Manuel Utilisateur

Projet réalisé par
Boulahfa Adam
Errazki Mohamed
Gaoua Marouane
Lachiri Ilias
Sekkal Amine

Encadré par Mr.Mathias Ramparison et Mr.Tarik Larja

Table des matières

1 Présentation de Decac et fonctionnalité

1.1 Introduction

Le compilateur **Decac** permet de compiler et d'exécuter du code dans le language Deca. Ce langages de programmation orienté objet permet comme en Java ou en C++ de créer des classes, implémenter des méthodes et d'implémenter une bibliothèque.
Un compilateur est un outil essentiel pour convertir un code source écrit dans un langage de haut niveau en un code exécutable par une machine. Le fait que notre compilateur respecte la grammaire et la syntaxe de Deca demande un travail minutieux pour garantir la qualité de notre produit.
Ce travail est important car il permet aux d'écrire des programmes en utilisant un langage de haut niveau qui est plus facile à lire et à comprendre, tout en garantissant que le code généré est efficace pour l'exécution sur une machine.

1.2 Installation du compilateur

1.2.1 Téléchargement des sources

Afin de pouvoir utiliser notre compilateur il suffit de cloner notre dépôt gitlab

```
git clone https://nicolasflamel.ensimag.fr/gl2023/gr7/gl32.git
```

1.2.2 Mise en place de Maven

Si Maven n'est pas installé, il faudra suivre les étaptes suivantes :

1. Allez sur le site web de Maven à l'adresse https ://maven.apache.org/download.cgi
2. Téléchargez la version de maven 3.8.4
3. Décompressez dans le local :

   ```
   tar xzvf apache-maven-3.8.4-bin.tar.gz -C /usr/local
   ```

4. Créez un lien

   ```
   ln -s /usr/local/apache-maven-3.8.4 /usr/local/maven
   ```

5. Créez un script .sh

   ```
   export M2_HOME=/usr/local/maven
   export PATH=${M2_HOME}/bin:${PATH}
   ```

6. Relisez le .sh

   ```
   source /etc/profile.d/maven.sh
   ```

7. Vérifiez la version mvn :

   ```
   mvn -v   // version devrait être 3.8.4
   ```

1.2.3 Mise en place de la machine IMA

Pour pouvoir utiliser le code assembleur généré par le compilateur, vous pouvez utiliser la machine IMA en suivant les étapes suivantes :

```
mkdir global
cd global/
tar xvfz ../Projet_GL/docker/ima_sources.tgz
bin/ima -v
```

ima2023a devra être affichée après l'exécution.

1.2.4 Compilation de nos classes de notre compilateur

Pour pouvoir utiliser le compilateur, suivez les étapes suivante :

```
mvn compile
mvn test-compile  // pour compiler les tests
```

Une fois cela fait, vous pouvez utilise le compilateur.
Remarque : Si un problème persiste dans la mise en place de l'environnement, veuillez consulter le lien suivant contenant plus de détails : https://4mmpgl.gricad-pages.univ-grenoble-alpes.fr/4mmpgl.pages.univ-grenoble-alpes.fr/environnement/machine_perso/

1.3 Utilisation du compilateur Deca

Pour pouvoir utiliser le compilateur, il faudra se déplacer dans le dépôt de test où se trouvent tous les fichiers *test deca* nécessaires (ou vous pouvez créer le votre et bien spécifier le path du fichier deca que vous avez créé).
Une pile de test est mise à disposition afin de pouvoir tester notre compilateur. Cela peut être fait en entrant dans le répertoire de *test context* et puis en exécutant :

```
cd /ensimag/GL/Projet_GL/src/test/deca/context/valid/provided
```

Après, faites un appel au compilateur :

```
decac test_utilisateur.deca
```

Suite à cette commande, un fichier assembleur portant le même nom avec l'extension *.ass* sera généré dans le même répertoire. Le programme assembleur peut être exécuté grâce à la commande :

```
ima test_utilisateur.ass
```

