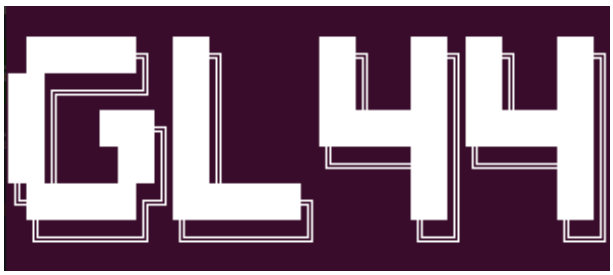


Documentations
Projet de Génie Logiciel

Version 2.3.1

GL44
Aymeric Baron - Carmel Benoit - Eliott Dumont
Jérôme Duveau - Jules Roques
January 26, 2024

Contents

1 Introduction

Ce document récapitule le projet Génie Logiciel mené par Aymeric Baron, Carmel Benoit, Eliott Dumont, Jérôme Duveau et Jules Roques du 18 décembre 2023 au 26 janvier 2024.

Vous y trouverez les informations concernant nos choix de conception, nos processus de validation, toutes les spécifications de l'extension, ainsi que quelques pages sur l'impact énergétique de notre projet.

2 Documentation de conception

2.1 Lexer et Parser

Les analyses lexicale et syntaxique du code Deca fournit en entrée sont effectuées dans les fichiers `DecaLexer.g4` et `DecaParser.g4` du répertoire `src/main/antlr4` . Le paragraphe suivant a pour but de présenter la démarche à suivre et les erreurs à ne pas faire pour faire évoluer le compilateur et afin qu'il reconnaisse une plus grande partie du langage.

De manière générale, l'ajout de règles supplémentaires est envisageable dans la mesure où les règles définissent une grammaire hors-contexte au format EBNF. La définition des tokens reconnus est faite dans le lexer. Ce fichier peut être enrichi à condition de ne pas répéter un token plusieurs fois et de prendre soin de commenter l'ajout de nouveaux tokens comme ce qui a déjà été fait.

Concernant le parser, l'ajout de règles se fait conformément à la syntaxe imposée par antlr. Se référer à la documentation ANTLR si besoin. Les classes créées en conséquence (par exemple `MethodCall.java`) doivent être mises dans `src/main/java/fr/ensimag/deca/tree` Ces classes peuvent hériter d'une classe abstraite déjà existante, ou alors la super classe doit aussi être créée dans ce même répertoire. Nous détaillerons en partie [2.4] une partie de l'architecture des classes afin d'y voir plus clair.

Dans chacune des classes devant être créée, en plus du constructeur et des méthodes abstraites héritées, des fonctions `decompile` et `prettyPrintChildren` doivent être implémentées. La première, decompile, permet la décompilation de l'arbre afin de vérifier que le code en sortie est compatible avec l'arbre ayant été crée par le lexer et le parser. La seconde, prettyPrintChildren, permet un affichage de l'arbre représentant le code Deca en entrée, non décoré. Cet affichage permet la vérification du bon fonctionnement du parser vis à vis du code en entrée. En plus de l'affichage des non terminaux, l'arbre précise aussi les locations des tokens lus grâce à la fonction `setLocation` . Cette fonction précise dans l'arbre la ligne et la colonne auxquelles se trouve un token reconnu.

2.2 Vérification contextuelle

En parallèle des vérifications contextuelles est faite la décoration de l'arbre. Toute cette partie se retrouve dans les fonctions `verifyXyz` du dossier `src/main/java/fr/ensimag/deca/tree`

Durant la totalité du projet, la stratégie sur cette étape était de commencer par coder la décoration de l'arbre dans le but de laisser passer un programme valide avant de relever les erreurs contextuelles des programmes ne pouvant pas être compilés en Deca. De cette façon, l'étape C peut débuter ses tests et donc son implémentation le plus rapidement possible ce qui est a prioriser car c'est la plus chronophage. Pour relever les erreurs nous nous sommes servi de la syntaxe contextuelle mais aussi de toutes les autres règles propre au Deca spécifiée dans la documentation à disposition. Comme par exemple, la déclaration de variable qui ne

peut pas être un `string` .

Un des problèmes rencontrés qui nous a coûté le plus de temps a été que lorsque nous cherchions un type dans EnvironmentType à l'aide du Symbol créé dans la partie A, cela nous renvoyait null. Nous avons vérifié que EnvironmentType était bien initialisé, que notre Symbol était le bon et que les Symbol étaient unique par rapport à leur nom. Cependant, dans notre cas, les Symbol sont créés dans deux tables différentes ce qui explique que nous ne les trouvions pas dans la table utilisée à l'étape B. Il est donc nécessaire à chaque recherche dans EnvironmentType de créer un nouveau Symbol dans la table courante (qui sera simplement récupéré si il existe déjà) à l'aide du nom de celui de l'étape A.

Pour la conversion en flottant dans le cas d'une opération arithmétique concernant un entier et un flottant, il a fallut créer ConvFloat. Pour se faire nous avons décider d'Overload la méthode verifyRValue présente dans AbstractExpr en une méthode avec un paramètre booléen supplémentaire dans le but de l'appeler seulement dans le cas d'un ConvFloat. Cela lui permet d'être spécifique car on ne devait pas y faire les mêmes opérations tout en laissant un code cohérent de par le nom de cette méthode qui devait correspondre à sa fonction.

Pour les déclarations de méthodes, nous avons choisi de modifier les paramètres de verifyType dans la classe `Identifier` . Nous y avons ajouté un booléen permettant de savoir si nous sommes dans le type renvoyé par une méthode ou non. En effet, dans tous les autres cas qui sont des déclarations de variable, il est interdit d'assigner le type `void` et la vérification se faisait dans cette fonction. Maintenant, si l'appelle se fait sur le type de retour d'une méthode, le type `void` est autorisé et ne relève pas l'erreur associée. Le but était dans ce cas d'utiliser au maximum la factorisation du code en ajoutant simplement un paramètre à la place de déclarer une nouvelle méthode.
Dans la même optique, nous avons remarqué que l'implémentation de `MethodBody` était parfaitement identique à celle du main faite pour le sans-objet. C'est pourquoi nous avons utilisé les mêmes méthodes de base en changeant simplement l'EnvironmentExp et la CurrentClass sur lequel elles sont appelées.

La dernière difficulté fut de comprendre ce qui était attendu à l'aide de la syntaxe contextuelle. Pour exemple, nous nous sommes demandé si dans une même classe, un attribut et un paramètre de méthode peuvent avoir le même nom. Nous avons codé en prenant en compte que ce n'était pas possible car la classe `ClassDefinition` ne possédait qu'un seul `EnvironmentExp` mais nous avons découvert trop tard grâce à un exemple du polycopié que c'était en fait possible. Voila donc la principale source d'amélioration possible du côté de cette étape.

2.3 Génération de code assembleur IMA

Pour générer le code assembleur, nous avons recours à plusieurs classes et méthodes.

Notons que c'est cette partie qu'il faut finir d'implémenter pour supporter le langage avec objet.

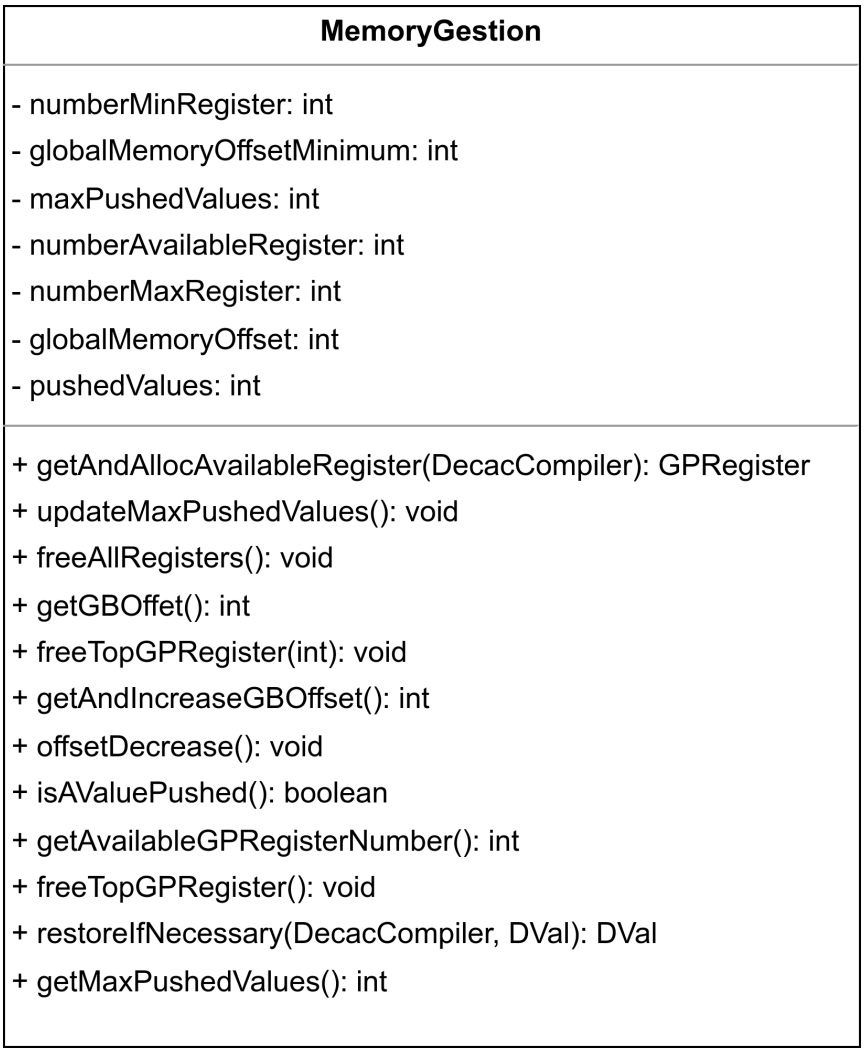


Figure 1: `class MemoryGestion`

2.3.1 Gestion de la mémoire

La classe `MemoryGestion` (figure 1) du package `fr.ensimag.deca.codegen` permets la gestion de la mémoire de la machine abstraite IMA. Elle traite en interne l'allocation des registres et de la pile en fonction du programme deca rentré. Cette classe offre notamment deux possibilité.

