

Decac

Groupe Génie Logiciel n°29



Compilateur pour le langage Deca

Manuel Utilisateur

Janvier 2024

Table des matières

1	Généralités	2
2	Utilisation du compilateur	3
2.1	decac -b	3
2.2	decac -p	3
2.3	decac -v	3
2.4	decac -n	4
2.5	decac -r X	4
2.6	decac -d	4
2.7	decac -P	4
2.8	decac -w	4
3	Lexique des messages d'erreur	5
3.1	Erreurs communes	5
3.1.1	Déclaration de variable	5
3.1.2	Assignation de valeur à une variable	5
3.1.3	Conversion en Float ou en Int	5
3.1.4	Opération	5
3.1.5	Print, Printx, Println, Printlnx	6
3.2	Erreurs spécifiques au compilateur essentiel	6
3.2.1	Erreur Passe 1	6
3.2.2	Erreur Passe 2	6
3.2.3	Erreur Passe 3	7
3.2.4	GetAttribut et CallMethod	7
3.3	Erreurs spécifiques au compilateur complet	8
3.3.1	Cast	8
3.3.2	InstanceOf	8
4	Limitations et spécificités du compilateur	9
4.1	Gestion de la pile	9
4.2	Gestion des impressions	9
4.3	Gestion du débogage	10
4.4	Limites	10
5	Extension TRIGO	11
5.1	Description générale	11
5.2	Utilisation	11
5.3	Limites de la classe Math	11
5.3.1	Portée théorique et fonctionnalité de la classe	11
5.3.2	Précision	12

1 Généralités

Ce document est un manuel d'utilisation adressé aux futurs utilisateurs du Deca Compiler, nommé Decac. Le manuel fournira des indications sur l'utilisation du compilateur, mais également sur une extension de la bibliothèque standard : l'extension TRIGO. Il est recommandé de compléter la lecture avec des ressources donnant une spécification détaillée du langage Deca. Si vous avez des questions à propos de notre compilateur, n'hésitez pas à nous contacter. Cela pourrait également permettre d'améliorer notre manuel d'utilisation, pour celles et ceux qui auraient les mêmes interrogations. Sur ce, nous vous souhaitons une bonne lecture, et nous espérons que vous apprécierez d'autant plus les possibilités de l'outil Decac à travers ce manuel d'utilisation.

2 Utilisation du compilateur

Tout commence avec la commande `./decac`. Elle permet d'afficher les différentes utilisations possibles du compilateur, ainsi qu'une brève description de chaque option, que nous vous fournissons également ci-dessous.

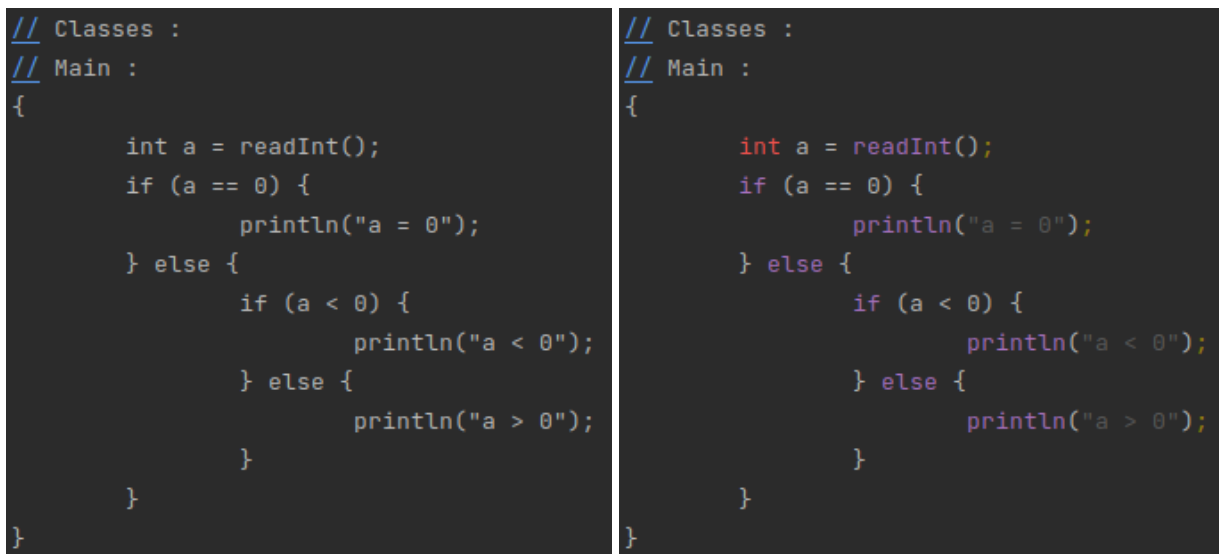
```
decac [[-p | -v] [-n] [-r X] [-d]* [-P] [-w] <fichier deca>...] |  
[-b]
```

