Projet GL 2021:
Manuel utilisateur

Équipe n°31 : Antoine Briançon, Axel Glorvigen, Thibault Launay, Maxime Martin et Pierre Pocreau

Sommaire:

I. Introduction

Ce document est un manuel utilisateur, permettant à quiconque souhaitant utiliser le compilateur “decac” pour la compilation d’un programme en langage “Deca” de comprendre son fonctionnement, ses limitations et ses principaux messages d’erreurs. Les différentes options du compilateur sont aussi abordées dans ce document, avec leur explication précise disponible dans le document des spécifications du projet. Ce document propose aussi une rapide explication des fonctionnalités disponibles dans l’extension “TRIGO”, implémentant quelques fonctions trigonométriques connues.

II. Limitations du compilateur

L’avancement du projet nous permet d’avoir un compilateur Decac avancé, fonctionnant sur la majorité du langage Deca. Cependant, il existe toujours des erreurs liées à une gestion des registres, ce qui ne permet pas un fonctionnement optimal du compilateur sur certains programmes. Les limitations du compilateur correspondent donc aux limitations du langage Deca, expliquées dans les spécifications du projet, en plus d’une fonctionnalité de gestion de registres non complète. On propose cependant de reprendre les points principaux des règles abordées dans les spécifications, et de préciser l’impact de la gestion de registres dans l'exécution du programme.

A. Types prédéfinis

Les types prédéfinies correspondent aux types utilisés selon les limitations imposées par le langage “Deca”, sous-ensemble du langage Java. On note ainsi les limitations suivantes au niveau des types :

- Les entiers doivent être codables sur 32 bits en tant qu’entiers signés positifs. On peut ainsi créer des entiers inclus entre (- 2^31 +1) et (2^31 - 1).

- Les flottants sont codés en simple précision sur 32 bits. La valeur minimale d’un flottant vaut 1.4e-45 et la valeur maximale vaut exactement (2-2^-23)*2^127. On ne peut pas convertir les flottants en entiers.

- Les chaînes de caractères ne peuvent être assignées à des variables. Le type “string” ne peut donc être utilisé qu’en entrant une chaîne de caractère dans une instruction “print” ou “println”.

- Les booléens prennent les valeurs “true” ou “false”, et ne ne sont pas interchangeables avec des entiers comme dans d’autres langages.

B. Classes et objets

Chaque classe instanciable crée un nouveau type, propre à cette classe. De nombreuses règles encadrent la programmation orientée objet du langage Deca, listées dans les spécifications du projet. On liste cependant les principales règles pour l'utilisateur :

- Un type est sous-type de lui-même.

- Les champs des classes peuvent être protégés ou publics, mais ils ne peuvent pas être privés. Pour accéder à un champ protégé, il faut que la classe courante soit un sous-type de la classe où le champ est déclaré.

- La redéfinition de méthode n'est possible que si la méthode a le même type (ou sous-type) de retour et la même signature que la méthode du même nom dans la super classe. Les méthodes qui retournent un type non vide sont obligées de retourner un objet qui est un sous-type du type de retour attendu.

- Il n'y a pas de constructeurs pour les objets, on utilise simplement un "A a = new A()" seul pour la création d'une instance de A.

- Les "casts" et "instanceof" ne sont pas implémentés.

C. Programme

Une des spécificités de langage Deca est que l'on peut faire un bloc "main" directement dans un fichier deca en dehors d'une classe, contrairement au langage Java. Ce bloc "main" commence par une accolade ouvrante "{" et se termine par une accolade fermante "}".

On ne peut pas déclarer des variables après une instruction, que ce soit dans le main ou dans une méthode. Les variables doivent donc être déclarées avant la première instruction Deca.

III. Messages d'erreur

A. Erreur du lexeur

Seuls les caractères ASCII sont acceptés par le lexer. Certains mots, précisés dans les spécifications du projet, sont réservés par le langage Deca et ne peuvent donc pas être utilisés comme nom de variable ou de classe.

B. Erreurs de syntaxe

Les erreurs de syntaxe sont dues au non-respect des règles de syntaxe du langage Deca. Celles-ci sont donc levées par le parser, et directement gérées par ANTLR. Voici la liste des erreurs possibles :

- *" No viable alternative at input '}' "*, pour une instruction incorrecte.
- *" Missing ('}' at '<EOF>' , missing ')' at ';' "* en encore *" Missing ';' at '}' "* si on oublie une parenthèse, une accolade ou un point virgule.
- "Entier non inclus entre -2^31 + 1 et 2^31 - 1" et "Flottant non valide", pour des instanciations d'entiers ou de flottants ne respectant pas les règles du langage Deca.