- une méthode pour récupérer un `GPRegister` libre, qui prends en compte le cas dans lequel tous les registres sont déjà utilisés (la pile rentre donc en compte)
- des méthodes permettant de gérer les déclarations de variables avec l'information sur le décalage mémoire par rapport à la base globale (`GBOffset`)

Cette classe peut être étendue dans l'optique de gérer le langage deca avec objet, en prenant en compte des opérations sur le tas.

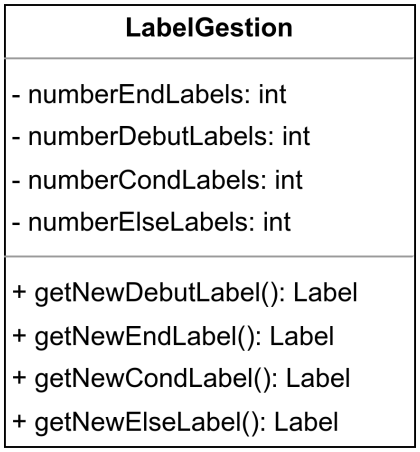


Figure 2: `class LabelGestion`

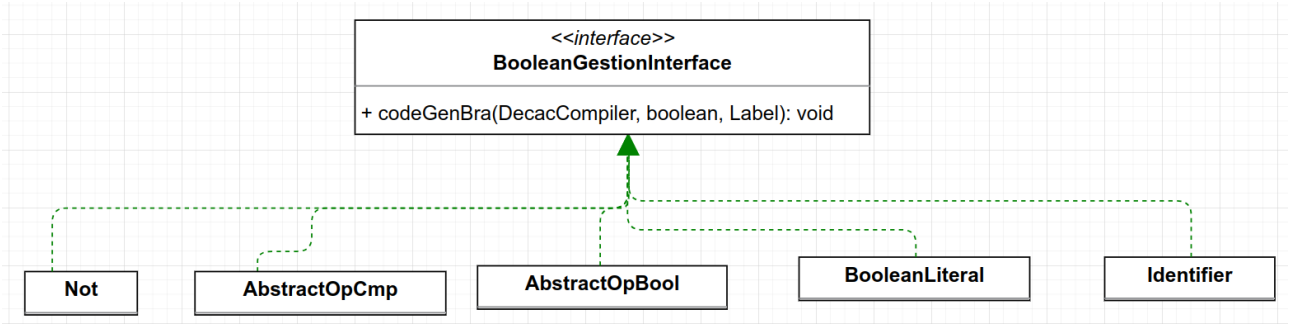


Figure 3: `interface BooleanGestionInterface` et ses classes filles

2.3.2 Gestion des labels

La classe `LabelGestion` (figure 2) du package `fr.ensimag.deca.codegen` permets la gestion des labels du programme assembleur. Elle garde des informations sur les labels déjà utilisés et peut en fournir des uniques.

2.3.3 Gestion des branchements avec conditions booléennes

Certaines expressions nécessitent une méthode particulière qui sert à la génération de code avec branchements. On matérialise ce besoin par l'interface `BooleanGestionInterface` (figure 3).

La méthode `codeGenBra()` (prototype en figure 3) permets de sauter vers un label si l'expression sur laquelle est appelée la méthode est évaluée à vrai ou faux. Cette méthode est très utile pour générer les instructions `if then else`, `while`, ainsi que les expressions booléennes.

2.3.4 Génération d'un noeud de l'arbre abstrait

N'importe quel noeud (classe dérivant de `Tree`) possède une méthode de génération de son équivalent en assembleur.

Par exemple, pour la classe `Main` qui dispose de deux attributs de types `ListDeclVar` et `ListInst` , on appelle la méthode de génération de code pour chacun d'entre eux. Ces classes possèdent elles-mêmes des attributs sur lesquels on appellera la méthode de génération de code, et on répète ce processus jusqu'aux classes feuilles qui implémenteront effectivement du code en assembleur.

2.3.5 Génération des expressions

Une grande partie des expressions (dérivant de `AbstractExpr`) doivent fournir une méthode `codeGenExpr` permettant de générer le code calculant cette expression et fournissant sa désignation de valeur (`DVal`).

L'algorithme de gestion des registres pour les expressions arithmétiques binaires n'est pas aussi naïf que celui présenté dans le cours vidéo. Il était proposé de systématiquement charger dans un registre les opérandes droite et gauche. Dans notre compilateur, seule l'opérande gauche est chargée, ce qui permet d'économiser un registre. Un exemple est donné en listing 1.

Listing 1: Optimisation des opérations arithmétiques binaires

```
; Calcul de i = 1 + 2
; version naive
LOAD #1, R2
LOAD #2, R3
ADD R3, R2
; version optimisee
LOAD #1, R2
ADD #2, R2
```

2.3.6 Affichage des expressions

Toutes les expressions pouvant être affichées implémentent la méthode `codeGenPrint()` .

- pour les expressions flottantes, entières ou booléennes (que ce soient des variables, des immédiats ou des expressions binaires), on charge dans le `GPRegister` R1 la désignation de valeur correspondante puis on l'affiche avec la commande appropriée;
- pour les chaînes de caractères l'affichage se fait sans passer par de registre intermédiaire.

2.4 Architecture des classes

3 Documentation sur la validation

3.1 Descriptif des tests

Pour créer des tests, nous avons suivis deux méthodologies différentes. La première sert à déblayer le terrain pour avoir une bonne base de tests et une bonne compréhension de comment notre compilateur devait fonctionner. Cela sert nous sert tant à tester que de comprendre la structure de compilateur, des comment les règles doivent s'imbriquer les unes dans les autres et quels sont les éléments à vérifier. Une fois cette première liste de tests non exhaustive créée, nous avons utilisé Jacoco afin de mettre a jour continuellement cette liste de tests pour améliorer au maximum notre taux de couverture.

Dans un premier temps nous avons établis une liste d'erreur potentielle au sein de chaque règle afin de comprendre les besoins de notre compilateur mais surtout de voir les potentielles problèmes que pourrait rencontrer notre compilateur , nous devions donc nous assurer que chaque potentielle problème devait être testé.Concrètement nous prenions une règle et essayions de comprendre ce qui devait être vérifier , dons essayions donc de créé un programme déca qui utilise cette règle , Pour chaque règles, nous faisions deux tests , un programmes valides pour s'assurer que l'on avait la bonne sortie et un programme invalide pour s'assurer que le compilateur vérifiait et traitait bien le problème (en renvoyant une erreur contenant le numéro de règle et en expliquant pourquoi le programme est invalide) .Ci dessous un exemple de la procédure que nous utilisions:

- **Etape 1:analyse d'une règle (3.24)**
  On voit ici que que cette règle concerne le return et doit vérifié que que le type retourné ne doit pas être 'void' et doit retourné correctement le type et la valeur de la variable.
- **Etape 2: Trouver un code qui permettrait d'entrer en conflits avec cette règle et un autre pour passer correctement par cette règle**
  après analyse de la règle ,il est évident qu'il suffit de retourné une variable lorsque le type de la méthode est 'void' pour générer une erreur. De plus pour passer correctement par cette règle , il faut retourné tout les types un peut faire un programme qui return un type générique et un autre qui retourne un type que l'on a créé.

```
1 public class A {
2             void method(int i ){
3                 return i ;
4             }
5         }
```

Listing 2: exemple erreur

```
1 public class A {
2             int method(int i ){
3                 return i ;
4             }
5         }
```

Listing 3: programme valid type générique

```
public class A {
           int a ;
}
public class B {
    A attribut;
    A getA(){
        return this.attribut;
    }
}
```

Listing 4: programme valid renvoie une classe

- **Etape 3: création des test et organisation**
  Une fois la règle bien comprise et que l'on a nos idées pour la tester , on se rend dans le dossier tests de notre projets , y ajoutons dans test/valide nos deux fichiers déca censé être valide et dans test/invalide notre programme censé produire une erreur nous exécutons ensuite nos script et vérifions que tout c'est bien passé ou si cela c'est mal passé cherchons a comprendre pourquoi et corrigeons notre code.

Dans un second temps, nous nous sommes beaucoup servis de Jacoco, afin de savoir plus précisement quelle partie du code à été testée. Ainsi nous avons vu les fonctions et branches du code dans lesquelles nous ne passions pas et avons pu corriger cela. Nous avons donc rajouter des tests afin de tester ces bouts de codes dans l'objectif d'optimiser notre taux de couverture.

Nous nous sommes par exemple rendu compte que nous n'avions jamais testé LEQ ou GEQ donc nous avons fait un test spécifique :

```
class TestInequality {
    void test() {
        int a = 5;
        int b = 10;
        if (a <= b) {
           println(a + " est inf rieur ou egal    " + b);
        }
        if (b >= a) {
            println(b + " est sup rieur ou egal    " + a);
        }
    }
}


{
    TestInequality test = new TestInequality();
    test.test();
}
```

Listing 5: LEQ et GEQ

Après ces tests, nous avons constaté que le code passait bien par les fonctions LEQ et GEQ de notre compilateur et ainsi nous avons amélioré notre taux de couverture.

3.1.1 Tests du lexer

3.1.2 Tests du parser

L'analyse contextuelle est une phase cruciale dans le processus de compilation, servant à vérifier la cohérence et la conformité du code source avec les règles sémantiques du langage Deca. Cette étape suit l'analyse syntaxique et précède la génération du code assembleur. Son rôle est d'assurer que le code Deca, bien que syntaxiquement correct, respecte également les contraintes sémantiques telles que la portée des variables, les types de données, et les règles de conversion de type.

Ci-après nous détaillerons la méthodologie adoptée pour mener à bien nos tests. Nous avons procédé de manière systèmatique.

- Identification des Règles Contextuelles :
  Nous avons commencé par identifier toutes les règles contextuelles pertinentes du langage Deca, telles que définies dans la spécification du langage. Ces règles incluent, mais ne sont pas limitées à, la vérification des types, la portée des identifiants, et les règles de surcharge des méthodes.