Après cet appel, le message suivant devra être affiché :

```
Bonjour , utilisateur du language Deca !
```

1.4 Utilisation des options du compilateur

Le compilateur *decac* fourni possède plusieurs options de compilation. Pour les afficher, il suffit d'exécuter :

```
decac
```

(Aucun argument ne doit être ajouté après la commande)
Cela permet d'afficher le manuel d'utilisation du compilateur suivant :

```
Comment utiliser Decac  :
decac [[-p | -v] [-n] [-r X] [-d]* [-P] [-w] <fichier deca>...] | [-b]

. -b (banner) : affiche une bannière indiquant le nom de l équipe
. -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 a
  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)
```

Par exemple, la commande suivant affichera les créateurs du compilateur ainsi que le numéro de groupe.

```
decac -b test_utilisateur.deca
```

1.5 Tests des fichiers Deca

Notre compilateur Deca effectue les étapes de vérification syntaxique et contextuelle afin de déterminer si un programme deca est correcte.
Pour vérifier l'analyse syntaxique et contextuelle, on peut utiliser les commandes suivantes :

```
//Attention à bien effectuer un mvn test-compile
test_syn test_utilisateur.deca --> Affiche l'arbre abstrait du code
test_context test_utilisateur.deca --> Affiche l'arbre décoré du code
```

Il est également possible de récupérer les Tokens du programme en effectuant la commande suivante :

```
test_lex test_utilisateur.deca
```

En cas d'erreur, le code Deca fourni est incorrect et ne pourra être compilé.

Si aucune erreur est produite dans les tests précédents, on peut procéder à la génération grâce à la commande decac qui génère le code assembleur. Celui-ci pourra ensuite être exécuté par la commande ima.

1.6 Tests de couverture automatisés

Afin de tester la fiabilité de notre compilateur, vous pouvez par exemple utiliser Jacoco et une pile de Test automatique. Il faudra tout d'abord donnner les droits d'éxecution des script fournis en exécutant la commande suivante sur Terminal :

```
~/ensimag/GL/gl32/src/test/script $ chmod +x *.sh
```

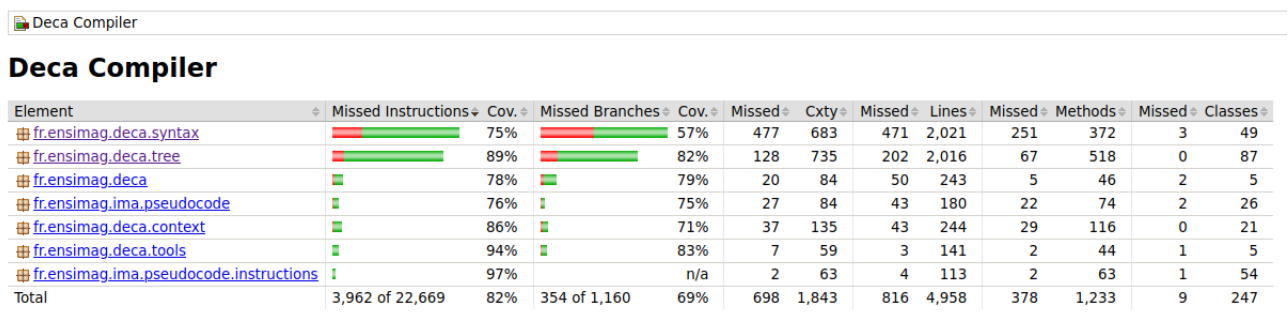
Exemple test-context.sh

```
#!/bin/sh
DEBUT=$(date +%s.%N)
for i in src/test/deca/syntax/invalid/provided/*.deca    # changez ici le chemin pour
                                                         # utiliser vos propres test
do
echo "$i"
test_context "$i" > "${i%.deca}".lis
if [ $? -ne 0 ]
then
    echo  "\033[0;32mTEST CONTEXT ERREUR ATTENDU \033[0m"
else
    echo  "\033[0;31mTEST CONTEXT PASSÉ INNATENDU \033[0m"
fi
done
FIN=$(date +%s.%N)
echo "TEMPS D'EXECUTION TOTAL DES TESTS  : $(echo "$FIN - $DEBUT" | bc) secondes"
```

