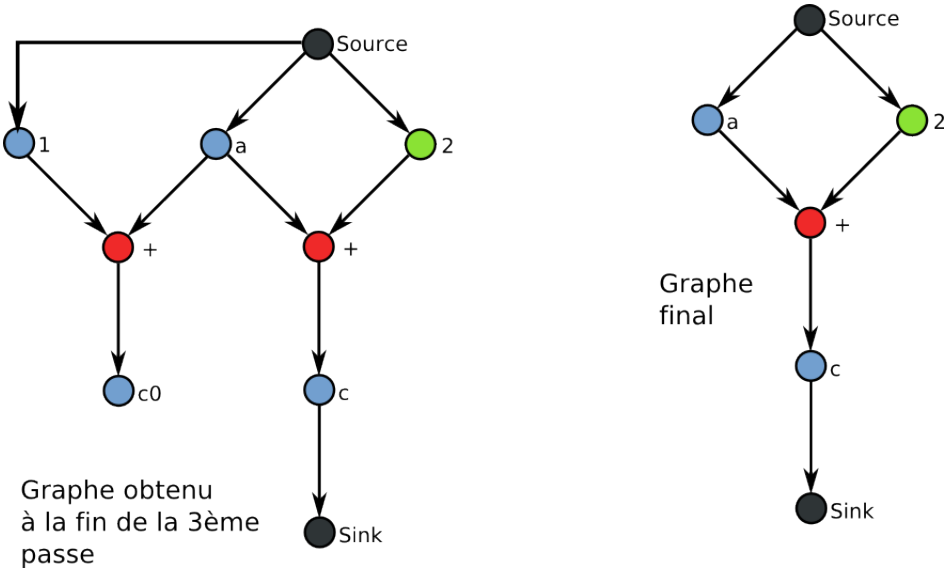


Fig. 3.17 – Exemple de suppression de chemins inutiles

En effet, la première affectation est inutile, sauf qu'elle est quand même parser. Nous avons donc ajouté à **GraphCleaner** l'algorithme récursif de suppression des chemins inutiles suivant : on parcourt l'ensemble des noeuds de type Variable du graphe et on supprime le noeud (et les arcs du noeud). On regarde pour chacun des noeuds anciennement prédécesseur du noeud courant s'il est lui-même sans successeur et on le supprime, etc...

3.5.3 Bipartition du graphe

Le graphe en sortie de la 3^{me} phase n'est pas toujours biparti : en effet, dans le cas d'affectation de type $a = b$;(expression à droite de l'affectation variable ou fonction), le graphe possède un noeud a ayant comme successeur le noeud b. Ce défaut s'explique par le fait que l'on ne puisse avoir 2 symboles différents (ici ceux de a et b) ayant une même référence à un noeud sans augmenter de manière conséquente la difficulté d'implantation du parser.

Pour résoudre ce problème, nous avons repris une classe développée par Bertrand Le Gal qui permettait justement de concaténer deux sommets de type variable successif dans le graphe. Pour la concaténation des sommets, on choisit un sommet à supprimer et on réaffecte les successeurs et les prédécesseurs du noeud supprimé au noeud gardé. L'élection du noeud que l'on doit garder se faisait suivant l'algorithme suivant (on appelera *pred* le noeud variable prédédesseur et *succ* le noeud variable successeur) :

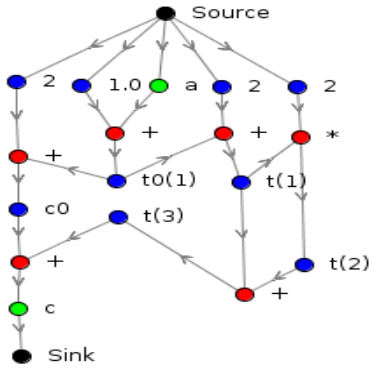
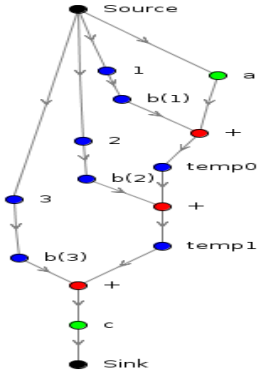
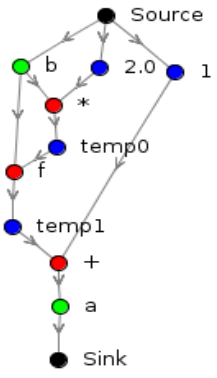
- si *pred* est une variable d'entrée, on supprime *succ*
- si *succ* est une variable de sortie, on supprime *pred*
- si *pred* et *succ* ne sont ni d'entrée, ni de sortie, on supprime *succ*

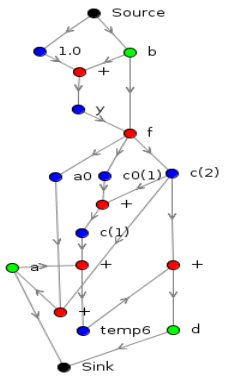
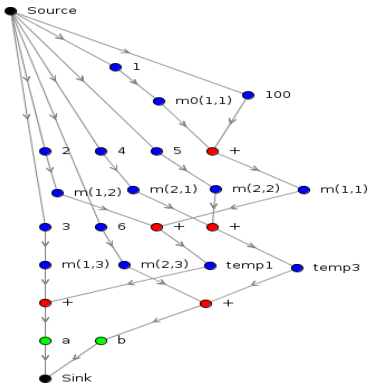
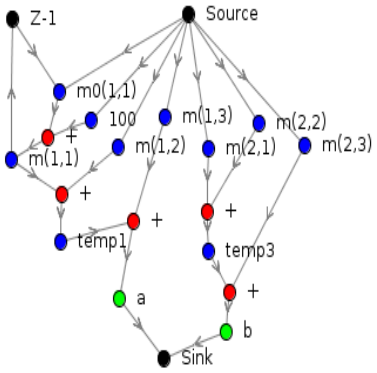
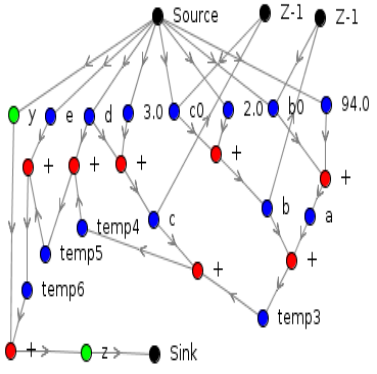
Nous avons repris cet algorithme auquel nous avons ajouté la gestion des noeuds $Z - 1$:

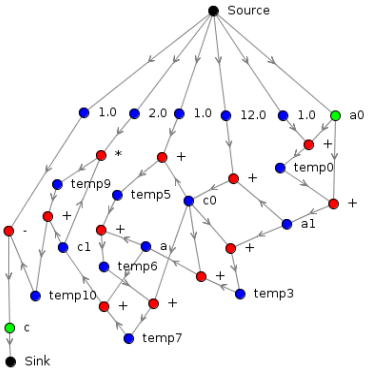
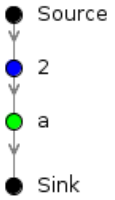
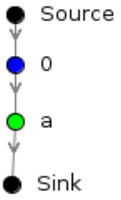
- si *pred* est une variable portant l'attribut **adaptation** ou **vieillissement**, alors on supprime *succ*

Il a également fallu empêcher la suppression de l'arc de la variable supprimée vers le noeud $Z - 1$.

3.6 Tests du Parser MatLab

Code Matlab à parser	Graphe généré	Remarques
function [c] = test(a) t(1) = a + 1 ; c = t(1) + 2 ; t(1) = t(1) + 2 ; t(2) = t(1) * 2 ; t(3) = t(2) + t(1) ; c = t(3) + c ;		Manipulation de tableau
function [c] = test(a) b = [1,2,3] ; c = a + b(1) + b(2) + b(3) ;		Declaration de tableau
function [a] = funcTest(b) [a] = f(b,2.b) + 1 ;		Fonction

function [a,d] = test(b) y = b + 1 ; [a, c(1), c(2)] = f(y,b) ; c(1) = c(1) + c(2) ; a = a + c(2) ; d = a + c(1) + c(2) ;		Fonction à plusieurs retours
function [a,b] = Matrix() m = [1,2,3 ;4,5,6] ; m(1,1) = m(1,1) + 100 ; a = m(1,1) + m(1,2) + m(1,3) ; b = m(2,1) + m(2,2) + m(2,3) ;		Matrice
function [a,b] = Matrix() global m ; m(1,1) = m(1,1) + 100 ; a = m(1,1) + m(1,2) + m(1,3) ; b = m(2,1) + m(2,2) + m(2,3) ;		Matrice globale
function [z] = f(y) a = b + (- (-94)) ; b = c + 2 ; c = d + 3 ; z = a + b + c + d + e + y ;		Z-1

function [c] = Test(a) a = a + 1 + a ; c = a + 12 ; a = c + a + c ; c = c + a + c + a + 1 ; c = c * 2 + (c - 1) ;		Label
function [a]=f() for i= -1 :2 c=i ; a=a*i ; end a=c ;		Boucle for
function [a]=f() for i= -1 :2 a=a*i ; end		Boucle for

4

Optimisations au niveau du graphe engendré

4.1 Présentation des différents types d'optimisation sur le graphe

Notre travail a consisté en l'ajout de la fonctionnalité d'optimisation d'un graphe.

Le terme "optimisation d'un graphe" peut se comprendre de différentes façons :

- le remplacement d'un motif constitué d'un noeud opérateur et de deux noeuds variables, comme par exemple le remplacement du motif représentant l'expression arithmétique **y=a+0** par le motif constitué du seul noeud représentant **y=a** que l'on peut voir sur la figure 4.1. Chacune de ces règles d'optimisation doit être explicitement définie par l'utilisateur de l'outil GraphLab, dans la partie "optimisation" d'un fichier nommé *optimization_file.xml*. Il ne s'agit pas là de calcul du motif de sortie à partir d'un motif d'entrée, mais seulement de remplacement de motif. Ce dernier est effectué à partir d'une règle, spécifiée par l'utilisateur, composée d'un motif d'entrée et d'un motif de sortie,
- la simplification via le calcul de l'expression arithmétique d'un motif, comme par exemple le remplacement du motif représentant l'expression arithmétique **y=sqr(4)** par le motif représentant le calcul de celle-ci, à savoir **y=2** visible sur la figure 4.2. Le client doit définir les règles de calcul pouvant être appliquées sur le graphe, dans la partie "simplification" du fichier *optimization_file.xml*. Il doit préciser un opérateur, ainsi qu'une ou plusieurs entrées, et écrire en sortie une expression arithmétique tout en pouvant faire reférence à une entrée grâce à son *id*. Celle-ci sera ensuite calculée par l'outil **Jep** lors de la rencontre de ce motif dans le graphe et sera remplacée par la valeur calculée,
- la parallélisation de motifs d'une hauteur maximale égale à 3 (la prise en compte de motifs d'une hauteur supérieure entrainant un temps d'exécution trop élevé). La parallélisation d'un motif consiste à remplacer un motif séquentiel (ou linéarisé) par le motif parallélisé correspondant. Par exemple, si l'on considère le motif linéarisé représentant l'expression arithmétique **y=(((a+b)+c)+d)**, la figure 4.3 montre le remplacement du motif par une forme parallélisée qui se traduit par l'expression arithmétique

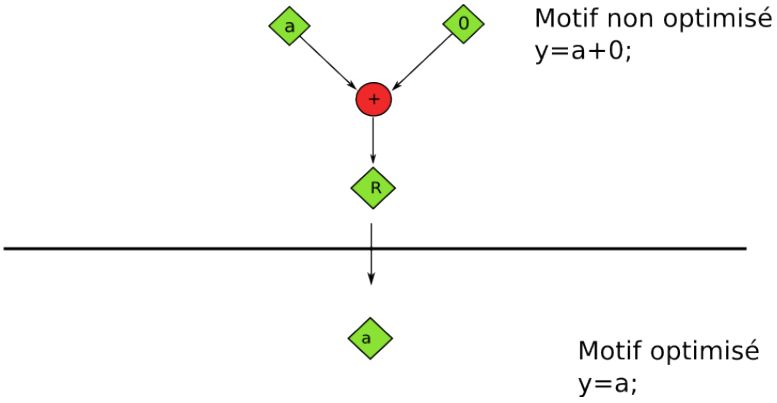


Fig. 4.1 – Détection d'un motif apparaissant dans une règle du fichier d'optimisations et remplacement par le motif optimisé

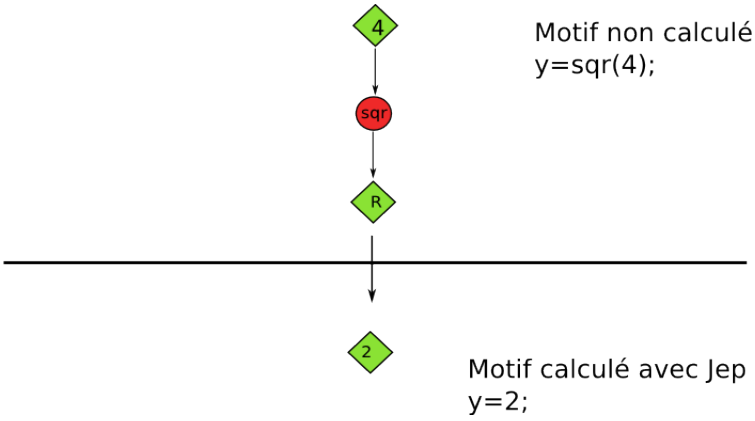


Fig. 4.2 – Simplification d'un motif cohérent avec une règle de simplification, via un calcul mathématique réalisé grâce à l'outil Jep

y=(a+b)+(c+d). Les règles de parallélisation de motifs doivent être écrites par le client dans le fichier *parallelization_file.xml*. La hauteur d'un motif linéaire détectable est fixée à 3. Nous considérons donc au total quatre sortes de motifs linéaires en entrée :

1. un peigne de hauteur 3 dont les arêtes sont toutes orientées à droite,
2. un peigne de hauteur 3 dont les arêtes sont toutes orientées à gauche,
3. un peigne de hauteur 2 dont les arêtes sont toutes orientées à droite, et
4. un peigne de hauteur 2 dont les arêtes sont toutes orientées à gauche,

ainsi que la sortie de forme parallélisée correspondante.
Dans chacune des règles, le motif d'entrée (un des quatre présentés ci-dessus) et le motif de sortie (le motif correspondant au motif d'entrée) sont imposés à l'utilisateur. Ce dernier ne pourra pas ajouter des règles d'optimisation qui sortiraient de ce cadre (à moins que des classes qui prennent en compte des optimisations de hauteur supérieure ne soient développées ultérieurement au projet), comme par exemple, créer une règle pour la reconnaissance d'un motif en forme de peigne de hauteur 7 par exemple. L'utilisateur a, à sa disposition, quatre sortes de règles, dans lesquelles il ne pourra compléter que les opérateurs et variables du motif d'entrée, ainsi que les opérateurs et variables du motif de sortie,

– la reconnaissance et la linéarisation de motifs parallélisés, comme par exemple, le remplacement du motif parallélisé **y=(a+b)+(c+d)** par le motif linéarisé **y=(((a+b)+c)+d)** comme le décrit la figure 4.4. Les règles de linéarisation de motifs doivent être écrites par l'utilisateur dans le fichier *linearization_file.xml* et, tout comme pour le fichier de parallélisation, les motifs d'entrée et de sortie sont imposés à l'utilisateur.

Nous décrirons par la suite les différents modes d'implantation de chacun de ces fichiers, ainsi que la détection dans le graphe de motifs compatibles avec un motif en entrée d'une règle donnée.

4.2 Remplacements de motifs simples et simplifications arithmétiques

Le plugin OptmizeGraph peut être séparé en deux phases distinctes :

1. la description et le chargement des règles d'optimisations en mémoire.
2. l'application de ces règles sur le graphe.

