Projet GL 2021:
Documentation de conception

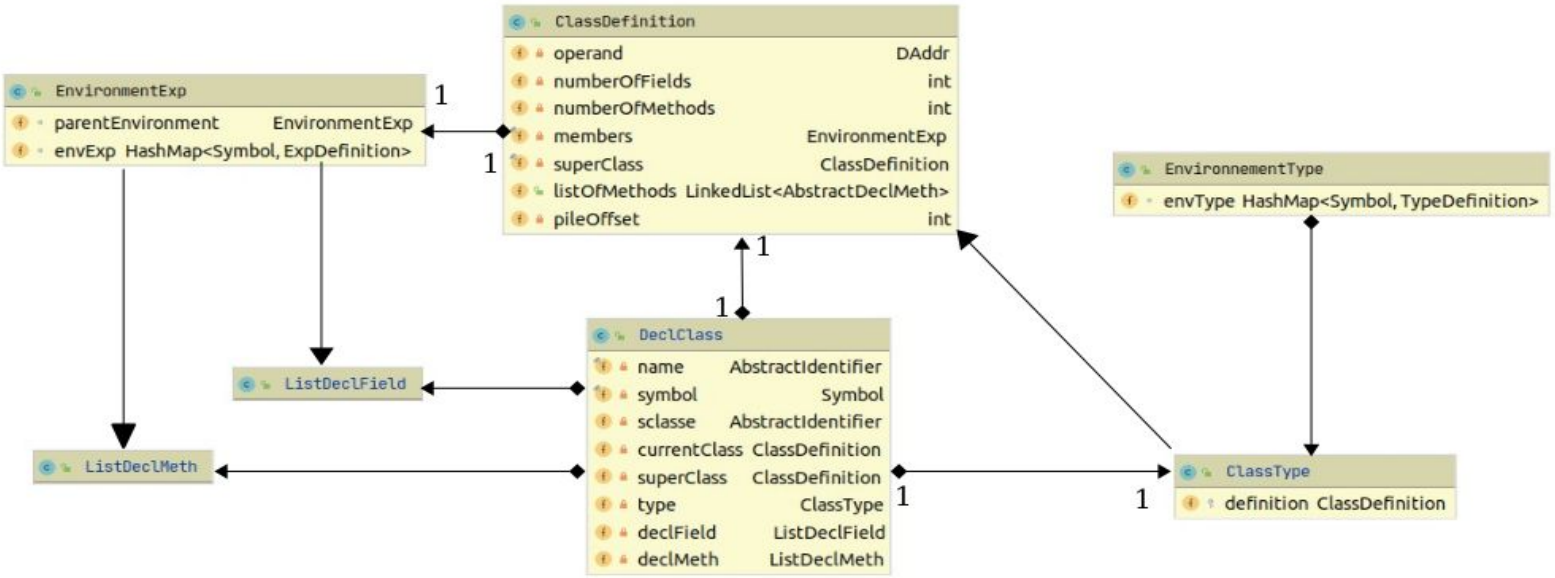
Équipe n°31 : Antoine Briançon, Axel Glorvigen, Thibault Launay, Maxime Martin et Pierre Pocreau

Sommaire:

I. Implémentation et architecture pour les classes et méthodes.

A. Classes

Dans le langage Deca il est possible de créer des classes, contenant des méthodes et des champs. Notre implémentation suit le diagramme de classe simplifié suivant:



Les classes du programme Déca que l'on souhaite compiler sont représentées par des *ClassDefinitions*, elles contiennent un environnement local qui est une instance *d''environnementExp*, dans lequel on peut retrouver les méthodes et champs par leur nom.

