

Manuel utilisateur

GL37

Avare Thomas, Quentin Candaele, Helena Cazals, Germain Vu

22 janvier 2022

Ceci est le manuel utilisateur du compilateur deca du groupe 37 du projet GL. Ce document a pour but de présenter le fonctionnement et l'utilisation du compilateur, ses options ainsi que la présentation de l'extension. L'extension est l'extension [TRIGO], ses différentes fonctions implémentées seront présentées, ainsi que les limites techniques de ces fonctions.

Table des matières

1	Compilation et options de compilation	2
1.1	Présentation de la commande et ses options	2
2	Messages d'erreurs	3
2.1	Messages à la compilation	3
2.2	message à l'exécution	9
3	Extension TRIGO	10
3.1	utilisation de l'extension trigo	10
3.2	Méthode ULP	10
3.3	Méthode sin	10
3.4	Méthode cos	10
3.5	Méthode arctan	11
3.6	Méthode arcsin	11

1 Compilation et options de compilation

La compilation d'un programme deca se fait via la commande **decac** dans le terminal suivi du chemin vers le fichier du programme deca. Pour l'exécuter de cette manière on doit cependant avoir initialisé le **PATH**, soit au préalable, soit directement dans les **PATH** dans notre fichier **.bash.rc** (ou **.zsh.rc** selon le système unix).

il ya quelques différences avec le compilateur de l'énoncé, le notre permet d'imprimer des booléens notamment (true, false ou résultats d'opérations booléennes), certaines options n'ont pas pu être intégrées à temps (-n, -P, instanceof, ...).

1.1 Présentation de la commande et ses options

La commande **decac** peut-être appelé seule, elle va alors afficher toutes les options de compilations énumérées ci-dessous :

decac **[[-p | -v] [-n] [-r X] <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 à $R_0 \dots R_{X-1}$, avec $4 \leq X \leq 16$
- 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)

On remarque notamment que les options **-p** et **-v** sont incompatibles.

2 Messages d'erreurs

2.1 Messages à la compilation

Les messages d'erreurs (lexicales, syntaxiques, contextuelles, et éventuelles limitations du compilateur) sont formatés de la manière suivante :

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

Voici un tableau montrant les messages d'erreurs générés par le compilateur :

Message d'erreur	Signification	Exemple de code faux
Identifieur 'toto' indéfini (règle 0.1).	Identificateur non déclaré	<pre>{ println (toto); }</pre>
Type indéfini (règle 0.2).	Le type doit être soit natif ou déjà défini.	<pre>{ Strange s; }</pre>
La superclass <identifieur> n'existe pas (règle 1.3).	La super-classe spécifiée ne peut pas faire office de super-classe.	<pre>class A extends B {}</pre>
Identifieur 'A' déjà déclaré (règle 1.3).	La classe de même nom a déjà été déclarée.	<pre>class A {} class A {}</pre>
Un champ ne peut pas être de type 'void' (règle 2.5).	Un champ déclaré ne peut pas être de type 'void'.	<pre>class A { void i; }</pre>

Message d'erreur	Signification	Exemple de code faux
La signature d'une méthode est incompatible à celle définie dans la super-classe (règle 2.7).	Réécriture impossible d'une méthode définie dans une super-classe en modifiant le type de retour.	<pre> class A { int getX() { return 4; } } class B extends A { void getX() {} } </pre>
La signature d'une méthode est incompatible à celle définie dans la super-classe (règle 2.7).	Réécriture impossible d'une méthode définie dans une super-classe en modifiant les paramètres.	<pre> class A { int getX(){ return 4; } } class B extends A { int getX(int i){ return i; } } </pre>
Le nom de champ est une méthode définie dans une superclasse (règle 2.7).	Les noms de champ doivent différer des nom de méthodes déjà déclarées.	<pre> class A { int m(){ } } class B extends A { int m; } </pre>
Identificateur champ de même nom déjà déclaré (règle 2.7).	Un champ de même nom a déjà été déclaré dans la classe.	<pre> class A { int i; float i; } </pre>
Missing parameters. Il manque un paramètre d'après la définition de la méthode dans la classe. (règle 2.7).	Attention à l'utilisation d'une méthode sans tous ses paramètres.	<pre> class A { void m(int i) {} } { A a = new A(); a.m(); } </pre>

