UGA – Grenoble INP

Projet Génie Logiciel

Documentation de validation

Grenoble, January 26, 2022.

Contents

1 Présentation des tests

1.1 Motivation

Les tests sont une partie centrale de la bonne réussite du développement d'un logiciel. Dans le cadre de ce projet, il est important de bien vérifier que chaque partie du code est bien écrite et n'induit pas de problème, car la moindre petite ligne de code défectueuse pourrait produire une erreur généralisée dans l'ensemble du compilateur.

Le projet étant de construire un compilateur, les principaux tests seront donc de vérifier le bon fonctionnement de notre compilateur en donnant en entrée différents tests sous forme de programme Deca valide ou invalide, puis de regarder la sortie standard de notre compilateur. A partir de cela, il est facile de savoir si le résultat est bien le résultat attendu.

1.2 Utilisation générale des tests

1.2.1 Exécution des 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.

1.2.2 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.

1.2.3 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 Les différents tests

Il existe différents types de tests, chacun correspondant à une étape particulière du fonctionnement de notre compilateur. Chacune de ces parties nécessite une attention particulière sur certains points.

2.1 Tests de lexicographie

2.1.1 Présentation des tests

Ces tests permettent de vérifier si, lors de la lecture d'un fichier, le compilateur distingue et génère bien les différents tokens du fichier. Cela correspond à la première partie de l'étape A de fonctionnement du compilateur. Un mauvais fonctionnement de cette partie induit une compilation totalement corrompue du fichier qui a été mal lu par le compilateur.

Pour tester cette partie, il faut utiliser la commande :

```
test_lex
```

qui se trouve dans :

```
./src/test/scripts/launchers/test_lex.sh
```

Les fichiers correspondant à ce test se trouvent dans le répertoire :

```
./src/test/deca/lexing
```

Ils sont triés en 4 sous-dossiers:

- Les cas donnés dans le squelette qui sont valides (*lexing/valid/provided*)
- Les cas donnés dans le squelette qui sont invalides (*lexing/invalid/provided*)
- Les cas rajoutés au squelette qui sont valides (*lexing/valid/self*)
- Les cas rajoutés au squelette qui sont invalides (*lexing/invalid/self*)

2.1.2 Résultats des tests

Pour un cas valide :

```
test_lex hello.deca
```

On observe que le programme suivant :

```
{
    println("Hello");
}
```

se décompose en une séquence de tokens comme ci-dessous :

```
OBRACE: [@0,136:136='{',<25>,10:0]
PRINTLN: [@1,146:152='println',<3>,11:8]
OPARENT: [@2,153:153='(',<30>,11:15]
STRING: [@3,154:160='"Hello"',<47>,11:16]
CPARENT: [@4,161:161=')',<31>,11:23]
SEMI: [@5,162:162=';',<27>,11:24]
CBRACE: [@6,164:164='}',<26>,12:0]
```

Le programme a donc été bien décomposé.

Pour un cas non valide, c'est-à-dire un programme qui comporterait un token non reconnaissable comme le programme :

```
"chaine pas finie
```

une erreur est alors soulevée :

```
src/test/deca/lexing/invalid/provided/chaine_incomplete.deca:10:0: token recognition
error at: '"chaine pas finie'
```

2.1.3 Automatisation des tests

Il existe deux scripts d'automatisation des tests de lexing. **basic-lex.sh** et **self-test-lex.sh**. Le second permet de lancer tous les tests du dossier *lexing* et de vérifier s'ils renvoient une erreur ou non, afin de contrôler s'ils ont le comportement attendu. Après exécution, le script donne le résultat suivant:

```
Cas valide donné
hello.deca  PASSED
simple_lex.deca  PASSED
Cas valide créé
calcul.deca  PASSED
comment_not_finish.deca  PASSED
filename.deca  PASSED
float.deca  PASSED
hexa.deca  PASSED
invalid-token.deca  PASSED
no_such_file.deca  PASSED
open_brace.deca  PASSED
wrong_function.deca  PASSED
Cas non valide donné
chaine_incomplete.deca  PASSED  (failure)
Cas non valide créé
no-token.deca  PASSED  (failure)
wrong-filename.deca  PASSED  (failure)
```

Ces deux tests sont également ajoutés dans le **pom.xml** du projet maven afin d'être exécuté lors d'une automatisation générale des tests.