2.1 decac -b

L'option `-b` (banner) permet d'afficher une bannière indiquant le nom de notre équipe, gl29.

2.2 decac -p

Le `-p` (parse) lance `decac` sur un fichier `deca`, et arrête son exécution juste après la construction de son arbre abstrait. Il affiche alors la décompilation de l'arbre. Il est possible d'ajouter de la couleur via l'option `--color` pour plus de visibilité. Un exemple d'utilisation sur le fichier `if.deca` vous est présenté ci-dessous.



```
// Classes :  
// Main :  
{  
  
    int a = readInt();  
    if (a == 0) {  
        println("a = 0");  
    } else {  
        if (a < 0) {  
            println("a < 0");  
        } else {  
            println("a > 0");  
        }  
    }  
}
```

Figure 1: Résultats de `/decac -p if.deca` (à gauche) et de `/decac -p --color if.deca` (à droite).

2.3 decac -v

Concernant le `-v` (verification), le principe est le même que pour `-p`, sauf que cette fois, l'exécution s'arrête après les vérifications lexicales, syntaxiques et contextuelles du fichier Deca. Si les vérifications n'ont détecté aucune erreur, la commande ne renvoie rien en sortie. Sinon, un message d'erreur s'affiche. Par exemple, le fichier `add-incompatible.deca`, où l'on tente d'additionner un élément de type `String` et un autre de type `int` renvoie le message suivant après l'exécution de la commande : `add-incompatible.deca:2:16: Exception: Incompatible types in arithmetic operation: string and int.`

2.4 **decac -n**

L'option **-n** (nocheck) ignore les cas d'arrêt d'exécution tels que les programmes incorrects (division par 0; débordement arithmétique; etc...), les erreurs de lecture (par exemple, un `readInt()` où l'utilisateur entre un flottant), ou encore un débordement mémoire.

2.5 **decac -r X**

L'option **-r X** (registers) permet de limiter les registres disponibles à `R0, ..., R[X-1]`, avec `X` compris entre 4 et 16.

2.6 **decac -d**

Le **-d** (debug) active les traces de débog lors de l'exécution. Il est possible de répéter l'option plusieurs fois pour avoir plus d'informations.

2.7 **decac -P**

L'option **-P** (parallel) permet d'optimiser le temps de compilation sur plusieurs fichiers, en les lançant en parallèle au lieu de les lancer successivement.

2.8 **decac -w**

Enfin, le **-w** (warning) affiche les avertissements qui pourraient être générés lors de la compilation d'un fichier Deca.

3 Lexique des messages d'erreur

3.1 Erreurs communes

3.1.1 Déclaration de variable

Lorsqu'on instancie une nouvelle variable, notre compilateur va d'abord vérifier dans DeclVar que le type n'est pas void. Si le type est void une erreur est levée : `"Exception : Variable type cannot be void"`.

