UGA – Grenoble INP

Projet Génie Logiciel

Documentation utilisateur

Grenoble, 24 janvier 2022.

Table des matières

1 Présentation de notre produit

Le langage Deca est une sorte de sous-langage de Java, ce qui signifie qu'il est orienté objet et repose sur un typage fort. Notre produit est un compilateur pour ce langage, il permet à l'utilisateur de transformer son code Deca en un code de plus bas niveau (langage d'assemblage).

Afin d'exécuter notre programme, il est obligatoire de télécharger *maven*, qui est un outil d'automatisation de la production de logiciels Java, et il est nécessaire d'avoir une version relativement récente de *Java*. (Java 16 ou plus récent).

Le produit est un package qui peut être téléchargé à partir du dépôt Git suivant (disponible uniquement pour les utilisateurs de gitlab.ensimag) :

https://gitlab.ensimag.fr/gl2022/g3/gl13/-/tree/master

Le paquet contient de nombreux fichiers dont un fichier *readme.md* qui présente certaines des fonctionnalités et certaines lignes de commande intéressantes pour l'exploitation du compilateur.

1.1 Compilation et exécution

Pour compiler le projet, il faut lancer la ligne de commande suivante dans un terminal :

```
mvn compile
```

Le programme exécutable se trouve dans le dépôt *decac* ; pour l'exécuter il faut lancer la commande suivante dans un terminal :

```
./src/main/bin/decac
```

Il existe différentes options du compilateur comme décrite dans la section 5. Pour une exécution normale et sans options du compilateur, il suffit d'écrire dans le terminal :

```
decac <fichier1.deca> <fichier2.deca> \dots
```

Un fichier assembleur *.ass* est alors généré. Pour exécuter ce fichier assembleur, il faut lancer dans le terminal :

```
ima fichier.ass
```

Le résultat apparaît alors dans le terminal.

En cas de mauvaise reconnaissance des chemins vers les différents scripts, il peut être nécessaire de lancer la commande suivante dans le terminal courant (notamment pour l'utilisation de l'extension) :

```
source env_settup.sh
```

1.2 Exécution de tests

Les tests sont placés dans *src/test/script*, afin de rendre un test exécutable, l'utilisateur est invité à exécuter la commande suivante :

```
chmod 0755 test-name.sh
```

Lorsqu'un test est en cours d'exécution, la présentation du test apparaît en caractères jaunes dans le terminal, et une fois exécuté, le résultat du test apparaît à côté de son nom. Le résultat du test est l'un des fichiers suivants :

- un fichier *.lis* si le test résulte d'un analyseur syntaxique ou d'un lexer
- un fichier *.ass* si le fichier .deca a été compilé en langage assembleur
- un fichier *.res* si le fichier résulte d'une exécution
- un fichier *.j* et *.class* si le fichier a été compilé en Jasmin (cf extension Bytecode)

Ces fichiers contiendront soit le résultat du programme montrant qu'il fonctionne correctement, soit les erreurs rencontrées par le programme lors de son exécution.

1.3 Automatisation des tests

Il y a quelques tests dans le fichier *pom.xml* ; ces tests peuvent être exécutés en utilisant la commande suivante dans un terminal :

```
mvn test
```

Ce programme s'arrête lorsqu'il y a des erreurs dans l'exécution. Afin d'éviter que cela ne se produise, l'utilisateur peut choisir d'ignorer les erreurs et d'exécuter tous les tests avec la commande suivante dans le terminal :

```
mvn test -Dmaven.test.failure.ignore
```

L'utilisateur peut aussi utiliser la ligne de commande suivante pour lancer les tests :

```
mvn verify
```

Toutefois, cette commande ne donne pas autant d'informations à propos des classes et de leur exécution que la première commande.

1.4 Couverture de tests

L'utilisateur peut obtenir une couverture complète des tests en exécutant la commande suivante dans un terminal :

```
mvn verify
```

Afin d'avoir un accès graphique de cette couverture, la commande suivante peut être exécutée :

```
jacoco-report.sh
```

Puis il faut ouvrir le fichier *index.html* en utilisant, par exemple, le navigateur firefox en exécutant la commande suivante dans un terminal :

```
firefox target/site/index.html
```

2 Limites de notre compilateur

Le compilateur présente plusieurs limitations dues soit aux limitations des différents outils utilisés soit à l'implémentation du compilateur.

— De part l'utilisation de l'API Java Integer.ParseInt(), les entiers doivent être compris entre -2^{-31} et $2^{31} - 1$. De même pour les flottants, ils seront donc compris entre 2^{-149} et $(2 - 2^{-23}).2^{127}$ dû à l'utilisation de ParseFloat().

