

Documentation de Validation

GL53



SOMMAIRE

SOMMAIRE	2
Descriptif des tests	3
Les scripts de tests	4
Gestion des risques	5
Risques internes associés aux tâches du projet	5
Risques internes associés aux tâches du projet	7
Gestion des rendus	10
Résultats de Cobertura	11
Autres méthodes de validation	11
Pistes pour amélioration du compilateur	12
Conclusion	13
Annexe	15

I. Descriptif des tests

Les tests rédigés en deca sont dans un premier temps classifiés selon les étapes du compilateur, les dossiers lexicographie/, syntax/, context/ et codegen/ correspondent respectivement à l'étape de l'analyse lexicale, syntaxique, contextuelle et la génération du code. A l'intérieur de chaque dossier se trouvent les tests "valid" et "invalid", les premiers sont utiles pour vérifier que les fonctionnalités implémentées marchent correctement, et les deuxièmes pour vérifier que le compilateur attrape correctement l'erreur. Notons que ces tests sont soit des tests unitaires ou des tests plus complexes pour une vérification supplémentaire et un débogage efficace.

Finalement, à l'intérieur de chacun de ces dossiers, on peut trouver les dossiers sansObjet/ et Objet/ pour les tests concernant les fonctionnalités de chacune des deux parties. On peut trouver également le dossier auto/ dans context/valid/, celui ci contient des tests générés d'une manière automatique par un script python, et qui est une couche supplémentaire dans notre batterie de tests.

Une différence majeure entre les tests rédigés et la manière dont ils sont automatisés (voir II) est entre les tests de la partie C et les autres parties. Les tests de la génération du code requiert une vérification de la sortie standard, et donc prend plus de temps à rédiger et à automatiser (d'où le nombre relativement limité de tests comparé aux autres parties).

La rédaction des tests et les scripts d'automatisation a été richement informative sur 3 niveaux :

- Assurer la bonne implémentation de l'étape de l'analyse contextuelle et les messages d'erreurs.
- Assurer l'intégrité du compilateur intégré à chaque commit. On cite par exemple l'implémentation de la partie objet du parseur qui a cassé instantanément le compilateur. Cette démarche est donc vivement recommandé pour quiconque qui souhaite continuer le travail sur ce compilateur (voir VI).

- Découvrir des bugs passés inaperçus, ceci a été particulièrement utile dans la partie objet, où il fallait gérer des notions telles que la récursivité, la bonne gestion du stack, etc..

II. Les scripts de tests

L'automatisation des tests était indispensable à cause de la quantité considérable des tests, les scripts de d'automatisation ont donc été rédigé dans la première semaine du travail.

Les scripts ont été essentiellement écrits en python, et se trouvent dans le dossier `src/test/script` :

- les tests `test_lexer.py`, `test_synt.py` et `test_context.py` ont été rédigé de la même manière, de façon à ce que le script attrape ou non, intentionnellement ou non, une erreur durant la compilation le test `test_gencode.py` compare la sortie de chaque fichier après compilation par `decac` et `ima` avec le résultat attendu, et donc l'ajout d'un fichier test dans la base doit être accompagné par une (petite) modification du script (voir annexe pour un exemple détaillé).
- le `tester.py` est un interface minimaliste sur le terminal qui permet à l'utilisateur de tester les différentes parties précédentes.
- le test `test_decac.sh` sert à tester les différentes options du compilateur `decac`, notamment la décompilation et la gestion du parallélisme.

Notons que les scripts ont été rédigé dans un esprit user-friendly, et le résultat est affiché joliment dans le terminal sans trop de détails techniques.

III. Gestion des risques

Classification des risques

Pour ce projet on peut identifier 4 catégories de risques:

- 1) Risques internes associés à l'organisation du projet
- 2) Risques internes associés aux tâches du projet
- 3) Risques internes associés au rendu des projets
- 4) Risques externes liés à l'environnement

Commençant par les **risques liés à l'environnement**, ici on s'intéresse surtout à l'environnement de travail, les packages dont on aura besoin pour tester et coder le compilateur, à l'exemple de ima pour la partie C, maven, ANTLR. Sur les machines de l'Ensimag on n'a pas de problèmes par rapport à ça mais on s'est assuré que tous les membres du groupes ont configurés l'environnement de travail sur leurs machines personnelles si un jour on n'a pas accès aux salles infos de l'ensimag, ou un problème avec ssh ou un problème de connexion.