2.2 Tests de syntaxe

2.2.1 Présentation des tests

Ces tests permettent de vérifier si, lors de la lecture d'un fichier, après avoir généré les différents tokens, ces derniers se sont correctement assemblés et donnent du sens du point de vue de la syntaxe de Deca.

Pour tester cette partie, il faut utiliser la commande :

```
test_synt
```

qui se trouve dans :

```
./src/test/scripts/launchers/test_synt.sh
```

Les fichiers correspondant à ce test ce trouvent dans le répertoire :

```
./src/test/deca/synt
```

Ils sont triés en 4 sous-dossiers :

- Les cas donnés dans le squelette qui sont valides (*synt/valid/provided*)
- Les cas donnés dans le squelette qui sont invalides (*synt/invalid/provided*)
- Les cas rajoutés au squelette qui sont valides (*synt/valid/self*)
- Les cas rajoutés au squelette qui sont invalides (*synt/invalid/self*)

2.2.2 Résultats des tests

Pour un cas valide :

```
test_synt hello.deca
```

On observe que le programme suivant :

```
{
    println("Hello");
}
```

permet d'obtenir un arbre des dépendances syntaxiques :

```
‘> [10, 0] Program
   +> ListDeclClass [List with 0 elements]
   ‘> [10, 0] Main
      +> ListDeclVar [List with 0 elements]
      ‘> ListInst [List with 1 elements]
         []> [11, 8] Println
             ‘> [11, 16] ListExpr [List with 1 elements]
                []> [11, 16] StringLiteral (Hello)
```

Le programme a donc été bien décomposé.

Pour un cas non valide, c'est à dire un programme qui comporterait un assemblage de tokens qui ne sont pas corrects selon la syntaxe Deca comme ci-dessous:

```
    idf _autre_idf
{
}
(
)
// un commentaire.
```

Il apparaît que Ce dernier programme est correct lexicalement mais pas syntaxiquement, une erreur est donc levée :

```
src/test/deca/syntax/invalid/provided/simple_lex.deca:11:0: no viable alternative at input
'idf'
```

2.2.3 Automatisation des tests

Il existe deux scripts d'automatisation des tests de lexing. **basic-synt.sh** et **self-test-pars.sh**. Le second permet de lancer tous les tests du dossier *synt* et de vérifier s'ils renvoient une erreur ou non, afin de contrôler s'ils ont le comportement attendu. Après exécution, il donne le résultat suivant:

```
Cas valide donné
hello.deca  PASSED
Cas valide créé
calcul.deca  PASSED
classDeclaration.deca  PASSED
comment_not_finish.deca  PASSED
DoubleInclude.deca  PASSED
filename.deca  PASSED
Cas non valide donné
chaine_incomplete.deca  PASSED  (failure)
simple_lex.deca  PASSED  (failure)
Cas non valide créé
circularinclude1.deca  PASSED  (failure)
circularinclude2.deca  PASSED  (failure)
float.deca  PASSED  (failure)
hexa.deca  PASSED  (failure)
invalidLValue.deca  PASSED  (failure)
invalid-token.deca  PASSED  (failure)
no_such_file.deca  PASSED  (failure)
no-token.deca  PASSED  (failure)
open_brace.deca  PASSED  (failure)
wrong-filename.deca  PASSED  (failure)
wrong_function.deca  PASSED  (failure)
```

Ces deux tests sont également ajoutés dans le **pom.xml** du projet maven afin d'être exécuté lors d'une automatisation générale des tests.

2.3 Tests Contextuels

2.3.1 Présentation des tests

Ces tests permettent de vérifier si, une fois l'obtention d'un programme syntaxiquement correct, ce même programme l'est également contextuellement. Cela sert en fait à vérifier si l'étape B du compilateur s'est correctement déroulée.

