Documentation de conception
Compilateur Deca

Projet Génie Logiciel, groupe 15

January 27, 2023

Contents

1 Architécture déjà présente

1.1 Compréhension du modèle présent

L'analyse contextuelle est une étape importante dans le processus de compilation d'un programme informatique. Elle a pour objectif de vérifier la validité des constructions syntaxiques d'un programme source et de déterminer leur signification en fonction du contexte dans lequel elles apparaissent. Cela permet de détecter des erreurs sémantiques dans le code source, telles que des références à des variables non déclarées ou des

types de données incompatibles.

L'analyse contextuelle se déroule après l'analyse lexicale et l'analyse syntaxique, qui ont respectivement pour but de décomposer le programme en mots et de vérifier la validité de la structure syntaxique. Elle utilise une représentation intermédiaire du programme, comme un arbre de syntaxe abstraite, pour faciliter sa tâche.

L'analyse contextuelle est divisée en deux sous-étapes : la vérification de la déclaration et l'analyse de la portée. La vérification de la déclaration consiste à vérifier que toutes les variables, fonctions et autres symboles utilisés dans le programme ont été correctement déclarés et définis. L'analyse de la portée, quant à elle, permet de déterminer la visibilité et la validité de ces symboles en fonction de leur emplacement dans le programme.

Une fois l'analyse contextuelle terminée, le compilateur peut générer du code machine ou du code intermédiaire pour l'exécution ultérieure du programme. C'est l'intégration de cette analyse dans le processus de compilation qui permet de détecter les erreurs sémantiques dans le code source et de garantir que le programme généré fonctionnera de manière correcte. Tout programme Deca bien typé doit être compilé en un programme assembleur dont le comportement à l'exécution respecte la sémantique du programme source. Réciproquement un programme Deca mal typé doit être rejeté par le compilateur avec un message d'erreurs appropriées.

La vérification contextuelle d'un programme Deca nécessite trois passages sur le programme. En effet, on peut, dans une déclaration de champ ou de paramètre d'une méthode, faire référence à une classe dont la déclaration apparaît après son utilisation. Pour cette raison, la première passe consiste à vérifier uniquement le nom des classes et la hiérarchie des classes. Lors de la deuxième passe, on vérifie les déclarations des champs et la signature des méthodes.

D'autre part, les méthodes peuvent être mutuellement récursives. Par conséquent, on vérifie le corps des méthodes au cours d'une troisième passe. Pour chaque passe, les vérifications à effectuer sont spécifiées formellement par une grammaire attribuée exprimée sur la syntaxe abstraite. Autrement dit, chacune de ces grammaires définit un ensemble d'arbres program (l'ensemble des arbres acceptées par cette grammaire attribuée) qui est un sous- ensemble de PROGRAMME.

Par définition, les programmes Deca bien typés sont les programmes dont l'arbre de syntaxe abstraite est acceptée successivement par chacune des grammaires attribuées de la syntaxe contextuelle.

Point d'Entrée:
Arbre Abstraite Non décorée.
Point de Sortie:
Arbre Décorée ou une erreur Contextuelle

Les identificateurs d'un programme Deca sont de différentes natures, On distingue les identificateurs de type et de classe d'une part. et les identificateurs de champ, de paramètre, de variable et de méthode d'autre part.

la conception des identificateurs de type:
les types du langage Deca comportent:
void, boolean, float, int et string. De plus à chaque classe A du programme correspond un type **typeClasse(A)**.
on a également le type "null" qui représente le type du littéral "Null".

Un identificateur de type est définie par son Symbole et un type Definition qui représente son type et sa Location. Donc chaque type représente une classe concret java qui hérite le champ Symbole de la classe Type et qui diffère aux autres par sa propre comportement qui sont les deux méthodes booléens : (boolean isSymbole()) qui est vrai si l'objet en question est un objet dont le Symbole est Symbole, et La fonction (boolean sameType(Type otherObj)) qui compare un objet avec l'objet de Symbole "Symbole".