Puis lors de l'initialisation, le compilateur va vérifier que le type de la variable est bien le même que celui de l'expression. Si ce n'est pas le cas, on a l'erreur suivante : `"Exception : Incompatible types in initialisation : type expression and type variable."`

Lors de la vérification de l'expression de la variable dans Identifier, notre compilateur va chercher dans l'EnvironnementExp local la définition de la variable. S'il n'existe pas, alors il cherche dans l'EnvironnementExp d'une classe si la classe mise en paramètre n'est pas nul. Cependant, si elle est nulle, alors on cherche dans l'EnvironnementType du compiler pour chercher l'expression de la classe. Si après toutes ces recherches, le compilateur n'a pas trouvé la définition, il renverra une erreur : `"Exception : Identifier VariableName is not defined"`.

3.1.2 Assignation de valeur à une variable

Lors de l'assignation d'une valeur, notre compilateur va d'abord vérifier que le type de la variable et le type de la valeur sont compatibles. S'ils ne le sont pas, alors une erreur sera levée : `"Exception : Incompatible types in assign : type variable and type valeur"`.

3.1.3 Conversion en Float ou en Int

Lorsqu'on veut convertir un int en float, notre compilateur va vérifier que le type mis dans cette conversion est bien un int, sinon une erreur est levée : `"Exception : Conversion vers un flottant d'un non entier"`.

Pour la conversion d'un float en int, la même vérification est faite. Si le type mis dans celle-ci n'est pas un float, on a l'erreur suivante : `"Exception : Conversion vers un entier d'un non flottant"`.

3.1.4 Opération

Avec le compilateur, on peut effectuer plusieurs opérations. Pour les opérations arithmétiques, booléennes, de comparaisons et d'inégalités, le compilateur va vérifier que le type de l'opérande de gauche et le type de l'opérande de droite sont les mêmes. Si ce n'est pas le cas, on a les erreurs suivantes :

Pour les opérations arithmétiques : `"Exception : Incompatible types in arithmetic operation: type gauche and type droit"`.

Pour les opérations booléennes : `"Exception : Incompatible types in boolean operation. type gauche and type droit"`.

Pour les opérations de comparaisons et d'inégalité : `"Exception : Incompatible types incompatible : type gauche and type droit"`.

Pour l'opération de modulo, le compilateur fait la même vérification et renvoie une erreur si les deux types ne sont pas les mêmes : `"Exception : Invalid type for modulo : type gauche and type droit"`.

Enfin concernant le moins unaire, si le type de la variable n'est pas `int` ou `float` alors une erreur est renvoyée : `"Exception : Unary minus can only be applied to int or float"`.

3.1.5 `Print`, `Printx`, `Println`, `Printlnx`

Lorsque l'on veut afficher un résultat, le compilateur va d'abord vérifier que ce que l'on veut afficher est de type `int`, `float`, `string` ou `boolean`. Si ce n'est pas le cas, on a l'erreur suivante qui s'affiche : `"Exception : Argument of print must be int, float, boolean or string"`.

3.2 Erreurs spécifiques au compilateur essentiel

3.2.1 Erreur Passe 1

Lors de la passe 1 on va avoir des vérifications lors de la déclaration d'une classe. En premier lieu, on vérifie que le type de sa Super Classe n'est pas nul. Si c'est le cas, on a l'erreur suivante : `"Exception : Super class Name SuperClass doesn't exist"`.

Une fois que cela est vérifié, on va déclarer la classe dans l'Environnement Type du compiler. Si cette classe est déjà définie dans cet environnement, on renvoie l'erreur suivante : `"Exception : Class Name Class already exists"`.

3.2.2 Erreur Passe 2

Pendant la passe 2, le compiler va faire des vérifications lors de la déclaration des attributs et des méthodes d'une classe.

Pour la vérification des attributs, le compilateur va d'abord vérifier que le type de l'attribut n'est pas `void`. En cas d'attribut de type `void`, l'erreur suivante sera levée : `"Exception : Variable type cannot be void"`.