On peut ainsi automatiser les tests et déterminer la couverture de notre code avec le jeu de test proposé. On peut utiliser Jacoco avec ces commandes suivantes :

```
mvn clean
mvn verify
```

Cela va générer un rapport de couverture dans le répertoire target/site sous le nom *index.html* que vous pouvez ouvrir sur votre navigateur préféré.
Le rapport généré devra ressembler à celui ci-dessous :


Deca Compiler

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		75%		57%	477	683	471	2,021	251	372	3	49
fr.ensimag.deca.tree		89%		82%	128	735	202	2,016	67	518	0	87
fr.ensimag.deca		78%		79%	20	84	50	243	5	46	2	5
fr.ensimag.ima.pseudocode		76%		75%	27	84	43	180	22	74	2	26
fr.ensimag.deca.context		86%		71%	37	135	43	244	29	116	0	21
fr.ensimag.deca.tools		94%		83%	7	59	3	141	2	44	1	5
fr.ensimag.ima.pseudocode.instructions		97%		n/a	2	63	4	113	2	63	1	54
Total	3,962 of 22,669	82%	354 of 1,160	69%	698	1,843	816	4,958	378	1,233	9	247

2 Messages d'erreur

Notre compilateur gère des erreurs de différents types.

2.1 Erreur d'entrée de fichier Decac :

L'erreur suivante est levée lorsqu'on exécute *decac* sans lui donner un fichier en argument :

```
Error during option parsing:
Pas de fichier a compiler
```

Si jamais on lui donne un fichier non deca, c'est l'erreur suivante est levée :

```
Error during option parsing:
Le fichier entré est pas un .deca
```

2.2 Erreur de contexte :

Lors de la génération de notre arbre décoré on doit prendre , en compte le contexte des opérandes ; instruction et variable qu'on utilise c'est pour cela que plusieurs condition de validité ont été mis en place. A chaque manquement de l'une de ces conditions dans un code , un message d'erreur est automatiquement généré par une exception.

- Non defined type
  Explication : Une délcaration de variable n'est pas correct, en effet son type n'appartient pas à l'environement.
  Les types accéptés sont : int , float , boolean , string ...

- Print only accepts strings, ints and floats
  Explication : Le type de l'expression contenu au sein de l'instruction n'est pas un type valable.
  Les types valables sont : int,float,string
  Exemple de type non accepté : println(true). Il s'agit d'un booléen

- Right value doesn't match the expected type
  Explication : Assignement ou initialisation de deux variables de types incompatibles.
  Exemple de cas correct : <**Type1**>Variable1 = <**Type1**>Variable2
  Exemple de cas incorrect : <**Type1**>Variable1 = <**Type2**>Variable2

- Variable already defined
  Explication : Lors de la décalartion d'une variable , une redéfinition a été commise.
  Exemple de cas incorrect :

```
int x = 2;
int x = 5;
```

- Expecting a class
  Explication : Ecrire un a.x nécessite que soit de type classe.

- Cannot cast a void type <**Type1**> <**Type2**>
  Explication : Le cast ne s'applique au type void.

- Cannot be applied to this type <**Type1**> <**Type2**>
  Explication : L'opérande utilisé ne peut pas être utiliser avec deux expressions de type différents.
  Exemple de cas incorrect :

```
int x = 5 ;
if ( 5 == true ) {
println("ceci est un code incorrect");
}
```

- Selection expects a field
  Explication :

- Can only be applied to ints
  Explication : L'oppérande utilisé ne peut être utiliser qu'avec des expressions d'un type donné.