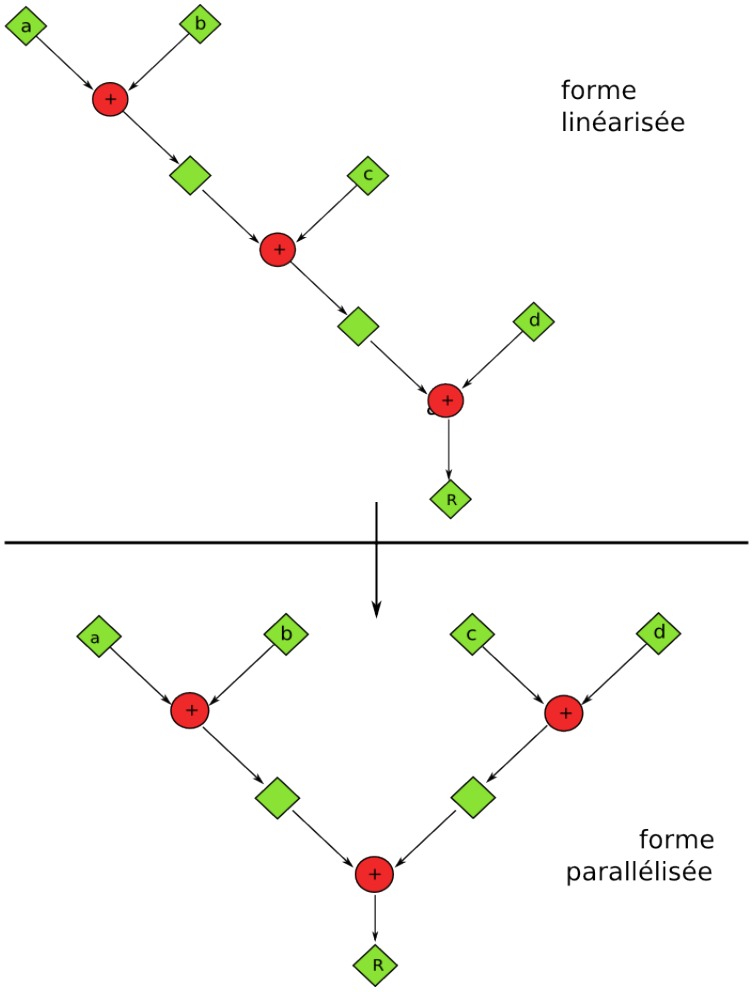


Fig. 4.3 – Parallélisation du motif représentant l'expression arithmétique **y=(((a+b)+c)+d)**

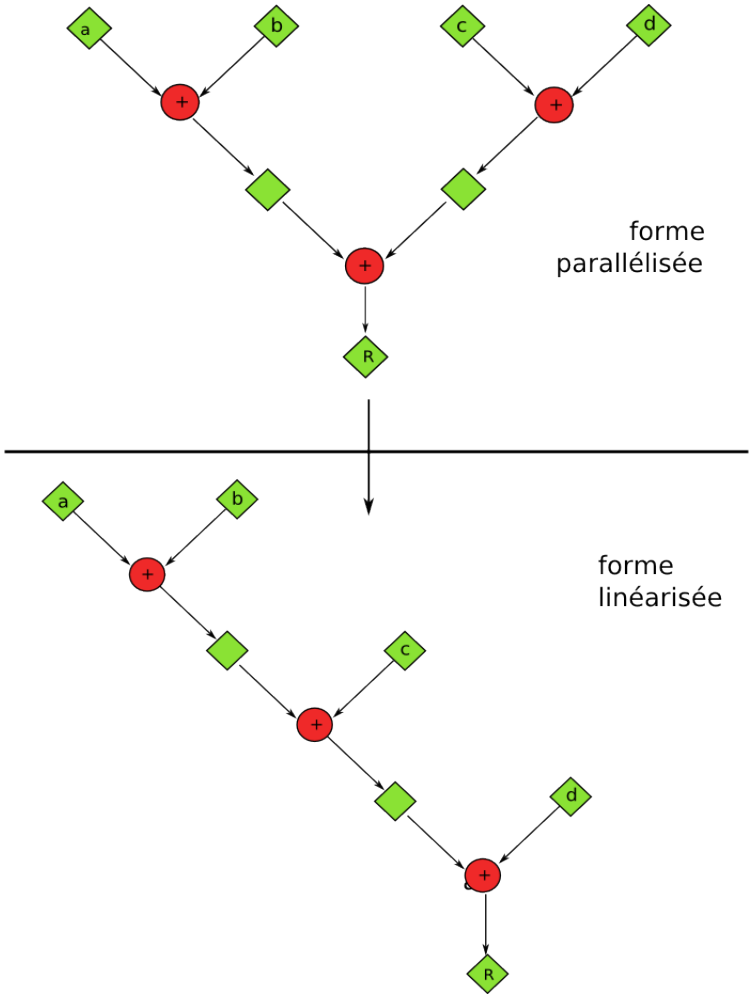


Fig. 4.4 – Linéarisation de motif

4.2.1 1ère phase : Fichier de description des règles d'optimisation

Ce fichier a été mis en place afin de décrire les règles d'optimisation applicables sur un graphe généré auparavant par l'analyseur syntaxique pour le langage MatLab.

Il a été écrit en X.M.L. car comparé aux langages de type binaire ou autres, X.M.L. a une syntaxe générique et extensible. Il permet de structurer une grande variété de contenus, car son "langage" (vocabulaire et grammaire) peut être redéfini.

Ces optimisations se font au niveau d'un noeud du graphe et sont de deux types :
- remplacements de motifs,
- simplifications arithmétiques.

Ainsi, nous avons dû introduire deux éléments de syntaxe différents pour ces deux types d'optimisations.

La racine du document est *<optimization_file>*. La bonne mise en forme du document est vérifiée par un analyseur syntaxique qui utilise à la fois une D.T.D. et une fonction que nous avons écrite et qui permet de vérifier ce que la D.T.D. ne permet pas.

Avec les erreurs syntaxiques vérifiées via la D.T.D., des messages assez explicites sont renvoyés dans la console de GraphLab, comme par exemple :

```
Error:
   Public ID: null
   System ID: file:/home/darkweaver/pfa/src/GraphLab/ressources/optimization_file.xml
   Line number: 8
   Column number: 30
   Message: Attribute "commutatives" must be declared for element type "inputs".
The document is not well formed !
```

Par contre, les erreurs renvoyées par notre fonction le sont moins car une fois la validation faite par la D.T.D., on ne peut plus dire que le travail s'effectue sur un fichier mais sur l'arbre D.O.M.. Il est donc difficile, textuellement, de se situer dans un arbre. Par exemple :

```
Error in /optimization_file/optimizations/optimization/old_node name="*"/inputs/variable
e12 does not match the valid id format e1 ... eN !
The document is not well formed !
```

Lorsqu'aucun problème syntaxique n'a été trouvé dans le fichier d'optimisations, le message *The document is well formed !* est affiché.

Les remplacements de motifs

Dans cette section, nous nous proposons de décrire la syntaxe du fichier d'optimisations en ce qui concerne la partie remplacement de motifs.

La partie remplacement de motifs du fichier d'optimisations est décrite entre les balises *<optimizations>* et *</optimizations>*.

La définition d'une optimisation (au sens remplacement de motif) est nécessairement sous la forme suivante :

```
1  <optimization>
2    <old_node name="-">
3      <inputs>
4        <variable id="e1"/>
5        <constant id="e2"/>
6      </inputs>
```

```
        <conditions>
          <predicat not="true" name="estPositif" idref1="e2"/>
        </conditions>
      </old_node>
      <new_node>
        <node_operator name="+">
          <node_variable idref="e1"/>
          <node_constant idref="e2" opposite="true"/>
        </node_operator>
      </new_node>
    </optimization>
```

Sur cet exemple, nous voyons comment décrire une règle qui permet de remplacer $a - b$ en $a + b$ dans le cas où b est négatif.

L'attribut *name* de *old_node* permet de spécifier le nom du noeud, c'est-à-dire le nom de l'opération correspondant à l'optimisation.

Dans le noeud *inputs*, il faut spécifier les entrées de l'opération qui peuvent être de type *variable* ou *constant*. Ce noeud peut également avoir un attribut *commutative*, par défaut à *false*, qui à *true* permet, sur un opérateur binaire, d'introduire de la commutativité dans la règle.

Les noeuds de type *variable* peuvent avoir plusieurs attributs :
- *id*, un identifiant unique. Les identifiants doivent être de la forme e1..e*n*,
- *idref*, une référence vers un identifiant déclarée préalablement. Ceci est utile lorsque l'on veut, par exemple, signaler que deux entrées sont égales (au sens syntaxique du terme, et non sémantique).

Les noeuds de type *constant* peuvent avoir plusieurs attributs :
- *id*, un identifiant unique. Les identifiants doivent être de la forme e1..e*n*,
- *value*, une valeur spécifique, comme par exemple l'entier 1.

Le noeud *conditions* est facultatif. Il permet de spécifier certains prédicats (balise *predicat*) à vérifier sur les entrées afin d'appliquer une règle. Les noeuds de type *predicat* peuvent avoir plusieurs attributs :
- *name*, le nom du prédicat. Par défaut, sa valeur est *none*, ce qui signifie qu'il n'y a pas de prédicat à vérifier. Il peut prendre les valeurs suivantes : *estMultiple*, *estOppose*, *estEgal*, *estInferieur*, *estSuperieur*, *estPositif* et *estPuissance*,
- *idref1* (facultatif), la référence vers le premier paramètre de la fonction prédicat à vérifier,
- *idref2* (facultatif), la référence vers le deuxième paramètre de la fonction prédicat à vérifier, s'il est nécessaire,
- *value1* (facultatif), une valeur pour le premier paramètre de la fonction prédicat à vérifier,(attention ceci remplace idref1),
- *value2* (facultatif), une valeur pour le deuxième paramètre de la fonction prédicat à vérifier, s'il est nécessaire,(attention ceci remplace idref2),
- *id* (facultatif), il s'agit d'un identifiant permettant de faire référence au résultat calculé lors de l'évaluation de prédicats fournissant aussi un résultat (en l'occurence estPuissance qui renvoie la puissance en question).
- *not*, par défaut à *false*, permet d'indiquer que l'utilisateur veut vérifier le prédicat contraire de celui spécifié.

Le noeud *new_node* permet de décrire le nouveau noeud de sortie, c'est-à-dire, après optimisation. Dans ce noeud doit être représentée l'expression de sortie sous la forme d'un arbre. Pour cela, les noeuds *node_operator*, *node_constant* et *node_variable* peuvent être utilisés.

Par exemple, pour l'expression $(x + 1) * y$, l'arbre doit être formé comme celui-ci :

```
<node_operator name="*">
    <node_operator name="+">
        <node_variable idref="x"/>
        <node_constant value="1"/>
```

```
5     </node_operator>
6     <node_variable idref="y"/>
7 </node_operator>
```

Les noeuds de type *node_operator* sont nécessairement les seuls pouvant contenir d'autres noeuds *node_operator*, *node_constant* et *node_variable*. Ils ont deux attributs :
- *name*, le nom de l'opérateur,
- *nb_out*, le nombre de racines des arbres, dans le cas où plusieurs expressions sont en sortie. Ce nombre est par défaut à 1.

Les noeuds de type *node_variable* n'ont qu'un seul attribut *idref* (obligatoire) qui est la référence vers un noeud de l'entrée.

Les noeuds de type *node_constant* peuvent avoir les attributs suivants :
- *value*, la valeur de la constante,
- *opposite*, par défaut à *false*, qui à *true* indique que l'opposé de la constante à laquelle nous faisons référence, doit être pris,
- *idref*, une référence à une constante de l'entrée.
- *predref*, une référence au résultat d'un prédicat.

Les simplifications arithmétiques

Dans cette section, nous nous proposons de décrire la syntaxe du fichier d'optimisations en ce qui concerne la partie simplifications arithmétiques.

La partie simplifications arithmétiques du fichier d'optimisations est décrite entre les balises *<arithmetic_computation>* et *</arithmetic_computation>*.

La définition d'une optimisation (au sens simplification arithmétique) est nécessairement sous la forme suivante :

```
1     <computation>
2       <old_node name="sin">
3         <inputs>
4           <constant id="e1"/>
5         </inputs>
6       </old_node>
7       <new_node>
8         <node_constant id="s1" eval="sin(e1)"/>
9       </new_node>
10    </computation>
```

Sur cet exemple, nous voyons comment décrire une règle qui permet de calculer automatiquement le sinus d'une constante.

Afin de rendre à l'utilisateur le remplissage du fichier de règles plus aisé, nous avons gardé la même syntaxe que pour les remplacements de motifs.

Ainsi, nous nous proposons ici d'axer notre propos sur les différences entre la déclaration d'une simplification arithmétique et d'un remplacement de motif.

Dans les noeuds *inputs*, les noeuds *variable* ne sont pas autorisés, un calcul ne pouvant se faire que sur des constantes.

De la même manière, un calcul sur des constantes ne donnant qu'une constante, seuls les *node_constant* sont autorisés dans les *new_node*.

Pour les simplifications arithmétiques, d'autres attributs s'ajoutent à ceux existant sur les *node_constant* :
- *id*, identifiant de la forme s1..s*n*,

– *eval*, une expression à évaluer (les références aux entrées sont incluses dans la (ou les) expression(s) arithmétique(s)).

4.2.2 1ère phase : Chargement des règles en mémoire

Le fichier X.M.L. est interprété par un parseur DOM qui stocke les règles une par une au sein d'une structure de données adaptée : la classe RuleSet. Cette classe sert à regrouper des instances de la classe Rule, cette dernière servant à stocker une règle. Le parseur crée puis stocke les règles une par une dans une instance de la classe RuleSet fourni au préalable. Ce sera cette instance de classe RuleSet regroupant toutes les règles qui sera utilisée par la suite pour transformer le graphe.

Création d'une règle

Un règle est structurée d'une façon assez semblable à sa description dans le fichier X.M.L.. Il existe deux types de règles à stocker : celles décrivant les remplacement de motifs et celles décrivant les simplification arithmétiques. Tout d'abord, le motif servant à reconnaître est décrit d'après deux critères distincts commun aux deux règles :

1. le noeud Opérateur qui est le premier critère utilisé afin de reconnaître un motif. L'élément essentiel dans une règle est le noeud Opérateur car c'est le premier critère servant à reconnaître un motif compatible.
2. les noeuds prédécesseurs du noeud opérateur qui permettent d'affiner la reconnaissance du motif. En effet, cela permet de spécifier des règles différentes en fonction des propriétés des "inputs" du noeud Opérateur.
3. la commutativité qui explicite le fait que la règle s'applique aussi aux motifs dont les "input" ont été commutés. Il a été convenu avec le client que la commutativité se limiterait seulement aux noeuds possédant deux "inputs".

Ensuite, les caractéristiques des deux types de règles divergent. Pour les remplacements de motifs, il faut stocker :

– les conditions à vérifier sur les "inputs". Ces condition sont fournies sous forme d'un vecteur de noeuds "predicats". Ces noeuds sont semblables aux noeuds stockant des fonctions sauf qu'ici le nom de la fonction est un nom de predicat.
– le motif de remplacement appelé "output_pattern", ce motif est fourni sous la forme d'une instance de la classe Pattern servant à stocker des motifs.

Pour les simplifications de motifs, il faut stocker :

– l'expressions ou les expressions à fournir à JEP pour la simplification.

Stockage

Les règles sont stockées au sein d'une instance de la classe RuleSet qui a été fournie au préalable au parser. La classe RuleSet encapsule un vecteur de règles et des méthodes d'ajout et de recherche pour les instances de la classe Rule qui y sont stockées.
A l'origine, il était prévu de stocker les règles dans une table de hâchage. Cependant, nous avons eu un problème technique. En effet, la table de hâchage que nous utilisions était celle fournie par Java. Or, quand on veut extraire les valeurs associées à une clef, on ne récupère qu'un seul élément et non la file de collision. Cela nous a amené à reconsidérer les raisons de l'utilisation d'une telle structure. Après réflexion, nous avons réalisé que très peu de règles étaient stockées au sein de RuleSet, environ une cinquantaine en moyenne. De ce fait, une table de hâchage ne se justifiait pas. En effet, une table de hâchage est efficace pour stocker un grand nombre d'éléments : mille et plus. Pour de petites structures de données, la recherche linéaire est ce qu'il y a de plus efficace. Nous avons alors décidé de remplacer la table de hâchage par un vecteur.