- Création de cas de Test :
  Pour chaque règle identifiée, nous avons créé des cas de test spécifiques. Ces tests étaient conçus pour couvrir tous les scénarios possibles. Nous avons donc créé deux types de tests, les tests valides qui vérifient que le compilateur fonctionne avec un programme correct, et les test invalides qui vérifient que le compilateur renvoie bien une erreur dans le cas d'une erreur contextuelle. Par exemple, pour tester la règle de portée des variables, nous avons écrit des programmes où les variables étaient déclarées dans différents blocs et contextes, afin de vérifier que le compilateur détecte correctement les erreurs de portée. Les test suivants vérifient la règle concernant la déclaration et l'utilisation de variables dans un bloc de code.

```
// Test pour la regle 3.74
// Date: 18/01/2024
// Auteur: Jerome

class ClasseAvecMethodes {
    void afficherMultiples(int a, int b, int c) {
        println("a: " + a + ", b: " + b + ", c: " + c);
    }
}

class TestListeExpressions {
    void executionTest() {
        ClasseAvecMethodes objet = new ClasseAvecMethodes();

        // Appel de methode avec une liste d'expressions correspondant a la
           signature
        objet.afficherMultiples(1, 2, 3);
    }
```

```
}

// Programme principal
{
    TestListeExpressions testProgramme = new TestListeExpressions();
    testProgramme.executionTest();
}
```

Listing 6: Test Valide

```
// Resultat erreur:
//    Types incompatibles de parametres dans un appel de methode (regle 3.74)
//
// Description:
//    Tests de programme incorrect contextuellement (types incompatibles de
    parametres)
//
// Auteur:
//    Jerome
//
// Historique:
//    Cree le 16/01/2024

class MaClasse {
    int maMethode(int param) {
        return param * 2;
    }
}

class Test {
    int testMethod() {
        MaClasse obj = new MaClasse();
        return obj.maMethode("texte");
    }
}
```

Listing 7: Test Invalid 1

```
// Resultat erreur:
//    Nombre incorrect de parametres dans un appel de methode (regle 3.74)
//
// Description:
//    Tests de programme incorrect contextuellement (nombre incorrect de
    parametres)
//
// Auteur:
//    Jerome
//
// Historique:
```

```
//    Cree le 16/01/2024

class MaClasse {
    int maMethode(int param) {
        return param * 2;
    }
}

class Test {
    int testMethod() {
        MaClasse obj = new MaClasse();
        return obj.maMethode(10, 20);
    }
}
```

Listing 8: Test Invalid 2

Analyse des Résultats
Après avoir exécuté ces tests, nous avons analysé les sorties du compilateur. Les cas où le compilateur a correctement identifié les erreurs contextuelles ont validé l'efficacité de l'analyse contextuelle. Les cas où le compilateur n'a pas détecté les erreurs attendues ont été utilisés pour affiner notre implémentation, en s'assurant que chaque aspect de l'analyse contextuelle est conforme aux spécifications du langage Deca. Ces test vérifient si le compilateur détecte correctement les erreurs de redéclaration de variables dans le même bloc (règle 3.74)

Utilisation de `test_synt` et `test_context` : Pour faciliter l'analyse des résultats, nous avons utilisé ces scripts tests fournis. Cela nous a permis de visualiser les arbres abstraits décoré générés par le compilateur, offrant une compréhension claire de la manière dont le compilateur interprète chaque construction du langage Deca.

3.1.3 Tests de la génération de code

3.2 Scripts de tests

Après avoir créé une grande base de fichiers tests en deca (valides ou invalides), il s'agit de vérifier comment le compilateur réagit avec chacun d'entre eux. Le faire un par un serait bien trop fastidieux, c'est pourquoi nous avons créé des scripts en Bash vérifiant tout cela pour nous. Dans chaque dossier de test se trouve un fichier README.md fournissant plus d'onformations sur la forme des fichiers Deca tests à adopter.

3.2.1 Organisation des scripts

Les scripts se répartissent dans 3 dossiers différents:

- Dans le dossier `launchers`, sont fournis des sripts permettant d'abord compiler le code Java du compilateur, puis de lancer le compilateur sur un fichier test Deca renseigné. Chaque script du dossier permet de lancer le compilateur jusqu'à une certaine étape (analyse lexical, syntaxique, et contextuelle). Notons que pour tester le compilateur dans sa totalité nous utiliserons directement la commande `decac` de `src/main/bin`.

Figure 4: Début d'exécution de `run_all_tests.sh`

- Dans le dossier `verification_scripts` , nous avons plusieurs scripts qui vérifient la sortie du compilateur lancé sur un programme Deca. Par exemple, `verif_echec_test.sh` vérifie que l'exécution d'un fichier test a bien échoué à la bonne position en renvoyant un certain message d'erreur. Nous reviendrons plus en détail sur les scripts de vérification ce dossier. Le script `program_on_deca_folder` permets de lancer ce genre de script de vérification sur un ensemble de fichiers Deca d'un certain répertoire.
- Dans le dossier `tests_runners` , nous avons les scripts de plus haut niveau qui lancent les vérifications de sortie du compilateur lancé sur les fichiers valides ou invalides d'une certaine étape. Par exemple, `stepB_invalid.sh` vérifie que les fichiers tests Deca du répertoire `src/test/deca/context/invalid/created` sont bien invalides pour l'étape B (donc Deca-invalides), et que position et message d'erreur attendus sont bien détectés.

L'exécution de `run_all_tests` (représentation en figure 4) permets de lancer tous les tests et d'avoir un résultat très visuel de l'avancement dans l'implémentation des fonctionnalités du compilateur.

3.2.2 Fichiers Deca invalides

Pour vérifier la sortie du compilateur lancé sur n'importe quel fichier invalide, nous proposons le script `verif_echec_test.sh` . Pour le bon fonctionnement de ce test, il faut renseigner en commentaire du fichier Deca les informations suivantes (exemple donné en listing 28):

- la position de l'erreur en ligne 2,
- le message d'erreur que l'on est censé avoir en en sortie du compilateur en ligne 3.

Si la sortie du compilateur ne correspond pas aux attendes, `verif_echec_test.sh` renvoie une erreur.

```
1  // Resultat erreur:
2  //    8:12
3  //    Condition non booleenne dans 'if' (regle 3.22)
4
5  class MaClasse {
6      void maMethode() {
7          int x = 5;
8          if (x) {
9          }
10     }
11 }
```

Listing 9: Exemple de programme invalide

3.2.3 Fichiers Deca valides

Pour les fichiers Deca valides, nous avons plusieurs manières de vérifier la sortie du compilateur.

- Pour tester l'analyse lexicale, `verif_succes_test_lex.sh` permet de vérifier la détection des tokens d'un programme Deca (on renseigne les tokens en commentaires dans le fichier Deca, cf README correspondant pour plus d'infos);
- Pour tester l'analyse syntaxique et contextuelle, nous avons plusieurs méthodes. Tout d'abord, `verif_tree` nous permets de vérifier la forme de l'arbre abstrait, qu'il soit décoré (analyse contextuelle) ou pas (analyse syntaxique). On doit renseigner des informations sur les noeuds attendus ainsi que leurs positions pour valider cette vérification (cf README correspondant). Cependant, cette méthode est très fastidieuse car les arbres deviennent vite très grands. Nous avons opté dans un second temps pour un méthode moins sûre, mais beaucoup efficace: grâce au script `verif_succes_basic.sh` nous ne vérifions pas la forme des arbres en sorte de ces étapes, nous vérifions seulement si le test lancé se termine sans erreur;
- Enfin, pour tester la génération de code, nous avons recours à `verif_excution.sh` . Ce script lance le compilateur sur un fichier Deca "fichier.deca" (censé être Deca-valide), et exécute avec `ima` le code assembleur généré "fichier.ass". On compare le résultat de cette exécution avec un fichier "fichier.exp", qui contient le résultat attendu de l'exécution.

3.3 Gestion des risques et des rendus

Dans le cadre de notre objectif de rendre un compilateur zéro défaut, et plus largement afin de réaliser un projet "zéro-défaut", nous nous sommes réunis afin de répertorier les risques que nous pourrions rencontrer concernant les rendus et comment nous devons y faire face.

3.3.1 Risques génériques

- Retard dans le rendu du projet et non respect des deadlines
  - Etablir un planning réaliste avec des marges de sécurité
  - Prendre en compte tous les membres et leurs spécialisations
  - Suivre régulièrement l'avancement du projet et ajuster en conséquence
- Manque de ressources humaines
  - Identifier les compétences de chacun le plus tôt possible
  - Prévoir plusieurs personnes sur chacune des facettes du projet pour assurer un back-up en cas d'absence imprévue
  - Respecter les délais le plus possible et être conscient qu'un décalage vis à vis du planning compromet le rendu du projet à la date imposée par le client
- Problèmes de communication externe et interne
  - Mise en place de canaux de communication clairs
  - Organisation de réunions régulières pour s'assurer que tous les membres soient informés des progrès et problèmes potentiels
- Problèmes de qualité des rendus et suivis
  - Mise en place de processus de test rigoureux au cours de chaque sprint
  - Organisation de réunions régulières pour s'assurer que tous les membres soient informés des progrès et problèmes potentiels potentiels
- Découragement d'un ou plusieurs membres du groupe au fur et à mesure que le projet avance
  - S'assurer du bien-être de chacun dans son travail
  - Ne pas se surmener trop tôt dans le projet, au profit d'un rendu intermédiaire, mais au détriment de la fin du projet

- Dépendance excessive envers un membre du projet, le projet repose trop sur une ou deux personnes
  - Distribuer équitablement la charge de travail et les responsabilités
  - Mettre en place des plans de secours pour les rôles clés
  - Encourager l'entraide et le partage de connaissance au sein de l'équipe
- Problème de conflits entre les membres du groupe
  - En discuter dans la charte d'équipe et discuter de solutions potentiels
  - Encourager la communication pour résoudre les problèmes efficacement

3.3.2 Risques techniques

- Problèmes d'exécution du programme lors d'un rendu
  - Cloner le git sur lequel on travaille
  - Compiler le programme issu directement du dépôt partagé et y effectuer les tests
- Casser le git
  - Communiquer avant d'effectuer une manœuvre importante sur git
  - Participation à l'amphi Git de tous les membres pour optimiser nos compétences et éviter toute erreur
- Ne pas prendre en compte la régression de notre projet vis à vis des tests: faire attention à ne pas dé-valider des tests acquis auparavant, au profit de tests «plus poussés»
  - Écrire des tests de non-régression. Ces tests seront exécutés lors du push, et le code ne sera pas push si les tests ne passent pas.
  - Organisation de réunions régulières pour s'assurer que tous les membres soient informés des progrès et problèmes potentiels
- Placer des programmes test dans un mauvais répertoire
  - S'assurer de la bonne exécution des programmes test régulièrement en lançant le script `run_all_test.sh`
  - Prendre soin de vérifier l'emplacement d'un programme test lorsqu'on est amené à le modifier
- Écrire des programmes tests qui ne relèvent pas l'erreur attendu par manque d'attention

  - Faire preuve de rigueur lors de l'écriture des programmes tests en prenant de soin de vérifier qu'il respecte la syntaxe déca
  - Prendre le temps de vérifier le résultat voulu lorsque celui-ci est nécessaire pour l'exécution des tests
  - Faire attention aux `printlnx()` et `printx()` qui affichent des quantités hexadécimales.

