

# Projet Génie Logiciel

## Manuel utilisateur

ENSIMAG 2ème année - 2022-2023

---

Groupe 1 - équipe 3

Jorge Luri Vañó  
Nils Depuille  
Vianney Vouters  
Virgile Henry  
Logan Willem



# Sommaire

## Sommaire

### I. Utilisation du compilateur

1. Option -b
2. Option -p
3. Option -v
4. Option -n
5. Option -r
6. Option -d
7. Option -d
8. Option -w
9. Option -o

### II. Limitations de compilation du langage deca

### III. Messages d'erreur

1. Erreurs lexicales
2. Erreurs syntaxiques
3. Erreur de parsing des int et float
4. Erreurs contextuelles
5. Erreurs d'exécution du code assembleur

### IV. Transformations et limitations de l'optimisation

1. Résolution des calculs évidents
2. Suppression des variables, attributs, méthodes et classes inutiles
3. Développement en calculs plus rapides :
4. Substitution des méthodes "inline"
5. Limitations de l'extension d'optimisation

## I. Utilisation du compilateur

Le compilateur permet de compiler un programme en langage deca pour obtenir un fichier assembleur exécutable par une machine virtuelle ima. Il s'utilise par l'intermédiaire de la commande **decac** de la manière suivante :

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

- Les options sont des lettres précédées d'un tiret et séparées d'un espace.
- Les fichiers sont des chemins de fichiers séparés d'un espace.

Il est donc possible de combiner les options et de compiler plusieurs fichiers en séquence ou en parallèle.

### 1. Option -b

Cette option permet d'afficher une bannière avec le nom de l'équipe, les membres qui la composent ainsi que leur(s) rôle(s) dans l'équipe. Cette option ne peut être appelée que seule, sans autre option ni fichier source. Si cette dernière condition n'est pas respectée, decac envoie comme message d'erreur : `“/!\ The option -b can only be used alone.”` puis affiche l'usage correct de la commande decac. Dans tous les cas, decac termine juste après avoir affiché, soit la bannière, soit le message d'erreur.

```
=====
Groupe 1 - GL03:
Nils DEPUILLE   : Chef de projet
Virgile HENRY   : Code master
Jorge LURI VAÑO : Git master & DRH
Vianney VOUTERS : Secrétaire Général
Logan WILLEM    : Test master
=====
```

## 2. Option -p

Cette option lance le compilateur en mode *parse and verify* qui comme son nom l'indique parse le fichier en argument. Juste après la vérification syntaxique, le compilateur effectue une décompilation de l'arbre. Quelques modifications sont alors effectuées sur le code retour comparé à celui d'entrée :

- Lorsqu'on déclare une classe, si elle extends `Object` on le note `class A extends Object` à la place de `class A`
- Dès qu'il y a une branche conditionnelle, un `if`, `else` est automatiquement ajouté, même s'il n'y était pas à l'origine.

Dans le cas où le programme n'est pas correct, une erreur est renvoyée.

Cette option n'est pas compatible avec l'option -v.

## 3. Option -v

Cette option permet d'arrêter le compilateur juste après l'étape de vérification. Elle retourne uniquement les erreurs. S'il n'y en a pas, rien n'est retourné. Cette option n'est pas compatible avec l'option -p.

## 4. Option -n

Cette option permet de supprimer certaines erreurs renvoyées à l'exécution. Les erreurs ignorées sont les suivantes:

- Division entière ou modulo par 0.
- Débordement arithmétique sur les flottants.
- Absence de `return` lors de l'exécution d'une méthode.
- Conversion de type impossible.
- Déréférencement de `null`.
- Stack overflow et Heap overflow.

Cette option permet entre autres de gagner quelques cycles à l'exécution.

## **5. Option -r**

L'option -r attend un argument supplémentaire. Il s'agit d'un entier compris entre 4 et 16. Cette option permet de limiter le nombre de registres utilisés. Ainsi `decac -r 4` demande au compilateur de n'utiliser que les registres R0 à R3.