Choix de la structure de données pour la vérification des types:
Pour faire la vérification de type en temps minimal: il est créé un Environnement de type une classe concret java qui représente un Dictionnaire (Map) qui associe le Symbole de chaque type à sa Type de définition. Donc on ajoute chaque type dans cette table puis à la vérification on utilise la méthode (CONTAINS) qui est O(1) pour les Maps cette Vérification permet de rejeter les erreurs de type:

```
{
    randomType <nom_variable>;
}
```

Et il suffit d'enrichir cet Environnement au cours de notre parcours à chaque fois qu'on rencontre une déclaration de classe qui n'est pas encore ajoutée dans celui-ci si le type de classe est déja présent dans l'environnement ou sa super classe n'est pas dans cet environnement on lève l'erreur de la Double déclaration de classe ou la non définition de sa super classe. tout est en O(1).

***la conception des identificateurs des expressions:**
-un champ est défini par son Symbole ,la classe dans la quelle est définie , sa visibilité et son index._ **-**une méthode est définie par son Symbole, son signature , son index.
-un paramètre est définie par son Symbole et son Type.
-une variable est définie par son Symbole et son Type.

il est crée une classe java concrét pour chaque identificateur de cette liste.

choix de la structure de données:

-à chaque classe on associe un Environnement d'Expression qui représente un Dictionnaire qui associe le Symbole de chaque identificateur définie dans cette classe à sa propre Expression de définition. par cette structure on peut accéder à chaque élément, vérifier son existence, et l'ajouter en temps O(1). ces opérations permettent de vérifier la Double déclaration d'un identificateur dans cette classe et la vérification de son existence lors de son utilisation dans une expression.

Pour accéder aux identificateurs de la Super classe il est ajouté dans cette Environnement un pointeur vers l'environnement Parent (Environnement des identificateurs de la Super classe).

-Une Signature est une ArrayList des types de paramètres d'une méthode donc l'accès à un élément de cette Liste est en O(1) en temps. On aura besoin de cette accès pour comparer la signature de deux méthodes par exemples au moment de la redéfinition d'une méthode.

Toutes ces classes et structures de données sont définies dans le package "contex".

-Conception pris pour parcourir l'arbre abstraite non décorée:

***à** chaque non terminal de la grammaire Deca est associé une classe Abstraite:
Exemple: "AbstractProgram", "AbstractMain", ..etc
***à** chaque terminal de la grammaire Deca est associé une classe concrètes:
Exemple: Programme, Main, ..etc
*à chaque règle de la grammaire X qui dérive vers Y alors (abstract) class Y extends X.
*à chaque règle de la grammaire X qui dérive vers Y[A B]
La classe Y (ou une classe de base) contient des champs de types A et B.

1.2 Implémentation pour Hello Word et SansObjet

Implémentation de Hello World:

```
{
    println("Hello World");
}
```

le parcours de l'arbre commence par le noeud "Programme"(Program.java), du programme on passe au "Main" (Main.java) , puis Listes des instructions (ListInst.java) , l'instruction Println (Println.java) . puis on vérifie chaque expression pris par "Println" comme argument donc on passe au nœud "String literal" (StringLiteral.java).

***Implémentation pour l'example sans Objet:**

```
{
   int x ;
   x=readInt() ;
   println(0.5*(x*x)) ;
}
```

le parcours de l'arbre commence toujours par "Programme" (Programme.java), puis "Main" (Main.java), puis on vérifie l'identificateur de type de chaque variable déclarée (ListDeclVar.java) puis (DeclVar.java) aprés on passe l'expression d'affectation (Assign.java) on vérifie le type et l'expression pour les deux côtés de l'affectation

puis on vérifie si l'affectation est possible. après on passe aux instructions comme dans l'exemple de "Hello Word". et à chaque passe s'il y a pas d'erreurs on décore le nœud associé a ce passe.

2 Complétion de l'architecture pour le langage essentiel

2.1 Modifications apportées

***Modifications apportées pour prendre en compte les programmes avec Object:**

on a suivi la même procédure en ajoutant les terminaux et les non terminaux spécifique à l'objet: Return.java, This.java, MethodCall.java, Selection.java, ...etc

-Problématique rencontrée:

comment faire la différence entre les champs d'une classe et les paramètres et les variables déclarées dans les corps des méthodes?

par exemple:

```
class A{
       int x;
       void setX(int x){
        this.x = x;
       }
}
```

ici dans cet exemple le "x" à gauche est le champ "x" alors que le "x" de droite est le paramètre "x".