— Pour les print/println, seuls peuvent être affichés les entiers, les flottants et les chaînes de caractères ; les booléens ne peuvent donc pas être affichés. De plus, seuls les flottants ont la possibilité d'être affichés en hexadécimal.

— Concernant les classes nous avons différentes limitations. Les champs d'une classe ne peuvent être déclarés qu'en *protected* ou en *public* ; ils ne peuvent donc pas être restreints à la classe seule avec le mot clef *private*. Un champ et une méthode ne peuvent pas avoir le même nom dans une classe donnée. Mais, une méthode peut être redéfinie dans une sous-classe sans avoir à utiliser le mot clef *Override*.

— Pour les Cast, seuls ont été traités les Cast d'un entier vers un flottant, le cas inverse ou les Cast triviaux d'un type vers le même type. Les Cast d'une classe vers sa sous-classe n'ont pas été traités car cela nécessiterait l'utilisation de InstanceOf qui lui même n'a pas été implémenté.

3 Messages d'erreurs

3.1 Erreur de Lexicographie

Les erreurs lexicographiques sont lancées quand le compilateur ne peut pas générer le token de l'information donnée. Ces erreurs apparaîtront dans les fichiers *.lis* qui sont générés automatiquement après le lancement du test automatique.

L'erreur est écrite sous le format :

./**location**/*file :a :b :* token recognition error

Où :

— a est le numéro de la ligne où se situe l'erreur dans le fichier

— b est le numéro de la colonne où se situe l'erreur dans le fichier

3.2 Erreurs Syntaxiques

Les erreurs syntaxiques sont lancées lorsque le programme n'est pas correct lors de l'assemblage des Token sans qu'il y ait un lien avec le contexte.

Ainsi, **Recognition Exception** correspond à un échec de l'analyse syntaxique. Elle est réduite lorsque le parser a reconnu l'élément manquant et le propose ensuite. Cela peut donc correspondre à un oubli de point virgule ou une parenthèse non fermée.

Ces erreurs sont générées automatiquement par la super classe AbstractDecaParser qui est responsable de l'implémentation de toutes les méthodes auxiliaires, des variables d'instance et des surcharges des méthodes ANTLR par défaut.

Il n'y a pas de nouvelles erreurs sauf celles par défaut.

3.3 Erreurs Contextuelles

Les erreurs contextuelles sont lancées par le compilateur lorsque les Token n'ont pas de sens lors de la lecture du code. Par exemple, cela peut regrouper un mauvais typage d'une variable ou alors une méthode non void ne possédant pas de return.

Les erreurs contextuelles sont générées par la classe ContextualError qui hérite de LocationException.

Les erreurs contextuelles se produisent s'il y a une :

— Erreur de Définition

— Erreur de Type

— Erreur de Déclaration de Variables

— Erreur d'Identifiant

— Erreur d'Initialisation

Ainsi, les erreurs contextuelles sont :

— **type <Type> is not supported by print or println** pour l'affichage d'une expression différente d'un nombre ou d'une chaîne de caractères.

```
println(true);
```

— **<Condition> should have been boolean** lorsque la condition dans un if n'est pas un booléen.

```
if(3){ ... }
```

— **The condition of a while loop should be a bool but here it is a <Condition Type>** est levée de même lorsque la condition du while n'est pas un booléen.

```
while(3){ ... }
```

— **Variables can't be void type** quand l'utilisateur essaye de déclarer une variable ayant un type void.

```
void x;
```

— **No such type <Type>** lorsque le type d'une variable n'a jamais été déclaré avant.

```
A a;
```

— **<Type Right> isn't a subtype of <Type Left> so it cannot be assign to it** lorsque l'utilisateur essaye d'assigner à une variable un type qui ne lui correspond pas.

```
boolean b = 3;  ou  int i = 2.3;  ou  float f = "hello";
```

— **This variable <Name> is not declared yet** quand une variable est utilisée sans avoir été déclarée avant.

```
int a = x+1        ici x n'est pas déclaré
```

— **Double definition if this identifier <Name>** quand une variable est définie plusieurs fois.

```
int x;
int x=1;
```

— **Variables, Parameters or Field can't be void type** quand l'utilisateur essaye de déclarer une variable ou des attributs ou des paramètres d'une méthode de type void.

```
void x;
class A {
    void x;
    int method(void x){ ... }
}
```

— **The two operands must be booleans for And/Or operation** car les opérations And et Or ne peuvent prendre que des booléens.

```
(false || ("string" && 1) )
```

— **! is only for boolean but the operand is : <Operand Type>** lorsque "!" est utilisé pour autre chose que sur un booléen.

```
int x = ! 3  ou  string str = ! "Hello"
```