- Écrire des programmes tests qui ne relèvent pas l'erreur attendu par manque d'attention
  - Faire preuve de rigueur lors de l'écriture des programmes tests en prenant de soin de vérifier qu'il respecte la syntaxe déca
  - Prendre le temps de vérifier le résultat voulu lorsque celui-ci est nécessaire pour l'exécution des tests
  - Faire attention aux `printlnx()` et `printx()` qui affichent des quantités hexadécimales.

- Le retard d'un membre bloque l'avancée d'un autre à cause de la dépendance des étapes
  - Maintenir une bonne communication concernant l'avancée de chaque étape pour ne pas être surpris qu'une tâche avant moins vite qu'imaginé
  - Demander du soutien, déléguer une sous-tâche pour avancer et terminer plus vite

- S'éparpiller sur les programmes de test et les scripts
  - Trouver l'équilibres entre exhaustivité des tests et perte de temps
  - Établir des scripts de test fonctionnels le plus vite possible sans trop vouloir peaufiner le résultat

3.4 Résultats de Jacoco

Nous avons réusssi à avoir un taux de couverture total de 58% sur l'ensemble du code , Nous avons testé principalement la classe DecaParseur car c'est celle que l'on devait implémenter et on est partit du principe que tout le code fournis dans le projet comme la classe (Decompile) était correct , de plus comme nous n'avons pas pu implémenter la génération de code pour la partie C, la plupart des fonctions au dessus ne sont évidement pas testé , infine , c'est surtout la classe DecaParser que nous avons testé .

Nous pouvons constater que la plupart de la couverture manquante proviens de la classe inequality_expr(int) ,nous avons faits un zoom juste dessous pour voir ce que nous avions pu oublié et finalement il d'avère que ce sont les expressions this et instanceof que nous n'avons pas implémenté donc nous estimons qu'on a couvert ce qui été à couvrir pu .

Pour finir la non couverture de la plupart des classes de fr.ensimag.tree proviens des classes liés à la génération de code

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.
fr.ensimag.deca.syntax		69%		52%
fr.ensimag.deca.tree		55%		61%
fr.ensimag.deca		20%		14%
fr.ensimag.ima.pseudocode		1%		0%
fr.ensimag.ima.pseudocode.instructions		0%		n/a
fr.ensimag.deca.codegen		14%		0%
fr.ensimag.deca.context		76%		67%
fr.ensimag.deca.tools		35%		33%
Total	7,877 of 18,965	58%	505 of 981	48%

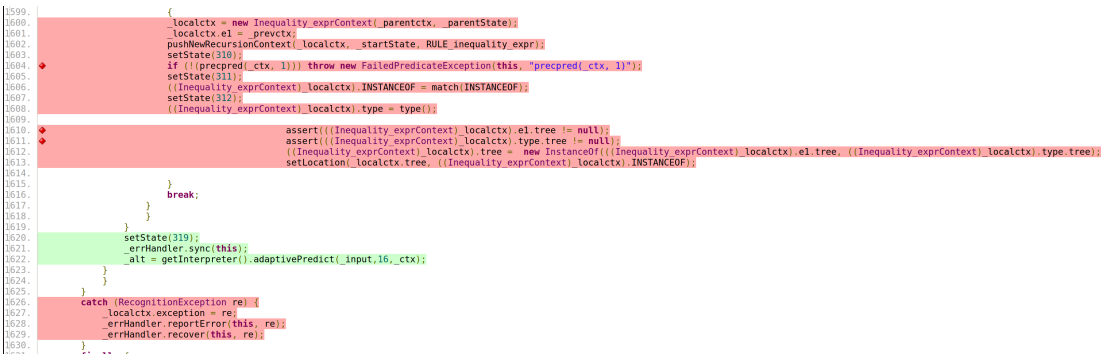
Figure 5: pourcentage de couverture de la classe DecaCompiler

Deca Compiler > fr.ensimag.deca.syntax > DecaParser

DecaParser

Element	Missed Instructions	Cov.	Missed Branches	Cov.
inequality_expr(int)		71%		47%
mult_expr(int)		83%		56%
primary_expr()		73%		51%
inst()		89%		63%
eq_neq_expr(int)		84%		56%
sum_expr(int)		84%		56%
decl_method()		64%		47%
sempred(RuleContext, int, int)		0%		0%
select_expr(int)		87%		61%
literal()		76%		60%
or_expr(int)		83%		56%
and_expr(int)		83%		56%
multi_line_string()		0%		0%
literal_sempred(DecaParser.LiteralContext, int)		0%		0%
inequality_expr_sempred(DecaParser.Inequality_exprContext, int)		0%		0%
class_decl()		78%		50%
unary_expr()		83%		60%
decl_field(Visibility, AbstractIdentifier)		82%		60%
mult_expr_sempred(DecaParser.Mult_exprContext, int)		0%		0%

Figure 6: zoom sur DecaParser



```
1599.                     {
1600.                     _localctx = new Inequality_exprContext(_parentctx, _parentState);
1601.                     _localctx.e1 = _prevctx;
1602.                     pushNewRecursionContext(_localctx, _startState, RULE_inequality_expr);
1603.                     setState(310);
1604.                     if (!(precpred(_ctx, 1))) throw new FailedPredicateException(this, "precpred(_ctx, 1)");
1605.                     setState(311);
1606.                     ((Inequality_exprContext)_localctx).INSTANCEOF = match(INSTANCEOF);
1607.                     setState(312);
1608.                     ((Inequality_exprContext)_localctx).type = type();
1609.
1610.                                             assert(((Inequality_exprContext)_localctx).e1.tree != null);
1611.                                             assert(((Inequality_exprContext)_localctx).type.tree != null);
1612.                                             ((Inequality_exprContext)_localctx).tree =  new InstanceOf(((Inequality_exprContext)_localctx).e1.tree, ((Inequality_exprContext)_localctx).type.tree);
1613.                                             setLocation(_localctx.tree, ((Inequality_exprContext)_localctx).INSTANCEOF);
1614.
1615.                     }
1616.                     break;
1617.                 }
1618.                 }
1619.             }
1620.             setState(319);
1621.             _errHandler.sync(this);
1622.             _alt = getInterpreter().adaptivePredict(_input,16,_ctx);
1623.         }
1624.         }
1625.     }
1626.     catch (RecognitionException re) {
1627.         _localctx.exception = re;
1628.         _errHandler.reportError(this, re);
1629.         _errHandler.recover(this, re);
1630.     }
```

Figure 7: code manquant dans inequality expr(int) DecaParser

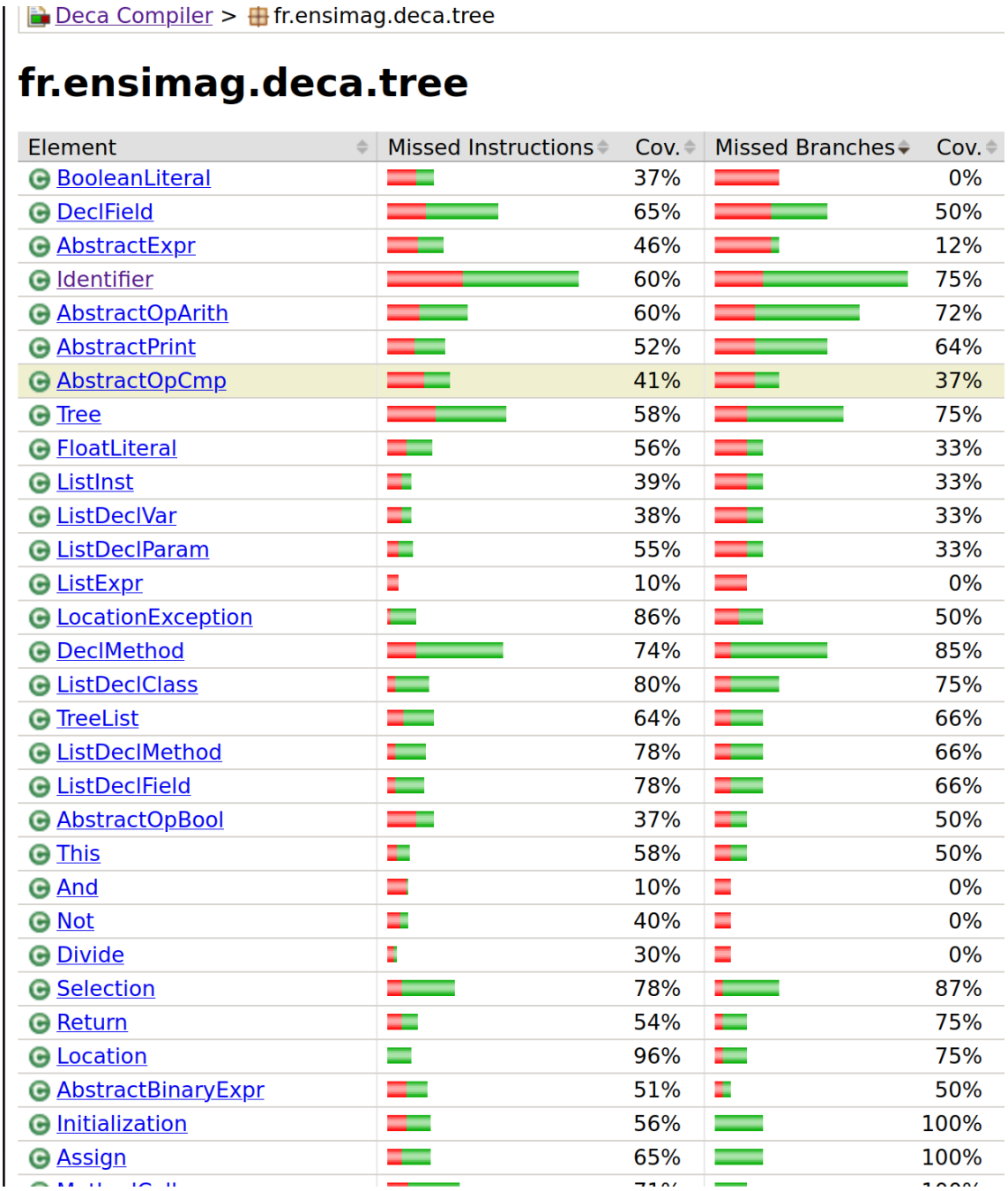


Figure 8: zoom sur Deca.tree

3.5 Démarche à suivre pour enrichir la base de tests

Plus concrètement, tous les fichiers tests ont été écrits dans le dossier `src/test/deca`. Ce dossier est subdivisé, en fonction de si les tests portent sur : le lexer (`lexicon`), la syntaxe (`syntax`), l'analyse contextuelle (`context`) ou la conversion en langage assembleur (`codegen`). Des explications sur les méthodes de tests adoptées sont présentes dans les ReadMe, dans chacun de ces dossiers.