Puis, on vérifie que le nom de cet attribut n'est pas déjà déclaré en tant que méthode dans sa Super Classe. Si c'est le cas, une erreur s'affichera : `"Exception : Field Name Field is already defined as a method in SuperClass"`.

Enfin, on vérifie que cet attribut n'est pas déjà défini dans cette classe. Sinon, on renvoie l'erreur suivante : `"Exception : Field Name Field is already defined"`.

Pour la vérification des méthodes, le compilateur va chercher si le nom de la méthode existe dans la Super Classe. Si c'est le cas, il va alors vérifier que ce nom est associé à une méthode pour pouvoir accepter le `Override`. Si ce nom est associé à un attribut, une erreur est renvoyée : `"Exception :`

Method Name Method is already defined as a field in SuperClass".

Sinon on va vérifier que cette méthode a le même nombre de paramètres que celle du même nom dans sa Super Classe. Si ce n'est pas le cas, on a l'erreur suivante : "Exception : Signature of method Name Method is not the same".

Sinon, on va vérifier que le type de la méthode est le même que celle dans la SuperClasse. Si ce n'est pas le cas, on a une erreur : "Exception : Type of method Name Method is not the same".

Une fois que toutes ces vérifications sont passées, il ne reste plus qu'à vérifier que cette méthode n'est pas déjà déclarée dans la classe, et si elle l'est déjà, on a l'erreur suivante : "Exception : Method Name Method already declared".

3.2.3 Erreur Passe 3

Pendant la passe 3, le compilateur va vérifier l'initialisation des attributs et le corps des méthodes.

Pour l'initialisation des attributs, l'erreur levée est la même que celle dite précédemment (section 3.1.1).

Si on a une méthode ASM, le compilateur va vérifier le string mis en paramètre.

Si on a une méthode classique, il y aura une vérification des variables et l'erreur levée à cette étape est la même que celle dite précédemment (section 3.1.1)

Il y aura ensuite une vérification des instructions.

Pour les instructions `While` et `IfThenElse`, le compilateur va vérifier les instructions à l'intérieur de ces instructions et renvoyer une erreur si besoin. Il y aura aussi une vérification de la condition qui renverra une erreur si le type renvoyé de cette condition n'est pas du type boolean : "Exception : Condition must be boolean".

Pour les instructions `Print`, `Printx`, `Println`, `Printlnx`, si la vérification relève une erreur, ces erreurs seront du même type que celles rencontrées dans la section 3.1.5.

Pour l'instruction `Return`, si le type à retourner est une classe, le compilateur va vérifier d'abord si le type de la classe mise est une sous-classe du type à retourner. Si ce n'est pas le cas, on a l'erreur suivante : "Exception : Return type is not a subclass of the method type".

Si le type retourné n'est pas une sous-classe du type à retourner, alors, il y a une vérification pour savoir si les deux types sont les mêmes. Si ce n'est le cas, on a alors l'erreur suivante : "Exception : Return type is not the same as the method type".

Si le type à retourner n'est pas une classe, mais un `int`, `float` ou `boolean`, le compilateur va donc vérifier si les deux types sont les mêmes et si ce n'est pas le cas, une erreur est renvoyée : "Exception : Return type is not the same as the method type".

3.2.4 GetAttribut et CallMethod

Lors d'un programme, le compilateur peut faire appel aux classes `GetAttribut` et `CallMethod`, et par conséquent des vérifications sont faites. Dans les deux classes, la première vérification est de s'assurer que le type de l'expression qui appelle l'attribut ou la méthode est bien une classe, et si ce n'est pas le cas, l'erreur suivante sera levée : "Exception : L'expression n'est pas de type

classe".

Si c'est le cas, on passera aux vérifications suivantes.

Dans GetAttribut, le compilateur récupère la définition de l'attribut appelé dans le programme. Si cette définition est égale à null, alors l'erreur suivante est levée : "Exception : L'attribut n'existe pas dans la classe".