Pour tester cette partie, il faut utiliser la commande :

```
test_context
```

qui se trouve dans :

```
./src/test/scripts/launchers/test_context.sh
```

Les fichiers correspondant à ce test se trouvent dans le répertoire :

```
./src/test/deca/context
```

Ils sont triés en 4 sous-dossiers :

- Les cas donnés dans le squelette qui sont valides (*context/valid/provided*)
- Les cas donnés dans le squelette qui sont invalides (*context/invalid/provided*)
- Les cas rajoutés au squelette qui sont valides (*context/valid/self*)
- Les cas rajoutés au squelette qui sont invalides (*context/invalid/self*)

2.3.2 Résultats des tests

Pour un cas valide :

```
test_context hello-world.deca
```

On observe que le programme suivant :

```
{
    println("Hello, world!");
}
```

permet d'obtenir un arbre des dépendances syntaxiques :

```
‘> [10, 0] Program
   +> ListDeclClass [List with 0 elements]
   ‘> [10, 0] Main
      +> ListDeclVar [List with 0 elements]
      ‘> ListInst [List with 1 elements]
         []> [11, 8] Println
             ‘> [11, 16] ListExpr [List with 1 elements]
                []> [11, 16] StringLiteral (Hello, world!)
                    type: string
```

Le programme a donc été bien décomposé. En effet, on voit que le typage du StringLitteral a correctement été attribué.

Pour un cas non valide, c'est-à-dire un programme qui comporterait une erreur contextuelle, comme ci-dessous :

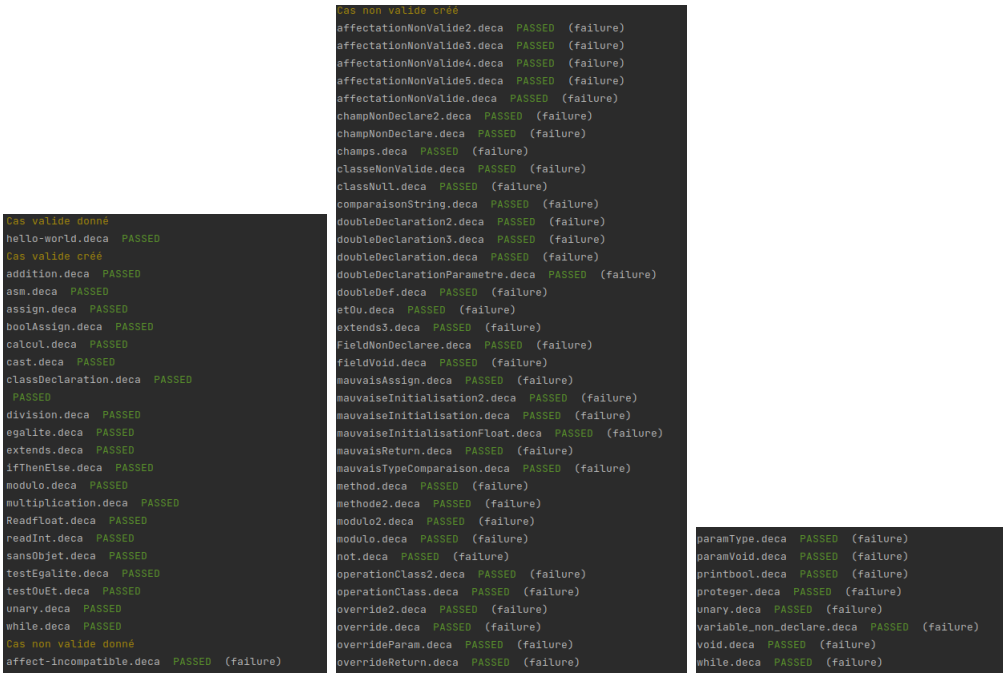
```
{
int a;
boolean b;

a = b;
}
```

Il apparaît que ce dernier programme est correct syntaxiquement, mais pas contextuellement car les deux variables sont de types incompatibles. Une erreur est donc levée :

```
src/test/deca/context/invalid/provided/affect-incompatible.deca:15:7: boolean isn't a
subtype of int so it cannot be assign to it
```

2.3.3 Automatisation des tests

Il existe deux scripts d'automatisation des tests du contexte du programme. **basic-context.sh** et **self-test-context.sh**. Le second permet de lancer tous les tests du dossier *context* et de vérifier s'ils renvoient une erreur ou non, afin de contrôler s'ils ont le comportement attendu. Après exécution, il donne le résultat suivant:



Ces deux tests sont également ajoutés dans le **pom.xml** du projet maven afin d'être exécuté lors d'une automatisation générale des tests.