Il est important de prendre soin à la pertinence de chacun des tests ajoutés. En particulier, ne pas rajouter de tests vérifiant les mêmes choses qu'un programme déjà existant. Pour éviter cela, les programmes ont été rangés minutieusement dans des sous-dossiers par thème, et les noms choisis doivent être explicites afin de comprendre directement de quoi il s'agit. Il est préférable d'ajouter un en-tête expliquant le but du test, conformément à ce qui a déjà été fait.

L'utilisation de Jacoco peut s'avérer pertinente afin de déterminer quels sont les manques de notre répertoire test vis à vis de la couverture du programme.

4 Documentation sur l'extension TRIGO

4.1 Introduction

Cette section présente une extension possible du compilateur Deca, en l'occurrence un ajout concernant les fonctions trigonométriques. L'objectif est de créer une bibliothèque des fonctions trigonométriques de base pour calculer le plus précisément possible des résultats de cos, sin, atan, asin (et ulp). Bien que réaliser les calculs avec une précision parfaite pour toutes les valeurs d'entrée soit un exercice très difficile, nous essaierons d'y parvenir pour chaque fonction pour des sous-domaines du domaine de définition.

4.2 Choix de conception

4.2.1 Les constantes

Pour le bon fonctionnement des méthode trigonométriques nous stockons au préalable quelques constantes afin de gagner en précision. C'est le cas par exemple de $\pi, 2\pi, \frac{\pi}{2} et \frac{\pi}{4}$

On stocke également les valeurs du plus grand et du plus petit flottant représentable en déca, ces valeurs servent au bon comportement de la fonction ulp.

4.2.2 Fonctions supplémentaires

Afin de rendre le code plus lisible et de factoriser le code, nous avons introduit plusieurs fonctions supplémentaires.

```
float _factorielle(int n) {
        if (n == 0 || n == 1) {
            return 1;
        } else {
            return n * _factorielle(n - 1);
        }
    }
```

Listing 10: factorielle

Tout d'abord la fonction factorielle qui est très utile dans le développement en série entière des fonctions trigonométrique

```
float _factorielleParite(int n) {
        if (n == 0 || n == 1) {
            return 1f;
        } else {
            return n * _factorielle(n - 2);
        }
    }
```

Listing 11: factorielle parité

La fonction "factorielleParite" calcul le produit d'un entier sur deux de 0 à n ou de 1 à n selon si n est pair ou impair. Cette méthode est utile pour calculer l' arcsinus.

```
float _puissance(float f, int n){
        if (n == 0){
            return 1;
        }
        if (n > 0){
            return f*_puissance(f,n-1);
        }
        return _puissance(f, n+1) / f;
    }
```

Listing 12: puissance

Une fonction qui calcule f puissance n

```
int _puissanceDeMoinsUn (int n){
        if (n % 2 == 0){
            return 1;
        } else {
            return -1;
        }
    }
```

Listing 13: puissance de -1

Dans notre extension, nous avons souvent besoin de puissances de (-1) donc nous avons créé une méthode spéciale permettant de ne pas avoir à calculer à chaque appelle la puissance. Cette méthode regarde simplement dans -1^n la parité de l'exposant pour en déduire si le résultat est 1 ou -1.

4.2.3 ULP

```
float ulp(float f) {

        float puissance_de_deux = 1;
        if (f<0) {
            f = -f;       // car ulp(f) = ulp(-f)
        }
        if (f == 0){
            return MIN_VALUE;
        }
        if (f == MAX_VALUE){
                return _puissance(2, 104);  //104=127-23
        }
        if (f < 1){
            while (f < puissance_de_deux){
                puissance_de_deux = puissance_de_deux / 2;
            }
            return puissance_de_deux * _puissance(2, -23);
        }
```

```
        while (f >= puissance_de_deux){
            puissance_de_deux = puissance_de_deux * 2;
        }
        return puissance_de_deux * _puissance(2, -22);
    }
```

Listing 14: implémentation de la fonction ulp

On cherche a tel que $2^{a+23} \leq x < 2^{a+24}$

D'après la spécification ulp(-f) = ulp(f). D'après la spécification, ulp(0) = "la plus petite valeur représentable en flottant".

4.2.4 Fonctions trigonométriques

Méthode générale

Pour les fonctions trigonométriques, nous avons choisi comme approche de nous baser sur le développement en série entière de ces fonctions. Dans les fonctions, la méthode de sommation peut sembler étrange, en effet pour avoir le moins de perte de précision possible, nous sommons selon la méthode de Ruffini-Horner

Actuellement, les fonctions trigonométriques inversées asin et atan sont moins précises que les fonctions cosinus et sinus. A la recherche d'une précision toujours accrue, une méthode aurait pu être une recherche dichotomique de y tel que

$$sin(y) = x$$

pour trouver alors

$$asin(x) = y$$

Néanmoins, après une étude théorique, nous avons conclu que le gain de précision de cette recherche dichotomique n'était pas rentable comparé à l'augmentation de la complexité engendrée. Il serait effectivement nécessaire d'appeler un grand nombre de fois la fonction sin ce qui multiplierait la complexité de la fonction asin par un ordre de grandeur d'environ 10^2.

fonctions cosinus et sinus

```
float cos(float theta) {
        float resultat;
        if (theta > PI) {
            while (theta > PI){
                theta = theta - DEUX_PI;
            }
        }
        if (theta < -PI) {
            while (theta < -PI){
                theta = theta + DEUX_PI;
            }
        }                                   //Maintenant -PI <= theta <= PI
        if (theta > 0){
            if (theta > DEMI_PI){
```

```
                resultat = -_sinTaylor(theta - DEMI_PI);
            }
            else{
                resultat = _cosTaylor(theta);
            }
        }
        else if (theta < 0){
            if (theta < -DEMI_PI){
                resultat = -_cosTaylor(theta + PI);
            }
            else{
                resultat = _sinTaylor(theta + DEMI_PI);
            }
        }
        else resultat = 0;
        if (resultat < -1f){
            resultat = -1;
        }
        if (resultat > 1f){
            resultat = 1;
        }
        return resultat;
    }
```

Listing 15: implémentation du cosinus

```
float sin(float theta) {
        float resultat;
        if (theta > PI) {
            while (theta > PI){
                theta = theta - DEUX_PI;
            }
        }
        if (theta < -PI) {
            while (theta < -PI){
                theta = theta + DEUX_PI;
            }
        }                           //Maintenant -PI <= theta <= PI
        if (theta > 0){
            if (theta > DEMI_PI){
                resultat = -_cosTaylor(theta - DEMI_PI);
            }
            else{
                resultat = _sinTaylor(theta);
            }
        }
        else if (theta < 0){
            if (theta < -DEMI_PI){
                resultat = -_sinTaylor(theta + PI);
```

```
            }
            else{
                resultat = -_cosTaylor(theta + DEMI_PI);
            }
        }
        else {
            resultat = 0;
        }
        if (resultat < -1){
            resultat = -1;
        }
        if (resultat > 1){
            resultat = 1;
        }
        return resultat;
    }
```

Listing 16: implémentation du sinus

On commence par se ramener dans l'intervalle $[-\pi;\pi]$ Ensuite en fonction du quart de cercle où on se trouve, on appelle soit la fonction sinus ou cosinus avec un argument compris entre 0 et $\frac{\pi}{2}$

Ensuite on vérifie le codomaine puis on renvoie le résultat

Intéressons nous maintenant au développement en série entière de Taylor:

développement du cosinus

```
float _cosTaylor(float theta){
        int n = 6;
        float somme = 0;
        float theta2 = theta * theta;
        while (n >= 1){
            somme = somme + _puissanceDeMoinsUn(n) / _factorielle(2*n);
            somme = somme * theta2;
            n = n - 1;
        }
        somme = somme + 1;
        return somme;
    }
```

Listing 17: développement du cosinus

Une analyse préalable à permis de montrer que la précision ne s'améliorait plus au delà de sept termes dans la série. On peut observer le développement du cosinus calculé avec la méthode de Horner.

développement du sinus

```
    float _sinTaylor(float theta){
        if (theta > PI_SUR_QUATRE){
            return _cosTaylor(DEMI_PI - theta);
        }
        int n = 5;
```

```
        float somme = 0;
        float theta2 = theta * theta;
        while (n >= 1){
            somme = somme + _puissanceDeMoinsUn(n) / _factorielle(2*n+1);
            somme = somme * theta2;
            n = n - 1;
        }
        somme = somme + 1;
        somme = somme * theta;
        return somme;
    }
```

Listing 18: développement du sinus

Si "theta" est entre $\frac{\pi}{4}$ et $\frac{\pi}{2}$ en entrée, on renvoie le cosinus de $\frac{\pi}{2}$ - theta. Pour plus de précision.