## **6. Option -d**

L'option -d permet de changer le niveau de debug et ainsi afficher plus de messages en fonction de ce niveau de debug. Plus l'option est spécifiée, plus le niveau de debug est élevé. Les niveaux de debug sont les suivants:

0. Quiet mode: L'option n'est pas spécifié, rien n'est affiché.
1. Info mode: Seules les informations sont affichées.
2. Debug mode: Les informations classiques et de debug sont affichées.
3. Trace mode: Les informations, les messages de debug et de trace sont affichées.
4. All mode: Tous les messages sont affichés.

## **7. Option -d**

Cette option permet de lancer la compilation en parallèle de tous les fichiers spécifiés en argument de la commande decac.

## **8. Option -w**

Cette option est reconnue mais n'est pas utilisée. Elle n'a pas d'effet.

## **9. Option -o**

Cette option permet de lancer le compilateur en mode optimisation. Les optimisations effectuées sont celles décrites dans la partie suivante. Cette option peut être combinée avec l'option -p qui dans ce cas ne compile pas le programme mais renvoie le programme décompilé, suite aux différentes optimisations appliquées sur le (ou les) fichier(s) en question.

## II. Limitations de compilation du langage deca

L'intégralité des fonctionnalités du langage deca avec et sans objet est implémenté. Toutefois, il n'est pas encore possible d'assigner une valeur à un attribut depuis le programme principal. Un exemple en est donné ci-dessous:

```
class A {  
    int x;  
}  
{  
    A a = new A();  
    a.x = 2;  
}
```

On se contentera donc de méthodes dédiées type “setters”, qui elle, fonctionnent:

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

En effet, la sélection sur un attribut d'un objet est considérée comme un field dans une méthode, et donc provoque une erreur d'adressage lors de son assignation.

### III. Messages d'erreur

#### 1. Erreurs lexicales

Une erreur lexicale est levée à la compilation lorsqu'un mot du programme n'est pas reconnu par le langage deca. Les erreurs se présentent sous la forme suivante :

<nom de fichier.deca>:<ligne>:<colonne>: token recognition error at: '<mot incorrect>'

#### 2. Erreurs syntaxiques

##### Suite incorrecte de mots :

Une erreur syntaxique est levée à la compilation lorsqu'un enchaînement de mots du programme n'est pas reconnu par la grammaire du langage deca. Les erreurs se présentent sous l'une des formes suivantes :

<nom de fichier.deca>:<ligne>:<colonne>: no viable alternative at input '<mot incorrect>'

<nom de fichier.deca>:<ligne>:<colonne>: mismatched input '<mot incorrect>' expecting {<liste de mots entre apostrophes>

##### Nom de fichier invalide importé:

Une erreur syntaxique est levée à la compilation lorsqu'un fichier inclus n'est pas trouvé par le compilateur. L'erreur se présente sous la forme suivante :

<nom de fichier.deca>:<ligne>:<colonne>: <nom de fichier>: include file not found

#### 3. Erreur de parsing des int et float

##### Entier codable sur 32 bits :

Une erreur syntaxique est levée à la compilation si un littéral entier n'est pas codable comme un entier signé positif sur 32 bits. Les erreurs se présentent sous la forme suivante :

<nom de fichier.deca>:<ligne>:<colonne>: rule literal failed predicate: {\$tree != null}?

##### Flottant arrondi à l'infini ou à 0 :

Les littéraux flottants sont convertis en arrondissant si besoin au flottant IEEE-754 simple précision le plus proche. Une erreur de compilation est levée si un littéral est trop grand et que l'arrondi se fait vers l'infini, ou bien qu'un littéral non nul est trop petit et que l'arrondi se fait vers zéro. Les erreurs se présentent sous la forme suivante :

<nom de fichier.deca>:<ligne>:<colonne>: rule literal failed predicate: {\$tree != null)?