2.4 Tests de compilation

2.4.1 Présentation des tests

On se concentre maintenant sur la génération des tests qui permettent de contrôler la bonne compilation du fichier source. Pour cela, on utilise directement la commande du compilateur :

```
decac
```

qui se trouve dans le répertoire :

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

Les fichiers correspondant à ce test se trouvent dans le répertoire :

```
./src/test/deca/codegen
```

Ils sont triés en 5 sous-dossiers :

- Les cas donnés dans le squelette qui sont valides (*codegen/valid/provided*)
- Les cas donnés dans le squelette qui sont pour la performance (*codegen/perf/provided*)
- Les cas rajoutés au squelette qui sont valides (*codegen/valid/self*)
- Les cas rajoutés au squelette qui sont pour la performance (*codegen/perf/self*)
- Les cas rajoutés au squelette qui sont invalides (*codegen/invalid*)

Le dernier sous-dossier fait référence aux générations des labels d'erreur pour IMA.

2.4.2 Résultats des tests

Lorsqu'on lance le compilateur sur un programme Deca, il en sort un programme en assembleur qui pourra ensuite être exécuté par IMA.

Ainsi, lorsqu'on lance :

```
decac entier1.deca
```

qui correspond au programme Déca :

```
{
   int i ;
   i = 1;
   println(i);
   i = i + 1;
   println(i);
}
```

on obtient un programme assembleur comme suit :

```
; start main program
TSTO #2
BOV stackOverflow.0
ADDSP #1
; Main program
; Beginning of main instructions:
LOAD #1, R3
STORE R3, 1(GB)
LOAD R3, R2
; value in R2 ignored
LOAD 1(GB), R2 ; AbsExpr
LOAD R2, R1
WINT
WNL
LOAD 1(GB), R3 ; AbsExpr
ADD #1, R3
STORE R3, 1(GB)
LOAD R3, R2
; value in R2 ignored
LOAD 1(GB), R2 ; AbsExpr
LOAD R2, R1
WINT
WNL
HALT
; end main program
inputError.0:
WSTR "Error: Input/Output Error"
WNL
ERROR
overFlowError.0:
WSTR "Error: Overflow during arithmetic operation"
WNL
ERROR
stackOverflow.0:
WSTR "Error: Stack Overflow"
```

```
WNL
ERROR
pilePleineLabel.0:
WSTR "Error: Heap full"
WNL
ERROR
noReturnLabel.0:
WSTR "Error: No return in the method"
WNL
ERROR
castFailed.0:
WSTR "Error: Cast Failed"
WNL
ERROR
equals.0:
LOAD -2(LB), R0
CMP -3(LB), R0
SEQ R0
RTS
```

Il existe deux scripts d'automatisation des tests de compilation du programme. **basic-gencode.sh** et **self-test-gen.sh**. Le second permet de lancer tous les tests du dossier *codegen* et de vérifier s'ils renvoient une erreur ou non, afin de contrôler s'ils ont le comportement attendu. Après exécution, il donne le résultat suivant :

```
Cas valide donné
cond0.deca  PASSED
ecrit0.deca  PASSED
entier1.deca  PASSED
exdoc.deca  PASSED
exmathdoc.deca  PASSED
Cas valide créé
addition.deca  PASSED
affectObject.deca  PASSED
asm.deca  PASSED
assign2.deca  PASSED
assign.deca  PASSED
boolAssign.deca  PASSED
boolObject.deca  PASSED
calculConvAuto2.deca  PASSED
calculConvAuto.deca  PASSED
calcul.deca  PASSED
cast1.deca  PASSED
cast2.deca  PASSED
cast.deca  PASSED
class2.deca  PASSED
class.deca  PASSED
compteurObject.deca  PASSED
convfloat.deca  PASSED
division.deca  PASSED
egalite.deca  PASSED
etOu2.deca  PASSED
extends2.deca  PASSED
extends.deca  PASSED
Greater.deca  PASSED
ifThenElse.deca  PASSED
initClass.deca  PASSED
modulo.deca  PASSED
moinsUnaire.deca  PASSED
multiplication.deca  PASSED
noOperation.deca  PASSED
notcondition.deca  PASSED
not.deca  PASSED
object.deca  PASSED
Readfloat.deca  PASSED
readInt.deca  PASSED
sansObjet.deca  PASSED
testEgalite.deca  PASSED
testOuEt.deca  PASSED
unary.deca  PASSED
while.deca  PASSED
Cas performance donné
ln2.deca  PASSED
ln2_fct.deca  PASSED
syracuse42.deca  PASSED
Cas performance créé
boolean_not.deca  PASSED
fibo42.deca  PASSED
```