4.2.3 2ème phase : Application des transformations sur le graphe

Cette partie explique comment est conçue la deuxième phase du module d'optimisation de graphe. Les données fournies sont :

- le graphe sur lequel doivent s'appliquer les transformations,
- l'ensemble des règles de transformation stockées sous forme de *RuleSet*. La structure RuleSet est issue de la première phase qui a créé les règles à partir d'un fichier XML et qui les a stockées au sein de cette instance de *RuleSet*.

Cette phase a pour but de rechercher des occurrences de motifs particuliers à travers le graphe pour ensuite les remplacer par un autre motif plus ou moins complexe. Ces motifs particuliers sont constitués d'un noeud opérateur ainsi que de ses noeuds parents et enfants. En pratique, on parcourt tous les motifs un par un. Pour chaque motif parcouru, on cherche si on peut lui appliquer une règle de transformation et, si c'est le cas, on lui applique la règle. On peut ainsi découper cette phase d'optimisation en deux étapes principales constituées de sous-tâches :

1. chercher les motifs du graphe
   (a) parcourir le graphe
   (b) extraire un motif
2. appliquer les optimisations aux motifs
   (a) rechercher une règle applicable sur le motif examiné.
   (b) créer le motif de remplacement selon la règle trouvée.
   (c) remplacer dans le graphe original l'ancien motif par le nouveau motif calculé.

Recherche des motifs

Choix du parcours de graphe

Il a d'abord fallu expliciter les caractéristiques du graphe fourni en entrée afin de définir les parcours envisageables.
Le graphe est :

- orienté,
- bipartie si l'on considère les opérateurs/fonctions et les variables,
- sans cycle si on ne tient pas compte des noeuds Z-1 qui sont des noeuds spéciaux.

Sachant que l'élément central dans une optimisation est le noeud symbolisant la fonction ou l'opérateur et qu'aucun cycle n'est présent dans le graphe du point de vue des noeuds opérateur/fonction, le parcours naturel semble être un parcours sur les noeuds opérateurs/fonctions.
Ensuite, grâce à cette absence de cycle, nous pouvons envisager de faire simplement un parcours en profondeur ou un parcours en largeur. Il n'y a pas de parcours plus efficace qu'un autre car cela va dépendre de la configuration du graphe. Le parcours en largeur a donc été choisi arbitrairement.

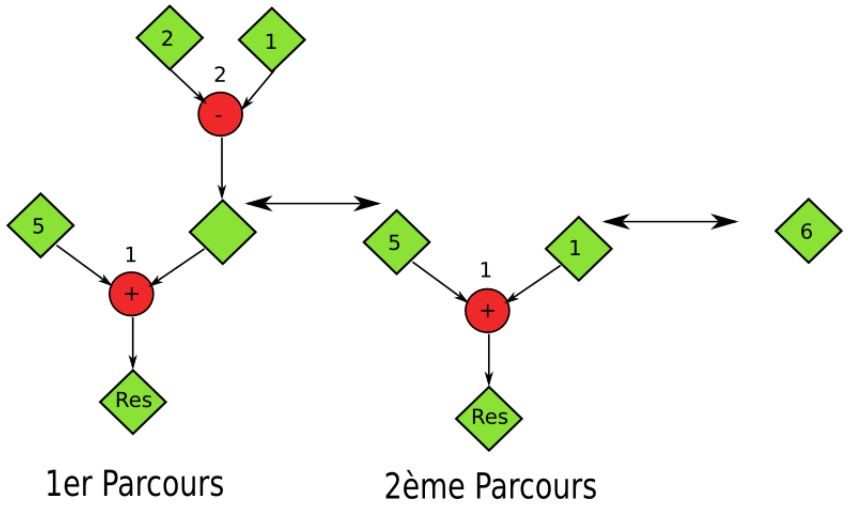


Fig. 4.5 – parcours en largeur

Un seul parcours en largeur ne suffit pas à effectuer toutes les optimisations. Il est possible qu'une optimisation apporte une autre optimisation dans son motif de remplacement ou rende possible de nouvelles optimisations en modifiant les noeuds parents d'un opérateur déjà traité.
Pour remédier au premier problème, si une optimisation a été effectuée lors d'un parcours, alors les noeuds du nouveau motif sont ajoutés à la file des des noeuds à traiter. Pour remédier au second problème, il suffit de préciser que, lorsqu'un noeud subit une optimisation, alors tous les opérateurs directement successeurs doivent être ajoutés à la file des noeuds à traiter. Cela permet de descendre les cascades d'optimisations potentielles tant que des optimisations sont possibles. En effet, ces noeuds opérateur directement descendants peuvent potentiellement être optimisés (cf figure 4.5).
Pour qu'un noeud ne soit pas traité plusieurs fois inutilement, il suffit de maintenir à jour une liste des noeuds déjà traités. Dans le processus normal de parcours du graphe, i.e. quand aucune optimisation n'est effectuée, un noeud opérateur fils est ajouté à la file des noeuds à traiter seulement s'il n'appartient pas encore à la liste des noeuds déjà parcourus.

Bouclage infini

Un nouveau problème se pose : si des optimisations sont effectuées tant qu'il reste des optimisations possibles, on prend le risque de créer des boucles infinies. En effet, il suffit par exemple que le motif de remplacement issu d'une optimisation puisse se transformer avec cette même optimisation (cf figure 4.6).

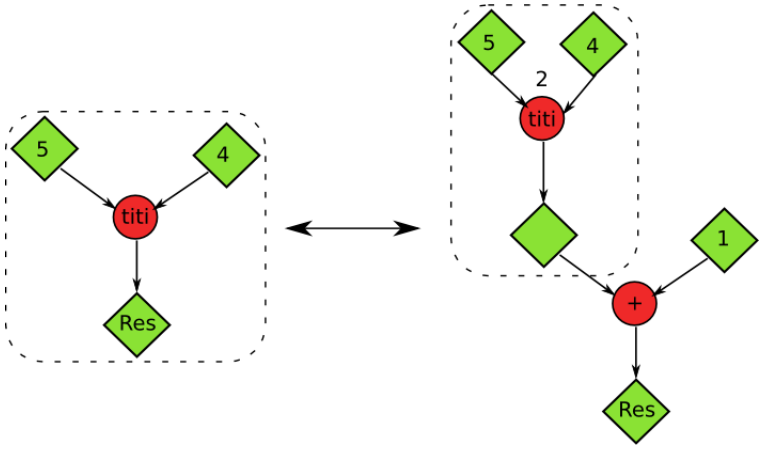


Fig. 4.6 – Cette Optimisation génère une boucle infinie car son motif de remplacement contient son propre motif à optimiser. Cette règle s'expand à l'infinie.

Ce cas a été discuté avec le client et il a été convenu qu'aucun mécanisme détectant les boucles infinies ne serait mis en place. En effet, sa mise en place serait un handicap en terme de ressource sur le projet. De plus, peu de personnes utilisent le logiciel à l'exception de son concepteur, monsieur Le Gal, et ces personnes sont des utilisateurs experts. Par conséquent, il est laissé à l'utilisateur la responsabilité de ne pas écrire de règles d'optimisation générant de telles boucles récursives.

Extraction de motifs

Il faut d'abord définir les caractéristiques utiles des motifs.
Pour identifier un motif, il est nécessaire de connaître :

- le noeud symbolisant un opérateur ou une fonction, que nous appellerons par abus de langage "noeud opérateur". Ce noeud est central car les règles d'optimisation s'effectuent autour d'un opérateur ou d'une fonction. C'est donc le premier critère de reconnaissance d'un motif.
- l'ensemble des noeuds parents du noeud opérateur qui permettent d'identifier des cas particuliers d'utilisation de cet opérateur ou fonction. Il s'agit du deuxième critère d'identification.

Les noeuds enfants du noeud opérateur ne sont pas pas pris en compte lors de la recherche de motif. En effet, la seule information que l'on peut en extraire est le nombre de résultats renvoyés par une fonction ou un opérateur. Or ce nombre est intrinsèque à l'opérateur et ne dépend pas de la manière dont ce dernier est utilisé. Donc leur connaissance n'apporte rien pour la recherche et l'identification d'optimisations. Cependant, ces noeuds enfants servent pour relier le nouveau motif au graphe. Il faut donc aussi les stocker mais pas pour la même raison.

Par conséquent, pour extraire un motif, il suffit de récupérer un noeud opérateur ainsi que ses noeuds parents et ses noeuds fils.

Application des optimisations aux motifs

L'application des optimisations aux motifs du graphe se fait en trois étapes successives :

1. rechercher une règle applicable sur le motif examiné,
2. créer le motif de remplacement selon la règle trouvée,
3. remplacer dans le graphe original l'ancien motif par le nouveau motif calculé.

Recherche d'une règle applicable sur le motif examiné

Il faut trouver une règle applicable sur un motif donné. On ne cherche pas à trouver toutes les règles applicables, une seule suffit. En effet, même si on obtenait toutes les règles, il faudrait en choisir une à appliquer. Comme il n'y a aucune raison pour qu'une règle soit plus prioritaire qu'une autre, il est plus simple de choisir la première que l'on trouve. De ce fait, le choix de la règle est dépendant de l'architecture du programme. En effet, les règles sont stockées dans l'ordre dans lequel elles sont lues dans le fichier X.M.L.. En pratique, ce sera la première règle adéquate du fichier X.M.L., quand on le lit du début vers la fin, qui sera choisie.

Il existe deux critères pour choisir une règle :
- l'opérateur. Ce critère est très simple à vérifier car les opérateurs sont les éléments centraux des règles et des motifs. Leur obtention et leur comparaison sont très simples et rapides.
- l'identification des noeuds parents qui correspondent aux arguments pour les fonctions ainsi qu'aux opérandes pour les opérateurs. Celle-ci est plus complexe car elle s'effectue sur deux critères distincts : le type et la valeur. Il existe deux types de noeuds variables dans le graphe : les noeuds variables symbolisant des constantes et les noeuds variables symbolisant des résultats de calculs ou des arguments de la fonction parsée. Si le noeud est de type constante, la règle peut s'appliquer quelle que soit la valeur ou uniquement sur une valeur précise. Par conséquent, dans certains cas, il est nécessaire de vérifier la valeur du noeud constante.

Ainsi, une règle est d'abord choisie en fonction du premier critère et ensuite en fonction du second. En effet, si l'opérateur sur lequel s'applique une règle ne correspond pas à celui du motif examiné, cela ne sert à rien d'effectuer une vérification du deuxième critère qui serait plus coûteuse.

Application du motif de remplacement selon la règle trouvée

Le motif de remplacement est tout d'abord calculé puis stocké au sein de la règle sous une forme standard. Il existe deux types de règles évaluées de façons distinctes :
- les règles ne nécessitant pas d'évaluation préalable ne faisant que du remplacement de motif. Ce sont les règles décrites dans la première partie du fichier X.M.L.. Il s'agit de règles possédant un motif dont les noeuds sont décrits explicitement dans le fichier X.M.L..
- les règles nécessitant une évaluation qui sont en fait des simplification arithmétiques. Ces règles sont décrites dans la deuxième partie du fichier X.M.L.. Le motif de remplacement est un noeud constante dont la valeur est calculée selon une expression arithmétique respectant la syntaxe de Jep.

Le fait de stocker le nouveau motif de façon standard au sein de la règle a pour avantage de permettre de remplacer l'ancien motif du graphe par le nouveau motif sans tenir compte du type de la règle.

Création du motif sans évaluation

Il est supposé ici que les conditions sur les noeuds parents appelés "input nodes" sont respectées.
La création d'un tel motif se déroule en deux étapes :

1. Tout d'abord, le motif de remplacement tel qu'il est défini dans la règle est cloné, c'est-à-dire que le motif est reproduit avec de nouveaux noeuds ayant les mêmes propriétés et de nouvelles arêtes ayant les mêmes propriétés et reproduisant la même structure que dans l'ancien motif. C'est ce nouveau motif qui sera transformé afin d'obtenir le motif final qui remplacera l'ancien motif.
2. Le remplacement des noeuds faisant référence à des noeuds du graphe. Certaines de ces références sont remplacées par des clones des noeuds originaux du graphe. Il existe trois types de références : les "idrefs" qui apparaissent explicitement dans le fichier X.M.L. et les "temprefs" qui correspondent aux noeuds fils de l'opérateur du nouveau motif qui sera connecté au graphe seront remplacés par des noeuds du graphe. Le nombre des noeuds "temprefs" est spécifié dans le fichier X.M.L. au niveau de l'attribut nb_out. Les "predrefs" sont des noeuds constante qui vont prendre la valeur d'une valeur auxiliaire calculée lors de l'évaluation d'un prédicat.

Un effet de bord du clonage des noeuds du graphe est que les arêtes reliant ces noeuds au reste du graphe sont aussi clonées. De ce fait, les noeuds clonés sont aussi connectés au graphe. Le nouveau motif est donc déjà partiellement connecté au graphe.

Création du motif avec évaluation

Le calcul du nouveau motif se fait en trois étapes :

- L'évaluation par Jep des expressions fournies par le fichier X.M.L. afin de calculer le nouveau motif. Au préalable, il faut transformer ces expressions en expressions évaluables par Jep. Au début, les expressions sont de la forme : "ceil(e1 * 2ê2)". Les identifiants e1 et e2 désignent des noeuds constantes possédant une valeur qui dépend du motif sur lequel est appliqué la règle. Il faut donc à chaque application de la règle remplacer les identifiants par les valeurs des noeuds associées afin de pouvoir fournir à Jep une expression qu'il sera en mesure d'évaluer. Ensuite, Jep évalue les expressions transformées et renvoie les valeurs associées sous forme de chaînes de caractère.
- Il faut ensuite vérifier si la valeur est entière ou non et transformer les chaînes de caractères renvoyées si besoin est. En effet, Jep renvoie la valeur en format flottant. Par exemple, la valeur 1 sera renvoyée en tant que "1.0". Le client a demandé que, quand la valeur est un entier, elle apparaisse comme un entier dans le graphe. Cependant, il ne faut pas pour autant tronquer toutes les valeurs, seulement les entiers doivent l'être.
- Enfin, il faut créer le nouveau motif. Pour ce faire, un noeud de type constante est créé pour stocker chacun des résultats. Ensuite, ces noeuds sont ajoutés au nouveau motif.

Ici, il n'y a aucun effet de bord. Le nouveau motif devra être entièrement connecté au graphe.

Remplacement dans le graphe original de l'ancien motif par le nouveau motif calculé

Le remplacement d'un motif se fait en trois étapes :

1. La liaison des noeuds du motif reliant le motif et le reste du graphe. Ces noeuds particuliers sont de deux types : les "input nodes" et les "output nodes". Les premiers sont les noeuds faisant le lien entre le graphe et le motif. Les deuxième sont ceux faisant le lien entre le motif et le graphe.

2. La déconnexion de l'ancien motif du graphe. Pour ce faire, le motif est entièrement détruit en supprimant toutes les arêtes de l'ancien motif.

