PROJET GÉNIE LOGICIEL

2023/2024

Manuel Utilisateur du Compilateur Deca

Antoine Borget
Pierre Lin
Anass Haydar
Yassine Safraoui
Jihane Zemmoura

Groupe GL-47

Table des matières

1 Introduction

Ce document est conçu pour fournir des informations essentielles à tout utilisateur actuel ou potentiel du compilateur Deca. Elle s'adresse spécifiquement aux utilisateurs familiers avec les spécifications du langage Deca. Ce manuel-utilisateur comprendra une description complète du compilateur, de ses commandes ainsi que ses options de compilation. De plus, une grande partie de la documentation répertoriera les messages d'erreur pouvant être générés par le compilateur, en précisant leur nature (lexicographique, syntaxique, contextuelle, d'exécution du code assembleur) et les configurations associées. Enfin, pour les utilisateurs intéressés par l'extension GameBoy du compilateur, une partie de la documentation fournira des instructions détaillées sur son utilisation ainsi que sur la bibliothèque développée.

2 Description du compilateur

Le compilateur Deca est un outil logiciel écrit en langage Java qui permet de traduire un fichier source écrit en langage Deca en un fichier assembleur compatible avec une machine abstraite. Grâce à sa mise en œuvre en Java, le compilateur Deca bénéficie de la portabilité offerte par la machine virtuelle Java (JVM), ce qui signifie qu'il peut être exécuté sur diverses plate-formes sans nécessiter de modifications majeures.

Le compilateur effectue l'analyse lexicale, syntaxique et contextuelle du code source Deca afin de produire un arbre abstrait décoré à partir duquel on génère un code assembleur exécutable par une machine abstraite nommée IMA.

Le compilateur développé est **complet** et ne présente aucune limitation. Il traite tous les sous-ensembles du langage Deca et toute sa lexicographie y compris les inclusions de fichiers (Include), les conversions (Cast) et les tests d'appartenance à une classe (InstanceOf).

3 Commandes et options du compilateur

Le programme principal « decac » est un compilateur Deca complet. Le fichier d'entrée doit être un fichier d'extension .deca, et, sans erreur, le résultat de la compilation génère un fichier nom_fichier.ass dans le même répertoire que le fichier source.

La syntaxe d'utilisation de l'exécutable decac est :
decac [[-p | -v] [-n] [-r X] [-d]* [-P] nom_fichier.deca] | [-b]

La commande decac, sans argument, affichera les options disponibles. On peut appeler la commande decac avec un ou plusieurs fichiers sources Deca.

Voici la description des options de compilation citées dans la syntaxe :

. **-b** (banner) : Affiche une bannière indiquant le numéro de notre 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 est 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 certains tests d'erreur à l'exécution. Les erreurs supprimées sont détaillés dans la section 4.4.1.

. **-r X** (registers) : Limite le nombre de registre utilisé à X, où $4 \leq X \leq 16$.

. **-d** (debug) : Active les traces de debug. Il est possible de répéter l'option jusqu'à 3 fois pour avoir plus de traces de debug.

. **-P** (parallel) : S'il y a plusieurs fichiers sources, cette option lance la compilation des fichiers en parallèle en utilisant un thread pour chaque fichier (pour accélérer la compilation).

Remarques :

— Les options '-p' et '-v' sont incompatibles.

— En l'absence des options '-b', '-p' et '-d', une exécution de decac ne produit aucun affichage si la compilation réussit.

— L'exécution de decac ne lit pas d'entrée sur son entrée standard.

— L'option '-b' ne peut être utilisée que sans autre option, et sans fichier source. Dans ce cas, decac termine après avoir affiché la bannière.

— Si un même fichier apparaît plusieurs fois sur la ligne de commande, il n'est compilé qu'une seule fois.