- New is used on classes
  Explication : L'opértande *New* a été effectué sur une expression qui n'était pas une classe.
  Exemple de cas incorrect :

```
int x = new int()
```

- Cannot apply the NotOperator to this type
  Explication : L'oppérande "!" utilisé ne peut être utiliser qu'avec des expressions de type booléen

- Can be applied only to booleans
  Explication : L'oppérande utilisé ne peut être utiliser qu'avec des expressions de type booléen

- Comparaisons only applied to ints & floats
  Explication : L'oppérande utilisé ne peut être utiliser qu'avec des expressions de type int/float

- "You can't use void as a type",
  Explication : Impossible de déclarer une variable ou paramètre en tant que void.

- In order to return, return type should be different than void
  Explication : On effectue un return sur une méthode qui est censé de rien retourner (ie type VOID)
  Exemple de cas incorrect :

```
class A {
 int x;
 void setX(int x) {return x;}
}
```

- Selection exp must be a class
  Explication :

- Can't acces this field from main
  Explication : Un appel a été fait sur un attribut ayant une visibilité *Protected* , cela ne devrait pas arriver.

```
class A {
 protected int x = 0;
}

{
A a = new A();
int y = a.x;
}
```

- **Class** is already defined
  Explication : Une redéfinition de classe a été effectuée. Exemple de cas incorrect :

```
class A {
 protected int x = 0;
}
class A {
 protected int z =1 ;
}
```

- Condition does not return a boolean
  Explication : La condition utilisé dans un if ou while ne correspond pas à un type booléen Exemple de cas incorrect :

```
{
    int x = 1 ;
    if (x) {                    ---> Le type de x est pas booléen
        println("erreur");
    }
}
```

- Cannot apply an arithmetic operation to this type
  Explication : On ne peut pas appliquer des opérations arithmétiques sur des types autre que float et int. Exemple :

```
{
    int x = 1 ;
    boolean b = true;
    x = b + x;
}
```

- Opérations Bool can only be applied to booleans
  Explication : On peut pas appliquer des op booléenes qu'aux booléans

```
{
    int x = 1;
    boolean b = true;
    b = b || x;
}
```

- <**Type**> is definetely not a class
  Explication : Déclaration d'une classe ou héritage d'une expression qui n'est pas de type class.
  Exemple de cas incorrect :

```
class A extends boolean{}
```

- A method is already defined by <**Expr**>
  Explication : Une méthode a été redéfinie :
  Exemple de cas incorrect :

```
class A {
void var(){
 println("var est une méthode") ;
 }
}
class B extends A {
   public int var = 2 ;
  }
```

- Signature of overriden method test doesn't match the method
  Explication : L'override d'une methode contient des paramétres (ie signature) différente de la classe métre héritée.
  Exemple de cas incorrect :

```
class A {
void var(){
 println("var est une méthode") ;
 }
}
class B extends A {
    void var(int x){
      println(x);
    }
  }
```

- Return type of the child is not a subtype of the parent's
  Explication : Le type de la valeur de retour de la méthode héritée et le type de la valeur de retour de la classe mére sont différents. Exemple de cas incorrect :

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

```
    int setX(int x ) {
       return x
     }
  }
```

- Invalid cast
  Explication : Un cast sur un type non permit a été effecuté. On peut cast que les types int et float. Exemple de cas correct :

```
{
int a  ;
float x = (float) a ;
}
```

  Exemple de cas incorrect :

```
{
int a  ;
float x = (boolan) a ;
}
```

- A field is already defined by "name"
  Explication : Quand on veut déclarer une méthode au même nom qu'un attribut qui existe dans la classe courante ou ses parents.

```
class A{
void setX(int x){}
}
class B extends A {
protected int setX; //là
}
```