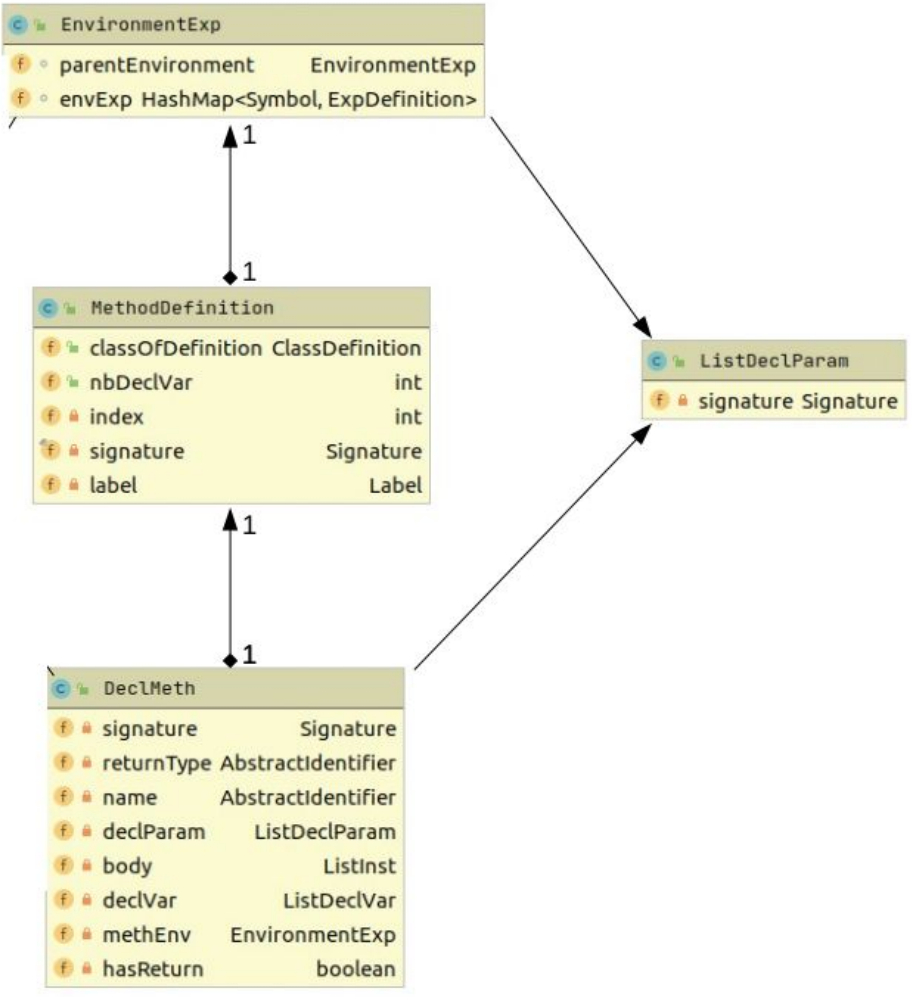
Les *ClassDéfinitions* sont stockées dans une HashMap représentée par l'environnement des Types, propre à l'instance de notre compilateur. On peut ainsi instancier de nouveaux objets, n'importe où dans notre programme déca.

On retrouve dans la classe *DeclClass* l'ensemble des méthodes propres à la vérification contextuelle, et à la génération de code.

Nous choisissons aussi de stocker directement la liste des méthodes dans une LinkedList, qui contient l'ensemble des méthodes de la classe, et de sa *superClass*. Ainsi nous avons accès aux méthodes lorsque nous manipulons la *classDefinition* d'une classe.

B. Méthodes

On peut définir dans une classe Déca, plusieurs méthodes. Voici le diagramme de classe résumant notre implémentation:

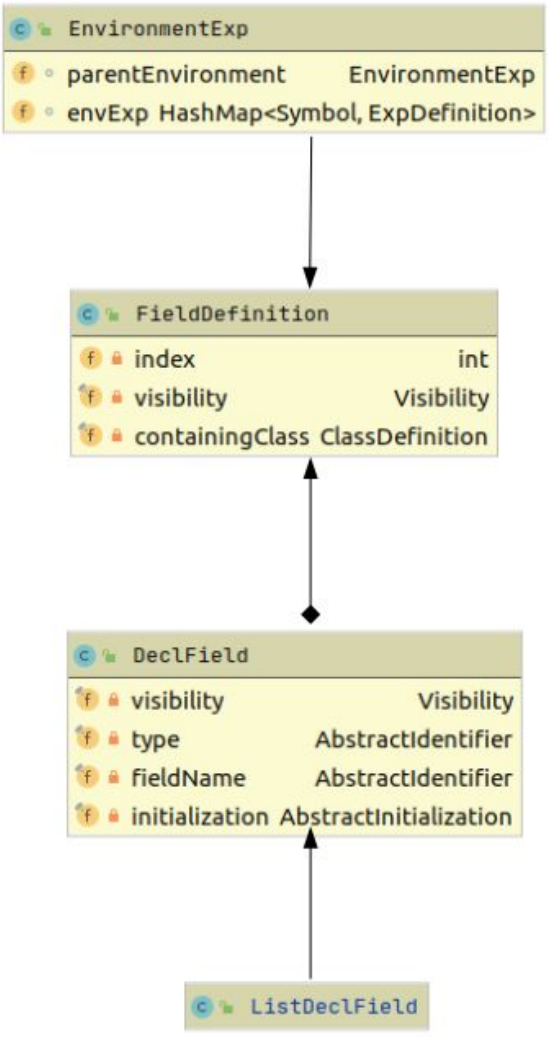


Une méthode possède son propre environnement local, instance de la classe *EnvironnementExp* dans lequel on peut trouver ses paramètres mais aussi l'ensemble des variables locales à la méthode, déclarées dans son corps. L'environnement local de la méthode hérite de l'environnement local de la classe, ainsi on y trouve aussi les champs et les autres méthodes.

Nous avons représenté uniquement la *ListDeclParam* sur le diagramme, mais une méthode possède aussi une liste de déclaration de variables et une liste d'instruction, (similaire à un *main*).

C. Champs

Enfin, le dernier point propre à la déclaration des classes sont les champs. Il sont associés à une définition de champs, qu'on peut retrouver dans l'environnement de la classe.



D. Autres classes relatives à la gestion des classes et méthodes

1. New

Permet grâce au mot réservé “new” d’instancier de nouveaux objets. Vérifie que la classe est bien présente dans l’environnement des types, et effectue la déclaration des champs. C’est une sous classe d’*AbstractExpr*.

2. This

Permet dans une classe, d’appeler des méthodes ou de sélectionner des champs de la classe. C’est aussi une sous classe d’AbstractExpr.

3. Null

Représente l’objet Null, par défaut les objets non initialisés sont initialisés à Null par le compilateur. Peut lever une erreur si on essaye de le déréférencer.

4. Sélection

Permet de sélectionner le champ d’un objet, pour s’en servir comme expression, ou bien pour lui assigner une valeur. C’est une sous classe *d’AbstractLValue* pour pouvoir être utilisée à gauche d’une assignation.

5. Return

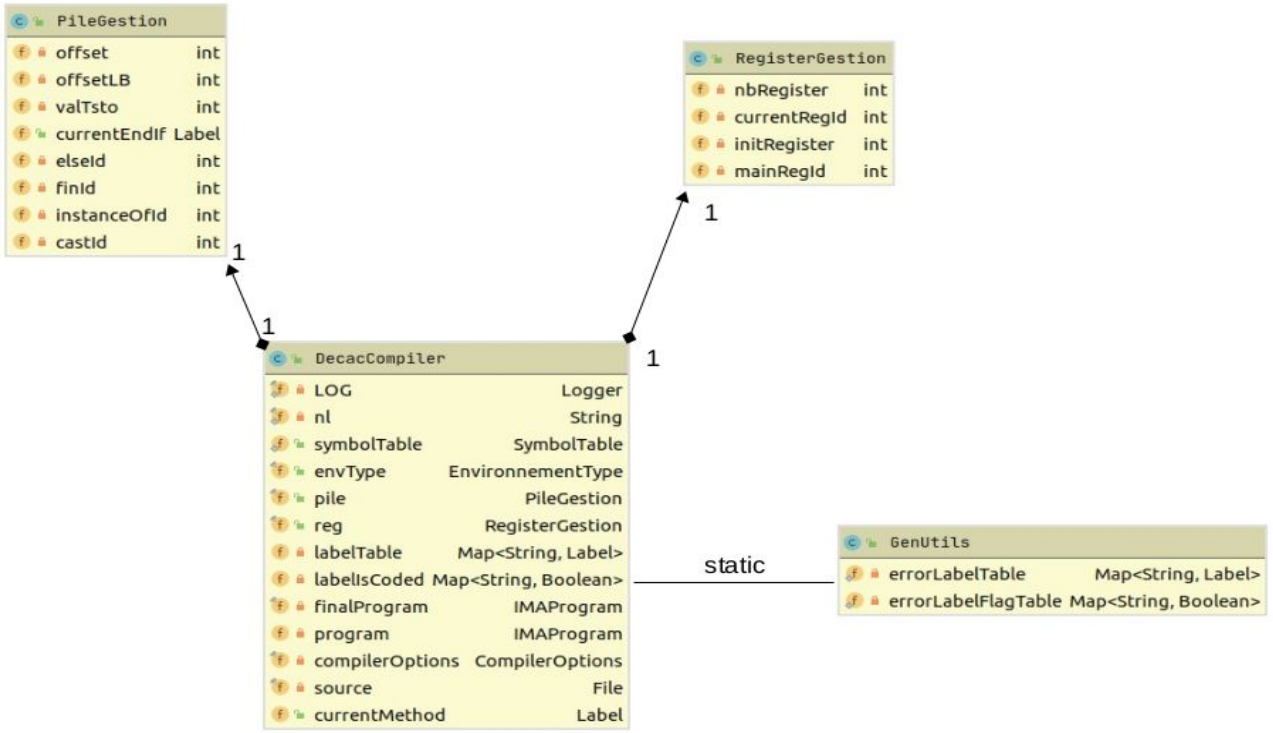
Le *return* est nécessaire pour toute fonction qui n’est pas de type *void.* Toute fonction qui n’est pas de type void possède un message d’erreur “Erreur : sortie de la méthode sans return” juste avant la fin du code de la fonction. Croiser une instruction *return* permet de sauter par dessus cette erreur et de retourner une valeur dans le registre *R0*. *Return* est une sous classe de la classe *AbstractInst*