C. Erreurs de contexte (Contextual Error)

Les erreurs contextuelles sont détectées lors de la partie B du fonctionnement du compilateur, c'est-à-dire lors de la vérification et décoration de l'arbre. On propose ici une liste des méthodes levant des erreurs liées à cette partie :

- *AbstractExpr.verifyCondition* : renvoie une erreur si une condition dans un if ou un while n'est pas de type boolean.
- *Abstract.OpArith.verifyExpr* : renvoie une erreur si les opérandes d'une opération arithmétique ne sont ni de type int ou float.
- *AbstractOpBool.verifyExpr* : renvoie une erreur si les opérandes d'une opération binaire ne sont pas de type boolean.
- *AbstractOpCmp.verifyExpr* : renvoie une erreur si les opérandes d'une comparaison ne sont pas de types int ou float

- *AbstractPrint.verifyInst* : renvoie une erreur si on tente d'afficher autre chose que int, float ou une chaine de caractères entrée manuellement.

- *DeclVar.verifyDeclVar* : renvoie une erreur si on tente d'assigner une variable de type void ou string, ou on donne un nom de variable de mot clé.

- *Identifier.verifyExpr* : renvoie une erreur si le nom de variable utilisé est défini dans l'environnement.

- *Initialization.verifyInitialisation* : vérifie que la valeur est intitialisée au bon type, grace à la fonction Type.assign_compatible(T1, T2)

- *Modulo.verifyExpr* : vérifie que l'on utilise l'opérateur "%" avec des entiers.

- *Not.verifyExpr* : renvoie une erreur si l'on utilise l'opérateur not avec autre chose qu'un boolean.

- *UnaryMinus.verifyExpr* : renvoie une erreur si l'on utilise l'opérateur unaire moins avec autre chose qu'un int ou float.

- *InstanceOf.verifyExpr* : Renvoie une erreur si l'on utilise des types non existants ou une expression illégale

- *Cast.verifyExpr* : renvoie une erreur si le type utilisé n'est pas cast_compatible avec le type du cast.

- *DeclClass.verifyClass* : renvoie une erreur si l'on tente d'hériter d'une classe non définie avant dans le code, ou une classe déjà existante

- *DeclField.verifyDeclField* : renvoie une erreur si l'on définit un attribut qui est une méthode de la superclasse, ou de type void, ou string, ou qui est défini

- *DeclMethod.verifyDeclMeth* : renvoie une erreur si l'on tente de déclarer une méthode avec un type de retour ou une signature non sous-type de celle définie en superclasse si elle existe, ou de définir une méthode déjà définie.

- *Selection.verifyExpr* : renvoie une erreur si l'on sélectionne une attribut inexistant ou protégé d'une classe

- *MethodCall.verifyExpr* : renvoie une erreur si l'on appelle une méthode inexistante, avec les mauvais arguments, ou un objet null

- *This.verifyExpr* : renvoie une erreur si on utilise le this en dehors d'une methode

- *New.verifyExpr* : renvoie une erreur si l'on essaie un new avec une classe qui n'existe pas

D. Erreurs à l'exécution

Les erreurs à l'exécution correspondent à des erreurs levées lors de la partie C de la compilation, avec la traduction du programme en langage assembleur. Nous avons ainsi la liste des erreurs suivantes :

- *div_zero_error* : Erreur de division par zéro.
- *overflow_error* : Dépassement lors d'une opération arithmétique.
- *stack_overflow_error* : Si on appelle trop de fois une fonction récursive.
- *io_error* : Erreur de lecture pour les méthodes readInt ou readFloat.
- *dereferencement.null* : Appel de méthode ou sélection sur un objet null.
- *tas_plein* : Allocation impossible car la pile est pleine.
- *cast_error* : Impossible de réaliser le cast.

IV. Exécution du compilateur et options

La compilation d'un programme Deca nommé "program.deca" se fait dans un terminal, avec la commande: "decac program.deca".

Sans options ni erreur de compilation, decac créé un fichier "program.ass", exécutable par la machine abstraite proposée dans ce projet par la commande suivante : "ima program.ass".

On peut ajouter des options au compilateur decac, permettant par exemple de compiler plusieurs fichiers en même temps.

La commande générale s'écrit "decac [options] [files]".

Les options disponibles sont :

- -b,--banner : Affiche une bannière indiquant le nom de l'équipe.
- -d,--debug : Active les traces de debug, option pouvant être repétée.
- -n,--no-check : Supprime les tests de débordement à l'exécution.