3. La mise à jour des listes du graphe triant les différents noeuds.

Lors de la liaison des "inputs nodes", il faut vérifier le type des noeuds et leur origine. En effet, seuls les noeuds de type constante créés soit par le parser, soit lors de la création d'un motif évalué, doivent être reliés. De plus, ces noeuds doivent être reliés à la source du graphe. Les autres "input nodes" sont déjà reliés au graphe. Cela provient de l'effet de bord précisé plus haut au moment du remplacement des noeuds faisant référence à des noeuds du graphe lors de la création d'un motif sans évaluation.(cf figure 4.7)

De même, lors de la liaison des "output nodes", les noeuds ayant remplacé les noeuds "tempref" sont déjà reliés au reste du graphe. Ils ne doivent donc pas être reliés une nouvelle fois. Cependant, il existe un cas où les noeuds doivent être connectés au graphe. Il s'agit du cas où il n'y a pas d'opérateur dans la partie du nouveau motif générant cet "output node".(cf figure 4.7)

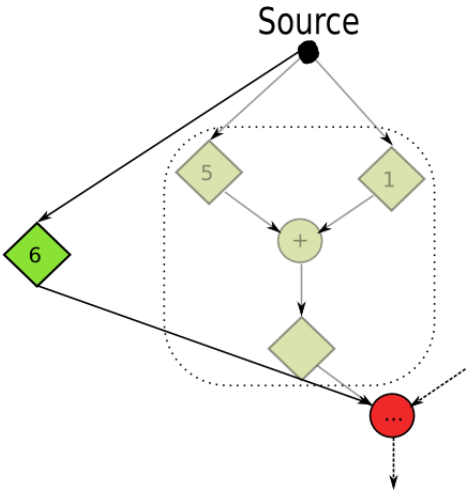


Fig. 4.7 – Remplacement du motif “5 + 1” par le motif “6”

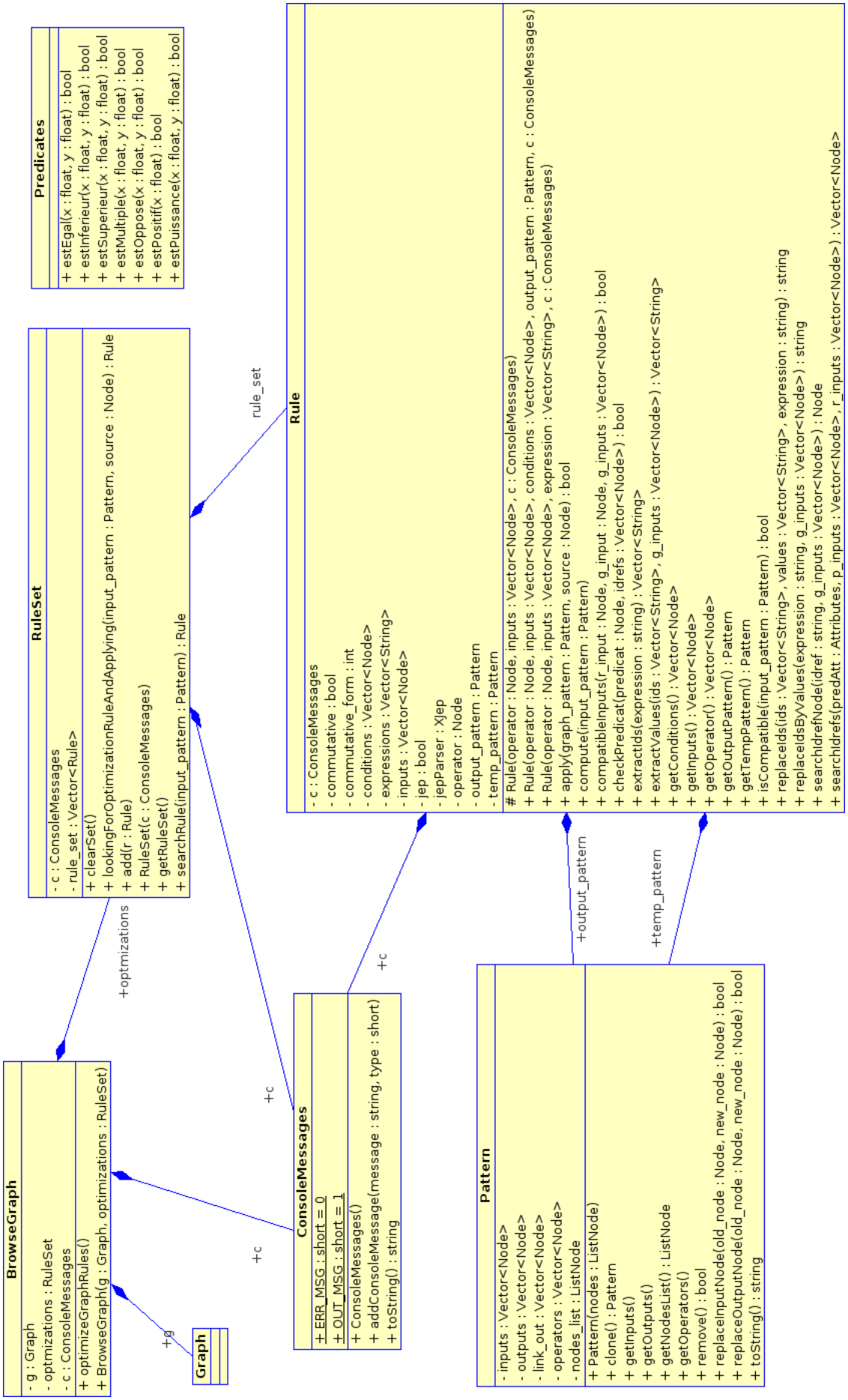


Fig. 4.8 – Diagramme de Classe

4.2.4 Implantation

Diagramme de Classe (cf figure 4.8)

Description des classes :

OptimizeGraph
Cette classe constitue la "racine" du Plugin. En effet, c'est à travers elle que le Plugin interagit avec le reste du logiciel.

ConsoleMessages
Cette classe a pour but de transmettre les messages d'erreur et d'information vers la console de GraphLab ainsi que vers le terminal.

XMLOptimizationFileParser
Il s'agit du parser de fichier X.M.L. utilisant SAX. Cette classe permet de remplir une structure RuleSet avec des instances de Rule représentant des règles de transformation pouvant être appliquées au graphe.

DTDValidation
Il s'agit d'une classe de gestion des erreurs de la D.T.D. dérivant de la classe ErrorHandler qui est la classe de gestion des erreurs de SAX.

BrowseGraph
Il s'agit de la classe exécutant l'algorithme général de parcours du graphe.

RuleSet
Cette classe sert à stocker les règles de transformation créées par la classe XMLOptimizationFileParser. Elle permet d'effectuer une recherche de règle compatible avec un motif fourni au préalable et de lui appliquer la transformation associée à la première règle trouvée.

Rule
Cette classe permet de stocker une instance de règle de transformation écrite dans le fichier XML parser par la classe XMLOptimizationFileParser. Elle permet de savoir si la règle en question peut s'appliquer à un motif fourni au préalable. Elle s'occupe aussi d'appliquer la règle au motif fourni.

Pattern
Cette classe permet de stocker un motif du graphe. Elle rend aussi possible sa suppression du graphe. Elle possède également deux méthodes spécifiques qui permettent de remplacer un "output node" ou un "input node" du motif par un autre noeud.

Predicates
Cette classe contient les fonctions d'évaluation des prédicats cités dans le fichier XML.

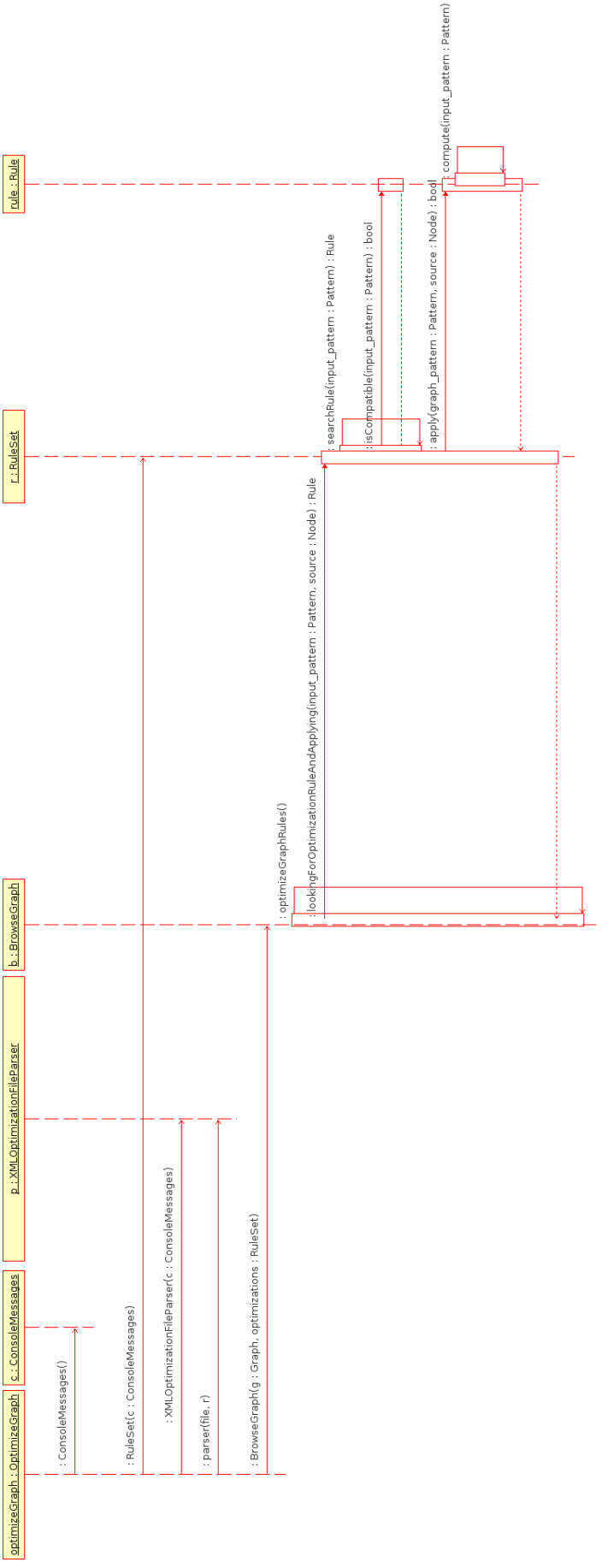


Fig. 4.9 – Diagramme de Séquence

Diagramme de Séquence

OptimizeGraph commence par instancier la classe *ConsoleMessages* pour pouvoir, par la suite, fournir cette instance aux autre instances. Ensuite, une instance de *RuleSet* est créée ainsi qu'une instance de *XMLOptimizationFileParser*. L'instance de RuleSet ainsi que le fichier X.M.L. sont fournis à l'instance du parser au travers de la méthode parser(). Par la suite, le graphe ainsi que l'instance de RuleSet qui a été remplie avec les règles de transformation contenues dans le fichier X.M.L. sont passés lors de l'instance de la classe *BrowseGraph* lors de sa création.
Après cette phase de préparation, l'instance de *BrowseGraph* exécute l'algorithme de parcours du graphe fourni grâce à la méthode *optimizeGraphRules()*. Pour chaque motif, elle fait appel à la méthode *lookingForOptimizationRuleAndApplying()* de l'instance de RuleSet fournie et lui donne en arguments le motif et la source du graphe. La méthode *lookingForOptimizationRuleAndApplying()* fait appel à une autre méthode de l'instance de RuleSet, *searchRule()*, avec pour argument le motif. Ensuite, cette méthode cherche une instance de Rule s'appliquant sur un motif possédant le même opérateur que le motif fourni. Si une règle est trouvée alors on fait appel à la méthode de l'instance de Rule, *isCompatible()* qui vérifie si le reste du motif est compatible. Si ce n'est pas le cas, alors *searchRule()* continue à chercher une instance de Rule compatible. Dès qu'une règle compatible est trouvée, elle est appliquée au travers de la méthode *apply()* de l'instance de Rule représentant la règle. Cette méthode commence par faire appel à la méthode *compute()* de l'instance de Rule pour calculer le nouveau motif. Une fois que ce motif est calculé, la méthode *apply* remplace l'ancien motif du graphe par le nouveau motif qui vient d'être calculé.
Enfin, une fois que le motif est appliqué, on revient à l'algorithme principal, exécuté par la méthode *optimizeGraphRules()* de l'instance de BrowseGraph, pour mettre à jour l'instance de graphe et continuer l'exécution de l'algorithme.

4.2.5 Tests

Voici une liste de tests avec à droite le graphe avant optimisation et à gauche le graphe après optimisation :