E. InstanceOf et Cast

Le langage Déca avec les classes et la méthode forme le langage essentiel. Le langage complet nécessite l'ajout du *cast* dynamique et de la classe représentant un *instanceOf*.
Ces deux classes héritent de la classe *AbstractExpr*. Parce que le cast est dynamique, il doit vérifier qu'il est possible lors de l'exécution en codant un *instanceOf*.

F. Gestion de la pile et des erreurs

Nous avons introduit plusieurs classes utilitaires permettant la gestion des piles, des registres des labels et des erreurs, qui sont instanciées par chaque instance de *DecacCompiler*, c'est important pour paralléliser la compilation de plusieurs fichiers.



1. Gestion de la pile et des labels

Une instance de *PileGestion*, contient plusieurs entiers qui servent représentent les offsets de GB et de LB. On y trouve aussi des indices pour créer des labels différents dans les if/then/else, dans les *instanceOf* et les *casts*. Les labels sont stockés dans la

hashmap *labelTable* de l'instance *DecacCompiler*. Enfin on y trouve un entier représentant la valeur du *TSTO*. On pourrait séparer les indices servant à la pile et ceux des labels dans des classes différentes.

2. Gestion des registres

La gestion des registres se fait par une instance de *RegisterGestion* instanciée dans chaque instance de *DecacCompiler*. Cette instance stock l'id courant du registre utilisé, cela fonctionne bien dans le cas du déca sans-objet, mais nous avons rencontré des problèmes lors de l'appel de méthodes, pour sauvegarder les registres. Il faut donc modifier cette classe pour nous permettre de le faire facilement. Ce problème avec la sauvegarde des registres est le principal souci de notre compilateur, mais nous l'avons diagnostiqué trop tard et n'avons malheureusement pas eu le temps de le résoudre.

3. Gestion des erreurs

Enfin la gestion des erreurs se fait dans la classe *GenUtils*, cette classe est statique. On y trouve toutes les erreurs prédéfinies, et des fonctions permettant d'insérer le saut sur le label d'une erreur dans le code assembleur, en vérifiant que l'option -n (no-check) du compilateur n'est pas activée. On y trouve aussi une hashMap qui permet de savoir si une erreur a été rencontrée et si l'on doit générer le code qui lui est associé.

Cette table ne devrait pas être statique et devrait se trouver dans chaque instance de *DecacCompiler*, dans notre implémentation actuelle on risque de générer le code d'une erreur si elle est rencontrée dans un autre fichier que l'on compile en parallèle, mais ce n'est pas critique.

On trouve aussi dans *GenUtils* les fonctions *Dval* et *Mnemo* utiles à la génération de code d'expressions arithmétiques et de comparaisons.

II. Spécificités

A. Choix d'implémentation pour la classe Object

La classe Object est implicitement définie pour chaque programme Deca qui utilise une classe. Nous utilisons une méthode statique pour l'initialiser (*DeclClass.declObject)*. On déclare également la seule méthode de cette classe en Deca qui est equals. Ensuite, on l'assigne en super classe à toute classe qui n'a pas de super classe explicitement déclarée.