Ces deux tests sont également ajoutés dans le **pom.xml** du projet maven afin d'être exécuté lors d'une automatisation générale des tests.

2.4.3 Cas de decac -p

Pour tester la décompilation d'un programme déca, il suffit de prendre un programme Deca, et de lancer la commande :

```
decac -p cond0.deca
```

On peut alors bien vérifier que le programme Deca :

```
{
   if (1 >= 0) {
      println("ok");
   } else {
      println("erreur");
   }
}
```

est transformé en un autre programme Deca correct :

```
{
        if((1 >= 0)){
                println("ok");
        } else {
                println("erreur");
        }
}
```

On peut alors recompiler le programme ainsi obtenu pour vérifier que ce programme décompilé est valide.

Il existe un script d'automatisation pour la commande *decac -p*. Il s'agit de **self-test-decac-p**. Il permet de lancer la décompilation sur plusieurs fichiers répartis dans les différents dossiers de tests du projet. Il permet de récupérer les erreurs de décompilation et donc de vérifier le bon comportement de ce processus. Ce test est bien évidemment rajouté au **pom.xml**.

2.4.4 Cas des autres options

Les autres erreurs n'ont pas de tests à proprement parlé. Ils peuvent néanmoins être testés en lançant la commande :

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

Des tests du squelette permettent de succinctement tester le bon fonctionnement des options. Il s'agit de **basic-decac.sh** et **common-tests.sh**. Ces deux tests sont également ajoutés au **pom.xml** pour l'automatisation généralisée.

2.5 Tests pour l'éxécution

2.5.1 Présentation des tests

Ces tests permettent de vérifier si, une fois notre programme parfaitement compilé, il n'y a pas d'erreur d'interprétation du code assembleur par IMA.

Pour tester cette partie, il faut utiliser la commande :

```
ima
```

qui doit préalablement être téléchargée dans la machine de l'utilisateur.

Les fichiers correspondant à ce test se trouvent dans le répertoire :

```
./src/test/deca/codegen
```

Ils sont triés en 5 sous-dossiers :

- Les cas donnés dans le squelette qui sont valides (*codegen/valid/provided*)
- Les cas donnés dans le squelette qui sont pour la performance (*codegen/perf/provided*)
- Les cas rajoutés au squelette qui sont valides (*codegen/valid/self*)
- Les cas rajoutés au squelette qui sont pour la performance (*codegen/perf/self*)
- Les cas rajoutés au squelette qui sont invalides (*codegen/invalid*)

2.5.2 Résultats des tests

Pour un cas valide :

```
ima entier1.ass
```

On observe que le programme suivant :

```
{
 int i ;
 i = 1;
 println(i);
 i = i + 1;
 println(i);
}
```

compile et s'exécute sans problème tout en donnant le résultat suivant :

```
1
2
```

Le programme a donc bien donné le résultat que l'on attendait de lui.