Si elle ne vaut pas null, alors on va vérifier que l'attribut voulu est bien un attribut de la classe. Si ce n'est pas le cas, l'erreur suivante sera renvoyée : "Exception : L'attribut n'est pas un champ de la classe".

Si les deux vérifications sont passées, il ne reste plus qu'à vérifier la visibilité de l'attribut. Si elle vaut protected, alors on aura l'erreur suivante : "Exception : L'attribut est protected, impossible d'y accéder depuis une autre classe".

Dans CallMethod, on procède de la même façon que dans GetAttribut. Si la définition de la méthode voulue vaut null, alors une erreur est renvoyée : "Exception : L'identificateur n'existe pas dans la classe".

Puis si elle ne vaut pas null mais que ce n'est pas une méthode, on aura l'erreur suivante :

"Exception : L'identificateur n'est pas une méthode".

Enfin, si c'est bien une méthode, mais que le nombre de paramètres mis dans le main diffère de celui défini dans la classe, on aura donc une erreur :

"Exception : Wrong number of arguments in method call : [Nombre paramètre Main] expected, [Nombre paramètre Classe] given".

3.3 Erreurs spécifiques au compilateur complet

3.3.1 Cast

Quand on cast une valeur, on a différents cas. Un float peut être cast en int, un int peut être cast en float, une classe peut être cast en une de ses Classes mères ou une de ses Classes filles. Si lors de la vérification, nous ne sommes dans aucun de ces cas, alors une erreur est renvoyée : "Exception : cast error, you can't cast Type Cast to Type Expression".

3.3.2 InstanceOf

Pour cette instruction, le type de la première variable doit être une classe ou null et celui de la deuxième doit être une classe. Si une des deux conditions n'est pas satisfaite, on a l'erreur suivante :

"Exception : type incompatible : Type variable 1 and Type variable 2".

4 Limitations et spécificités du compilateur

Tout d’abord, le compilateur présente des limitations liées au langage IMA et, par conséquent, aux langages assembleurs 32 bits (moins de précision sur les nombres flottants et des bornes d’entiers plus restreintes par rapport à du 64 bits). Comme spécifié, DECA est un langage moins étendu que Java (*i.e.*, *absence de définition de variables dans le corps, absence de gestion des tableaux par défaut...*).

4.1 Gestion de la pile

La gestion de la pile diffère de celle initialement décrite dans le manuel utilisateur. En supposant que $r :=$ Nombre de registres disponibles, $r \in \llbracket 4; 16 \rrbracket$. Parmi les variables définies¹ dans le corps (du main ou de la méthode), nous allons calculer leur nombre d’apparitions dans le code² afin d’obtenir une table avec les variables en fonction de leur nombre d’utilisation. Ensuite, nous placerons les $r - 4$ variables utilisées dans les registres et les suivantes dans la pile.

Prenons le programme suivant :

```
1 {   int a = 12; // Programme inutile
2   int b;
3
4   b = 0;
5   print(b);
6   while (a > 0) {
7       a = a - 1;
8       println(a);
9       b = b + 1;
10  }
11  println(b); }
```

Nous obtenons la table suivante :

Nom de la variable	Nombre d’apparitions	Zone de stockage
a	5	1 (GB)
b	6	R2

Table 1: Exemple de table pour la gestion des variables, en supposant qu’il y a seulement 5 registres utilisables (R0 et R1 sont réservés et R3 et R4 réservés pour les autres opérations).

Cette technique permet d’optimiser le nombre d’accès à la pile, qui est plus coûteux en opérations.

4.2 Gestion des impressions

Une deuxième divergence par rapport à la spécification concerne la possibilité d’afficher des booléens. Le programme peut maintenant afficher à l’écran les valeurs `true` et `false`. De plus, il est désormais possible d’afficher des chaînes de caractères qui se trouvent sur plusieurs lignes.

¹À l’exception des *fields* et des *params* qui ont leur propre place réservée dans la pile

²Cette métrique est la plus simple à obtenir, même si dans des boucles comme dans l’exemple, elle ne représente pas l’utilisation réelle de la variable.