Une analyse préalable à permis de montrer que la précision ne s'améliorait plus au delà de six termes dans la série. On peut observer le développement du sinus calculé avec la méthode de Horner.

fonction arc-sinus

```
float asin(float x) {
        if(x < 0){
            x = -x;
        }
        return _asinTaylor(x);
    }
```

Listing 19: implémentation de l'arc-sinus

On utilise le fait que l'arc-sinus est impaire.

développement de l'arc-sinus

```
float _asinTaylor(float x){
        int n = 17;
        float somme = 0;
        float x2 = x * x;
        while (n >= 1){
            somme = somme + _factorielleParite(2*n-1) / (_factorielleParite
               (2*n) * (2*n+1));
            somme = somme * x2;
            n = n - 1;
        }
        somme = somme + 1;
        somme = somme * x;
        return somme;
    }
```

Listing 20: développement de l'arc-sinus

Une analyse préalable à permis de montrer que la précision ne s'améliorait plus au delà de dix-huit termes dans la série. On peut observer le développement de l'arc-sinus calculé avec la méthode de Horner.

fonction arc-tangente

```
float atan(float f) {
        if (f < -1 || f > 1) {
            if (f > 1){
                return DEMI_PI - _arctanTaylor(1/f);
            }                                   //Car atan(f)=signe(f)*PI/2 -
               atan(1/x)
            else {
                return -DEMI_PI - _arctanTaylor(1/f);
            }
        }

        return _arctanTaylor(f);
    }
```

Listing 21: implémentation de l'arc-tangente

On utilise la propriété de l'arc-tangente: $atan(f) = signe(f) * \frac{\pi}{2} - atan(\frac{x}{)}$.

développement de l'arc-tangente

```
float _arctanTaylor(float x) {
        int n = 100;
        float somme = 0;
        float x2 = x * x;
        while (n >= 1){
            somme = somme + (float)_puissanceDeMoinsUn(n) / (2 * n + 1);
            somme = somme * x2;
            n = n - 1;
        }
        somme = somme + 1;
        somme = somme * x;
        return somme;
```

Listing 22: développement de l'arc-tangente

Une analyse préalable à permis de montrer que la précision ne s'améliorait plus au delà de cent termes dans la série. On peut observer le développement de l'arc-tangente calculé avec la méthode de Horner.

4.2.5 Algorithme de CORDIC

Une méthode longtemps envisagée fut la méthode Cordic qui est utilisé dans les calculatrices scientifiques par exemple pour calculer les fonctions trigonométriques. Mais après de nombreux essais cette méthode c'est révélée moins précise.

4.3 Analyse de la précision

Notations : $M = M_f + \Delta \cdot M$ où on note

- M la valeur réelle
- M_f la valeur flottante
- $\Delta \cdot M$ l'erreur

4.3.1 Fonctions `sin`

```
float _sinTaylor(float theta){
        if (theta > PI_SUR_QUATRE){
            return _cosTaylor(DEMI_PI - theta);
        }
        int n = 5;
        float somme = 0;
        float theta2 = theta * theta;
        while (n >= 1){
            somme = somme + _puissanceDeMoinsUn(n) / _factorielle(2*n+1);
            somme = somme * theta2;
            n = n - 1;
        }
        somme = somme + 1;
        somme = somme * theta;
        return somme;
    }
```

Listing 23: implémentation du sinus

On se place entre $[0, \frac{\pi}{2}]$:

$$\Delta(\theta^2) = 2 \cdot \theta(\Delta \cdot \theta) + \frac{ulp(\theta^2)}{2} = \frac{ulp(\theta^2)}{2} + \theta \cdot \frac{ulp(\theta^2)}{2}$$

d'où

$$\Delta(\theta^2) \leq ulp(\theta) + \frac{ulp(\theta^2)}{2} \leq 1.5 * ulp(\theta)$$

(Car $\theta < 2$)

Boucle : On note I la somme intermédiaire et S la somme.

$$\Delta I_n = \Delta S_{n+1} + \frac{ulp(\frac{1}{(2n+1)!})}{2} + \frac{ulp(I_n)}{2}$$

$$\Delta S_n = S_n \cdot \left(\frac{\Delta I_n}{I_n} + \frac{\Delta(\theta^2)}{\theta^2} \right) + \frac{ulp(S_n)}{2}$$

Initialement, $\Delta I_n = \frac{ulp(I_n)}{2} \leq \frac{2^{-49}}{2} = 2^{-50}$

$$I_n = \frac{1}{(2n+1)!} \quad \text{et} \quad S_n = \frac{\theta^2}{(2n+1)!}$$

$$\Delta S_n = \frac{\theta^2}{(2n+1)!} \cdot \left(2^{-50}(2n+1)! + \frac{\Delta(\theta^2)}{\theta^2}\right) + \frac{ulp(S_n)}{2}$$
$$\approx \frac{\Delta(\theta^2)}{(2n+1)!}$$
$$\leq 2^{-25} \cdot ulp(\theta)$$

On voit que

$$\forall n \in [\![1;5]\!], \Delta I_n \approx \frac{ulp(I_n)}{2}$$

et donc

$$\Delta S_n = \frac{\Delta(\theta^2)}{(2n+1)!}$$

Finalement

$$\Delta S_1 \approx \frac{\Delta(\theta^2)}{6} \leq 0,25 \cdot ulp(\theta)$$

$$\Delta I_0 \leq 0,25 \cdot ulp(\theta) \quad \text{et} \quad \Delta S_0 \leq ulp(\theta^2)$$

Donc

$$S_0 = \sum_{n=0}^{5} (-1)^n \cdot \frac{x^{2n+1}}{(2n+1)!} \pm \Delta S_0$$

Et donc

$$\Delta sin(\theta) \leq \Delta S_0 + \frac{\theta^{13}}{13!} \leq ulp(\theta^2) + \frac{\theta^{13}}{13!}$$

4.3.2 Fonctions cos

```
float _cosTaylor(float theta){
        int n = 6;
        float somme = 0;
        float theta2 = theta * theta;
        while (n >= 1){
            somme = somme + _puissanceDeMoinsUn(n) / _factorielle(2*n);
            somme = somme * theta2;
            n = n - 1;
        }
        somme = somme + 1;
        return somme;
    }
```

Listing 24: implémentation du cosinus

On se place entre $[0, \frac{\pi}{4}]$:

$$\Delta(\theta^2) = 2 \cdot \theta(\Delta \cdot \theta) + \frac{ulp(\theta^2)}{2} = \frac{ulp(\theta^2)}{2} + \theta \cdot \frac{ulp(\theta^2)}{2}$$

d'où

$$\Delta(\theta^2) \leq ulp(\theta) + \frac{ulp(\theta^2)}{2} \leq 1.5 * ulp(\theta)$$

(Car $\theta < 2$)

Boucle : On note I la somme intermédiaire et S la somme.

$$\Delta I_n = \Delta S_{n+1} + \frac{ulp(\frac{1}{2n!})}{2} + \frac{ulp(I_n)}{2}$$

$$\Delta S_n = S_n \cdot \left(\frac{\Delta I_n}{I_n} + \frac{\Delta(\theta^2)}{\theta^2} \right) + \frac{ulp(S_n)}{2}$$

Initialement, $\Delta I_n = \frac{ulp(I_n)}{2} \leq \frac{2^{-51}}{2} = 2^{-52}$

$$I_n = \frac{1}{2n!} \quad \text{et} \quad S_n = \frac{\theta^2}{2n!}$$

$$\Delta S_n = \frac{\theta^2}{2n!} \cdot \left(2^{-52} 2n! + \frac{\Delta(\theta^2)}{\theta^2} \right) + \frac{ulp(S_n)}{2}$$

$$\approx \frac{\Delta(\theta^2)}{(2n+1)!}$$

$$\leq 2^{-27} \cdot ulp(\theta)$$

On voit que

$$\forall n \in [\![2; 6]\!], \Delta I_n \approx \frac{ulp(I_n)}{2}$$

et donc

$$\Delta S_n = \frac{\Delta(\theta^2)}{2n!}$$

Finalement

$$\Delta S_1 \approx \frac{\Delta(\theta^2)}{2!} \leq 0,75 \cdot ulp(\theta)$$

$$\Delta I_0 \leq 0,75 \cdot ulp(\theta) + 2^{-23} \quad \text{et} \quad \Delta S_0 \approx 0.75 ulp(\theta) + ulp(\theta^2) + 2^{-23} \leq 2^{-23}$$

Donc

$$S_0 = \sum_{n=0}^{6} (-1)^n \cdot \frac{x^{2n}}{2n!} \pm \Delta S_0$$

Et donc

$$\Delta cos(\theta) \leq \Delta S_0 + \frac{\theta^{14}}{14!} \leq 2^{-23} + \frac{\theta^{13}}{13!} \approx 2^{-23}$$

4.3.3 Fonctions `atan`

```
float _arctanTaylor(float x) {
        int n = 100;
        float somme = 0;
        float x2 = x * x;
        while (n >= 1){
            somme = somme + (float)_puissanceDeMoinsUn(n) / (2 * n + 1);
            somme = somme * x2;
            n = n - 1;
        }
        somme = somme + 1;
        somme = somme * x;
        return somme;
    }
```

Listing 25: implémentation de l'arctangente

On se place entre $[0,1]$:

$$\Delta(x^2) = 2 \cdot x(\Delta \cdot x) + \frac{ulp(x^2)}{2} = \frac{ulp(x^2)}{2} + x \cdot \frac{ulp(x^2)}{2}$$

d'où

$$\Delta(x^2) \leq ulp(x) + \frac{ulp(x^2)}{2} \leq 1.5 * ulp(x)$$

(Car $x < 2$)

Boucle : On note I la somme intermédiaire et S la somme.

$$\Delta I_n = \Delta S_{n+1} + \frac{ulp(\frac{1}{(2n+1)})}{2} + \frac{ulp(I_n)}{2}$$

$$\Delta S_n = S_n \cdot \left(\frac{\Delta I_n}{I_n} + \frac{\Delta(x^2)}{x^2} \right) + \frac{ulp(S_n)}{2}$$

Initialement, $\Delta I_n = \frac{ulp(I_n)}{2} \leq \frac{2^{-30}}{2} = 2^{-31}$

$$I_n = \frac{1}{(2n+1)} \quad \text{et} \quad S_n = \frac{x^2}{(2n+1)}$$

$$\Delta S_n = \frac{x^2}{(2n+1)} \cdot \left(2^{-50}(2n+1) + \frac{\Delta(x^2)}{x^2} \right) + \frac{ulp(S_n)}{2}$$

$$\approx \frac{\Delta(x^2)}{(2n+1)}$$

$$\leq 2^{-31} \cdot ulp(x)$$

On voit que

$$\forall n \in [\![1;100]\!], \Delta I_n \approx \frac{ulp(I_n)}{2}$$

et donc

$$\Delta S_n = \frac{\Delta(x^2)}{(2n+1)}$$

Finalement

$$\Delta S_1 \approx \frac{\Delta(x^2)}{3} \leq 0.5 \cdot ulp(x)$$

$$\Delta I_0 \leq 0,5 \cdot ulp(x) \quad \text{et} \quad \Delta S_0 \leq ulp(x^2)$$