Concernant les risques associés à l'**organisation du projet**

Risques	Probabilité [0..5]	Importance [0..5]	Conséquences	Solutions
Une mauvaise répartition des tâches	2	3	Le non respect des délais sur le planning	Une réunion de groupe pour rééquilibrer les tâches
Une démotivation d'un membre de groupe	2	3	Un retard sur une partie du projet	Il faudra essayer d'expliquer les implications

				qu'aura cette démotivation sur l'ensemble du groupe, entendre les raisons et essayer de rééquilibrer le travail si cette démotivation est liée à une pression excessive que sent ce membre, le rééquilibrage des tâches est indispensable pour remédier à ce retard
Mauvaise entente au sein du groupe + problèmes de communication	0.5	3	L'ambiance au sein de l'équipe se détériore	On a parlé de ce point dans notre charte, il faudra respecter les points évoqués et remédier à cela
Une absence de longue durée d'un membre du groupe	1	3	Un retard majeur sur la partie développée	Il faudra tenir au courant les profs de ce genre d'imprévu, un

			par ce membre	réunion de groupe s'impose pour le partage des rôles sur cette partie
--	--	--	---------------	-----------------------------------------------------------------------

Risques internes associés aux tâches du projet

Retard majeur sur une partie qui est dépendante de l'avancement des autres parties [La partie B ou C]	1	4	Ce risque aura un impact sur l'avancement global de notre projet	Il faudra qu'un autre membre de groupe bascule sur cette partie en tant que renfort pour essayer de rattrapper ce retard
Retard irréversible sur une partie du projet	1	5	Temps très serré pour la suite du développement, on sent que la pression monte	Il faudra réévaluer les objectifs de l'équipe, à l'exemple du choix de développer le compilateur de base et non le compilateur complet deca (avec instanceof, ..), une réunion de groupe s'impose pour remédier à ce problème
Une information importante lors d'un suivi ou envoyée par mail au sujet du compilateur	0.5	1	Un temps perdu qui peut décaler l'avancement de notre projet	On a désigné un coordinateur, qui devra s'assurer que l'information est bien passée dans le groupe pour remédier à ce

dépassée, mal interprétée				genre de problèmes
<p>Les tests importants qui ne marchent pas (common-tests.sh) (basic*.sh)</p> <p>mvn compiler</p> <p>mvn test-compile</p> <p>mvn test</p>	4	5	<p>La note de groupe va être affectée, le développement des parties n'a pas été bien structuré , des bugs non traités</p>	<p>Le testeur doit relever ces problèmes au plus vite, les tests doivent anticiper le développement des partis pour que les développeurs puissent avoir une structure prête pour vérifier leurs travail, la révision du code par d'autres membres est indispensable pour éviter ce genre de problèmes</p>

Risques internes associés au rendu des projets

Mauvaise utilisation de git	1	5	Perturbation au sein du groupe, une partie développé, finie manquante dans le master	Tous les membres du groupe ont assisté à l'Amphi git, l'avantage quand on on développe tous dans la même salle est que forcément un autre membre pourrait régler les problèmes de merge, ou de stach, si on n'arrive pas à puller dus à des conflits
Un push de dernière minute, qui précède les dates clés qui intègre les mauvais fichiers	3	5	Grave , perte du travail, le non respect des cahiers des charges lors du pull général des professeurs, ⇒ La note sera vue à la baisse	Il faut terminer le travail un jour avant les dates clés, aucun push ne doit être émis quelques minutes avant le pull général, des copies du projet sont faites quotidiennement pour s'assurer que le projet est sauvegardé ailleurs que le git principal, aucun push d'une partie qui ne compile pas et qui n'est testée (le minimum de tests unitaires) n'est effectué, la charte du groupe recadre cette éventualité

Gestion des rendus

Ce projet demande plusieurs documentations à fournir (une doc utilisateur, une doc pour l'extension, la doc de validation, ..), on a prévu de finir le développement du code à temps pour pouvoir commencer la rédaction, tous les membres du groupe sont impliqués à la rédaction, chacun rédige la majeure partie de la documentation pour sa partie, suivant l'avancement des parties un autre membre pourra commencer la rédaction. Les dates des rendus sont des dates clés pour le groupe, cela nécessite que le code soit fonctionnel pour les parties demandés lors du rendus (intermédiaire où il faudra finir le sans objet ou le rendu final), on ne rend que la partie du code qui compile et qui est testée, toute partie qui ne soit pas aboutie ne sera mise dans notre branche master.