Pour un cas non valide, c'est-à-dire un programme qui comporterait une erreur d'exécution comme ci-dessous :

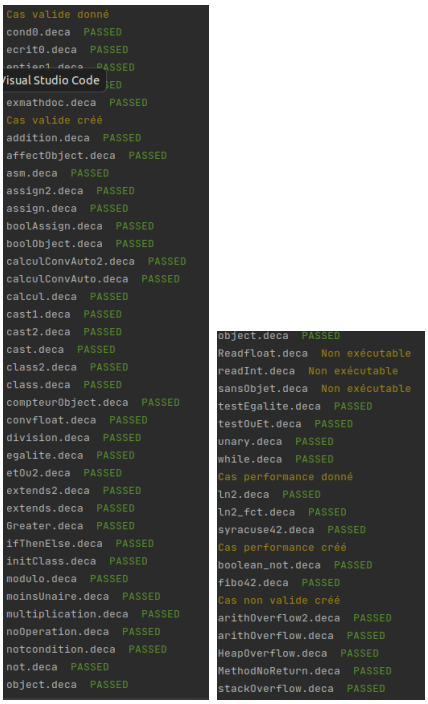
```
{
    println(1 /0);

}
```

Ce dernier programme comporte une erreur car il possède une division par 0. Il renvoie donc le message d'erreur :

```
Error: Overflow during arithmetic operation
```

2.5.3 Automatisation des tests

Il existe un script d'automatisation des tests d'exécution du programme : **self-test-exec.sh**. Il permet de lancer tous les tests du dossier *codegen* et de vérifier s'ils renvoient une erreur ou non, afin de contrôler s'ils ont le comportement attendu. Après exécution, il donne le résultat suivant :



Ce test est également ajouté dans le **pom.xml** du projet maven afin d'être exécuté lors d'une automatisation générale des tests.

2.6 Tests de l'extension

2.6.1 Présentation des tests

Ces tests permettent de vérifier l'option **decac -j** de notre compilateur qui correspond à l'extension Bytecode que nous proposons dans notre compilateur.

Pour tester cette partie, il faut utiliser la commande :

```
decac -j
```

Les fichiers correspondant à ce test se trouvent dans le répertoire :

```
./src/test/deca/bytecode/valid/self
```

2.6.2 Résultats des tests

Lorsqu'on lance la commande :

```
decac -j addition.deca
```

sur un fichier qui correspond à :

```
{
    println("Addition de 5+3+4 : ", 5+4+3);
    println("Addition de 5+7-20 : ", 5+7-20);
    println("Soustraction de 7-5-1 : ", 7-5-1);
    println();
    println("Maintenant pour les float ! ");
```

```
    println("Addition de 5.0+3.1+4.2 : ", 5.0+4.1+3.2);
    println("Addition de 5.3+7.5-20.4 : ", 5.3+7.5-20.4);
    println("Soustraction de 7.2-5.1-1.1 : ", 7.2-5.1-1.1);
}
```

La commande produit un fichier reformaté **Addition.class**. On peut alors tester le bon fonctionnement de la génération du fichier *.class* en exécutant ce fichier depuis une JVM classique, en utilisant :

```
java Addition
```

qui produit le résultat :

```
Addition de 5+3+4 :
12
Addition de 5+7-20 :
-8
Soustraction de 7-5-1 :
1
Maintenant pour les float !
Addition de 5.0+3.1+4.2 :
12.3
Addition de 5.3+7.5-20.4 :
-7.5999994
Soustraction de 7.2-5.1-1.1 :
0.9999999
```

Aux erreurs d'arrondis des flottants près, on remarque que le résultat produit est bon.

2.6.3 Automatisation des tests

Il existe un script d'exécution des tests automatiques pour l'extension. Il s'agit de **self-test-extension.sh** qui va générer les fichiers *.class* pour chacun des fichiers du dossier *bytecode*. Une fois exécuté, on trouve le résultat suivant :