4 Messages d'erreurs

On pourra tomber sur plusieurs type d'erreurs au cours de l'utilisation du compilateur, les erreurs lexicales, syntaxiques, et contextuelles s'afficheront sous cette forme :

<chemin/du/fichier.deca> :numLigne :numColonne : message d'erreur

4.1 Les erreurs lexicales

Le programme est faux au niveau lexical si le compilateur ne reconnaît pas un mot.

On aura alors des erreurs de ce type :

<unknown> :1 :0 : token recognition error at : 'ç'

4.2 Les erreurs syntaxiques

Le programme est faux syntaxiquement : la structure du code est erronée. Par exemple s'il manque un mot attendu, ou si l'ordre des mots n'est pas respecté.

Par exemple :

```
{ print "bonjour"; }
```

donnera : *<unknown> :2 :7 : missing '(' at '"bonjour"'*

Ou encore :

```
{ print("hello world");
  int a = 1; }
```

donnera : *<unknown> :3 :5 : no viable alternative at input 'a'*

En effet, les déclarations de variables doivent être mises avant les instructions dans un bloc.

4.3 Les erreurs contextuelles

Le programme est juste syntaxiquement, mais il y a des incohérences de type, des expressions non déclarées etc...

Par exemple :

```
{ int a = "bonjour"; }
```

donnera :*<unknown> :2 :9 : type 'string' must be compatible with type 'int'.*

4.3.1 Erreurs d'identificateur

- *"Undeclared identifier : 'name'."*

- *"Undeclared type identifier : 'name'."*

4.3.2 Déclarations de classes :

- *“Undeclared super class identifier : ‘name’.”*

- Pour la classe super :
"A class identifier is required."

- *"Class or type already exists."*

- *"’name’ is a field and a method at once."*

Déclarations de champs :

- *"Field ‘name’ already defined."*

- *“Field type cannot be void”*

Déclarations de méthodes :

- *“Method ‘name’ already defined.”*

- *“A field ‘name’ already exists in super class”*

- *“Method defined in super class with another signature”*

- *“Return type of override must be subtype of the return type of the method declared in super class”*

- *“Parameter type cannot be void”*

- *“Parameter ‘name’ already exists”*

4.3.3 Dans un bloc

Déclarations de variables :

- *“Variable ‘name’ already declared”*

- *“Variable type cannot be void”*

- *“Cannot use ‘return’ in void method.”*

- Assignation ou expression passée en argument :
“Type ‘type2’ must be compatible with type ‘type1’”

- Dans un bloc if/else ou while :
“Condition is of type ‘type’, it must be boolean”

- *“Binary operation ‘op’ cannot have operands of types : ‘type1’, ‘type2’”*

- *“Unary operation ‘op’ cannot have operands of types : ‘type’”*

- Usage de ’new’ :
“‘type’ is not a class.”

- *“‘this’ cannot refer to Object class.”*

- Cas d’une selection ’a.x’ :
“Expression type must be a class.”

- Selection sur un champs non visible :
"’expr’ is protected."

- Si l’on tente de réassigner une méthode :
“‘name’ must be a field, a parameter or a variable, but it is a method.”

- Dans une selection :
“‘name’ must be a field.”

“Method call on native type.”

“‘name’ is not a method identifier.”

Appels de méthodes :

- *“Too few arguments, expected type : ‘type’.”*

- *“Too many arguments.”*

4.4 Les erreurs à l’exécution

Ces erreurs ne sont pas détectées au cours de la compilation mais lors de l’exécution. En l’absence de l’option ’-n’, l’exécution provoquera soit un arrêt du programme soit un comportement indéfini.

4.4.1 Cas d’arrêt de l’exécution du programme

— Division entière par 0 : *"Error : Division by 0"*.
• Cette erreur s’affiche lorsque l’on fait une division entière (et seulement entière) par 0. Cela inclut le modulo entre deux entiers.

— Débordement arithmétique sur les flottants : *"Error : Float Operation Overflow"*.
• Cette erreur se produit lorsqu’une opération arithmétique entre deux flottant dépasse le maximum autorisé, le maximum étant 3.4028235×10^{38}. La division flottante par 0.0 est incluse dans cette erreur.