-Pour résoudre ce problème on a ajoutée un environnement dans (ListParam.java) cet environnement est spécifique à chaque méthode, et contient ses paramètres et ses variables locales et sera empilé à la vérification des corps des méthodes . donc après cet empilement on va commencer la recherche de "x" dans cet environnement et on le cherche pas dans l'environnement de la classe que si on le trouve pas dans la première.

2.2 diagrammes UML Final

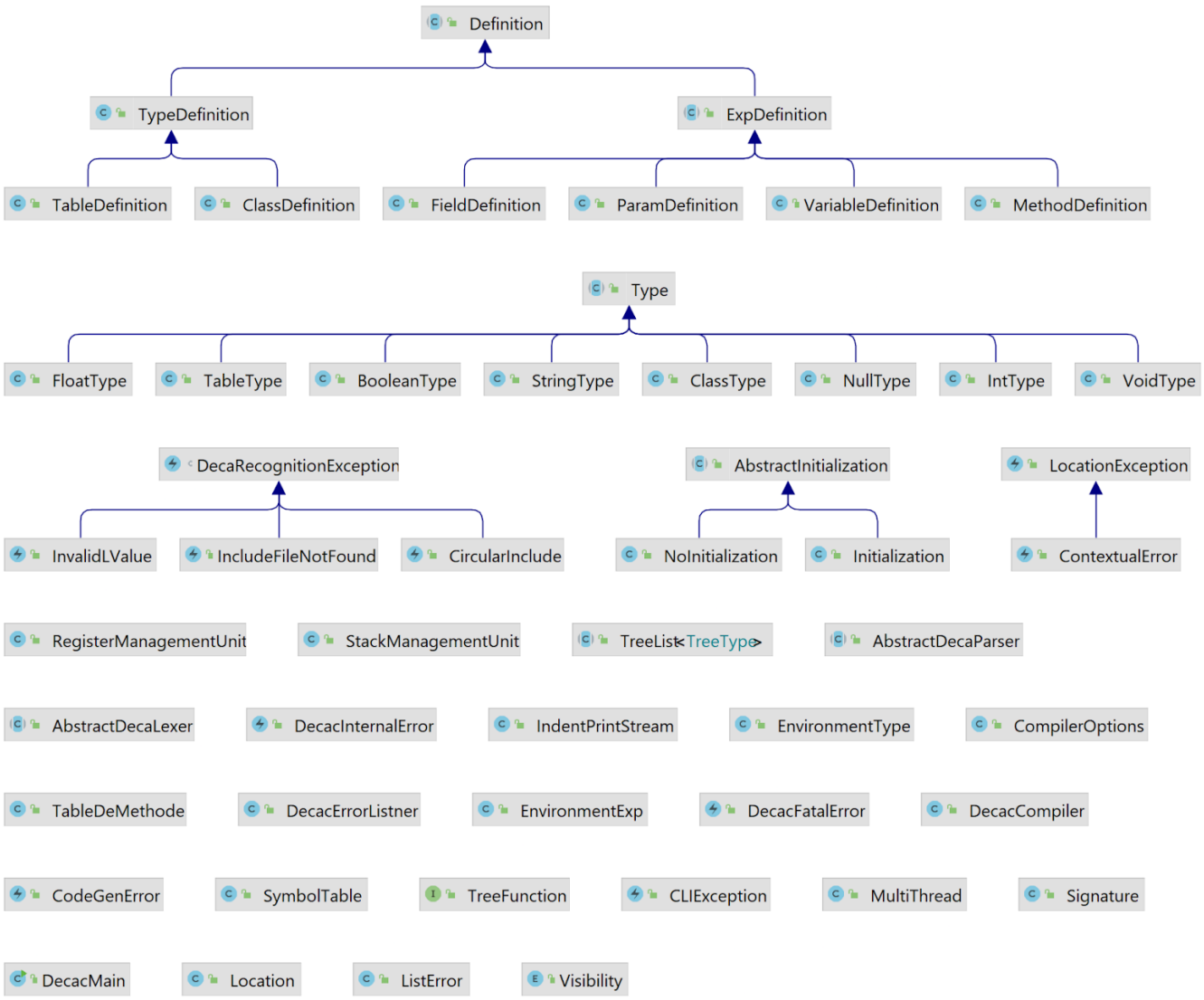

Figure 1: Ce Diagramme Uml les classes qui dérive de définition et les différent types, et ca montre aussi tout les autres classes du projet sauf ceux qui dérivent de AbstractInst