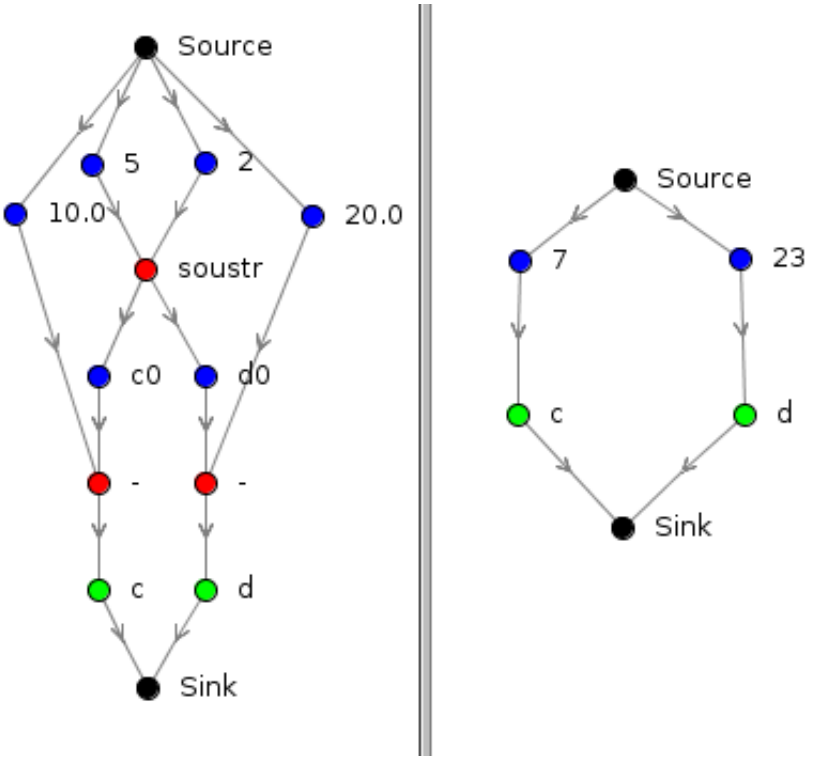


Fig. 4.10 – motif possédant 2 sorties

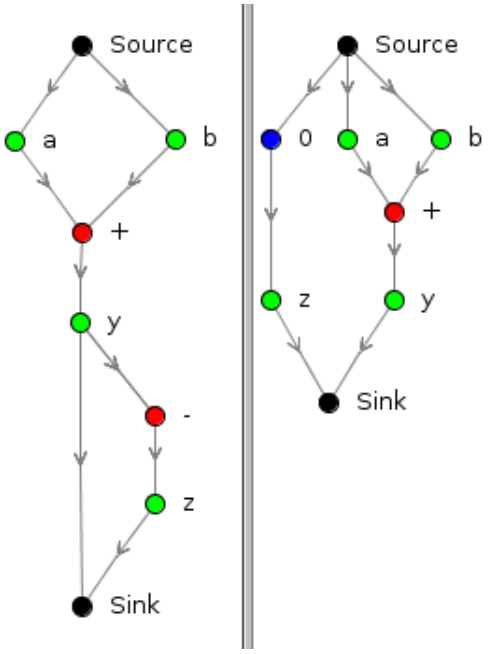


Fig. 4.11 – optimisation ne devant pas supprimer un noeud de type output

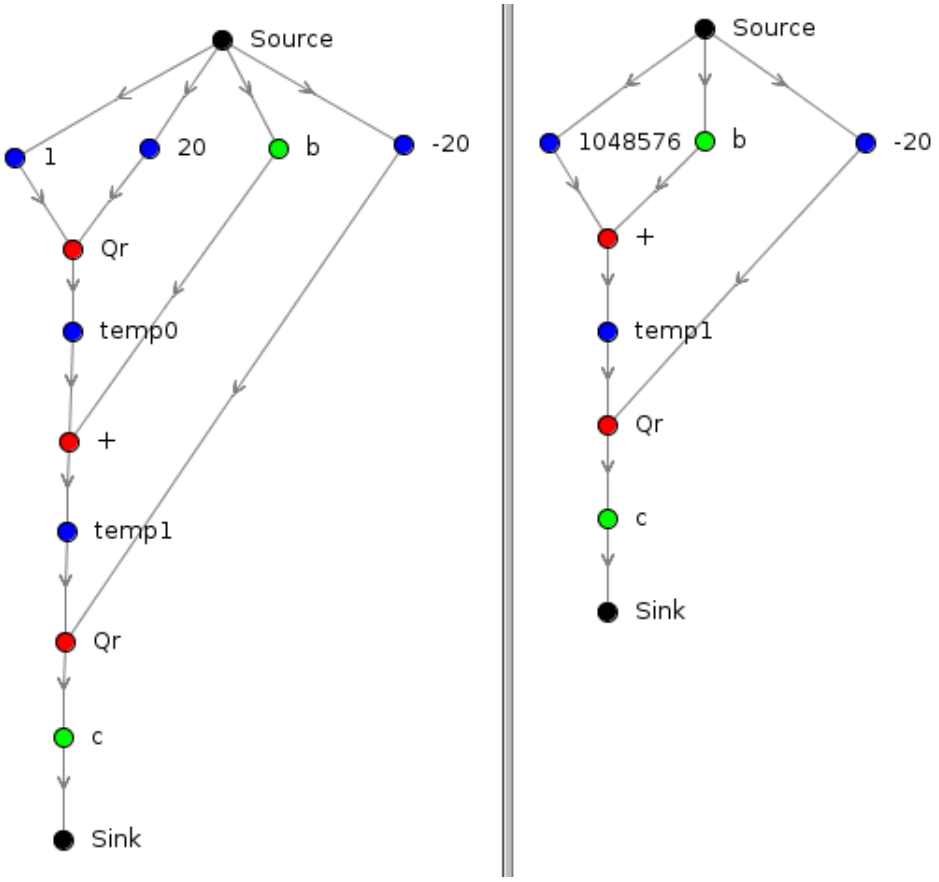


Fig. 4.12 – optimisation avec un noeud Qn

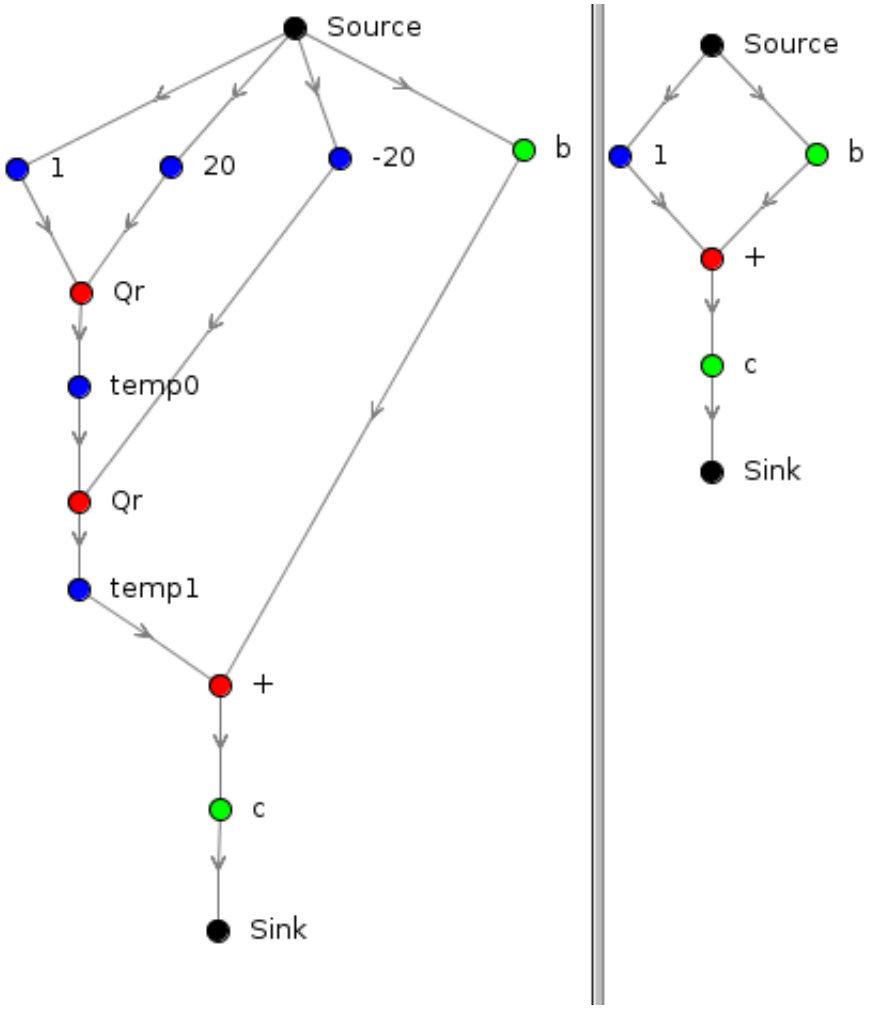


Fig. 4.13 – optimisation avec deux noeuds Qn se suivant

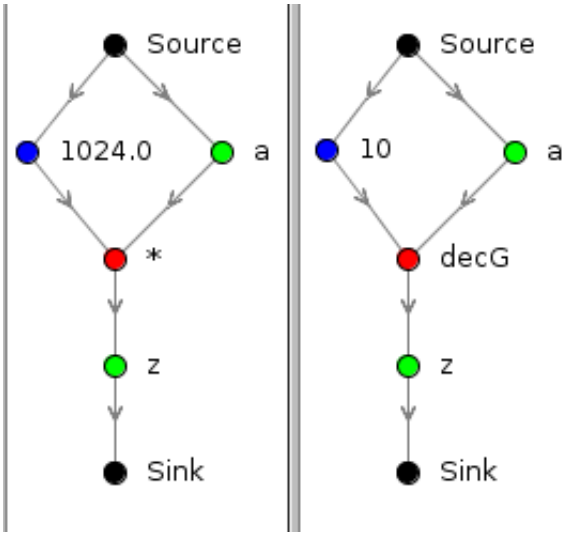


Fig. 4.14 – test avec un shift vers la gauche

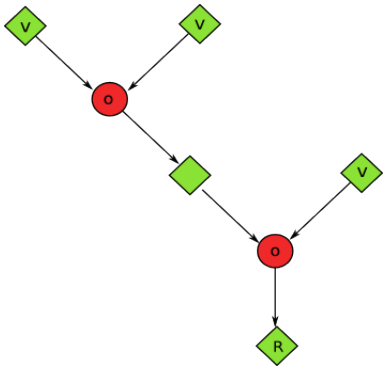


Fig. 4.15 – Motif en forme de peigne de hauteur 2 orienté à gauche

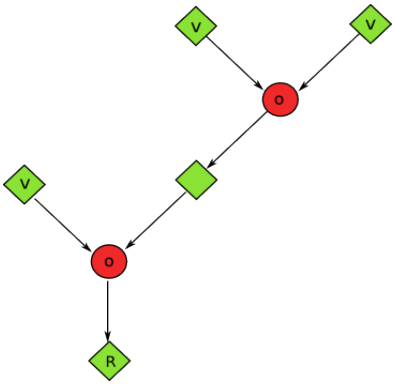


Fig. 4.16 – Motif en forme de peigne de hauteur 2 orienté à droite

4.3 Parallélisation et linéarisation de motifs détectés dans un graphe

4.3.1 Fichier de description des règles de parallélisation

Ce fichier a été créé pour décrire des règles de parallélisation qui pourront être appliquées sur un graphe obtenu ou non après analyse d'un code MatLab.

Tout comme le fichier décrivant les règles d'optimisation et de remplacement de motif, ce fichier est écrit en X.M.L..

Nous imposons les motifs apparaissant dans les règles de ce fichier. En effet, nous limitons la reconnaissance aux motifs qui ont une hauteur maximale de 3, dans un souci de limiter le temps d'exécution de l'algorithme de parcours et de reconnaissance d'un motif dans le graphe. Ainsi, seuls les motifs en forme de peigne de hauteur 2 ou 3 pourront être parallélisés.

Un peigne de hauteur 3 ou 2 peut avoir deux formes différentes : en ce qui concerne le cas d'un peigne de hauteur 2, le fichier 4.15 montre la forme orientée à gauche, et la figure 4.16 montre la forme orientée à droite (*o* désignant un opérateur et *v* désignant une variable).

Ces deux formes sont équivalentes si l'opérateur est commutatif. En effet, si nous prenons par exemple l'expression arithmétique **y=c*(a+b)**, celle-ci est équivalente à **y=(a+b)*c** car l'opérateur $*$ est commutatif. Et il en est de même si nous raisonnons avec un peigne de hauteur 3.

Nous n'avons pu, faute de temps, spécifier la commutativité d'un opérateur lors de la description du fichier contenant les définitions de règles de parallélisation. C'est la raison pour laquelle nous avons choisi de décrire séparément les règles qui consistent à paralléliser un motif 2 ou 3 orienté à gauche et les règles qui consistent à paralléliser un motif 2 ou 3 orienté à droite. Au niveau fonctionnel, cela revient au même que

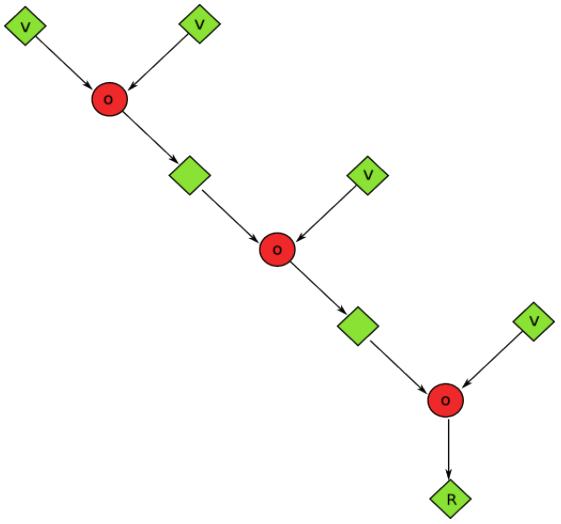


Fig. 4.17 – Motif en forme de peigne de hauteur 3

si nous avions décrit une seule règle avec une possibilité de préciser la commutativité, la seule différence est que, si un opérateur est commutatif, il faudra écrire une règle pour le motif 2 (respectivement 3) de gauche et une règle pour le motif 2 (respectivement 3) de droite.

La figure 4.17 représente l'allure d'un motif en forme de peigne de hauteur 3. La hauteur 3 indique qu'il s'agit d'un motif en forme de peigne possédant 3 opérateurs. Sur cette figure, *v* désigne une variable d'entrée, *t* une variable temporaire et *o* un opérateur.

Description de la règle de parallélisation d'un motif de hauteur 3

Etant donné que nous imposons une hauteur maximale de 3, nous imposons les motifs apparaissant dans les règles. Par souci de facilité de compréhension, nous ne faisons pas apparaître les variables temporaires dans les motifs présents pour aider l'utilisateur à décrire des règles de parallélisation. Ainsi, l'utilisateur de l'outil possède une information quant à l'application de la règle, pour l'aider à comprendre son but (cf. figure 4.18).

Lors de l'écriture de la règle, l'utilisateur devra décrire des informations concernant l'ancien motif et le nouveau motif. Par "ancien motif", nous entendons le motif à détecter, ici un motif *linéarisé*, et par "nouveau motif", nous entendons le motif résultant après application de la règle, à savoir un motif *parallélisé*.

Dans l'ancien motif, l'utilisateur devra spécifier les différents champs correspondant aux opérateurs *opr*1, *opr*2, *opr*3, c'est-à-dire que leur valeur devra être explicitement déclarée. Celle-ci doit être "+", "-", "*" ou "/" car nous ne prenons en compte que les opérateurs binaires usuels, conformément aux souhaits du client. En outre, l'utilisateur de l'outil ne devra pas modifier les identifiants correspondants aux variables car ceux-ci sont génériques et nous permettront de détecter dans le graphe un motif ayant pour variables des variables d'entrée aussi bien que des variables temporaires.

Par exemple, si nous considérons l'expression arithmétique **y=(((a+b)+c)+d)**, le graphe correspondant a l'allure d'un peigne de hauteur 3, comme nous pouvons le voir sur la figure 4.19. Lors de la parallélisation, cette expression arithmétique deviendra **y=(a+b)+(c+d)**. Le graphe correspondant à cette transformation est présenté en figure 4.19.

Pour définir une telle parallélisation, l'utilisateur devra décrire dans le fichier X.M.L. les informations suivantes :

- dans l'ancien motif :
  - le premier opérateur, noté *opr*1, vaut "+",
  - le deuxième opérateur, noté *opr*2, vaut "+",

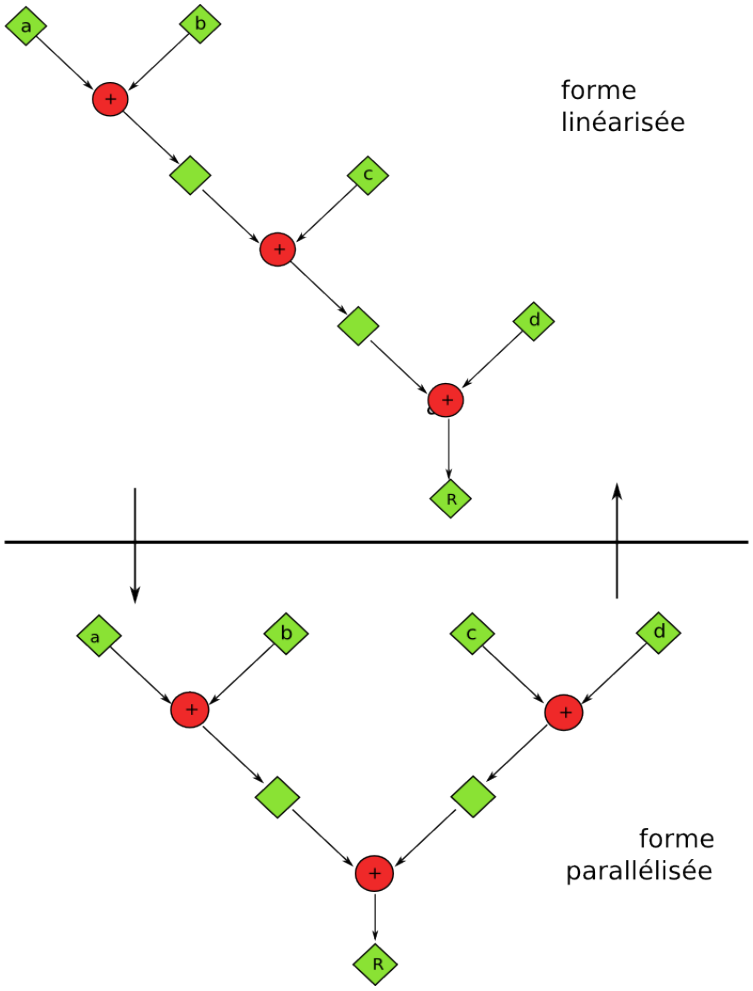


Fig. 4.18 – Exemple de motif linéarisé, en forme de peigne de hauteur 3, et sa forme parallélisée