Donc

$$S_0 = \sum_{n=0}^{100} (-1)^n \cdot \frac{x^{2n+1}}{(2n+1)} \pm \Delta S_0$$

Et donc

$$\Delta sin(x) \leq \Delta S_0 + \frac{x^{201}}{201} \leq ulp(x^2) + \frac{x^{201}}{201}$$

4.3.4 Fonctions `asin`

```
float _asinTaylor(float x){
        int n = 17;
        float somme = 0;
        float x2 = x * x;
        while (n >= 1){
            somme = somme + _factorielleParite(2*n-1) /
                (_factorielleParite(2*n) * (2*n+1));
            somme = somme * x2;
            n = n - 1;
        }
        somme = somme + 1;
        somme = somme * x;
        return somme;
    }
```

Listing 26: implémentation de l'arcsinus

On se place entre $[0,1]$:

$$\Delta(x^2) = 2 \cdot x(\Delta \cdot x) + \frac{ulp(x^2)}{2} = \frac{ulp(x^2)}{2} + x \cdot \frac{ulp(x^2)}{2}$$

d'où

$$\Delta(x^2) \leq ulp(x) + \frac{ulp(x^2)}{2} \leq 1.5 * ulp(x)$$

(Car $x < 2$)

Boucle : On note I la somme intermédiaire et S la somme. On voit que

$$\forall n \in [\![1;17]\!], \Delta I_n \approx \frac{ulp(I_n)}{2}$$

et donc

$$\Delta S_n = \frac{\Delta(x^2)(2n)!}{2^{2n}(n!)^2(2n+1)}$$

Finalement

$$\Delta S_1 \approx \frac{\Delta(x^2)}{6} \leq 0.25 \cdot ulp(x)$$

$$\Delta I_0 \leq 0,5 \cdot ulp(x) \quad \text{et} \quad \Delta S_0 \leq ulp(x^2)$$

Donc

$$S_0 = \sum_{n=0}^{17} 7 \frac{x^{2n+1}(2n)!}{2^{2n}(n!)^2(2n+1)} \pm \Delta S_0$$

Et donc

$$\Delta sin(x) \leq \Delta S_0 + \frac{x^{35}(34)!}{2^{34}(17!)^2(35))} \leq ulp(x^2) + \frac{x^{35}(34)!}{2^{34}(17!)^2(35))}$$

4.3.5 Fonctions `ulp`

```
float ulp(float f) {

        float puissance_de_deux = 1;
        if (f<0) {
            f = -f;       // car ulp(f) = ulp(-f)
        }
        if (f == 0){
            return MIN_VALUE;
        }
        if (f == MAX_VALUE){
                return _puissance(2, 104);  //104=127-23
        }
        if (f < 1){
            while (f < puissance_de_deux){
                puissance_de_deux = puissance_de_deux / 2;
            }
            return puissance_de_deux * _puissance(2, -23);
        }
        while (f >= puissance_de_deux){
            puissance_de_deux = puissance_de_deux * 2;
        }
        return puissance_de_deux * _puissance(2, -22);
    }
```

Listing 27: implémentation de la fonction ulp

On remarque que ulp(x) = a tel que $2^{a+23} \leq x < 2^{a+24}$ C'est ce que l'on fait ici. (On rappelle que ulp(-x) = ulp(x))

4.4 Validation de l'implémentation

4.4.1 ULP

Pour valider l'implémentation de la fonction `ulp` , nous calculons ses résultats pour des antécédents puissances de 2 allant de -150 à +127 (valeurs pour lesquelles `ulp` change) et comparons avec les images de `ulp` de Java.

4.4.2 Fonctions trigonométriques

Afin de valider les performances de précision de nos fonctions trigonométriques, nous fonctionnons comme pour la validation de la fonction `ulp` . Nous comparons nos résultats pour un ensemble linéaire de valeurs choisies dans un intervalle pertinent (lié au domaine de définition des fonctions, ou bien à un domaine auquel on peut se ramener).

Les bilans des fonctions `sin` et `cos` sont donnés en figure 9. Pour le sinus, dans l'intervalle $[0, \frac{\pi}{2}]$, nous avons une très bonne précision: 77% des valeurs sont à la précision maximale, 23% diffèrent d'un ULP. Pour le Cosinus les résultats ne sont pas aussi satisfaisants tout en restant corrects: plus de 90% des valeurs calculées ont strictement moins de 3 ULP d'écart avec la valeur exacte.

Pour les fonctions réciproques `asin` et `atan` , les bilans sont donnés en figure 10. Dans l'intervalle $[0, 1]$, la fonctions arctangente est assez précise: plus de 95% des points de cet intervalle ont une image à strictement moins de 2 ULP d'écart avec la valeur exacte. Pour l'arcsinus, les résultats sont clairement moins convaincants. Dans l'intervalle choisit, seulement 15% des valeurs atteignent la précision maximale, et 40% des valeurs s'en approchent à moins de 10 ULP.

4.5 Bibliographie

Développement en série de Taylor.

Méthode de Ruffini-Horner.

Algorithme de Cordic.

4.6 Conclusion

```
Rapport de comparaison pour Sinus
Fenêtre : [0.0, 1.5707964]
Nombre de points : 10000
ULP non fiables             Nombre d'occurrences          Proportion              Cumul.
0                           7712                          77.1 %                  77.1 %
1                           2288                          22.9 %                  100.0 %

Rapport de comparaison pour Cosinus
Fenêtre : [0.0, 1.5707964]
Nombre de points : 10000
ULP non fiables             Nombre d'occurrences          Proportion              Cumul.
0                           6697                          67.0 %                  67.0 %
1                           2096                          21.0 %                  87.9 %
2                           407                           4.1 %                   92.0 %
3                           203                           2.0 %                   94.0 %
4                           114                           1.1 %                   95.2 %
5                           69                            0.7 %                   95.9 %
6                           49                            0.5 %                   96.3 %
7                           40                            0.4 %                   96.8 %
8                           32                            0.3 %                   97.1 %
9                           35                            0.3 %                   97.4 %
10                          19                            0.2 %                   97.6 %
```

Figure 9: Bilan Sinus et Cosinus

```
Rapport de comparaison pour Asin
Fenêtre : [0.0, 0.5]
Nombre de points : 10000
ULP non fiables             Nombre d'occurrences          Proportion              Cumul.
0                           1575                          15.8 %                  15.8 %
1                           1307                          13.1 %                  28.8 %
2                           268                           2.7 %                   31.5 %
3                           167                           1.7 %                   33.2 %
4                           114                           1.1 %                   34.3 %
5                           91                            0.9 %                   35.2 %
6                           90                            0.9 %                   36.1 %
7                           80                            0.8 %                   36.9 %
8                           58                            0.6 %                   37.5 %
9                           65                            0.6 %                   38.2 %
10                          58                            0.6 %                   38.7 %

Rapport de comparaison pour Atan
Fenêtre : [0.0, 1.0]
Nombre de points : 10000
ULP non fiables             Nombre d'occurrences          Proportion              Cumul.
0                           7587                          75.9 %                  75.9 %
1                           1921                          19.2 %                  95.1 %
2                           29                            0.3 %                   95.4 %
3                           13                            0.1 %                   95.5 %
4                           12                            0.1 %                   95.6 %
5                           7                             0.1 %                   95.7 %
6                           9                             0.1 %                   95.8 %
7                           8                             0.1 %                   95.9 %
8                           4                             0.0 %                   95.9 %
9                           5                             0.1 %                   96.0 %
10                          5                             0.1 %                   96.0 %
```

Figure 10: Bilan Asin et Atan

5 Documentation sur l'impact énergétique du projet

5.1 Introduction

L'écologie étant un des enjeux majeurs de notre génération, dans notre école Grenoble INP - Ensimag nous suivons de nombreux cours visant à sensibiliser à cette cause. Ceux-ci sont d'autant plus importants dans notre cursus lorsque l'on sait que le numérique représente aujourd'hui 4% des émissions de gaz à effet de serre de notre planète (ajouter des choses plus précisément sur l'exécution de programmes). Cette documentation s'inscrit dans ce processus, en effet, ici nous allons expliquer les moyens mis en œuvre pour analyser l'impact énergétique de notre projet et le réduire durant les trois semaines d'implémentation de notre compilateur Deca.

5.2 Evaluation de la consommation énergétique de notre projet

5.2.1 Mise en place

Avant de commencer à optimiser l'impact énergétique d'un projet, il est évidemment important d'être à même de le quantifier. Pour cela nous avons créé un programme qui a recours à la bibliothèque pyhton subprocess. Le but de ce programme est d'abord de récolter une valeur tampon de l'énergie utilisée par le processeur puis de lancer le script voulu et enfin de récupérer à nouveau une valeur de l'énergie. Après cela, une simple soustraction permet de savoir l'énergie dépensée par nos scripts. Nous avons décidé de lancer deux scripts différents qui nous semblaient les plus énergivores parmi ceux que nous avons créés. Ils sont respectivement :

- Le compilateur Deca sur les tests que nous avons créés pour valider l'étape C. Cela permet dans un premier temps d'évaluer nos scripts de tests mais aussi dans un deuxième temps d'évaluer notre compilateur. En effet, l'avantage des tests de l'étape C est que l'intégralité du processus de compilation est traversé ce qui permet d'avoir une vu d'ensemble du compilateur contrairement à des tests sur les étapes A ou B qui ne passeraient tous les deux pas par la création du code assembleur de l'étape C. Plus généralement, nous avons décidé d'analyser la consommation de nos tests car nous pensons que la compilation et l'exécution d'une grande base de tests représente un coût important sur la réalisation du projet.
- L'extension, qui sera compilée en Java et non par notre compilateur car celui-ci n'a pas encore les fonctionnalités nécessaires à sa compilation. De plus, l'extension est la seule partie de notre projet qui était parfaitement libre, donc où les choix de conception sont les plus intéressants. En effet, du côté du compilateur, la conception était très guidée avec une architecture globale qui nous était donnée et que nous devions simplement nous approprier. Au contraire, l'extension partant de zéro, tous les choix sont propre à notre équipe. C'est pourquoi son analyse est d'autant plus importante.