IV. Résultats de Cobertura

Cobertura est un outil qui nous permet de mesurer la couverture du code par la base des tests, et donc nous fournit une idée sur la qualité des tests rédigés.

On obtient les résultats suivants après compilation et exécution des 5 scripts de test (voir l'annexe pour le rapport détaillé):

- 76% de couverture totale des lignes.
- 59% de couverture totale des branches.

Analyse des résultats:

L'ensemble de l'équipe est plutôt satisfait du résultat de Cobertura, sachant qu'une couverture totale du code est très difficile à achever et peu prioritaire par rapport aux bugs qui persistent dans le compilateur.

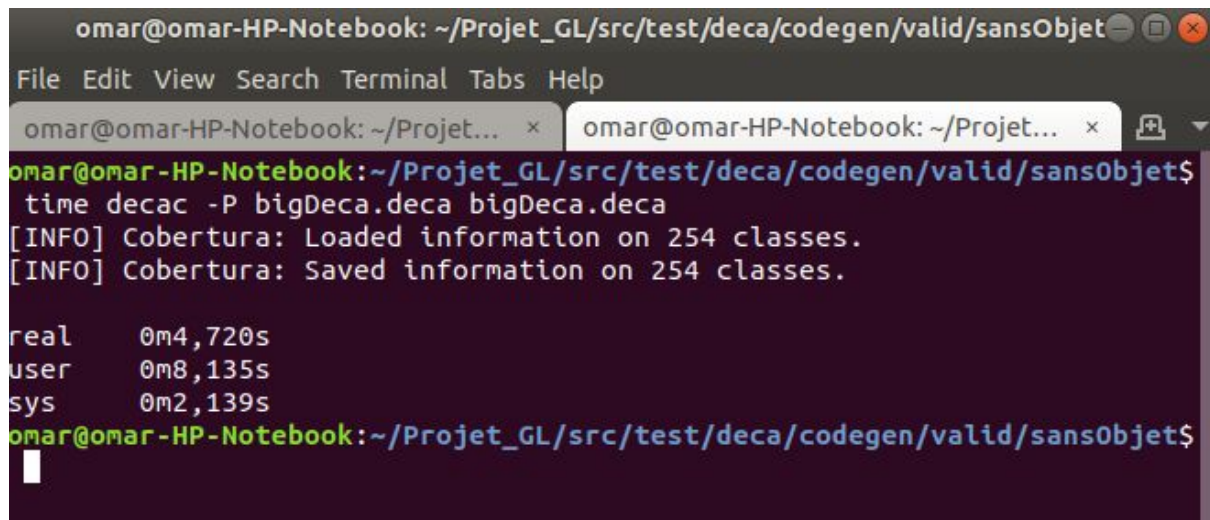
On note par contre quelques raisons qui pourraient justifier le résultat :

- La non utilisation de plusieurs instructions `ima` dans notre implémentation, qui rend les fichiers associés inutiles.
- La conception du code rend quelques bouts dans des classes inaccessibles, souvent par un `Override` dans une classe fille. Une optimisation de la qualité du code va remédier sûrement à ce problème.

V. Autres méthodes de validation

Durant la totalité des projets, on a utilisé des méthodes de validation du compilateur autre que les tests, notamment la vérification du code par une tierce personne dans le groupe. Par exemple, les membres qui ont été occupé par la partie C durant la quasi totalité du projet faisaient souvent des retours sur le code de la partie B, ainsi s'assurant que les erreurs sont bien gérés et les éléments nécessaires (i.e : les `set()`) bien implémentés.