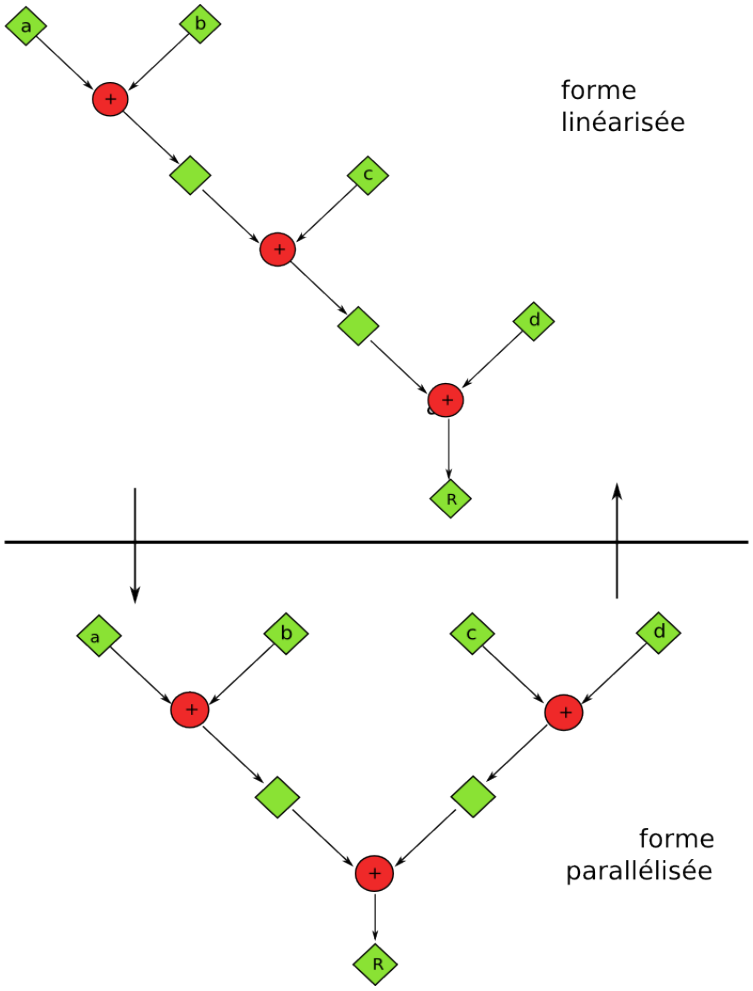


Fig. 4.19 – Motif en forme de peigne de hauteur 3 et sous forme parallélisée

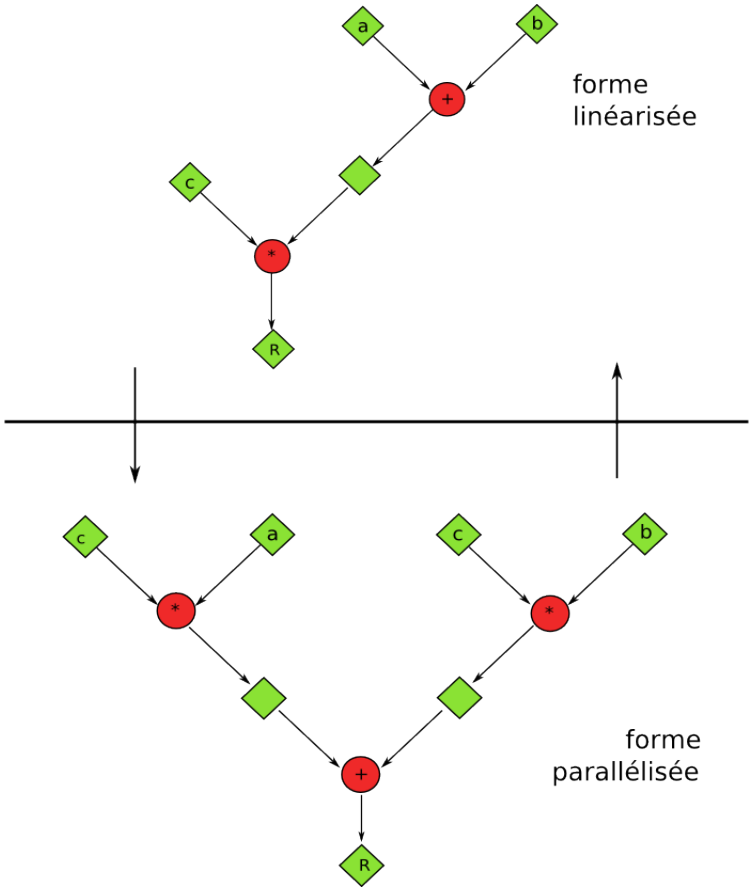


Fig. 4.20 – Motif en forme de peigne de hauteur 2 orienté à droite et sous forme parallélisée

  - le troisième opérateur, noté $opr3$, vaut "+",
- dans le nouveau motif :
  - le premier opérateur, noté $s1$, vaut "+",
  - le deuxième opérateur, noté s, vaut "+",
  - le troisième opérateur, noté s, vaut "+",
  - la première variable, notée $w1$, fait référence à la première variable de l'ancien motif, notée $v1$,
  - la deuxième variable, notée $w2$, fait référence à la deuxième variable de l'ancien motif, notée $v2$,
  - la troisième variable, notée $w3$, fait référence à la troisième variable de l'ancien motif, notée $v3$,
  - la quatrième variable, notée $w4$, fait référence à la quatrième variable de l'ancien motif, notée $v4$.

Description de la règle de parallélisation d'un motif de hauteur 2

Tout comme pour le motif en forme de peigne de hauteur 3, il est possible de spécifier une règle de parallélisation pour des motifs en forme de peigne de hauteur 2.

Nous avons conservé le même principe de description de la règle que celui utilisé pour la parallélisation de motif de hauteur 3.

Si nous prenons l'exemple d'un motif linéarisé de hauteur 2 orienté à droite, représenté par l'expression arithmétique **y=c*(a+b)**, nous obtenons la règle d'optimisation visible sur la figure 4.20 qui conduit à un motif parallélisé qui représente l'expression arithmétique **y=(c*a)+(c*b)**.

De même, si nous prenons l'exemple d'un motif linéarisé de hauteur 2 orienté à gauche, représenté par l'expression arithmétique **y=(a+b)*c**, nous obtenons la règle d'optimisation visible sur la figure 4.21 qui conduit à un motif parallélisé qui représente l'expression arithmétique **y=(a*c)+(b*c)**.

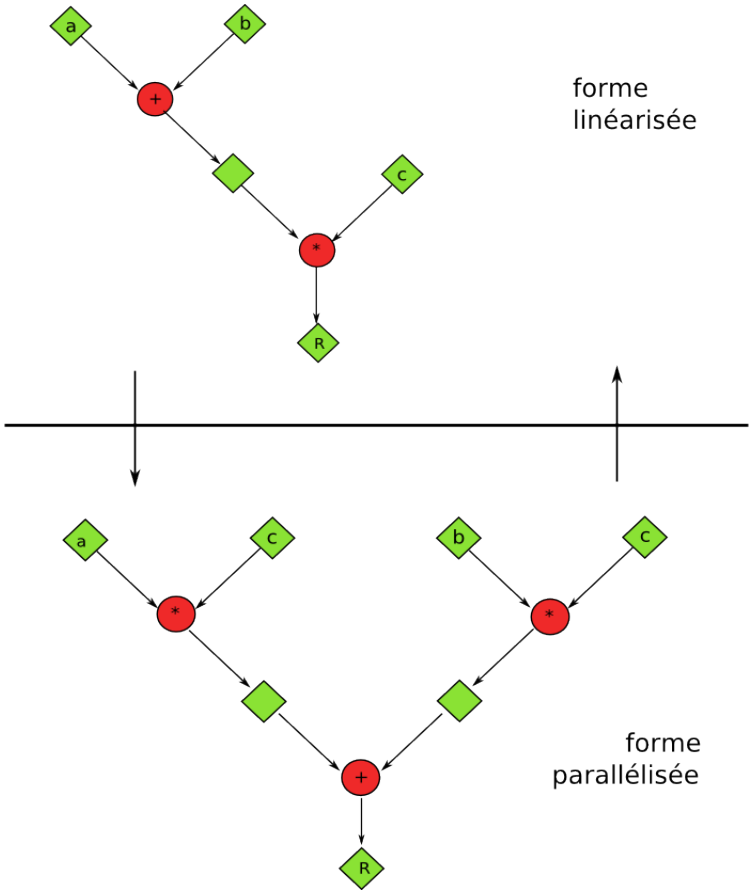


Fig. 4.21 – Motif en forme de peigne de hauteur 2 orienté à gauche et sous forme parallélisée

4.3.2 Fichier de description des règles de linéarisation

Ce fichier a été créé pour décrire des règles de linéarisation qui pourront être appliquées sur un graphe obtenu ou non après analyse d'un code MatLab.

Tout comme le fichier décrivant les règles de parallélisation, ce fichier est écrit en X.M.L..

Nous imposons les motifs apparaissant dans les règles de ce fichier. En effet, nous limitons la reconnaissance aux motifs parallélisés, c'est-à-dire bien équilibrés, qui ont trois opérateurs. Ainsi, le motif linéarisé obtenu aura une hauteur maximale de 3. Nous avons fait ce choix pour les mêmes raisons que celles évoquées précédemment.

Lors de l'écriture de la règle, le fichier des linéarisations se complète de la même façon que le fichier des parallélisations. En effet, dans l'ancien motif, l'utilisateur devra également spécifier les différents champs suivants :

- dans l'ancien motif :
  - le premier opérateur, noté $opr1$, vaut "+",
  - le deuxième opérateur, noté $opr2$, vaut "+",
  - le troisième opérateur, noté $opr3$, vaut "+",
- dans le nouveau motif :
  - le premier opérateur, noté $s1$, vaut "+",
  - le deuxième opérateur, noté s, vaut "+",
  - le troisième opérateur, noté s, vaut "+",
  - la première variable, notée $w1$, fait référence à la première variable de l'ancien motif, notée $v1$,
  - la deuxième variable, notée $w2$, fait référence à la deuxième variable de l'ancien motif, notée $v2$,
  - la troisième variable, notée $w3$, fait référence à la troisième variable de l'ancien motif, notée $v3$,
  - la quatrième variable, notée $w4$, fait référence à la quatrième variable de l'ancien motif, notée $v4$.

4.3.3 Description de la règle de linéarisation d'un motif de hauteur 3

Si nous considérons l'expression arithmétique **y=((a+b)+(c+d))**, le graphe correspondant a l'allure d'un arbre bien équilibré, comme nous pouvons le voir sur la figure 4.22. Lors de la linéarisation, cette expression arithmétique deviendra **y=(((a+b)+c)+d)**.

Le graphe correspondant à cette transformation est présenté en figure 4.22.

Il est possible de générer un peigne orienté à gauche ou bien à droite, selon l'écriture des règles.

4.3.4 Description de la règle de linéarisation d'un motif de hauteur 2

Si nous considérons l'expression arithmétique **y=((c*a)+(c*b))**, le graphe correspondant a l'allure d'un arbre bien équilibré, comme nous pouvons le voir sur la figure 4.23. Lors de la linéarisation, cette expression arithmétique deviendra **y=c*(a+b)**.

Le graphe correspondant à cette transformation est présenté en figure 4.23.

Les règles de production d'un motif linéarisé orienté à gauche ou à droite se font selon les propriétés du motif parallélisé. En effet, nous pouvons voir que si les variables $v1$ et $v3$ sont identiques dans le motif parallélisé, nous obtiendrons un motif linéarisé orienté à droite, et si ce sont les variables $v2$ et $v4$ du motif d'entrée qui sont égales, nous obtiendrons un motif linéarisé orienté à gauche.

4.3.5 Vérification de la D.T.D. et analyse du fichier

Les fichiers de parallélisation et de linéarisation décrit en langage X.M.L. possèdent une D.T.D. qui leur est intrinsèquement associée. Celle-ci décrit la structure de chacun des noeuds de l'arbre D.O.M. associé au fichier X.M.L..

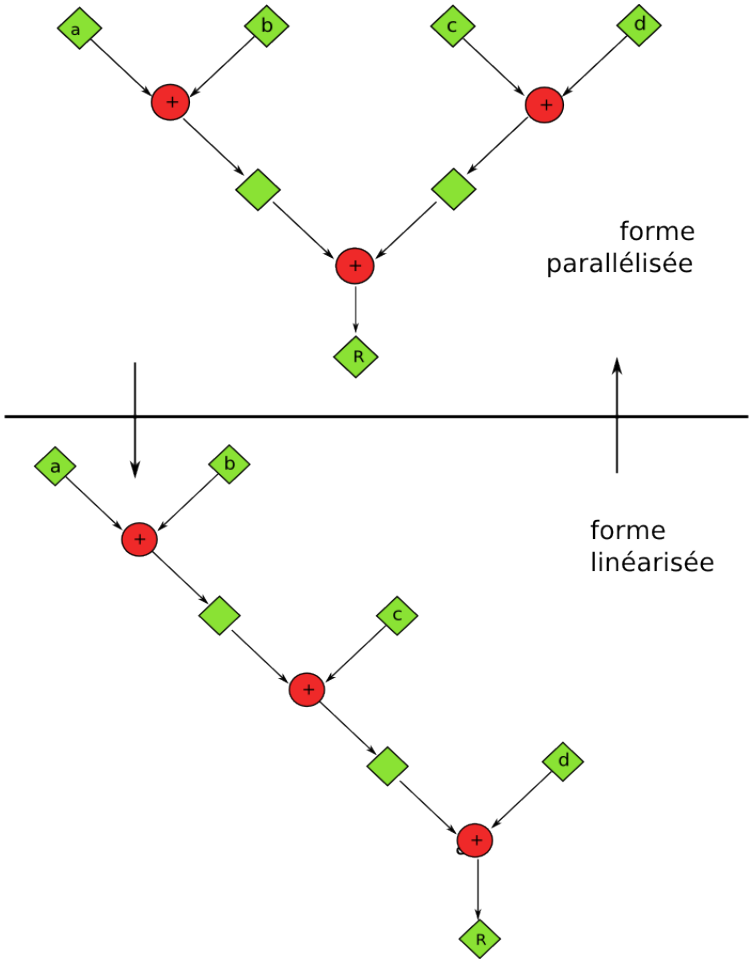


Fig. 4.22 – Motif parallélisé et sous forme linéarisée de hauteur 3 orienté à gauche

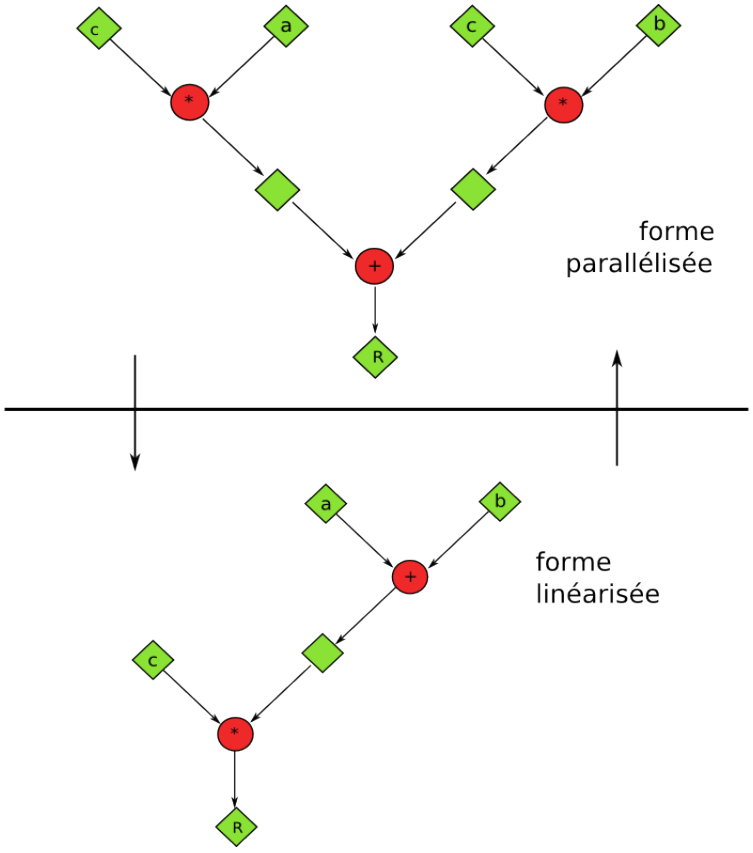


Fig. 4.23 – Motif parallélisé et sous forme linéarisée de hauteur 2, orienté à droite

Le client a souhaité que les fichiers de description des règles de parallélisation et d'optimisation soient simples à renseigner et simples à concevoir. Ce choix a entraîné le fait que toutes les règles ne pouvaient pas être vérifiées uniquement par la D.T.D. du fichier X.M.L. correspondante, à cause de la syntaxe trop légère du fichier.