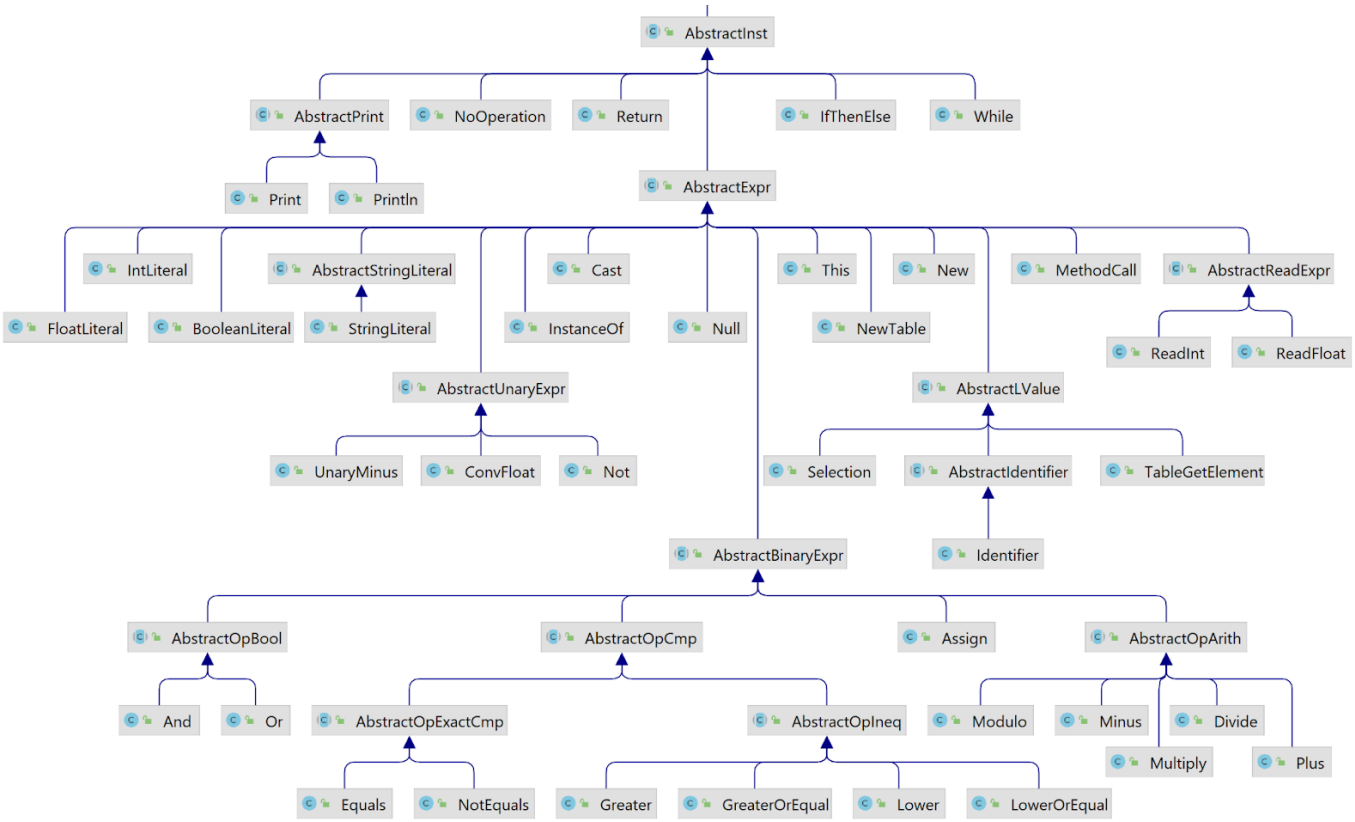

Figure 2: Ce Diagramme Uml les classes qui dérivent de la classe AbstractInst et leur dépendance

Les deux diagrammes uml suivant montrent toutes les classes ajoutées dans le projet et leur dépendance et l'architecture globale du projet. Cette répartition de classe a été cruciale pour la factorisation du code et avoir une base de code organisé.