Message d'erreur	Signification	Exemple de code faux
Too much parameters. Il y a trop de paramètres d'après la définition de la méthode dans la classe (règle 2.7).	Attention à l'utilisation d'une méthode et de ses paramètres.	<pre>class A { void m(int i) {} } { A a = new A(); a.m(); }</pre>
Paramètres de méthode portant le même nom (règle 2.8).	Impossible de définir deux fois le même paramètre dans une méthode.	<pre>class A { int getX(int x, int x){} }</pre>
Un paramètre d'une méthode ne peut pas être de type "void" (règle 2.9).	Un paramètre d'une méthode ne peut pas être de type "void".	<pre>Class A { void getX(void x) {} }</pre>
Types incompatibles pour l'affectation (règle 3.8).	Types incompatibles pour l'affectation.	<pre>{ int x = true; }</pre>
Une variable ne peut pas être de type 'void' (règle 3.17).	Une variable ne peut pas être de type 'void'.	<pre>{ void x; }</pre>
La condition n'est pas un booléen.	Un type boolean doit être dans la condition d'un 'if'.	<pre>{ if (1.0) {} }</pre>
La condition n'est pas un booléen (3.25).	Un type boolean doit être dans la condition d'un 'while'.	<pre>{ while (1.0) {} }</pre>

Message d'erreur	Signification	Exemple de code faux
Entier, flottant , booléen ou chaîne de caractère attendu (règle 3.31).	On ne peut print que des entiers, flottants , booléens ou chaînes de caractères.	<pre> class A {} { A a = new A(); println(a); } </pre>
Conversion de <type1> en <type2> impossible (règle 3.39).	Conversion de type1 en type2 impossible (règle 3.39).	<pre> { float i; i = (float) ("toto"); } </pre>
Les <type1> et <type2> sont incompatibles (règle 3.39).	Le membre de droite de instanceof doit être une classe.	<pre> { boolean b; b = 5 instanceof float; } </pre>
"this" ne peut pas être utilisé dans un programme principal (règle 3.43).	"this" ne peut pas être utilisé dans un programme principal.	<pre> { this = true; } </pre>
<type1> attendu mais <type2> donné (règles 3.54-59-60-61).	Une opération arithmétique ne se fait qu'entre int et float.	<pre> { int var; var = 1 / true; } </pre>
Opération impossible entre <type1> et <type2> (règles 3.49-50-51-52-53).	Une opération arithmétique ne se fait qu'entre int et float.	<pre> { int var; var = 1.0 % 1; } </pre>
Both operand must be booleans (règle 54-55-56-57).	Une opération binaire booléenne se faire entre booléens.	<pre> { int var; var = 1.0 % 1; } </pre>

Message d'erreur	Signification	Exemple de code faux
Impossible de comparer <type1> à <type2> (règles 3.56-57-58-59-60-61).	Une égalité/inégalité ne doit faire appel qu'à des flottants ou entiers.	<pre>{ boolean var; var = 1 > true; }</pre>
Un entier ou un flottant est nécessaire (règle 3.62).	L'opérateur unaire '-' ne prend que des float et int en argument.	<pre>{ int var; var = -true }</pre>
Un attribut protégé ne peut-être appelé en dehors de la classe.	Une variable protégé ne peut-être appelé que dans sa classe.	<pre>class A{ protected int i; } class B{ A a : new A(); int b; b = a.i; }</pre>
Signature non conforme, pas le bon nombre de paramètres (2.9).	Il n'y a pas le bon nombre de paramètres.	<pre>class A { void m(int i) {} } { A a = new A(); a.m(); }</pre>
Signature non conforme, le paramètre <ident> devrait être de type1 mais est de type2.	Signature non conforme.	<pre>class A { void m(int i) {} } { A a = new A(); a.m(); }</pre>