— **Unary Minus not possible for type <Type>** lorsque le moins unaire est utilisé pour autre chose qu'un entier ou un flottant.

```
string a = -"string"
```

— **Modulo should be between Int but operand <Left/Right> is type <Type>** pour un modulo avec un ou des types interdits : un type différent d'un entier.

```
int a = 1 % 2.0
```

— **Can't do aritmetical operation between types <Type left> and <Type right>** lorsqu'une opération arithmétique est faite entre des types autres que des entiers ou des flottants.

```
"hello" - "world" ou  1 + true
```

— **Impossible to compare <Type Left> and <Type Right>** lors de l'utilisation des opérations de comparaison ($<, >, \leq, \geq$) pour des types différents d'entiers ou flottants.

```
true < false  ou  "hello" > "word"
```

— **Class <Class> and <Class> are not compatible** lors d'un cast entre deux classes où la classe à transtyper n'est pas une sous-classe de l'autre.

```
class A { ...}
class B extends A { ... }
B b = new B();
A a = new A();
a = (A) (b); ici B n'est pas une sous-classe de A donc cela n'est pas possible
(mais l'inverse l'est)
```

— **Cast is not possible between <Type To Cast> and <Type>** quand le type de l'expression n'est pas transtypable.

```
int a = (int) ("Hello");
```

— **<Class> already declared** quand une classe possède déjà ce nom.

```
class A{ ... }
class A{ ... }
```

— **class null** lorsqu'une classe définie est nulle, c'est à dire qu'elle n'a pas de définition propre.

```
class A extends B{ ... }
{
    A a = new A();
}
```

— **Type <Nom> isn't a class** lorsqu'un identifiant est utilisé comme une classe alors qu'il n'est pas une classe.

```
class A extends float { ... }
```

— **<Field> already declared** lorsqu'un champ portant le même nom a déjà été déclaré dans cette classe.

```
class A {
    int x;
    int x;
}
```

— **The field <Field Name> is not declared yet** lorsque dans une classe un attribut est utilisé sans avoir été déclaré avant.

```
class A {
    int add(int x){
        return this.y + x;
    }
}
```

— **<Method> already declared** quand dans cette classe il existe déjà une méthode possédant ce nom

```
class A {
    void method(){...}
    void method(){...}
}
```

— **The method <Name> doesn't exist** quand une méthode est appelée sans avoir été définie dans la classe.

```
class A { ... }
A a = new A();
a.method();
```

— **Methods have different number of arguments** lorsqu'une méthode étant redéfinie dans une classe fille ne possède pas le même nombre d'arguments que lors de sa définition initiale.

```
class A {
    int add(int x){
        return x+1;
    }
}
class B extends A {
    int add(int x, int y){
        return x+y+1;
    }
}
```

— **Methods have <Type 1 arg> and <Type 2 arg> types for the same paramater** lorsqu'une méthode est redéfinie mais que le typage d'un même identifiant n'est pas respecté.

```
class A {
    int add(int x){
```

```
            return x+1;
        }
    }
    class B extends A {
        int add(float x, int y){
            return x+2;
        }
    }
```

— **Cannot Override <Field/Method Name>** lorsqu'une classe fille essaye de déclarer un attribut portant le même nom qu'un autre attribut de la classe mère mais ayant une signature différente.

```
class A {
    int a;
}
classe B extends A {
    int a(){ ... }
}
```

— **<Parameter> already in the parameters** lorsque l'on déclare deux fois une variable dans une fonction.

```
class A{
    void inutile(int x, int x){
        int y = x+x;
    }
}
```

— **Bad return type** quand une méthode ne renvoie pas le type lié à sa signature.

```
class A {
     int method(){
          return 2.0;
     }
}
```

— **Must be a Class Type before the selection** quand l'objet n'est pas une classe et qu'on utilise une sélection.

```
int a;
a.x = 1;
```

— **Access to the protected member <Name> denied** lorsqu'on essaye d'accéder à un attribut *protected* depuis une classe qui n'est pas l'une de ses sous-classes.

```
class A {
    protected int x=1;
}
A a = new A();
println(a.x);
```

— **Identifier <Name> cannot be used as an expression because it is of kind <Nature>** lorsqu'un objet est utilisé comme une expression alors que ce n'est pas possible.

```
class A {
    void m(){
        println(m);
    }
}
```

3.4 Erreurs d'exécution du code assembleur

Les erreurs liées à l'exécution du code sont dues à la génération de différents labels d'erreurs. Ces labels d'erreurs permettent de protéger les débordements que peuvent rencontrer les programmes lorsque l'on exécute les programmes assembleurs avec IMA. Ils préviennent ainsi des erreurs qui sont syntaxiquement et contextuellement indétectables, mais qui, à cause de l'architecture machine, sont problématiques.