#### 4. Erreurs contextuelles

Les erreurs contextuelles sont affichées de manière précise sous la forme suivante :

<nom de fichier.deca>:<ligne>:<colonne>: <description informelle du problème>.

La description informelle du problème est toujours terminée par la règle de vérification contextuelle. Dans chaque règle, ce qu'il y a entre les accolades est remplacé à chaque fois que l'erreur est appelée pour inclure le nom de l'objet sur lequel s'applique l'erreur. Voici la liste de toutes les erreurs contextuelles présentes dans l'ordre des règles :

- "The identifier {name} doesn't exist (rule 0.1)"

Cela signifie que l'utilisateur essaie d'accéder à un identifiant (variable, paramètre, attribut, méthode, ...) qui n'est pas connue dans l'environnement courant.

- "The type {type} doesn't exist (rule 0.2)"

Dans ce cas, le type n'est pas connu par le compilateur. On notera le fait que le type string n'existe pas en Déca car il n'est pas possible d'instancier une chaîne de caractères.

- "{Name} is already a type/class (rule 1.3)"

Lors de la déclaration d'un type ou d'une classe, on ne peut pas utiliser un nom qui existe déjà dans l'environnement courant.

- "{SuperIdentifier} is not a class (rule 1.3)"


Cette erreur se produit lorsqu'on veut étendre une classe d'une entité qui n'est pas une classe.

- "The field {fieldName} has already been declared (rule 2.4)"

Lorsqu'on souhaite déclarer un field, il ne faut pas que ce nouveau field ait le même nom qu'un field déjà existant dans l'environnement courant.

- "The field's type can't be void (rule 2.5)"





Il n'est pas possible de déclarer un field avec un type void en Déca.

- "The name {fieldName} is already used for a method in the superclass (rule 2.5)"

Il est possible en Déca de déclarer un field avec un nom déjà utilisé dans une classe mère, néanmoins ce nom doit être celui d'un field et non d'une méthode. Ainsi si ce nom est celui d'une méthode on renvoie cette erreur

- "The method {methodName} has already been declared (rule 2.6)"

Dans une même classe, il n'est pas possible de créer deux méthodes avec le même nom.

- "The name {methodName} is already used for a field in the superclass (rule 2.7)"

De même qu'avec le règle 2.5 avant, une méthode d'une classe ne peut pas avoir le même nom qu'un field dans une de ses classes mères.

- "The method {methodName} doesn't have the same signature as the method defined in the superclass (rule 2.7)"

Lorsqu'on Override une méthode, il faut que la méthode dans la classe fille ait la même signature que dans la classe mère.

- "The return type is not the same as defined in the superclass (or not a subtype) (rule 2.7)"

Au moment d'Override une méthode, le type de retour doit au moins être un sous-type du type de retour de la classe mère. Si ce n'est pas possible, on renvoie cette erreur pour signifier que le type de retour n'est pas le bon.

- "The parameter's type can't be void (rule 2.9)"

On ne peut pas déclarer un paramètre avec le type void, c'est la règle 2.5 pour les paramètres.


- "The parameter {paramName} has already been declared (rule 3.12)"

Lorsqu'on déclare plusieurs paramètres en même temps, il ne faut pas que deux paramètres aient le même nom.

- "The variable's type can't be void (rule 3.17)"

On ne peut pas déclarer une variable avec le type void.

- "The variable {varName} has already been declared (rule 3.17)"



Lorsqu'on déclare plusieurs variables en même temps, il ne faut pas que deux variables aient le même nom.

- "Return cannot be used when method has void type (rule 3.24)"

Comme il n'est pas possible d'instancier un objet en void, une méthode ayant comme type de retour void ne doivent pas contenir de return.

- "The condition is not a boolean (rule 3.29)"

Une condition doit être de type boolean à l'intérieur des if et while.