4.3 Gestion du débogage

Comme mentionné précédemment, certaines fonctionnalités ont été ajoutées dans les fonctions de débogage. La décompilation avec coloration syntaxique³ permet de s'assurer que le programme a été correctement analysé. Ensuite, lorsque le débogage de troisième niveau est activé, des commentaires sont ajoutés dans le code assembleur généré afin de faciliter le processus de débogage.

4.4 Limites

Le compilateur est complet dans l'implantation de DECA. Il passe l'ensemble des nos tests sur les étapes A, B et C ainsi que la dé-compilation et la limitation des registrer. Notre banque de tests étant pas parfaite, des bugs peuvent subsister ; notamment lors de calcules nécessitant beaucoup de registres dans les méthodes. De plus certaines parties ne sont pas optimisés, pour les méthodes, on sauvegarde l'ensemble des registres disponibles au lieu de ceux qu'on va effectivement modifier.

³Cette fonctionnalité n'est pas activée par défaut car elle n'est pas compatible avec les tests automatiques en raison des caractères d'échappement. Elle nécessite un terminal compatible.

5 Extension TRIGO

5.1 Description générale

Dans le cadre de l'extension TRIGO, nous avons enrichi la bibliothèque standard de Deca d'un fichier `Math.decah` définissant une classe `Math`. Cette classe comporte les méthodes suivantes :

- `float cos(float f)`
- `float sin(float f)`
- `float asin(float f)`
- `float atan(float f)`
- `float ulp(float f)`

Ces fonctions étaient imposées par le cahier des charges de l'extension, mais nous avons pris la liberté d'ajouter des méthodes et des attributs à la classe `Math` qui facilitaient l'implémentation des méthodes susmentionnées:

- `int sign(float f)` renvoie `+1` si le flottant est positif, `-1` sinon
- `float abs(float f)` : fonction valeur absolue
- `float pow(float f, int exp)` renvoie f^{exp}
- `int fact(int n)` renvoie $n!$

5.2 Utilisation

Pour utiliser la classe `Math`, il faut tout d'abord inclure le fichier `Math.decah` pour qu'il soit compilé en même temps que votre fichier principal. Il faut ensuite instancier la classe `Math` pour pouvoir appeler les différents attributs et méthodes de la classe `Math`. Par exemple:

```
1 #include "Math.decah"
2
3 {
4     Math m = new Math();
5     println(m.cos(3.4785523));
6 }
```

5.3 Limites de la classe Math

5.3.1 Portée théorique et fonctionnalité de la classe

Le fichier `Math.decah` ne fournit pas une bibliothèque mathématique complète. Seules les fonctions listées précédemment sont implémentées, en plus de quelques fonctions qui ne sont pas destinées à l'utilisateur.

Ces méthodes commencent par `_` ce qui laisse une plus grande liberté à l'utilisateur pour développer

et compléter la bibliothèque mathématique, dans une classe `UMath` qui étendrait la class `Math` par exemple.

Par ailleurs, contrairement au langage Java et sa classe `Float`, il n'existe pas de méthode permettant d'obtenir la représentation binaire d'un flottant sous forme d'un entier 32 bits au sens de la norme IEEE 754. Les arrondis décimaux sont donc inhérents à la représentation des flottants dans le langage Deca et des imprécisions sont à prévoir.

5.3.2 Précision

Les algorithmes utilisés s'avèrent assez précis dans la majorité des cas. Les résultats obtenus avec les fonctions trigonométriques de la classe `Math.decah` diffèrent significativement de ceux obtenus avec la classe `Math` de Java lorsqu'on manipule de grands nombres (*arcsin*(x) avec x proche de 1 par exemple). Cela vient du fait que nous n'utilisons pas la Fusion Multiplication Addition qui permettrait de limiter les imprécisions lors des calculs sur les flottants. Les imprécisions sont alors décuplées lorsqu'on manipule de grands nombres.