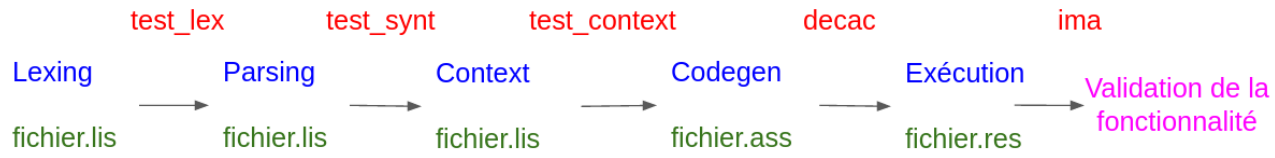
```
Cas valide créé
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/Addition.class
addition.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/Assign.class
assign2.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/Assign.class
assign.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/BoolAssign.class
boolAssign.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/Calcul.class
calcul.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/Convfloat.class
convfloat.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/Division.class
 PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/Egalite.class
egalite.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/EtOu.class
etOu2.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/Greater.class
Greater.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/Helloworld.class
hello-world.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/IfThenElse.class
ifThenElse.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/Modulo.class
modulo.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/Multiplication.class
multiplication.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/NoOperation.class
noOperation.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/Notcondition.class
notcondition.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/Readfloat.class
Readfloat.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/ReadInt.class
readInt.deca  PASSED
Generated: /home/utilisateur/Bureau/Workspace/2A/ensimag/GL/Projet_GL/./src/test/deca/bytecode/valid/self/SansObjet.class
sansObjet.deca  PASSED
```

Ce test est également ajouté dans le **pom.xml** du projet maven afin d'être exécuté lors d'une automatisation générale des tests.

3 Méthodologie des tests

3.1 Gestion des risques

Chaque fonctionnalité est développée selon une procédure stricte afin de limiter les erreurs au maximum.



Entre chaque étape, le code est relu et subit le test associé à l'étape. De plus, lors de la finalisation d'une implémentation, un **mvn verify** est lancé pour vérifier que la fonctionnalité n'a pas détruit le code déjà présent. Ce système permet de facilement identifier les problèmes et de les régler au fur et à mesure en isolant la partie problématique.

3.2 Gestion des rendus

Pour éviter que les rendus ne soient contaminés par des erreurs résiduelles, nous avons décidé de ne mettre sur la branche master que les rendus finaux. Une fois tous les tests passés sur une branche annexe, nous relançons alors tous les tests sur une cette branche.

Si tous les tests sont bons, nous clonons alors le dépôt sur une autre machine et lançons un **mvn verify** sur ce nouveaux dépôt pour être sûrs qu'un nouvel utilisateur puisse utiliser notre produit.

4 Analyse des résultats

4.1 Résultat de couverture

Nous avons configuré le projet pour obtenir une couverture de test de notre code avec jaccoco, une extension permettant de vérifier quelle partie du code a été testée ou non. Voici les résultats :

Deca Compiler

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		75 %		54 %	522	735	508	2 024	241	368	0	48
fr.ensimag.deca.tree		84 %		75 %	190	848	253	2 076	80	579	2	89
fr.ensimag.deca		78 %		61 %	40	108	98	374	8	54	2	6
fr.ensimag.deca.context		84 %		76 %	24	136	25	209	17	119	0	20
fr.ensimag.ima.pseudocode		87 %		77 %	19	87	24	185	14	76	1	26
fr.ensimag.ima.pseudocode.instructions		79 %		50 %	15	63	25	113	14	62	10	54
fr.ensimag.ima.pseudocode.jasmin		87 %		83 %	4	29	4	54	2	23	0	10
fr.ensimag.deca.codegen		96 %		80 %	7	54	1	92	0	36	0	4
fr.ensimag.ima.pseudocode.instructions.jasmin		95 %		50 %	4	40	3	64	2	38	2	36
fr.ensimag.deca.tools		94 %		100 %	1	16	3	41	1	13	0	3
Total	4 655 of 23 506	80 %	502 of 1 429	64 %	826	2 116	944	5 232	379	1 368	17	296

Nous obtenons donc une couverture de notre code de 80%. Les pourcentages restants sont dûs à plusieurs facteurs :

- Cas d'erreurs inatteignables avec notre implémentation
- Fonctions non prises en compte dans notre implémentation (*instanceOf* par exemple)
- Fonctionalité du squelette que nous n'avons jamais utilisée

4.2 Bilan de la phase de test

Les tests ont pu, dans le sprint final, nous permettre de corriger de nombreuses petites erreurs d'imprécisions de notre code. Le taux de couverture de 80% nous semble correct et démontre que nous avons constitué une base solide de test qui nous a permis d'aboutir à un compilateur plutôt robuste, dont nous connaissons bien les faiblesses.