- "Arguments of a print can only be float, int or string (rules 3.31)"

En Déca, à l'intérieur d'un print, il ne peut y avoir que des floats, int, string, ainsi on ne peut pas print des booléens ou des type\_classes.

- "A boolean operation has to be done only between 2 booleans (rule 3.33)"

Les opérations binaires booléennes comme les && ou || doivent être effectués avec deux booleans.

- "A boolean can only be compared to another boolean (rule 3.33)"
- "A class (or null) can only be compared to another class (rule 3.33)"

Les opérations binaires de comparaison doivent avoir le même type (ou alors un type castable vers l'autre type) dans l'opérande de gauche que celle de droite. Ainsi si on a un booléen à gauche, on doit en avoir un à droite, si on a une classe ou Null à gauche on doit avoir la même chose à droite.


- "The left operand of a comparison operation has to be an int or a float (rule 3.33)"
- "The right operand of an comparison operation has to be an int or a float (rule 3.33)"

De plus pour les opérations binaires de comparaisons, si ce n'est pas une opération entre deux booléens ou des classes, alors ce doit être des int ou float.

- "The left operand of an arithmetical operation has to be an int or a float (rule 3.33)"
- "The right operand of an arithmetical operation has to be an int or a float (rule 3.33)"

Pour une opération arithmétique, le type des opérandes doit int ou float.

- "A modulo can only be done between 2 int (rule 3.33)"



Le modulo est une opération arithmétique spéciale car elle ne s'applique pas aux flottants, ainsi les opérandes doivent être entières.

- "A unary minus is only followed by an int or a float (rule 3.37)"

Pour ce qui est du moins unaire, il ne s'applique qu'aux entiers ou flottants.

- "A not is only followed by a boolean (rule 3.37)"

Le not unaire n'est utilisé qu'avec un booléen.

- "Unable to cast type {type1} to {type2} (rule 3.39) "

Cette erreur apparaît lorsque l'utilisateur essaie de faire un cast impossible entre deux types. Pour rappel, pour qu'un cast soit possible, il faut qu'on ait un type à int et l'autre à float, ou bien que {type1} soit un sous-type de {type2}.

- "instanceof argument has to be a class (rule 3.40)"

InstanceOf ne doit être utilisé qu'avec une classe, il n'est pas possible de le faire avec les autres types.

- "New is only for classes (rule 3.42)"

Le new ne doit être utilisé qu'avec des classes, il n'est pas possible de créer un autre type par cette méthode.

- "This can only be used in a class (rule 3.43)"

This ne concerne que les classes, il ne peut pas être utilisé dans un Main, mais uniquement dans une classe.

- "The object of the selection is not of type class (rule 3.65)"

Il n'est pas possible de faire une sélection sur un autre type que class

- "The variable is protected (rule 3.66)"

La variable ne peut pas être atteinte car elle est protégée et nous ne sommes pas dans une sous-classe de la classe où elle est définie donc on ne peut pas faire de sélection dessus.

## **5. Erreurs d'exécution du code assembleur**

- "Error : Division by zero"

Une division entière par zéro a été exécutée.

- "Error : Heap overflow"

Un nouvel objet a été créé, mais il n'y a plus de place sur le tas.

- "Error : Wrong input format"

L'entrée fournie par l'utilisateur n'a pas pu être interprétée dans le format demandé.

- "Error : No return"

Une méthode ayant un type de retour différent de void n'a rien retourné.

- "Error : Null reference exception"

Une méthode ou un champ a été accédé sur un objet valant null.

- "Error : "Floating point number operation overflow"

Un débordement sur une opération flottante a eu lieu.

- "Error : Remainder by zero"

Un reste par zéro a été effectué.

- "Error : Stack overflow"

La taille maximale de la pile a été dépassée.