Une méthode bien particulière qu'on a utilisé pour tester l'option `-P` du compilateur a consisté de générer (par le biais de python) un fichier `deca` assez volumineux (nommé `/src/test/deca/codegen/valid/sansObjet/bigDeca.deca`) et de mesurer le temps de compilation de plusieurs copies de ce fichier à l'aide de la commande `time` dans le Bash. On retrouve que le parallélisme diminue le temps d'exécution par 50%, ce qui est totalement prévisible et valide complètement la gestion du parallélisme dans le compilateur.



```
omar@omar-HP-Notebook: ~/Projet_GL/src/test/deca/codegen/valid/sansObjet$
File Edit View Search Terminal Tabs Help
omar@omar-HP-Notebook: ~/Projet... x omar@omar-HP-Notebook: ~/Projet... x
omar@omar-HP-Notebook:~/Projet_GL/src/test/deca/codegen/valid/sansObjet$
time decac -P bigDeca.deca bigDeca.deca
[INFO] Cobertura: Loaded information on 254 classes.
[INFO] Cobertura: Saved information on 254 classes.

real    0m4,720s
user    0m8,135s
sys     0m2,139s
omar@omar-HP-Notebook:~/Projet_GL/src/test/deca/codegen/valid/sansObjet$
```

VI. Pistes pour amélioration du compilateur

On propose dans ce qui suit quelques pistes sur lesquels on pourra travailler pour améliorer ce compilateur :

- *Optimisation du code* : Même si le code a été rédigé durant les ¾ du projet dans l'esprit de la lisibilité et de l'extensibilité, l'ensemble de l'équipe a été relativement forcé d'écrire un code fonctionnel pour la partie objet, et qui n'est pas nécessairement optimal dans cette optique.
- *Débogage*: Un effort particulier a été fait dans le développement des parties B et C pour minimiser le nombre de bugs éventuelles. On est comme même convaincu que plusieurs erreurs de compilation persistent et qu'ils doivent être adressé prioritairement. Parmi les bugs qu'on a pu détecter mais malheureusement pas eu le temps de corriger :
 1. la mauvaise gestion du "return" dans une méthode, les instructions après le return sont exécutés comme même.

2. la récursivité ne marche pas dans des cas complexes (i.e : fonction à plusieurs paramètres).
3. *Extensions* : Un troisième point sur lequel on pourra améliorer le compilateur est d'étendre les fonctionnalités de ce dernier. On pourra dans un premier temps le compléter en implémentant l'instruction "instanceof". Dans un deuxième temps, on pourra :
 - a - travailler sur l'optimisation du code deca, on effectuant des passes supplémentaires.
 - b - implémenter les tableaux et différentes structures de données similaires : ArrayList, LinkedList, HashSet, HashTree..
 - c - Génération du bytecode Java.

VII. Conclusion

Le travail sur un projet de cette taille requiert sans doute une validation tout au long du travail. L'écriture des tests et l'automatisation en utilisant des scripts peut sembler comme une tâche contre productive au début, mais s'avère indispensable à la fin du projet, à cause de l'implémentations de nombreuses fonctionnalités par différents développeurs, ce qui cause inévitablement des conflits.

On recommande vivement cette méthodologie pour quiconque souhaitant élargir ce compilateur, on pourra également penser à utiliser des scripts qui mesurent la performance de compilation pour valider les extensions OPTIM ou TRIGO.

Annexe

échantillon du script test_context.py

```
#!/usr/bin/env python3
import os
class color:
    HEADER = '\033[95m'
    OKBLUE = '\033[94m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
def files(path):
    for file in os.listdir(path):
        if os.path.isfile(os.path.join(path, file)):
            yield file
def valid_context():
    counter = 0
    tmp = 0
    test_context = "launchers/test_context"
    valid_context_SO = "../deca/context/valid/sansObjet"
    x = 0
    for file in files(valid_context_SO):
        tmp += 1
        execute = test_context + " " + valid_context_SO + "/" + str(file) + " " + "2>
{}.log".format(str(file))
        if x == 1:
            os.system(execute)
        if x == 0:
            os.system(execute + "> synt.txt")
```

```

os.system("rm synt.txt")
if os.stat("{} .log".format(str(file))).st_size != 0:
    print(file+color.BOLD+color.WARNING+" *** [Test FAILED
UNEXPECTED] *** "+color.ENDC)

else:
    print(file+color.BOLD+color.HEADER+" *** [TEST PASSED EXPECTED]
*** "+color.ENDC)
    counter+=1
    print("=====")
    os.system("rm *.log")

    print(color.BOLD+color.OKBLUE+" X+X+X+X+X [TEST SANS OBJET
Context] X+X+X+X+X "+color.ENDC)

print("~=====~")

val = 1
if val == 1:
    y = valid_context()
    if (y[0] == y[1]):
        print(color.BOLD+ color.OKGREEN+" .-~-.~-.~[{} TESTS VALID
CONTEXT SUCCESS].-~-.~-.~".format(str(y[1]))+color.ENDC)

    print("~=====~")
    else:
        print(color.BOLD+ color.FAIL+" .-~-.~-.~[{} TESTS VALID
CONTEXT ERROR].-~-.~-.~".format(str(y[1] -y[0]))+color.ENDC)

print("~=====
=====~")