- field already exist
  Explication : Quand on redéfinit un attribut définit auparavant dans la même classe

```
class A{
protected int x = 1;
protected int x = 2;
}
```

- "name is undefined"
  Explication : Utiliser une variable non défini avant.

```
{
float b = 2.5;
a = b;
}
```

- "Type error while using instanceof"
  Explication : Non respect des règles imposées pour instanceof.

```
class A{}
class B{}
{
int a;
if (a instance of B){ //ici
    print("llllll");
}
}
```

- can't call a method in main without a class behind
  Explication : Ne pas appeler une méthode dans main sans une variable de type classe avant. Exemple :

```
class A{
 int x;
 void setX(int x){
    this.x = x;
}
}
{
setX(2); //error
}
```

- type class expects a field or a method **Explication :** une classe attend soit une selection soit une méthod-call. Exemple :

```
class A{
int x;
}
{
A a = new A();
int b;
print(a.b);//errooor
}
```

- A method is called by a non class type
  Explication : Appel d'une méthode par un type qui n'est pas une classe. Exemple :

```
{
int a;
a.setX(3); //errooooor
}
```

- Wrong signature for method
  Explication : Appel d'une méthode d'une classe avec des paramètres de types incompatibles avec ceux de la définition. Exemple :

```
class A{
int x;
void setX(int x){
    this.x = x;
}
}
{
A a = new A();
boolean t = true;
a.setX(t);//error
}
```

- new is only used on classes
  Explication : On ne peut instancier que des types classes.

- Modulo can only be applied to ints
  Explication : Modulo ne s'applique qu'aux entiers.

- cannot apply the Not Operator to this type
  Explication : l'opérateur Not est utilisé uniquement pour des booléans.

- Unary minus can only be applied to integers and floats
  Explication : L'utilisation de la soustraction se fait que sur les entier et flottants.

- Can't acces this field from main
  Explication : Faire appel à un attribut protégé dans le main.

```
 class A{
 protected int x;
 }
```

```
{
A a = new A();
print(a.x); //error
}
```

- Expression type is not a subtype of current class
  Explication : Faire appel à un attribut protégé par une classe qui n'est pas un sous-type de la classe courante.

- we're not in a subtype of the class where the field is defined
  Explication : Faire appel à un attribut protégé dans une classe qui n'est pas un sous-type de la classe ou l'attribut est défini. Exemple :

```
class A{
protected int x;
}
class B{
void met(){
    A a = new A();
    print(a.x);
}
}
```

- Can't call 'this' inside main .

2.3 Erreur de génération de code :

Les erreurs a l'exécution se divisent en deux grandes catégories :

2.3.1 Programmes incorrects

- Division entière (et reste de la division entière) par 0 : Ajout de l'instruction BOV en assembleur après les divisions qui réalise un saut vers le Label suivant : opArith, qui affiche l'erreur suivante :Erreur : Stack Overflow
- Débordement arithmétique sur les flottants (inclut la division flottante par 0.0) Ajout de l'instruction BOV en assembleur après les opéations arithmétiques, qui réalise un saut vers le Label suivant : opArith, qui affiche l'erreur suivante : Erreur : Stack Overflow déréférencement de null :
- Ajout de l'instruction CMP en assembleur qui compare le contenu qui risque d'être null avec null , qui réalise un saut vers le Label suivant en cas d'égalité : dereferencement-null, qui affiche l'erreur suivante :Erreur :dereferencement-null

2.3.2 Programmes corrects l'exécution dépasse les limites de la machine

- Débordement mémoire (pile ou tas) Ajout de l'instruction TSTO et BOV en assembleur qui vérifie si l'immédiat spécifié après TSTO est supérieur a l'espace disponible dans la pile et dans ce cas aller vers le label suivant pile_pleine qui affiche l'erreur suivante Erreur : pile pleine
- Ajout de l'instruction en assembleur après les divisions qui réalise un saut vers le Label suivant : opArith, qui affiche l'erreur suivante : Erreur : Stack Overflow

2.4 Erreur des options compilation :

- Le paramètre n'est pas un entier !
  Explication : L'utilisation de decac -r X n'a pas été faite correctement , il y'a pas d'entier spécifiant le nombre de registre à utiliser.