## IV. Transformations et limitations de l'optimisation

L'extension optimisation peut être exécutée avec l'option -o pour obtenir un fichier assembleur optimisé. Il est aussi possible de combiner cette option avec l'option -p pour visualiser les optimisations effectuées. Une description succincte des techniques d'optimisation est présentée ci-dessous afin d'éclairer l'utilisateur sur les effets de bord qu'il pourrait rencontrer.

Il est important de noter que l'optimisation réalisée par le compilateur peut potentiellement éliminer des erreurs qui seraient survenues à l'exécution sans compilation optimisée. Une division par 0 dans un calcul inutile en est un exemple. Un programmeur s'attendant à obtenir des erreurs à l'exécution doit donc compiler ses programmes sans l'option -o d'optimisation.

### 1. Résolution des calculs évidents

Les calculs portant sur des constantes uniquement sont directement simplifiés.

<u>Programme initial:</u>	<u>Programme optimisé décompilé:</u>
<code>int a = 2 + 3;</code>	<code>int a = 5;</code>

### 2. Suppression des variables, attributs, méthodes et classes inutiles

Si une variables, une méthode, un attribut ou une classe n'est utilisé nulle part, il est retiré du programme.

<u>Programme initial:</u>	<u>Programme optimisé décompilé:</u>
<code>int a = 2; int b = 4; println(a);</code>	<code>int a = 2; println(a);</code>

### 3. Développement en calculs plus rapides :

Les calculs sont reformulés pour pouvoir être exécutés rapidement par la machine abstraite ima. C'est notamment le cas des multiplications quelconque qui sont reformulées (jusqu'à une certaine limite) de sorte à obtenir des multiplications par puissance de 2. De telles opérations sont réalisées par de simples décalages bien moins coûteux en nombre de cycles que des multiplications.

<u>Programme initial:</u>	<u>Programme optimisé décompilé:</u>
<pre>int a = readInt(); int b = 12 * a + 2 * a + a + 1; println(b);</pre>	<pre>int a = readInt(); int b = (8 * a) + (4 * a) + (2 * a)       + a + 1; println(b);</pre>

#### **4. Substitution des méthodes “inline”**

On appelle méthode inline une méthode qui ne consiste qu'en une seule instruction, un **return**, et qui ne comporte aucune utilisation de field ou d'assignation. On peut donc remplacer les appels de ces méthodes par leur expression en substituant les paramètres par les valeurs ou variables données en entrée. Une telle substitution n'est réalisée que si les valeurs et variables données en entrée sont “atomiques” pour ne pas dupliquer des opérations lourdes. On considère comme “atomiques” les littéraux et les variables.


<u>Programme initial:</u>	<u>Programme optimisé décompilé:</u>
<pre>class A {     int square(int x) {return x*x;} } {     A a = new A();     print(a.square(3)); }</pre>	<pre>class A {     int square(int x) {return x*x;} } {     A a = new A();     print(3*3); }</pre>

#### **5. Limitations de l'extension d'optimisation**

Deux erreurs ont été relevées dans l'optimisation de la compilation du decac. Ces deux erreurs combinées peuvent causer l'impossibilité d'exécuter le fichier assembleur optimisé généré. Les deux erreurs sont les suivantes:

Dans une simple expression s'effectuant entre deux variables, l'étape de factorisation considère chacune de ces variables et leur occurrence dans l'expression afin de factoriser l'expression. Ainsi le résultat de la factorisation devient  $(a*1)+(b*1)$ . Cette erreur seule pourrait ne pas être impactante, si ce n'est en terme de nombre de cycles.

La seconde erreur est la non prise en considération du type des variables lors de la factorisation. Ainsi, si une variable est de type flottant, la factorisation ne va pas ajouter de nœud ConvFloat permettant de spécifier une conversion entre cette variable et son facteur entier.



Ces deux erreurs combinées vont entraîner dans notre cas, une multiplication en assembleur entre un entier et un flottant.