On code la table de la classe objet et la fonction equals uniquement si l'on déclare des classes (donc si *ListDeclClass* est non vide.)

B. Choix d'implémentation des chaînes de caractères

En Deca, les chaînes de caractères sont utilisées uniquement pour afficher des messages dans la console. Il n'y a aucune affectation ou manipulation possible. Nous avons fait le choix de néanmoins définir le string comme un type (on ne peut donc pas faire de classe string), mais l'utilisateur ne peut pas l'utiliser. Une future amélioration pourrait être de développer des fonctionnalités sur ce type pour effectuer des manipulations dessus, comme avec l'objet String en java.

C. Choix d'implémentation pour le code assembleur

Le compilateur Decac créer, si le programme deca est valide, un programme en langage assembleur exécutable par la machine abstraite IMA, déjà paramétrée dans ce projet. Ce programme assembleur est, comme le programme Deca, réparti en plusieurs parties : déclaration des classes, déclaration des champs et méthodes, déclaration des variables, code des méthodes, initialisation des champs, etc ... Ces différentes parties forment des blocs distincts mais sont bien sûr reliées dans le programme assembleur final. L'ordre dans lequel ces blocs sont codés est très important en assembleur, ce qui nous a poussé à choisir l'implémentation suivante.

Certaines commandes, comme ADDSP, TSTO ou encore la sauvegarde des registres, prennent en paramètre une variable que l'on peut connaître seulement après avoir parcouru la totalité d'un bloc. Par exemple, pour TSTO, il s'agit du nombre de variables créées, le nombre de temporaires utilisés etc. La difficulté est que ces commandes sont à inscrire au début d'un bloc, en amont de tout autre commande de ce bloc, et qu'il n'est pas possible de connaître le début du bloc en cours lorsque celui-ci est situé dans le programme complet. Nous disposons seulement de méthodes nous permettant d'ajouter des commandes en tête et en queue de programme.

Nous avons donc choisi, dans la classe DecacCompiler, de créer deux instances de IMAProgram : finalProgram, qui contient la totalité des blocs créés, et program, qui est un buffer contenant le bloc en cours de création, permettant de manipuler ce bloc comme un programme complet, avec insertion de commande en tête et en queue. Ainsi, à chaque nouveau bloc, on copie la totalité du contenu de program à la fin de

finalProgram, on vide program et on peut réécrire ce nouveau bloc dans program sans écraser le bloc précédent. Cette technique permet d'écrire facilement en tête de chaque bloc les instructions voulues, tout en conservant le programme complet qui ne sera pas modifié.

D. Parallélisation

L'option -P de la commande decac permet de compiler plusieurs programmes .deca en parallèle, c'est-à-dire en appelant un ensemble de threads qui compileront simultanément un programme chacun. La syntaxe de la commande est telle que:

decac -P <fichier1.deca> <fichier2.deca> etc.

Pour mettre en place la parallélisation, on utilise la librairie *java.util.concurrent* de Java. Quand le compilateur va détecter l'option dans la classe DecacMain, il va créer une instance de *ExecutorService* ainsi qu'une liste de threads d'instance *Future*. Chacun de ces threads va être chargé de la classe *DecacThread* qui contient la méthode *call()*, propre à la libraire concurrent. Cette méthode appelle la fonction de compilation avec pour arguments le fichier à compiler et les autres options rentrées par l'utilisateur.

Une fois l'initialisation de tous les threads terminés, on les lance simultanément. Si une erreur survient lors de la compilation d'un des fichiers, le thread concerné renvoie une erreur qui mettra fin au programme du compilateur qui enverra à son tour une erreur à l'utilisateur. A noter que si les autres fichiers, s'ils sont corrects, seront compilés sans problèmes.

Ci-dessous la partie du code de la classe *DecacMain* gérant la parallélisation:

```
if (options.getParallel()) {
    ExecutorService executor = Executors.newFixedThreadPool(options.getSourceFiles().size());
    List<Future<Boolean>> futures = new ArrayList<>(options.getSourceFiles().size());
    for (File source : options.getSourceFiles()) {
        futures.add(executor.submit(new DecacThread(options,source)));
    }
    for(Future<Boolean> future : futures){
        try{
            if(future.get()){
                error = true;
            }
        } catch (ExecutionException | InterruptedException e){
            e.printStackTrace();
        }
    }
    executor.shutdown();
```

D. Implémentation de la librairie math

Une fonctionnalitée que l'on pourrait facilement introduire dans notre compilateur, est la conversion d'instruction codant les expressions a + x*b par des instructions FMA. Cette instruction ne diminue pas le nombre de cycles effectués, mais elle améliore la précision en effectuant un arrondis contre deux si l'on fait l'opération avec un ADD et un MUL. La précision est très importante pour la librairie Math, de plus les opération de la forme a + x*b sont critiques, notamment pour effectuer des développements de Taylor, avec un algorithme de Horner.