```

échantillon du script test_gencode.py

```
def valid_gencode():
    counter = 0
    tmp = 0
    test_gencode = "../main/bin/decac"
    valid_gencode_SO = "../deca/codegen/valid/sansObjet"

    print("~=====~")
    print(color.BOLD+color.OKBLUE+"  X+X+X+X+X [TEST SANS OBJET
GENCODE] X+X+X+X+X "+color.ENDC)
    print("~=====~")

    for file in files(valid_gencode_SO):
        if (str(file))[len(str(file))-5:] == ".deca":
            tmp += 1
            execute = test_gencode + " " + valid_gencode_SO + "/" + str(file)
            os.system(execute)
            if os.path.isfile(str(file)[-4]+"ass"):
                print("Fichier {}ass non généré".format(str(file)[-4]))
                exit()
            resultat = os.system("ima {}".format(valid_gencode_SO +
"/"+str(file)[-4]+"ass" + "> resultat.log")
            fichier = open("resultat.log", "r")
            resultat = fichier.readlines()[0]
            resultat = resultat[:len(resultat)-1]
            os.system("rm resultat.log")
            if str(file) == "Cast.deca":
                attendu = "3.60000e+00 3 3 3.00000e+00 28"
            elif str(file) == "castIntFloat.deca":
                attendu = "(int) 2.50000e+00 = 2"
```



```

elif str(file) == "Mult.deca":
    attendu= "1.51250e+03"
    [...]
else:
    print("fichier pas encore traité")
    exit()

if resultat==attendu:
    print(str(file)+color.BOLD+color.HEADER+" *** [TEST PASSED]
*** "+color.ENDC)
    counter+=1
    #print("Tout va bien pour " + str(file))
else:
    print(str(file)+color.BOLD+color.WARNING+" *** [Test FAILED]
*** "+color.ENDC)
    #print("Tout ne pas va bien pour " + str(file))
    print("Le résultat obtenu :", resultat)
    print("Le résultat attendu :", attendu)
    os.system("rm " + "../deca/codegen/valid/sansObjet" + "/" + "*.ass")
print("=====")
    print(color.BOLD+color.OKBLUE+" X+X+X+X+X [TEST OBJET
GENCODE] X+X+X+X+X "+color.ENDC)
print("~=====~")

```

Packages

All
 It.ensimajo.dica
 It.ensimajo.dica.codegen
 It.ensimajo.dica.context
 It.ensimajo.dica.syntax
 It.ensimajo.dica.tools
 It.ensimajo.dica.tree
 It.ensimajo.lma.pseudocode
 It.ensimajo.lma.pseudocode.instructions

All Packages

Classes

[ADD](#) (100%)
[ADDSP](#) (50%)
[AbstractBinaryExpr](#) (84%)
[AbstractDecalExpr](#) (80%)
[AbstractDecalParser](#) (80%)
[AbstractDecalClass](#) (100%)
[AbstractDecalField](#) (100%)
[AbstractDecalMethod](#) (100%)
[AbstractDecalParam](#) (100%)
[AbstractDecalVar](#) (100%)
[AbstractExpr](#) (91%)
[AbstractIdentifier](#) (25%)
[AbstractInstruction](#) (100%)

Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	254	76%	59%	1,711
It.ensimajo.dica	6	79%	71%	2,718
It.ensimajo.dica.codegen	2	78%	66%	1,583
It.ensimajo.dica.context	22	75%	60%	1,791
It.ensimajo.dica.syntax	52	67%	45%	2,003
It.ensimajo.dica.tools	4	85%	70%	2,294
It.ensimajo.dica.tree	87	86%	83%	1,534
It.ensimajo.lma.pseudocode	27	78%	72%	1,253
It.ensimajo.lma.pseudocode.instructions	54	64%	N/A	1

Report generated by Cobertura 2.1.1 on 30/01/20 00:11.