— Absence de return : *"Error : Exiting function [className.methodName()] without return"*.
• Ce message s’affiche si on sort d’une méthode (de type différent de void) sans passer par un return. Si la méthode est de type void, il n’est pas obligatoire de passer par un return.

— Déréférencement de null : *"Error : Dereferencing Null Pointer"*.
• Cette erreur se produit lorsqu’on essaie de déréférencer un objet null, que ce soit pour accéder à un attribut ou pour appeler une méthode.

— Erreur de lecture : *"Error : Input/Output Error"*.
• Cette erreur est affichée dans le cas où la saisie de l’utilisateur du programme est une valeur incorrecte pour un readInt() ou un readFloat(). Saisir un entier tel que 6 pour un readFloat() n’est pas possible, il faut écrire 6.0.

— Débordement de la pile : *"Error : Stack Overflow"*.
• Ce message s'affiche quand la place restante dans la pile n'est pas suffisante pour exécuter correctement le programme principale, ou une méthode.

— Débordement du tas : *"Error : Heap Overflow"*.
• Cette erreur se produit quand la place restante dans le tas n'est pas suffisante pour allouer une instance d'une classe.

— Cast invalide : *"Error : Failed to cast variable of type [className1] to class [className2]"*.
• Cette erreur s'affiche quand on essaie de faire une conversion de type impossible

L'option '-n' du compilateur supprime les tests à l'exécution pour chacune de ces erreurs, sauf pour l'erreur de lecture.

4.4.2 Cas du comportement indéfini

— L'utilisation d'une variable globale non initialisée dans le programme principal peut soit provoquer une erreur IMA, soit avoir une valeur indeterminée. Il est cependant possible d'utiliser les attributs d'une classe sans initilisation, ceux-ci sont automatiquement initialisés à 0 par défaut.

— L'utilisation d'une méthode écrite en assembleur (asm) peut provoquer un comportement indéfini si elle ne respecte pas les conventions de liaison ou la syntaxe de l'assembleur. La sauvegarde des registres et le retour d'une fonction sont à la charge du programmeur.

5 L'Extension GameBoy

5.1 Installation et Compilation vers une GameBoy

En plus de compiler vers la machine IMA, notre compilateur decac permet aussi de compiler un code Deca pour la console Nintendo GameBoy (sortie en 1989).

Pour lancer le jeu GameBoy créé par le compilateur, il faut tout d'abord installer le compilateur RGBDS qui sera l'intermédiaire entre l'assembleur généré et l'assembleur exécuté par la GameBoy. Ensuite, il faut installer un émulateur GameBoy, nous recommandons Emulicious.

Par la suite, il faut utiliser le compilateur decac avec le flag '-g' pour compiler le programme vers un fichier .asm. Ensuite, il faut compiler ce fichier avec RGBDS à travers les commandes suivantes (voici un article qui explique ces commandes) :
- **rgbasm -L -o file.o file.asm**
- **rgblink -o file.gb file.o**
- **rgbfix -v -p 0xFF file.gb**

Finalement, on peut simuler le jeu en ouvrant le fichier **file.gb** avec Emulicious ou tout autre simulateur GameBoy.

Pour faciliter cette phase de compilation, vous pouvez utiliser le flag '-gb' du compilateur Deca pour directement générer le fichier .gb final. Néanmoins, cette option nécessite d'avoir mis RGBDS dans les variables d'environnements.

Par exemple sur Linux, on mettrait dans le fichier .bashrc :
- **PATH=$HOME/Chemin/Vers/rgbds :"$PATH"**
- **export PATH**

5.2 Introduction à la GameBoy

Les *tiles* sont des carrés de 8×8 pixels que l'on peut placer aux emplacements dédiés sur deux cartes de *tiles* : la *background map* et la *foreground map*, on ne peut pas les placer à cheval sur deux emplacements.