Les différentes erreurs d'exécution sont les suivantes :

— **Error : Input/Output Error** correspond à une erreur lors d'une lecture sur l'entrée standard ou d'une écriture sur la sortie standard. Par exemple, lors d'un ReadInt() l'utilisateur ne donne pas un entier mais un floattant ou un booléen.

```
int x = readInt()     //Input = "Hello World"
```

— **Error : Pas de Return dans la méthode** correspond à une erreur de réalisation d'une méthode qui n'a pas d'instruction *return* alors que le contexte le lui impose.

```
class A{
    int sub(int x, int y){
        int z;
        z=x-y;
    }
}
```

— **Error : Overflow during arithmetic operation** correspond à un dépassement arithmétique. Les deux cas principaux peuvent être :

— une division par 0 :

```
{
    int y;
    y = 10/0;
}
```

— une opération dépassant les valeurs maximales possibles par boucle infinie, par exemple :

```
{
    float compteur = 10.0;
    while(true){
        compteur = compteur * 100000.0;
        println(compteur);
    }
}
```

— **Error : Cast Failed** correspond à un cas où le transtypage d'une classe à une autre ne marche pas. Cela est sûrement du au fait que la classe n'est pas une sous-classe de l'autre.

```
class A { ... }
class B extends A { ... }
{
    A a = new A();
    B b=(B)(a);        car B est une sous classe de A et non l'inverse
}
```

— **Error : Heap full** lorsque le tas est plein et il n'est donc plus possible d'allouer de nouveaux objets.

```
class A { ... }
{
    A a;
    int c = 0;
    while(c<=1000){
        a = new A();
        c=c+1;
    }
}
```

— **Error : Stack Overflow** correspond à un dépassement de la taille de la pile impliquant alors une impossibilité de faire de nouveaux appels aux classes.

```
class A {  A a = new A(); }    ici la classe A s'appelle à l'infini
                                provoquant le débordement de la pile
{
    A obj = new A();
}
```

4 Extensions de la librairie standard

Nous avons travaillé sur l'extension de la librairie standard permettant de générer du code Jasmin depuis les programmes en Deca. Cela permet de les traduire en fichiers *.class* exécutables par la machine virtuelle Java.

Le compilateur Jasmin permet alors de générer, depuis un fichier *.deca* un autre fichier auxiliaire *.j* (pour Jasmin). Celui-ci est alors ensuite transformé par Jasmin en fichier *.class*. Pour effectuer cela avec un fichier *program.deca*, il suffit de lancer la commande suivante :

```
decac -j program.deca
```

Ainsi, en lançant cette commande, le fichier *.j* est généré, mais n'apparaît nulle part, car il est directement transformé en fichier *.class*. On peut toutefois retrouver l'ensemble du programme traduit en syntaxe Jasmin générée par le compilateur en lançant la commande suivante :

```
decac -j -d -d program.deca
```

Pour exécuter le fichier *.class* ainsi généré par *decac -j*, il suffit d'utiliser une JVM classique par exemple en écrivant dans le terminal :

```
java fichier.class
```

L'ensemble des informations liées à cette extension Jasmin est fournie dans un document annexe, "Documentation Bytecode".

5 Mode de fonctionnement

Pour les différents modes d'opérations de notre compilateur il y a :

— **-b** : Permet d'écrire une bannière avec le nom de notre groupe.

— **-p** : Permet de stopper le compilateur juste après la création de l'arbre et d'ensuite afficher la décomposition.

— **-v** : Arrête le compilateur après les étapes de vérifications, seules les erreurs sont affichées.

— **-n** : Permet de passer les tests de programme incorrect comme la division par 0, l'overflow arithmétique, l'absence d'un return dans une méthode, une conversion de type impossible,... et aussi les tests qui vérifient que l'exécution ne passe pas la mémoire limite de la machine (pile ou heap OverFlow). Cela permet pour un programme correct d'optimiser le temps et la consommation d'énergie.

— **-r** X : Limite le nombre de registres non marqués libres pour le compilateur. Cela doit rester dans l'intervalle de $4 \leqslant X \leqslant 16$.

— **-d** : Active les traces de débugage. Répéter plusieurs fois cette option pour avoir plus de traces.

— -P : S'il y a plusieurs fichiers sources, lance la compilation des différents fichiers en parallèle (accélère la compilation).

— **-j** : Permet de générer le *.class* en bytecode du fichier *myfile.deca* exécutable par la JVM (Java Virtual Machine) classique.

N.B. Il n'est pas possible d'utiliser les options **-p** et **-v** en même temps.

-b peut seulement être utilisé sans aucune autre option après, ni aucun fichier. Dans ce cas-là le *decac* finit après avoir affiché la bannière.

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