On pourrait l'implémenter de la même manière que la simplification des LOAD/STORE décrite dans la partie Optimisation du compilateur.

Concernant l'implémentation de la librairie math, il faut se référer au document sur l'extension trigo.

Notre compilateur possède des problèmes de gestion de pile et de registres lors de l'appels de fonctions, notamment lorsqu'elles s'appellent entre elles. Cette dynamique d'appel récursifs en "ping-pong" est clef dans l'implémentation des fonctions trigonométriques et met à mal notre compilateur. Ainsi les fonctions cosinus et sinus passent nos tests mais pas les fonctions arcsin et arctan.

III. Optimisation du code assembleur

Cette partie peut être retrouvée dans la documentation d'impact énergétique.

A. Store/Load:

Lorsque l'on évalue une expression, celle-ci est calculée dans le registre courant. Quand elle sert pour la déclaration ou l'assignation d'une variable, on effectue à un store pour charger l'adresse mémoire de la variable avec la valeur que l'on vient de calculer. Si on utilise dans l'instruction suivante, le compilateur va Load la variable dans un registre, parfois le même registre que celui dans laquelle l'expression a été calculée. Le schéma est le suivant:

```
Store Rx y(LB)
Load y(LB) Rx
```

Comme la valeur est déjà dans Rx, on peut supprimer le Load et gagner ainsi une instruction et 2 cycles internes.

Notre optimiseur est rudimentaire et parcourt le *finalProgram* pour y trouver le motif précédent (sur deux lignes consécutives), on pourrait l'améliorer pour supprimer le Load si la valeur contenue dans Rx n'a pas été écrasée.

Avec cette technique simple, nous pouvons gagner par exemple 288 cycles sur le programme *ln2.deca* soit environ 2%.

B. Permutation d'opérandes lors d'opérations commutatives:

Lorsque l'on code une opération arithmétique, on évalue l'expression de gauche dans le registre courant *n*, puis l'expression de droite dans le registre *n+1*. Le résultat de l'opération est stockée dans le registre *n*. Dans le cas où l'opérande de gauche est une variable, nous devons donc la charger dans un registre avant de réaliser l'opération alors que nous pouvons réaliser une opération avec uniquement une adresse de variable.

Pour réduire le nombre de load, nous permutons les opérandes des opérations commutatives (addition et multiplication) lorsque l'opérande de gauche est un littéral ou une variable.

Illustration:

Optimisé

```
19|  LOAD #0x1.0p0, R2
20|  DIV #0x1.3bp12, R2
21|  MUL 5(GB), R2
22|  LOAD #0x1.0p0, R3
23|  DIV #0x1.68p9, R3
24|  ADD R3, R2
25|  MUL 5(GB), R2
26|  LOAD #0x1.0p0, R3
27|  DIV #0x1.ep6, R3
28|  ADD R3, R2
29|  MUL 5(GB), R2
30|  LOAD #0x1.0p0, R3
```

Non Optimisé

```
16|  ADD 2(GB), R2
17|  DIV #0x1.0p1, R2
18|  STORE R2, 5(GB)
19|  LOAD 5(GB), R2
20|  LOAD 5(GB), R3
21|  LOAD 5(GB), R4
22|  LOAD 5(GB), R5
23|  LOAD 5(GB), R6
24|  LOAD 5(GB), R7
25|  LOAD 5(GB), R8
26|  LOAD #0x1.0p0, R9
27|  DIV #0x1.3bp12, R9
28|  MUL R9, R8
```

Dans le code de droite, on load plusieurs fois la valeur stockée dans la pile globale, à droite on effectue directement les opérations. On diminue le nombre d'instructions et on limite le nombre de registres utilisés.

Cette optimisation nous permet de gagner 330 cycles sur *ln2.deca* soit environ 2.2% du nombre de cycles obtenus sans optimisation.