- -p,--parse : Arrête decac après l'étape de construction de l'arbre, et affiche la décompilation de ce dernier.
- -P,--parallel : S'il y a plusieurs fichiers source, lance leur compilation en parallèle.
- -r,--registers <n> : Précise le nombre de registres utilisables, avec 3 < n < 17.
- -v,--verification : Arrête decac après l'étape de vérifications.

V. Extension TRIGO : Méthodes et Limitation

A. Présentation de la bibliothèque

Le compilateur decac est complété d'une extension « TRIGO » qui permet l'utilisation des fonctions trigonométriques sinus, cosinus, arctangente et arcsinus. Une fonction nommée ULP, pour Unit in the Last Place, est également implémentée. Ces fonctions sont présentes dans la bibliothèque standard du compilateur, plus précisément dans la classe Math du fichier src/main/resources/include/Math.decah. Pour utiliser ces fonctions dans un programme deca, il est nécessaire d'ajouter la mention « #include "Math.decah" » en tête du fichier. Puis, il suffit de déclarer une variable de type Math dans le programme et d'utiliser new pour créer une instance de la classe et de l'associer à cette dernière. Cela peut être effectué en une seule déclaration (cf. exemple ci-dessous). Ensuite, les méthodes de la classe Math peuvent simplement être appelées à partir de l'objet. Voici un exemple avec la fonction sinus :

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

B. Les méthodes de la classe Math

La classe Math permet l'utilisation de quatre fonctions trigonométriques ainsi que d'une fonction de calcul de la précision sur les flottants. Ces méthodes prennent toutes en argument un flottant et retournent également un flottant.

- **float ulp(float a)** : Cette méthode retourne la distance, toujours positive, entre le flottant a et le flottant qui le succède directement.

- **float sin(float a)** : Cette méthode retourne une approximation du sinus de a. Si l'argument passé en entrée est inférieur ou égal à π/4, une approximation précise de la valeur réelle est calculée directement par un développement en série de Taylor avec les 7 premiers termes. Si l'argument est supérieur à π/4,

l'approximation du sinus est calculée par équivalence avec un appel de la méthode cosinus, l'objectif étant de se retrouver avec un angle inférieur à π/4.

- **float cos(float a)** : Cette méthode retourne une approximation du cosinus de a. Si l'argument passé en entrée est inférieur ou égal à π/4, une approximation précise de la valeur réelle est calculée directement par un développement en série de Taylor avec les 8 premiers termes. Si l'argument est supérieur à π/4, l'approximation du cosinus est calculée par équivalence avec un appel de la méthode sinus, l'objectif étant, comme pour le calcul du sinus, de se retrouver avec un angle inférieur à π/4.

- **float asin(float a)** : Cette méthode retourne une approximation de l'arcsinus de a comprise entre l'équivalent en flottant de -π/2 et π/2 (-1.5707964 et 1.5707964). L'argument doit être compris entre -1 et 1 sinon la méthode s'arrête et retourne 0. Si l'argument est supérieur à -0.5 et inférieur à 0.5, l'approximation est calculée par un développement en série de Taylor avec les 4 premiers termes. Sinon, l'approximation est calculée par une série utilisant π/2 et une racine carrée.

- **float atan(float a)** : Cette méthode retourne une approximation de l'arctangente de a. Si l'argument est compris entre -1 et 1, l'approximation est obtenue en calculant π/4x + 0.273x (1−|x|). Sinon l'approximation est obtenue en calculant 2arctan(a / (1 + sqrt(1 + a ²))) avec sqrt le calcul de la racine carrée.

C. Les limites de la classe Math

La fonction ulp retourne une valeur exacte alors que les méthodes trigonométriques retournent une valeur approximative de la valeur réelle. En effet, l'utilisation obligatoire des développements en série pour un nombre de termes finis implique une différence entre la valeur réelle et le résultat obtenu. De plus, la manière dont les flottants sont construits implique parfois un arrondi de la valeur renvoyée. Lors de l'utilisation de la classe Math il faut avoir conscience que la valeur obtenue n'est pas toujours parfaitement la valeur réelle. Cependant, les quatre méthodes trigonométriques retournent des valeurs très précises avec une marge d'erreur relativement petite et le plus souvent négligeable. Il faut tout de même prendre en compte que plus l'angle passé en entrée est grand, plus l'imprécision sera importante. Par exemple, pour des angles de l'ordre des dizaines de milliers de radians, la marge d'erreur peut être au niveau de l'unité ce qui fait perdre toute pertinence au résultat obtenu.