3 Génération du code

3.1 Gestion des registres

Dans la génération du code, un des premiers problématiques qu'il faut résoudre est la gestion des registres et l'objectif est d'avoir une solution fiable qui peut être exploitée dans le code sans ajouter ajouter de traitement spécial à chaque méthode.

La première chose qu'on fait c'est l'ajout d'une classe avec le nom RegisterManagementUnit Cette classe là fera tout ce qui est nécessaire pour attribuer des registres aux expressions qui en ont besoin.

La structure de donnée utilisée est un tableau de hachage, la raison pour laquelle on a utilisée un tableau de hachage sont les suivants, pour chaque registre on aura besoin d'attribuer un boolean pour indiquer si le registre est occupée ou non et il fallait aussi introduire un compteur pour le nombre de fois que ce registre a été utilisée, ceci sera important pour deux choses :

-Le calcul de nombre de variable temporaire dont on aura besoin au maximum

-Ca permet d'attribuer des registres d'une manière alternante au lieu d'attribuer le même registre occupé à la fois pour une expr.

Une autre solution possible est d'utiliser un min heap et modifier la classe des registres en ajoutant tous les champs nécessaires. Cette approche aurait dû être plus efficace mais cette approche nous restreint car si un registre est libéré par une expression on décrémente son usage et comme le min heap ne se balance pas automatiquement, il fallait éliminer le registre de heap et le remettre pour qu'il y ait la bonne position et dans ce cas le tableau de hachage est beaucoup plus efficace. Pour conclure sur ce point le min heap était notre première solution mais sa manipulation est moins pratique que la structure donnée choisie.

3.2 Gestion de la mémoire

Les méthodes de cette classe seront appelées quand on aura besoin de réserver un registre dans le cas ou on a besoin de mettre une valeur dans un registre afin de l'exploiter ou quand il faut rendre le resultat d'une operation dans un registre comme par exemple l'appel d'une méthode ou l'attribution d'une valeur(Assign opération). On utilisera aussi les méthodes de cette classe pour réduire l'occupation des registres quand on aura plus besoin de la valeur stockée dedans.

Chaque bloc de code qu'on utilisera aura sa propre RegisterManagementUnit donc chaque méthode de classe, chaque méthode d'initialisation de classe et le main. On stockera dans la classe tous les registres utilisés qui sont différents de R0/R1 afin de pouvoir les mettre dans la stack au début des méthodes et les retirer à la fin de la méthode

Comme les valeurs stockées dans les registres ne peuvent être exploitées dans la prochaine instruction ou la prochaine déclaration de variable, tous les registres stables (R2-R15) sont considérés libres à la fin d'instruction.

Afin d'associer une position dans le stack au variables qu'on utilise, il fallait dans ce cas introduire une nouvelle class avec le nom StackManagementUnit, cette classe contient trois compteur principaux : gbCounter, lbCounter et paramCounter.

Le bgCounter est utilisé pour associer une position en mémoire au tableau de méthodes et les variables globales. Le lbCounter est utilisé pour associer aux variables locales des position en mémoire et paramCounter est utilisée pour les paramètre d'une méthode.

Chaque méthode aura sa propre StackManagementUnit afin de pouvoir associer les adresses uniquement à ces variables et ne pas interférer avec les autres méthodes. On calcule aussi dans cette classe la taille du stack dont on aura besoin de réserve dans le début d'une méthode ou le programme principal

3.3 Gestion des erreurs d'exécution

Pour pouvoir ajouter les erreurs d'exécution, il fallait définir une nouvelle classe avec le nom ListError au quel on place un tableau de hachage avec la liste des erreur possible en runtime dont on aura besoin et on associe aux éléments de ce tableau de hachage un booléen pour indiquer si l'erreur a été activée au nom.

Lors de la génération d'instruction, chaque fois qu'on appelle un label d'une erreur, on passe cette classe pour activer l'erreur qui à la fin du programme ajoutera toutes ces erreurs au code du programme principale.