5.2.2 Analyse des donnnées

Voici l'implémentation de notre évaluation de consommation énergétique.

```python
#!/usr/bin/python3

import subprocess

# Current energy counter in micro joules
commande = "cat /sys/class/powercap/intel-rapl/intel-rapl\:0/energy_uj"

res = subprocess.run(commande, shell=True, check=True, stdout=subprocess.PIPE
    , text=True)
val1 = int(res.stdout)
print("Buffer value :")
print(val1)

subprocess.run("src/test/script/tests_runners/run_all_tests.sh > /dev/null",
    shell=True, check=True)

res3 = subprocess.run(commande, shell=True, check=True, stdout=subprocess.
    PIPE, text=True)
val2 = int(res3.stdout)
print("End value :")
print(val2)

# Used energy in micro joules
diff = val2-val1
print("Used energy in micro joules = ", diff)
# Used energy in kiloWatt hour
energy = diff / (3.6*(10**9))
print("Used energy in watt-hour = ", energy)
```

Listing 28: Programme d'évaluation de la consommation énergétique de notre projet

Grâce à cela nous savons que notre compilateur consomme en moyenne 0,3Wh sur notre base de test entière. En estimant que nous l'avons lancé une centaine de fois cela revient à 3Wh à l'échelle du projet ce qui est équivalent à la consommation d'un appareil électroménager en veille durant 1h.

De même la consommation de l'extension sur 10 000 appelle de chaque fonctions trigonométriques est de 0,0007Wh. Il faudrait donc $4 * 10^8$ appelle à une fonction trigonométrique pour arriver à une consommation équivalente à toute la base de tests de notre compilateur.

5.3 Comment en est-on arrivé là ?

5.3.1 Optimisation au niveau de l'extension

Ayant choisi l'extension TRIGO, le but d'un point de vue énergétique s'aligne bien avec les consignes qui sont d'implémenter dans le langage Deca les 4 fonctions trigonométriques principales : cos, sin, asin, atan. Ces quatre fonctions doivent approcher au maximum la précision parfaite pour toutes les valeurs d'entrée. D'un autre côté, la vitesse d'exécution

des fonctions doit être la plus élevée possible, ce qui revient à en diminuer la complexité. Une réduction de la complexité revient évidemment à diminuer l'impact énergétique de notre extension. Dans cet optique, différents choix ont été fait que nous allons exposer ici :

- Pour commencer, dans notre vie quotidienne nous avons l'habitude de dire que chaque petit geste une fois répété un grand nombre de fois compte énormément. Il en est de même avec la façon de coder en informatique où éviter un grand nombre de petits calculs est une façon relativement simple de diminuer la complexité d'un algorithme. Dans notre extension, nous avons souvent besoin de puissances de (-1) donc nous avons créé une méthode spéciale permettant de ne pas avoir à calculer à chaque appelle la puissance. Cette méthode regarde simplement dans $(-1)^n$ la parité de l'exposant pour en déduire si le résultat est 1 ou -1.
- L'implémentation de l'extension a commencé par une période de recherche théorique. Des analyses ont été faites pour les quatre fonctions trigonométriques dans le but de déterminer le nombre minimum de passages pour chacune des boucles.
- Actuellement, les fonctions trigonométriques inversées asin et atan sont moins précises que les fonctions cosinus et sinus. A la recherche d'une précision toujours accrue, une méthode aurait pu être une recherche dichotomique de y tel que

  $$sin(y) = x$$

  pour trouver alors

  $$asin(x) = y$$

  Néanmoins, après une étude théorique, nous avons conclu que le gain de précision de cette recherche dichotomique n'était pas rentable comparé à l'augmentation de la complexité engendrée. Il serait effectivement nécessaire d'appeler un grand nombre de fois la fonction sin ce qui multiplierait la complexité de la fonction asin par un ordre de grandeur d'environ 10^2.

5.3.2 Optimisation au niveau des tests

Au début de la création et validation de nos tests, le but sur des programmes valides était de vérifier non seulement le bon passage à travers le compilateur et donc la bonne création des arbres (ou décoration des arbres pour l'étape B) mais aussi de vérifier si l'arbre créé était le bon. Ce processus était non seulement chronophage pour la création de notre base de tests mais aussi énergivore car le script vérifiait ligne par ligne si l'arbre était identique à l'arbre attendu. C'est pour ces deux raisons que nous avons décidé d'arrêter ces vérifications. Nous sommes passés à un script qui vérifie simplement si un arbre est créé en sortie des étapes A et B pour des programmes valides. Cela signifierait que la compilation s'est bien passé. Du côté de l'étape C nous avons écrit au maximum des tests avec une sortie sous forme de print pour pouvoir vérifier cette sortie que nous connaissons à l'avance. Nous avons pris cette décision sans craintes car si une erreur s'est glissée dans un des arbres, il y a plus de chances de la remarquer à cause d'une étape future qui rencontrerait un défaut due à cette erreur qu'en écrivant l'arbre à la main pour le vérifier. D'autant plus que les chances de se tromper à l'écriture de l'arbre attendu sont grandes.

A posteriori, nous pensons qu'une amélioration possible de nos scripts de tests serait d'utiliser l'option -P du compilateur qui permet de lancer la compilation de plusieurs fichiers de tests à la fois. De cette façon, nous pensons qu'appeler le compilateur une seule fois sur plusieurs tests serait moins énergivore que de l'appeler autant de fois que le nombre de tests à lancer. Néanmoins, ce ne sont que des suppositions car nous avons eu cette idée trop tard et nous n'avons donc pas pu la tester.

5.4 Petits gestes annexes

Dans cette section, nous allons exposer d'autres aspects de l'impact énergétique de notre projet auquel nous avons réfléchi.

Un premier élément important est le transport. En effet, le transport est l'activité qui contribue le plus aux émissions de gaz à effet de serre en France. En 2019, il représente 31% des émissions de notre pays d'après le rapport Secten 2020 de CITEPA. Il nous semblait donc intéressant de préciser que dans notre groupe, nous venions à l'Ensimag tous les jours pour pouvoir mieux travailler en équipe mais 1 personne venait à pied, 2 personnes venaient en vélo non-électrique et les deux derniers venaient en tramway. Nous avons calculé grâce au site www.impactco2.fr, développé par le gouvernement pour sensibiliser la population, que si tous les membres du groupe étaient venus chaque jour en voiture à moteur thermique, nos émissions de gaz à effet de serre auraient été multiplié par 28 simplement sur le transport.

6 Manuel utilisateur

6.1 Spécificités de notre compilateur

Cette section présente notre compilateur sous tous ses aspects, sur son fond (ses avantages et faiblesses) et dans sa forme (utilisation et limites).

6.1.1 Ses avantages

Le compilateur permet de compiler l'ensemble des programmes Deca n'utilisant pas les notions de cast, instance of ainsi que tout ce qui tourne autour des classes. De ce fait, il est capable de convertir tous les fichiers contenant instructions, structures conditionnelles et tout ce que cela implique, à condition que le programme soit correct (syntaxiquement et contextuellement).

De plus, notre compilateur est robuste aux erreurs de l'utilisateur. Les erreurs sont relevées correctement pour tous les programmes n'utilisant pas les notions de cast et instance of (fonctionne aussi pour les programmes avec classe).Dans le cas d'une erreur lexicale (token non reconnu, par exemple) ou syntaxique, le compilateur renvoie un message d'erreur précis avec la position de l'erreur. Sinon, dans le cas d'erreur contextuelle, le compilateur saura indiquer à l'utilisateur l'origine de l'erreur afin de simplifier la correction de celle-ci.

Le compilateur n'utilise pas de registres inutilement. En effet, l'algorithme de gestion de mémoire n'est pas aussi naïf que celui présenté dans le cours vidéo. Par exemple, lors d'une opération arithmétique binaire, il est proposé de systématiquement charger dans un registre les opérandes droite et gauche. Dans notre compilateur, seule l'opérande gauche est chargée, ce qui permet d'économiser de l'espace mémoire.

De plus, la gestion de pile est fonctionnelle (testée par plusieurs programmes), et assure la cohérence des résultats même lorsque beaucoup de registres sont utilisés simultanément.

Notre compilateur est également doté d'une classe Math dans la bibliothèque standard qui permet à l'utilisateur de calculer le sinus, le cosinus, l'arc-sinus, l'arc-tangente et l'ulp d'un flottant. La spécification de la méthode ulp est la même qu'en java. La précision des fonctions trigonométriques sera détaillée dans la documentation propre à l'extension TRIGO.

6.1.2 Ses limites

Comme spécifié dans le paragraphe précédent, ce compilateur ne permet pas de compiler des programmes utilisant des classes.

6.2 Gestion des messages d'erreur

Différentes erreurs sont relevées par le compilateur :

- Les messages d'erreurs provenant du lexer et du parser. Ils renvoient la place de l'erreur dans le fichier Deca ainsi que sa nature.

- Les messages d'erreurs dus au contexte. Ils renvoient le fichier Deca relevant un problème suivi de la localisation de l'erreur ainsi que la nature de celle-ci sous la forme :

```
*fichier.deca:ligne:colonne: Nature du probleme
```

  Pour les erreurs dérivant d'une règle de la syntaxe contextuelle de Deca, celle-ci est spécifiée dans le message d'erreur.

6.3 Utiliser notre compilateur

Pour utiliser le compilateur , il suffit de créer un fichier Deca, puis on utilise le script suivant :

```
$src/main/bin/decac option *fichier.deca
```

Par exemple grâce à la commande :

```
$src/main/bin/decac  src/test/deca/syntax/valid/provided/hello.deca
```

on exécutera le fichier hello.deca sans options.

Ci-après la liste des options disponibles sur notre compilateur :

- -b (banner) : affiche une bannière indiquant le nom de l'équipe (doit être utilisé seule).
- -p (parse) : arrête decac après l'étape de construction de l'arbre, et affiche la décompilation de ce dernier (i.e. s'il n'y a qu'un fichier source à compiler, la sortie doit être un programme deca syntaxiquement correct).
- -v (verification) : arrête decac après l'étape de vérifications (ne produit aucune sortie en l'absence d'erreur).
- -n (no check) : supprime les tests à l'exécution spécifiés dans les points 11.1 et 11.3 de la sémantique de Deca.
- -r X (registers) : limite les registres banalisés disponibles à R0 ... RX-1, avec 4 <= X <= 16.
- -d (debug) : active les traces de debug. Répéter l'option plusieurs fois pour avoir plus de traces.
- -P (parallel) : s'il y a plusieurs fichiers sources, lance la compilation des fichiers en parallèle (pour accélérer la compilation).
- -s (show) : affiche le fichier assembleur généré sur la sortie standard.