La GameBoy dispose de 128 *tiles* différentes au maximum, cependant, l'écran étant de dimension 160×144 pixels, il est possible d'afficher 360 *tiles* en même temps, donc des *tiles* seront utilisées plusieurs fois, comme des blocs de ciel ou de terre.

On dispose aussi des *tilemaps* qui permettent de remplir tout l'écran et pas qu'une seule case avec un ensemble de *tiles* prédéfinis. Par exemple, les *tilemaps* sont utiles pour afficher les écrans d'accueil sans avoir besoin de mettre chaque tile à son endroit manuellement avec Deca.

En mémoire, une *tile* est composée de 16 octets (64 pixels $\times$ 4 couleurs). **On peut générer une *tile* ou une *tilemap* entière à l'aide d'outils comme celui-ci, à l'aide d'une image, ou comme celui-là, avec lequel on peut designer manuellement les *tiles*, ou bien avec Emulicious.**

Si on veut afficher un objet sur l'écran avec plus de précision, on peut se servir des *sprites*. Un *sprite* est un bloc de 8×8 pixels que l'on peut placer sur n'importe quel position (en pixels) dans l'écran, on peut aussi les retourner verticalement et/ou horizontalement, ce qui est très utile pour afficher les caractères du jeu ainsi que les objets spéciaux. Par exemple, dans jeu de Super Mario, on peut les utiliser pour afficher le personnage principal Mario, les monstres, les blocs, et ainsi de suite. Cependant, la limitation des *sprites* est qu'on ne peut pas afficher plus que 40 *sprites* au total et 10 *sprites* par ligne.

5.3 Utilisation de la librarie GameBoy

Pour utiliser l'éxtension, nous devons instancier un objet de la classe GameBoy, disponible dans **GameBoy.deca**.

Il faudra appeler la méthode **init()** avant toute intéraction avec la GameBoy, celle-ci mettra le fond en blanc par defaut.

Pour afficher une *tile* donnée dans un endroit spécifique de l'écran, il suffit d'appeler la méthode **setColor(Color c, int x, int y)** de la classe GameBoy, on peut accéder aux quatres couleurs avec les constantes de la classe *GameBoy* : **WHITE**, **LIGHT**, **DARK**, **BLACK**.

Pour changer la couleur du fond, il faut utiliser **setBackgroundColor(Color c)**.

Pour utiliser ses propres *tiles*, l'instruction **#includeTiles** permet d'importer une ou plusieurs *tiles* pour les utiliser plus tard dans le programme. L'instruction **#includeTilemaps** fait la même chose pour les *tilemaps*.

On pourra utiliser ces *tiles* avec les méthodes **setTile(int tileIndex, int x, int y)** pour placer une *tile* sur une position, ou encore **setBackgroundTile(tileIndex)**, où *tileIndex* est l'indice de la *tile* dans le(s) fichier(s) que l'on a inclus.

Pour afficher sur l'écran une *tilemap* importée pour être un fond, on utilisera la méthode **setBackgroundMap(int index)**, où *index* est l'ordre dans les imports.

On peut accéder aux *input* de l'utilisateur via la méthode **keyPressed(int key)**, on passera en paramètre l'une des constantes de la classe *GameBoy* : **UP_KEY**, **DOWN_KEY**, **LEFT_KEY**, **RIGHT_KEY**.

Pour afficher à l'écran les changements, on utilisera **updateScreen()**, qui renverra *vrai* si les changements ont été affichés, et *faux* sinon. En effet on ne peut afficher nos changements que pendant une certaine période du rafraîchissement de l'écran, la fonction **updateScreen()** n'attend pas cette période. De cette manière, on peut avoir une boucle plus rapide que le rafraîchissement de l'écran, et ainsi on peut prendre les *input* de l'utilisateur en compte de manière plus sûre. Par exemple :

```
{
    ...
    while (playing) {
        if(keyPressed(...) {
            // prise en compte des input
            ...
        }
        updated = gb.updateScreen();
        if(updated) {
            // changements suivants
            ...
        }
    }
}
```

5.4 Limitations du langage Deca pour la GameBoy

— La GameBoy n'ayant aucune console pour afficher du texte, les erreurs à l'exécution citées dans la partie 4.4 ne sont pas gérées. Toutes les erreurs de la partie 4.4.1 provoquent un comportement indéfini.