3.4 Génération de code pour les expressions

3.4.1 Génération pour les opération binaires

la plupart des opération binaires sauf le assign et les opération booléenne qui nécessite un traitement spécial passe par la même méthode codeGenInst défini dans AbstractBinaryExpr, cette méthode est utilisée pour explorer les deux fils quelle posséde qui sont le left operand et le right operand, et dans notre architecture chaque expression possède un champ qui s'appelle le registre de retour qui possède un régistre auquel la valeur de calcul de tous ces fils existe.

Cette méthode commence initialement par explorer le left operand qui fera tout le calcule

nécessaire pour enfin retourner une valeur qui sera mis dans un registre de retour dans l'expr du left operand.

On explore maintenant la droite, une chose importante lors de l'exploration de la droite est que dans certains cas on n'a pas besoin de passer un registre pour exploiter la valeur dans l'expression car on peut directement faire l'opération avec l'adresse si on travaille avec un identifiant, cela n'est pas nécessaire mais peut réduire le nombre d'opérations considérablement.

Ainsi, maintenant on a deux opérandes qui possède des valeurs qu'on peut accéder pour faire une opération et dans cette situation on appel la méthode executeBinaryOperation qui est définit dans les opération binaire et qui ajoute les instructions qui concerne l'opération qu'on veut réaliser: addition, différence, multiplication, division ...

3.4.2 Attribution des registres aux littéraux et au identificateurs.

Une méthode a été créé dans abstractExpr avec le nom LoadGencode, cette méthode la est utilisée pour reservée un registre et elle prend aussi comme argument un booléen indiquant s'il faut appeler la méthode loadItemintoRegister qui est défini dans les classes qui possède une valeur qu'on mettra dans un registre dans un registre comme IntLiteral, FloatLiteral, Identifier et d'autres.

Lors de la réservation d'un registre, on peut se trouver dans la situation ou tout les registres sont occupées et dans ce cas il faut mettre prend un registre occupée et stockée l'ancien valeur qu'il possède dans le stack mais quand on termine d'utiliser ce registre de cette expr spécifique il faut remettre dans le registre l'ancienne valeur qu'il possédait afin qu'on ne la perd pas. Mais le problème est de savoir quand on doit remettre la valeur stockée dans la mémoire du registre.

Pour résoudre ce problème, on a défini un stack dans chaque expr qui contient tous les registres que l'expression a besoin de libérer dans le bon ordre.

Une méthode qui s'appelle pop registers a été ajouté qui remet les registres à leur valeurs initiaux en ajoutant des instruction de pop. Cette méthode sera appelées quand la valeur stockée dans le registre n'est plus important pour notre calcul, par exemple dans les opération binaire on remet le résultat de l'opération dans le registre de gauche ainsi on n'aura plus besoin au registre de l'opérande droite donc on pop les registres utilisées, et on transfert les registre dans le stack de l'opérande gauche vers l'expr du opération binaire afin de retourner les valeurs stockées dans le registre de gauche.

La chose la plus importante lors de l'attribution d'un registre à une expression dans le cas ou tout les registres sont réservées et ne pas prendre le même registre deux fois à la filées et dans le cas où le nombre de registre différent de R0 et R1 sont égal à deux, il faut toujours alterner entre les registre sinon on aura deux expr dans le même registre donc il faut passer un registre intermédiaire afin de pouvoir faire le calcul.

3.5 Gestion du tableau de méthode

Une classe a été ajoutée dans le paquetage codegen, afin de stocker les adresses des tableaux de méthodes de chaque classe, un tableau de hachage a été utilisé afin de stocker ces informations, ce tableau de hachage associé à chaque classDefinition l'adresse du tableau de méthode de classe. Ce choix de structure de données a été réalisé grâce à la rapidité d'accès quand on a la définition de la classe.

4 Conclusion

La conception de la partie C était beaucoup plus difficile que la partie B comme l'architecture de cette partie a été expliquée dans le poly. D'autres méthodes ont été réalisées dans cette phase de conception mais c'est un peu difficile de tout expliquer sans entrer dans les détails du code. Pour conclure, on croit le compilateur peut être optimisée et les instructions assembleur peuvent aussi être réduite et si on a choisi l'extension optim on aurait due instroduire plus de structure données pour mieux exploiter ce qui est dans les registres.