La D.T.D. vérifie donc un certain nombre de règles concernant la syntaxe du fichier X.M.L. :
- chaque noeud nommé *parallelization_motif3* doit avoir un unique fils *old_pattern* et un unique fils *new_pattern*,
- les noeuds *old_pattern* et *new_pattern* doivent posséder des noeuds *operator* et des noeuds *variable*,
- les noeuds *operator* doivent posséder un attribut *name* ainsi qu'un attribut *value* et les noeuds *variable* doivent posséder un attribut *id*,
- le noeud *variable* fils du noeud *new_pattern* doit posséder en plus un attribut *idref*.

Cette vérification permet de vérifier que l'arbre D.O.M. correspondant au fichier X.M.L. est *bien formé*.

Pour compléter la vérification suite à la D.T.D., nous avons créé un parser de fichier X.M.L. pour vérifier les autres propriétés suivantes que ne pouvait pas prendre en compte la D.T.D. :
- les noeuds *old_pattern* fils de *parallelization_motif3* contiennent exactement 3 noeuds *operator* et 4 noeuds *variable*,
- les noeuds *new_pattern* fils de *parallelization_motif3* contiennent exactement 3 noeuds *operator* et 4 noeuds *variable*,
- les noeuds *old_pattern* fils de *parallelization_motif2gauche* contiennent exactement 2 noeuds *operator* et 3 noeuds *variable*,
- les noeuds *new_pattern* fils de *parallelization_motif2gauche* contiennent exactement 3 noeuds *operator* et 4 noeuds *variable*,
- tout attribut *idref* d'un noeud *variable* fils d'un noeud *new_pattern* doit référencer un attribut *id* d'un noeud *variable* fils du noeud *old_pattern* correspondant à la même règle.

Le non-respect de la D.T.D. ou la non-validité des règles précédentes lors de l'analyse du fichier X.M.L. entraîne l'apparition de messages d'erreur à l'utilisateur.

4.3.6 Diagramme de classes de la partie "Parallélisation/Linéarisation"

Avant de commencer l'écriture des fichiers contenant les règles de parallélisation et de linéarisation, nous avons accordé une grande part à la conception. En effet, nous avons souhaité établir un modèle généraliste dans le but de faciliter la maintenabilité et l'évolutivité du code. Nous avons pu réaliser cela au moyen de mécanismes comme le polymorphisme, présents dans le langage Java, permettant la factorisation de code, ainsi qu'avec un raisonnement global lors d'une recherche d'algorithme. Par exemple, nous souhaitions écrire un algorithme de parcours du graphe qui soit valable pour la recherche de motifs parallélisés, mais aussi pour la recherche de motifs linéarisés, nous avons donc réfléchi à un algorithme assez général, et non spécifique à chacune des recherches.

La figure 4.24 représente le diagramme de classes de la partie "Parallélisation/Linéarisation".

Le client doit spécifier s'il souhaite effectuer des parallélisations ou des linéarisations sur le graphe. En effet, nous ne permettons pas à l'utilisateur de paralléliser et linéariser des motifs lors d'une même exécution car l'application de règles qui se "complètent", c'est-à-dire qui s'annulent, entraînerait un couple *détection et remplacement de motif* infini. Ainsi, l'utilisateur doit préciser un champ lors du lancement de la méthode de parallélisation/linéarisation pour choisir le type d'optimisation à effectuer, ainsi qu'un champ correspondant au nom du fichier contenant les règles correspondantes (*parallelization_file* ou *linearization_file*).

Par exemple, en supposant que le fichier de parallélisations se nomme *parallelisation.xml* et qu'on veuille paralléliser le graphe, il faudrait exécuter la commande suivante dans la console GraphLab : *parallelizeLinearize(ressources/parallelization_file.xml, p)*.

Lors de l'analyse d'un des deux fichiers, nous remplissons une *PatternRuleSet*, c'est-à-dire un ensemble de règles, avec chacune des règles, ou *PatternRule* rencontrée dans le fichier considéré. Nous avons choisi d'implanter une classe générique, *PatternRule*, qui décrit une règle de manière générale, et dont les autres classes décrivant chacun des différents types de règle héritent. Ce choix a été fait d'une part, pour permettre

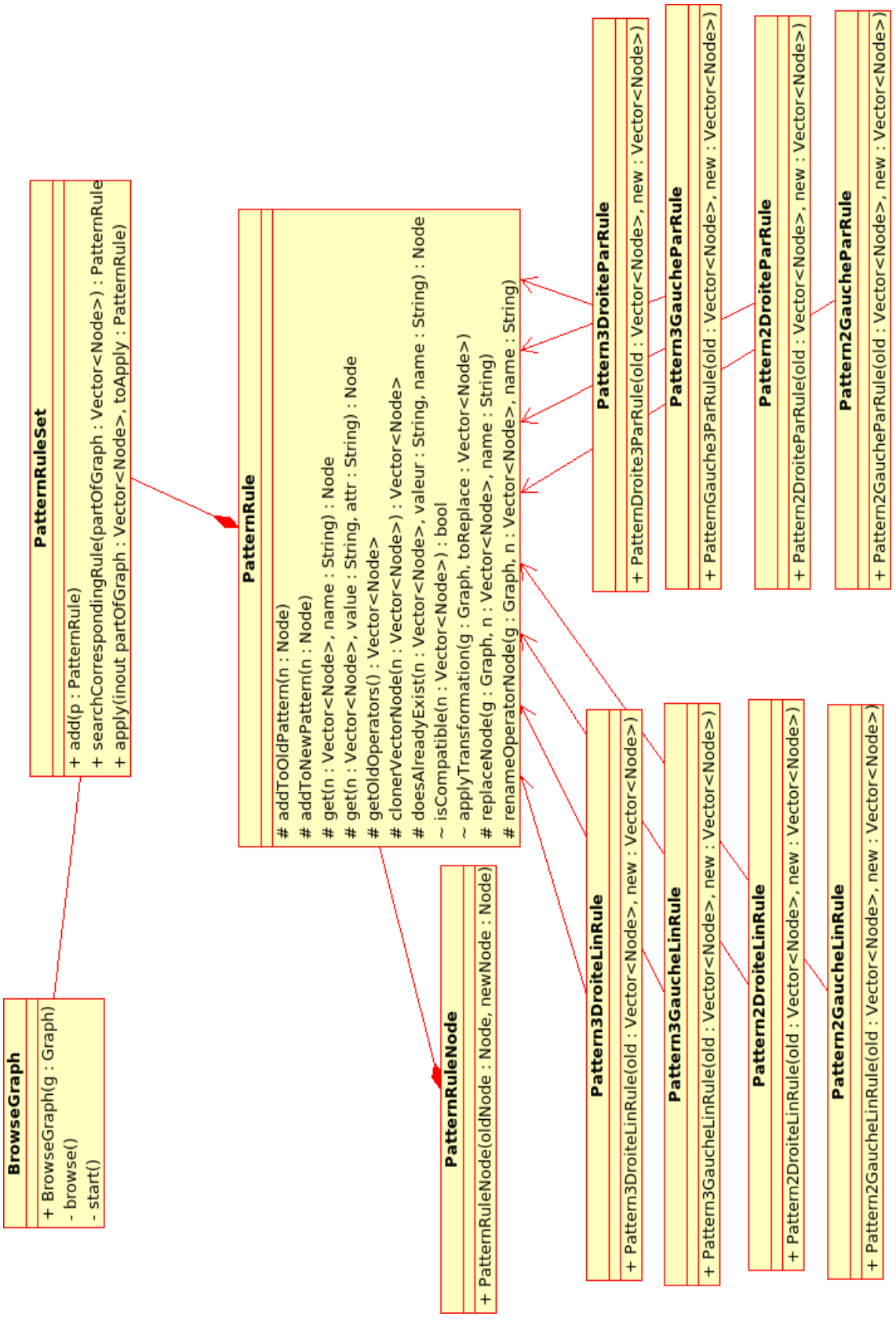


Fig. 4.24 – Diagramme de classes pour la partie “Parallélisation/Linéarisation”

le partage de code commun à ces classes, et, d'autre part, pour permettre à l'utilisateur (au sens mainteneur du logiciel) futur de rajouter des classes définissant d'autres règles, et ce, sans perturber l'architecture pré-existante.

Nous avons donc créé une classe pour chaque type de règle et d'optimisation rencontrée :
- au niveau de la parallélisation :
  1. *Pattern3GaucheParRule* : règle de parallélisation d'un peigne d'une hauteur 3 orienté à gauche,
  2. *Pattern3DroiteParRule* : règle de parallélisation d'un peigne d'une hauteur 3 orienté à droite,
  3. *Pattern2GaucheParRule* : règle de parallélisation d'un peigne d'une hauteur 2 orienté à gauche,
  4. *Pattern2DroiteParRule* : règle de parallélisation d'un peigne d'une hauteur 2 orienté à droite,
- au niveau de la linéarisation :
  1. *Pattern3GaucheLinRule* : règle de linéarisation d'un peigne d'une hauteur 3 orienté à gauche,
  2. *Pattern3DroiteLinRule* : règle de linéarisation d'un peigne d'une hauteur 3 orienté à droite,
  3. *Pattern2GaucheLinRule* : règle de linéarisation d'un peigne d'une hauteur 2 orienté à gauche,
  4. *Pattern2DroiteLinRule* : règle de linéarisation d'un peigne d'une hauteur 2 orienté à droite,

La classe *BrowseGraph* contient l'algorithme de parcours du graphe. Lors de ce parcours, l'ensemble des règles de parallélisation (ou de linéarisation) est stocké dans une *PatternRuleSet*. Ainsi, lorsque nous parcourons le graphe à la recherche de motif, nous considérons une règle de remplacement de motif, et cherchons dans le graphe un motif qui pourrait correspondre à cette règle. La classe *PatternRuleNode* est utile dans ce cas car elle permet d'établir un lien entre les noeuds d'une règle de parallélisation (ou de linéarisation) et les noeuds d'un motif détecté dans le graphe.

4.3.7 Implantation

Comme pour la partie remplacements de motifs et simplifications arithmétiques (cf. section 4.2.2), nous avons dû développer un plugin.

Ainsi, le plugin se situe dans le répertoire *PlugIngs/ParallelizationLinearization/src*. Il utilise les codes présents dans le package *ParallelizationLinearizationPackage*, c'est-à-dire dans le répertoire *PlugIngs/ParallelizationLinearization/src/ParallelizationLinearizationPackage/*.

De plus, un analyseur lexical a été développé pour les fichiers de description. Cet analyseur utilise un *ErrorHandler SAX* pour la vérification de la D.T.D. et pour les autres vérifications qui ne peuvent pas être effectuées via une D.T.D., nous avons implanté des méthodes travaillant sur un arbre D.O.M..

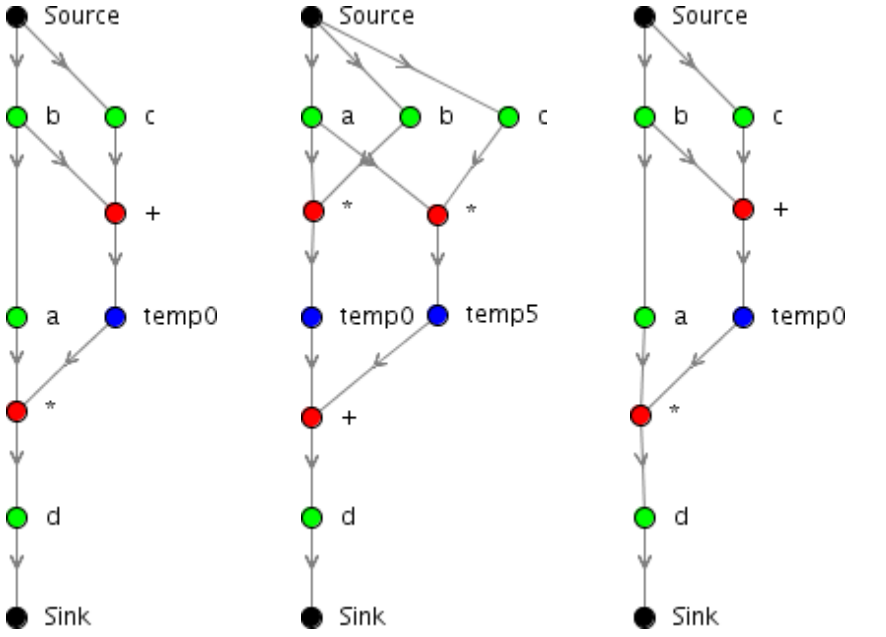
D'autres méthodes s'occupent, après vérification du fichier X.M.L., de remplir le *PatternRuleSet* (cf. 4.24). Ce dernier est un ensemble de règles d'optimisations sur le graphe.

L'implantation suit le diagramme de classe présenté en section **??**. La manière dont nous avons pensé l'architecture logicielle du plugin permet au mainteneur d'ajouter, éventuellement, d'autres types de parallélisation / linéarisation en implantant une classe héritant de *PatternRule* et en modifiant le parser pour qu'il la prenne en compte.

Le principe de l'algorithme implanté dans le fichier *BrowseGraph.java* est assez simple. Pour chaque règle R de l'ensemble de règles de linéarisation/parallélisation, nous cherchons dans le graphe un motif sur lequel on peut appliquer R, et cela, tant que nous pouvons appliquer R. Lorsque R est appliquée, dans la console de GraphLab, un message du type *The optimisation class ParallelizationLinearizationPackage.Pattern2DroiteParRule has been applied* est affiché.

4.3.8 Tests (exemples)

Dans cette section, nous nous proposons de montrer un ensemble non-exhaustif de règles applicables sur un graphe.



(a) Graphe d'origine (b) Après parallélisation de 4.25(a) (c) Après linéarisation de 4.25(b)

Fig. 4.25 – Parallélisation/linéarisation de hauteur 2 à droite

En figure 4.25 est présentée une parallélisation de hauteur 2 à droite du code Matlab (représentée par la figure 4.25(a)) suivant :

```
function [d] = f(a, b, c)
  d = a * (b + c);
```

Vérification (mathématique, certes) : $a*(b+c) = (a*b)+(a*c)$.

En figure 4.26 est présentée une parallélisation de hauteur 2 à gauche du code Matlab (représentée par la figure 4.26(a)) suivant :

```
function [d] = f(a, b, c)
  d = (b + c) * a;
```

Vérification (mathématique, certes) : $(b+c)*a = (b*a)+(c*a)$.

En figure 4.27 est présentée une parallélisation de hauteur 3 à gauche du code Matlab (représentée par la figure 4.27(a)) suivant :

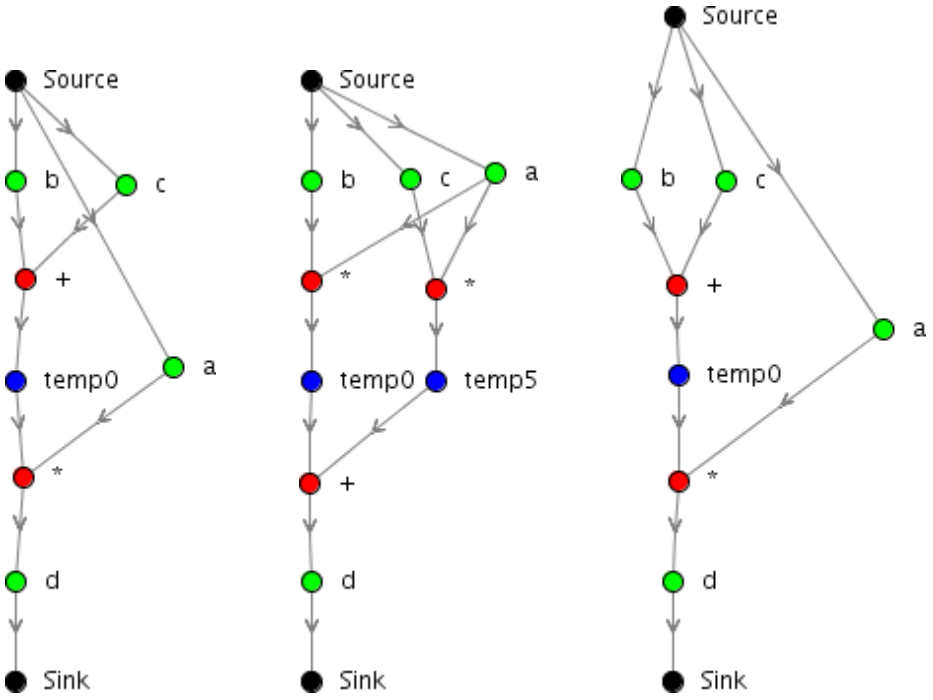
```
function [e] = f(a, b, c, d)
  e = d + (c + (a + b));
```

Vérification (mathématique, certes) : $d+(c+(a+b)) = (d+c)+(a+b)$.