— Dû aux limitations des instructions assembleurs disponibles en GameBoy, il n'y a pas de type dynamique pour les objets à l'exécution. C'est-à-dire que tous les appels de méthode se font par rapport au type statique.

Par exemple, si on a :

```
class A { int f() { return 1; } }
class B { int f() { return 2; } }
{ A a = new B();
int x = a.f(); }
```

Alors, la méthode appelée sera celle de A et non de celle B, contrairement à ce qu'il se passerait sur IMA. La variable x contiendra donc 1 et non 2.
En résumé, il n'y a pas de polymorphisme pour la compilation GameBoy.

— Les registres de la GameBoy ne pouvant stocker que des entiers, toutes les expressions de type flottant dans le programme sont convertis en entier (arrondi vers le bas). Il est donc inutile d'utiliser le type *float* si on veut programmer pour la GameBoy.

— Comme les registres de la GameBoy sont tous sur 8-bits, les entiers (et booléens) sont codés sur 8-bits. De plus, ces entiers sont tous positifs. Une variable de type *int* peut donc stocker un entier de 0 à 255. Comme cet intervalle est très petit, il faut faire très attention à la manipulation de ces variables. Essayer de stocker un nombre négatif ou supérieur à 255 dans une variable provoque un comportement indéfini.

— La mémoire de la GameBoy étant très réduite, il est possible de stocker jusqu'à 4096 variables (un peu moins en réalité car il faut compter les appels de méthodes et la sauvegarde de registre temporaire aussi) avant d'avoir un comportement indéfini. Ceci inclus aussi les attributs de classes qui sont tous alloués dans la pile, la GameBoy n'ayant pas de tas. Il est donc très important de ne pas utiliser *new* n'importe comment, car la libération de mémoire est impossible en Deca.

Par exemple, il faut éviter de faire un *new* dans une boucle qui est exécutée un grand nombre de fois. De même, il faut éviter de faire un *new* dans une méthode appelée beaucoup de fois.

Néanmoins, les variables des types de bases (*int*, *boolean*) peuvent être déclarées dans les méthodes sans problème, car elle seront libérées à la fin de la méthode.

On pourra se référer à cette formule qui donne une borne supérieure au nombre de cases utilisées dans la pile :
(Nombre de variables globales)
+ (Nombre d'attributs pour chaque 'new' ayant été fait depuis le début de l'exécution)
+ ($3\times$Nombre d'appels de méthode imbriqués)
+ (Nombre d'argument de tous les appels de méthode imbriquées)
+ (Nombre de variables locales de tous les appels de méthode imbriquées)
+ (Nombre de valeurs sauvegardées temporairement dans la pile)
< 4096.

— À cause de certaines limitations, il faut respecter la formule suivante pour chaque méthode (sinon le comportement est indéfini) :
(Nombre de paramètres de la méthode courante)
+ (Nombre de variables locales dans la méthode)
+ (Max(Nombre de paramètres des méthodes appelées dans la méthode))
< 64.
Par exemple pour la méthode suivante :

```
void f(int x) {
    A a = new A();
    int y;
    int z = 2;
    f1(x, y);
    f2(x, y, z, a);
}
```

La formule donne : $1 + 3 + \text{Max}(2, 4) = 8$.
Cette limitation ne devrait pas être restrictif pour la plupart des programmes.

En dehors de tout cela, Deca pour la GameBoy fonctionne de la même manière que pour IMA.