Message d'erreur	Signification	Exemple de code faux
Définition multiple de champs.	Champ multiple au sein d'une classe.	<pre> class A { int a; int a; } </pre>
Double définition de méthode.	On ne peut définir qu'une seule fois une méthode.	<pre> class A { int a() {} int a() {} } </pre>

2.2 message à l'exécution

Message d'erreur	Interprétation
Division par 0 impossible.	On ne peut pas diviser par 0.
Modulo par 0 impossible.	On ne peut pas faire de modulo par 0.
Infinite values not allowed.	Les int et les float sont limités.
NaN values not allowed.	Les valeurs NaN (Not a Number) ne sont pas acceptées en tant que int ou float.

les messages d'erreurs de type d'entrée pour des programmes deca interactifs n'ont pas été faits et une exception est lancée si un mauvais type est donné.

Les autres sorties sont des erreurs lancées par la machine ima.

3 Extension TRIGO

3.1 utilisation de l'extension trigo

L'extension est présente dans le fichier `Math.decah` et est appelée en l'incluant dans le code deca hors du main à l'aide de l'appel `#doinclude Math.decah`.

On peut alors appeler les fonctions de l'extension, à savoir :

- `cos` : la fonction trigonométrique `cosinus`.
- `sin` : la fonction trigonométrique `sinus`
- `acos` : la fonction trigonométrique `arccosinus`
- `asin` : la fonction trigonométrique `arcsinus`
- `ulp` : Unit in the Last Place

Pour plus de précisions, voir la documentation de l'extension.

3.2 Méthode ULP

La méthode ULP ne donne aucune approximation, le résultat est donc exact sur l'intervalle des flottants représentables. Nous avons réalisé des tests pour des valeurs allant de 0 à 2^{31} .

3.3 Méthode sin

La méthode `sin` est construite grâce à l'algorithme CORDIC amélioré et un décalage de π pour la gestion des grands nombres. Voici donc en fonction de l'angle dont nous souhaitons approcher le cosinus la précision (en bits) :

Intervalle	Erreur max (Bits)	Erreur moyenne (bits)	pas
$[0; \pi/2]$	5	3.5	2^{-22}
$[\pi/2; 2\pi]$	10	4.2	2^{-20}
$[100\pi; 100\pi + 2]$	13	9.84	2^{-16}

remarque. *Il y a une baisse de précision pour les grandes valeurs à cause du décalage de $n*\pi$ pas assez précis. Cf document extension.*

3.4 Méthode cos

La méthode `cos` est construite grâce à l'algorithme CORDIC amélioré et un décalage de π pour la gestion des grands nombres. Voici la précision en fonction de l'angle dont nous souhaitons approcher le cosinus (en bits) :

Intervalle	Erreur max (Bits)	Erreur moyenne (bits)	pas
$[0; \pi/2]$	5	3.5	2^{-22}
$[\pi/2; 2\pi]$	11	4.4	2^{-20}
$[100\pi; 100\pi + 2]$	14	9.80	2^{-16}

3.5 Méthode arctan

La méthode `arctan` est implémentée grâce à une application de la moyenne arithmético-géométrique d'une suite. Par imparité de la fonction arctangente, on ne donne le résultat que pour des nombres positifs.

Intervalle	Erreur max (Bits)	Erreur moyenne (bits)	pas
$[0; 2\pi]$	11	1.5	2^{-20}
$[100\pi; 100\pi + 2]$	4	2.7	2^{-16}
$[1000\pi; 1000\pi + 2]$	5	3.9	2^{-13}

3.6 Méthode arcsin

La méthode `arcsin` est déduite à partir de l'`arctan` à l'aide d'une simple formule géométrique. Par imparité de la fonction `arcsinus`, on ne donne le résultat que pour des nombres positifs.

Intervalle	Erreur max (Bits)	Erreur moyenne (bits)	pas
$[0; 1]$	4	2.2	2^{-20}