En figure 4.28 est présentée une linéarisation de hauteur 3 à droite du code Matlab (représentée par la figure 4.27(a)) suivant :

```
function [g] = f(a, b, c, d)
  e = a + b;
  f = c + d;
  g = e + f;
```

Vérification (mathématique, certes) : $(a+b)+(c+d) = a+b+c+d$.



(a) Graphe d'origine (b) Après parallélisation de 4.26(a) (c) Après linéarisation de 4.26(b)

Fig. 4.26 – Parallélisation/linéarisation de hauteur 2 à gauche

4.4 Analyse entre nos engagements et le produit final

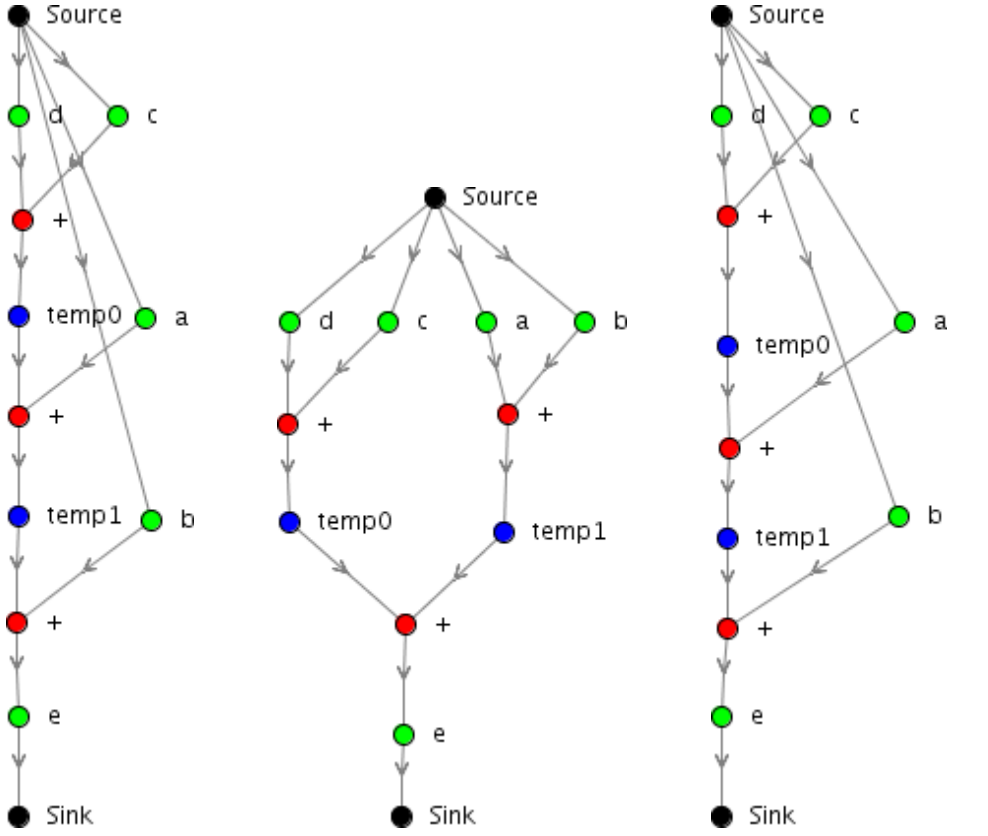
4.4.1 Concernant la partie Optimisation

Nous avons très largement sous-estimé l'ampleur de la tâche. En effet, ce qui était censé se faire en un peu plus d'une semaine s'est finalement prolongé sur quasiment toute la durée du pfa. Plusieurs éléments ont été mal estimés :

- la conception du langage de spécification des règles de transformation a pris beaucoup plus de temps que prévu. En effet, durant cette phase, nous nous sommes aperçus que mettre en place un tel langage qui soit relativement simple et compréhensible mais qui puisse contenir suffisamment d'informations était une tâche plutôt complexe et longue nécessitant de nombreuses réunions avec le client. Nous avions choisi que l'utilisateur modifierait directement le fichier X.M.L. pour gagner du temps. Au final, cela aurait peut-être été plus simple et plus rapide de mettre en place une interface car nous n'aurions pas eu de telles contraintes de simplicité de l'architecture du fichier X.M.L..
- Ensuite, sa mise en place, son débogage et les tests ont pris au final trois semaines. Cette partie a aussi été sous-estimée. Nous pensions que nous pourrions séparer le code en trois parties plus ou moins égales, ce qui aurait permis de maximiser la rentabilité. Au final, le coeur du Plugin n'a pas pu vraiment vraiment séparé en parties équitables.

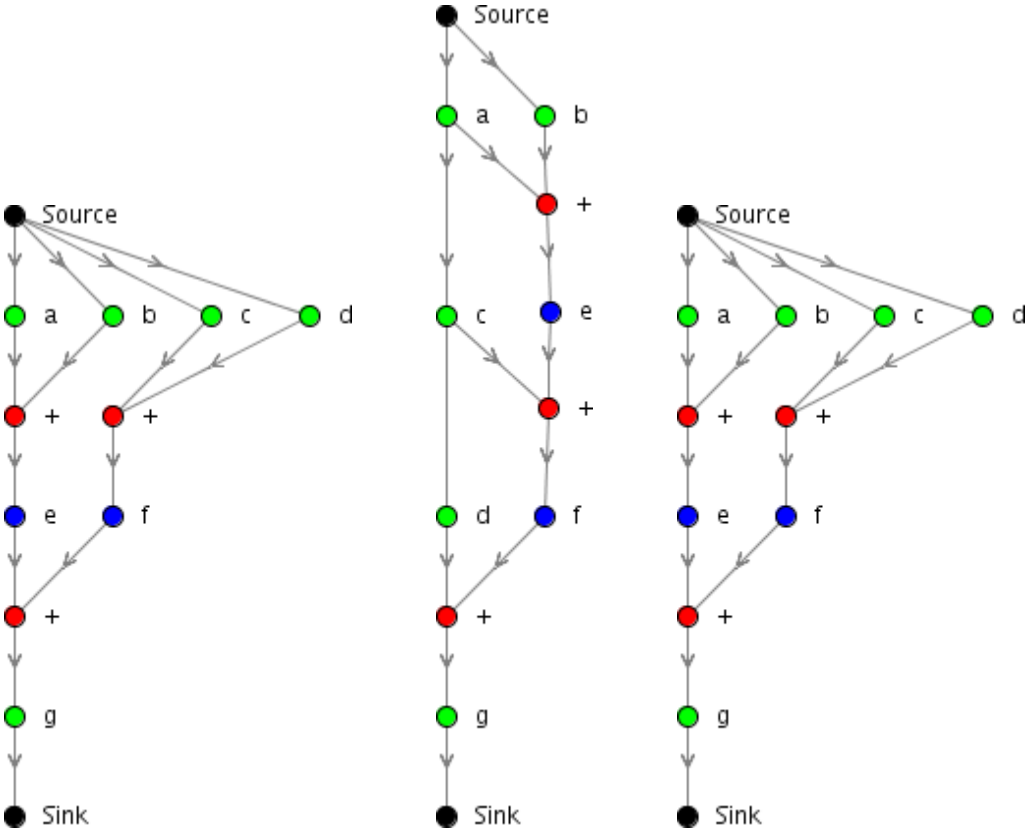
En pratique, nous avons dû scinder le trinôme en deux parties. Un membre est resté pour finir le code et faire le débogage pendant que les deux autres commençaient le travail sur le Plugin de Séquentialisation/Parallélisation. Ce qui était au début provisoire s'est au final transformé en définitif. En effet, comme le code et le débogage ont pris beaucoup de temps, le membre n'a pas pu rejoindre les deux autres membres à temps pour pouvoir les aider efficacement.

Pour conclure, ce qui devait normalement être une simple "mise en jambe" pour le plugin de Séquentialisation/Parallélisation s'est transformé en ce qui aurait pu constituer un projet à part entière.



(a) Graphe d'origine (b) Après parallélisation de 4.27(a) (c) Après linéarisation de 4.27(b)

Fig. 4.27 – Parallélisation/linéarisation de hauteur 3 à gauche



(a) Graphe d'origine

(b) Après linéarisation de 4.28(a)

(c) Après parallélisation de 4.28(b)

Fig. 4.28 – Linéarisation/parallélisation de hauteur 3 à gauche

4.4.2 Concernant la partie Parallélisation/Linéarisation

Le travail que nous avons effectué concernant la détection de motifs en forme de peigne de hauteur maximale 3 à paralléliser, ainsi que la détection de motifs à linéariser en peigne de hauteur maximale 3, est, selon les tests que nous avons effectués, conforme aux attentes du client.

Un point reste cependant à soulever : lors de l'écriture du cahier des charges, nous avions plutôt parlé de "Factorisation et Développement" de motifs dans un graphe. Nous avions alors interprété cela au sens mathématique du terme, c'est-à-dire que nous considérions la factorisation et le développement de motifs correspondant à des expressions arithmétiques. Nous avions donc commencé à réfléchir sur la partie conception et avions commencé à développer, suite à des idées d'algorithme, quelques classes, notamment celle concernant le parcours et la reconnaissance de motifs *factorisés* ou *développés* dans un graphe.

Après avoir montré notre début de travail au client, ce dernier nous a en fait expliqué que par "factorisation" il entendait en fait "parallélisation", et, de même, par "développement" il entendait "linéarisation", car l'interprétation de ces notions se faisait au sens électronique du terme. Le cahier des charges n'ayant pas été assez explicite à ce sujet, nous nous sommes donc conformés aux souhaits du client, et avons repris notre réflexion au point initial sur cette partie, à savoir la réflexion sur une manière de concevoir ces nouvelles données. Nous avons donc tenu compte de ces changements dès que nous en avons pris connaissance, et nous sommes adaptés pour développer dans le temps restant les fonctionnalités que souhaitait le client.

Nous avions pris du retard suite à ce changement, mais nous avons réussi à y faire face. Cet première expérience d'un travail sur un réel projet soumis à des contraintes de temps, de délai et à des engagements de production (notamment par le biais du cahier des charges) nous a permis de mieux prendre conscience des notions qui avaient été évoquées lors du cours de Génie Logiciel, mais également qui avaient été évoquées par notre encadrant technique. En effet, ce dernier nous avait précisé lors de la rédaction du cahier des charges, que toutes les notions devaient être clarifiées, et qu'il fallait faire en sorte que le client explicite davantage ses souhaits. Les changements qui ont eu lieu et que nous avons réalisés à cause du flou sur ce point dans le cahier des charges, nous ont fait prendre conscience de la réalité d'un projet.

5 Conclusion

«««< .mine

La réalisation du projet a bien abouti dans l'ensemble. En effet, bien que l'estimation des tâches en rapport avec l'optimisation des graphes ait été mal considérée, les plugins associés, à savoir les plugins d'optimisation et de Parallélisation/linéarisation, ont été entièrement implantés. Il a été parfois question de reconsidérer quelques structures du code, comme par exemple modifier la table de hâchage existante en un vecteur, permettant ainsi de tirer profit au maximum de la linéarité. Le scindement du trinôme au cours du projet a démontré finalement son efficacité vis-à-vis de la qualité des résultats. Par ailleurs, et bien que le parser tel qu'il a été récupéré présentait d'une part un code non maintenable et, d'autre part, des choix architecturaux non évolutifs et de nombreux défauts au niveau de la simplification des expressions arithmétiques, la refonte du code a conduit à l'aboutissement des grands traits des besoins fonctionnels exigés par notre client. Au-delà des quelques limitations qu'impose **Jep**, il serait profitable de développer par la suite quelques fonctionnalités telles que la gestion des tableaux multidimensionnels et la gestion des erreurs.

Enfin, le projet a permis à chacun de nous, d'une manière globale, de profiter d'une expérience sur un projet concret, encadré, formalisé, et ce, à plusieurs niveaux, aussi bien technique, en l'occurrence en matière de programmation orientée objet et en compilation, qu'organisationnel ou relationnel. =======

La réalisation du projet a bien abouti dans l'ensemble. En effet, bien que l'estimation des tâches en rapport avec l'optimisation des graphes ait été mal considérée, les plugins associés, à savoir les plugins d'optimisation et de Parallélisation/linéarisation, ont été entièrement implantés. Il a été parfois question de reconsidérer quelques structures du code, comme par exemple modifier la table de hâchage existante en un vecteur, permettant ainsi de tirer profit au maximum de la linéarité. Le scindement du trinôme au cours du projet a démontré finalement son efficacité vis-à-vis de la qualité des résultats. Par ailleurs, et bien que le parser tel qu'il a été récupéré présentait d'une part un code non maintenable et, d'autre part, des choix architecturaux non évolutifs et de nombreux défauts au niveau de la simplification des expressions arithmétiques, la refonte du code a conduit à l'aboutissement des grands traits des besoins fonctionnels exigés par notre client. Au-delà des quelques limitations qu'impose **Jep**, il serait profitable de développer par la suite quelques fonctionnalités telles que la gestion des tableaux multidimensionnels et la gestion des erreurs.

Enfin, le projet a permis à chacun de nous, d'une manière globale, de profiter d'une expérience sur un projet concret, encadré, formalisé, et ce, à plusieurs niveaux, aussi bien technique, en l'occurrence en matière de programmation orientée objet et en compilation, qu'organisationnel ou relationnel.

»»»> .r487

Glossaire

D.O.M.

Le Document Object Model (ou DOM) est une recommandation du W3C qui décrit une interface indépendante de tout langage de programmation et de toute plate-forme, permettant à des programmes informatiques et à des scripts d'accéder ou de mettre à jour le contenu, la structure ou le style de documents. Le document peut ensuite être traité et les résultats de ces traitements peuvent être réincorporés dans le document tel qu'il sera présenté.

D.T.D.

La Document Type Definition (D.T.D.), ou Définition de Type de Document, est un document permettant de décrire un modèle de document SGML ou X.M.L.. Une D.T.D. indique les noms des éléments pouvant apparaître et leur contenu, i.e. les sous-éléments et les attributs.

F.P.G.A., ou Field-Programmable Gate Array

Réseau de portes programmables in-situ.

Fichier MatLab

Les fichiers MatLab manipulés sont des suites d'opérations mathématiques, arithmétiques et logiques.

JEP, ou Java Expression Parser

JEP est une bibliothèque Java pour analyser et évaluer des expressions arithmétiques

M.O.E.

Maîtrise d'ouvrage

P.F.A.

Projet de Fin d'Année

Paquetage

Un paquetage est un conteneur logique permettant de regrouper et d'organiser les éléments dans le modèle U.M.L..

U.M.L.

U.M.L. (en anglais Unified Modeling Language, « langage de modélisation unifié ») est un langage graphique de modélisation des données et des traitements. C'est une formalisation très aboutie et non-propriétaire de la modélisation objet utilisée en génie logiciel.

V.H.D.L., ou Very high speed integrated circuit Hardware Description Language

Le V.H.D.L. est un langage de description matériel destiné à décrire le comportement et/ou l'architecture d'un système électronique numérique.

X.M.L.

XML (eXtensible Markup Language, "langage de balisage extensible") est un langage informatique de balisage générique. Le World Wide Web Consortium (W3C), promoteur de standards favorisant l'échange d'informations sur Internet, recommande la syntaxe X.M.L. pour exprimer des langages de balisages spécifiques.