- Le nombre de registre doit être compris entre 5 et 16
  Explication : L'utilisation de decac -r X n'a pas été faite correctement , le nombre de registre est supérieur à 16 ou inférieur à 5

- -v et -p sont incompatible

  Explication : L'utilisation de decac fait appel à p et v qui ne peuvent être fait en même temps.

3 Extension Trigo et ses limitations

Notre bibliothèque standard contient un fichier Math.decah qui implémente la méthode **ulp**, la méthode racine carré **sqrt**, la méthode **pow** qui calcule les puissances, ainsi que les méthodes trigonométriques : **cos, sin, asin, atan**. Pour utiliser la bibliothèque standard dans votre programme deca, il faut tout d'abord inclure "Math.decah" et puis créer une instance de la classe Math qui permettra d'appeler les méthodes implémentées. Par exemple :

```
#include "Math.decah"
{
    Math m = new Math();
    println("cos(0.0) = ", m.cos(0.0));
}
```

L'approximation des fonctions trigonométriques n'est pourtant pas sans défauts. La précision de notre implémentation, mesurée en ULP, est décrite ci-dessous :

3.1 Fonction sin

- Sur l'intervalle $[0, \pi]$, l'erreur est usuellement égale à 0 ulp et peut atteindre 1 ou 2. Notre implémentation est aussi très précise au voisinnage de π^- avec une erreur maximale de 2 ulp. Au voisinnage de π^+, l'erreur maximale est de 723 ulp.
- Sur l'intervalle $[\pi, 100]$, l'erreur usuelle est entre 0 et 2 ulp.
- Pour des valeurs entre $[10^2$ et $10^5]$, l'erreur usuelle est entre 0 et 5 ulp mais peut atteindre parfois avoir une valeur entre 10^2 et 10^3 lorsqu'il s'agit des multiples de π.
- Pour l'intervalle $[10^5, 10^6]$, l'approximation reste précise par rapport à l'argument d'entrée pour un très grand nombre de valeurs (erreur ne dépassant pas 100ulp). Il arrive pourtant dans pas mal de cas que notre approximation déborde de la valeur exacte.
- Pour des valeurs supérieures à 10^6, notre approximation est très loin d'être précise.

3.2 Fonction cos

- On obtient la même qualité d'approximation que pour **sin**, sauf qu'elle est beaucoup plus précise autour de π^+ et légèrement mieux sur les valeurs aberrantes.

3.3 Fonction atan

- Grâce aux formules trigonométriques, l'approximation est très précise sur toutes les valeurs, donnant une erreur entre 0 et 2 ulp.

3.4 Fonction asin

- L'erreur maximale est usuellement 0 ou 1 ulp, sauf pour des valeurs extrêmes comme 0.999999 où elle peut atteindre une valeur de l'ordre de 10^3.

3.5 Fonction ulp

- La valeur de retour est exacte.

4 Limitation de du compilateur

Certaines limitations peuvent être retrouvées au niveau des fonctionnalités du compilateur :

1. Concernant les Cast ils ne peuvent être appliqué qu'au int vers float ou float vers int. Les autres cast entre objet sont analysé lors du test-contextuel mais ne sont pas fonctionnel au niveau de la génération de code . Il est donc imposible d'effectuer un Cast de ce type /

```
Objet1 a ;
Objet2 b ;
b = (Objet2) a
```

Car la methode instanceOf n'a pas été traité dans la partie C de génération de code.

2. La fonction print est limité au string , int ou float.
3. La gestion de très grand nombre n'est pas faisable.