Document de validation

GL 03

M. Migné[*] E. Tellier[†] A. El Ahmar[‡] A. Benmoussa[§]
A. Fuentes[¶] Y. Amoura[‖] J. Vandeputte[**]
F. Constans[††]

21 juin 2023

[*] Matthieu.Migne@grenoble-inp.org>
[†] Emilien.Tellier@grenoble-inp.org
[‡] Anas.Elahmar@grenoble-inp.org
[§] Alae.Benmoussa@grenoble-inp.org
[¶] Aubin.Fuentes@grenoble-inp.org
[‖] Yanis.Amoura@grenoble-inp.org
[**] Jules.Vandeputte@grenoble-inp.org
[††] Florence.Constans@grenoble-inp.org

Table des matières

1 Description des tests

1.1 Tests de la partie sans-objet

1.1.1 Tests arithémtiques

Les tests des opérateurs arithmétiques sont pour la plupart des tests unitaires vérifiant une seule fonctionnalité précise (exemple : { int y = 1 + 1; } vérifie le bon fonctionnement de l'opérateur *plus* sur l'opération simple 1 + 1). Nous avons créé de nombreux tests permettant de vérifier un maximum de combinaisons possibles pour chacune des fonctionnalités implémentées.

Ces tests sont assez simples, mais d'autres tests plus complexes vérifiant des combinaisons d'opérations ont été créés.

Par exemple, le code suivant :

```
1 {
2       int i=1;
3       while (i<99)
4       {
5           i=i+1;
6       }
7       int y=((((2+3)))+((4+5))+6+(7+8))+9;
8       i=i+y;
9 }
```

Permet de vérifier le bon fonctionnement de l'opérateur plus dans des situations plus complexes.

Enfin, nous avons créé des tests vérifiant plusieurs opérateurs à la fois comme suit (voir le fichier de test "calcul_complexe.deca") :

```
1 {
2       float y = (3*2/(4+6*(3+3))-3)*(2+1)-1+80;
3       int x = ((3+6)%2)*(6-(3%4));
4       println(y);
5       println(x);
6 }
```

Le test ci-dessus vérifie à la fois l'implémentation des opérateurs multiplication, division, addition, soustraction et modulo.

1.2 Tests de la partie objet

Après avoir fait les tests de la partie sans objet, nous avons fait les tests pour tester la partie objet du compilateur.

La partie objet se compose principalement de tests unitaires, simplement, cette fois-ci, se sont des classes qui sont testées.

Aucun test n'a été spécifiquement implémenté pour le mot "return" obigatoire dans les méthodes autres que void. Cependant, il est souvent utilisé dans les autres tests ce qui suffit à vérifier son fonctionnement.

Pour tester le bon fonctionnement de l'implémentation objet, le fonctionnement général des tests est le même que l'exemple suivant :
une classe voiture hérite d'une classe véhicule et implémente différentes méthodes dont au moins un constructeur, des getters et des setters ainsi que les différentes méthodes abstraites de la classe véhicule.

1.3 Tests des options du compilateur

Les options du compilateur (-p, -v, -r, ...) sont testées via le script basic-decac.sh contenu dans le dossier src/test/script/.

Chaque option est utilisée sur tous les fichiers tests Deca contenus dans le dossier src/test/deca/context/valid/. En effet, le code du test basic-decac boucle sur chacun de ces tests Deca avec la commande "decac -option $cas_de_test" avec cas_de_test correspondant à l'élément courant de la boucle.

Au lancement du script, pour chaque fichier Deca utilisé pour tester l'option de compilateur, le programme affiche "OK" en cas de réussite et "Error", avec la description de l'erreur, en cas d'échec.

Exemple de résultat obtenu dans le terminal au lancement d'un test decac :

```
1 ------ Test de decac -r  pour le fichier src/test/deca/codegen/
      valid/ecrit0.deca : ------
2 ------ OK ------
3
4 ------ Test de decac -r  pour le fichier src/test/deca/codegen/
      valid/globalTests.deca : ------
5 /user/1/.base/fuentesa/home/gl03/src/test/deca/codegen/valid/
      globalTests.deca:33:32: token recognition error at: '>'
6 ERREUR: decac -r 3 a termine avec un status different de zero
```

Chaque test cible une option du compilateur de manière bien précise, sauf pour -r qui est utilisé en combinaison avec -P pour paralléliser les tests avec des valeurs de -r différentes (afin aussi de limiter la durée de ce test assez copieux). Par exemple, le test pour l'option -r ("decac -r $nombre") teste tout les nombres allant de 1 à 20. Le résultat attendu est son bon fonctionnement pour les nombres allant de 2 à 16 et une erreur pour les autres.

1.4 Objectif et conclusion des tests

1.4.1 Description de l'objectif

L'objectif des tests est de minimiser le nombre d'erreurs non prises en compte lors du développement du compilateur et de s'assurer que le logiciel répond à l'ensemble des besoins et des attentes définies dans le cahier des charges.

Il doivent couvrir le plus grand nombre d'erreurs possibles ou combinaisons possibles dans la syntaxe (exemple : "a // b" et "a / /b" correspondent à 2 tests bien distincts de l'opération division entière).

1.4.2 Organisation des tests

Les tests peuvent vérifier le bon fonctionnement d'une fonctionnalité tout comme le fait que la bonne erreur soit levée lorsqu'elle est censée l'être. Ils sont ainsi contenu soient dans un dossier /valid pour les tests vérifiant une bonne implémentation, soient dans un dossier /invalid pour les tests vérifiant le bon comportement des erreurs.

Le nom d'un fichier de test doit aussi donner un maximum d'information sur le contenu du test (au moins pour les tests les plus simples). Ainsi le test suivant, qui vérifie que la somme d'un float et d'un string dans une variable float est contextuellement incorrect, est contenu dans le fichier "plus_floatvar_float_plus_string_context" :

```
1 {
2     float y=  3.5 + "Bonjour";
3 }
```

1.4.3 Conclusion des tests

Seulement 3 tests ne sont pas validés. On peux donc dire que notre compilateur couvre presque l'ensemble des possibilités auxquelles nous avons pensé.

Il est cependant nécessaire de rappeler qu'ils nous est impossible de couvrir l'ensemble des situations possibles en seulement 3 semaines de travail. Il est certains que la couverture de nos tests est loin d'être parfaite. Le but ici reste de couvrir un maximum de possibilités afin de rendre un produit final de la meilleure qualité possible.

Exemple de script où tout les tests sont validés :

```
1 Compilation parallèle avec limitation des registres :
2 ### SCORE: 14 PASSED / 14 TESTS ###
3
4 Programme à nombre de registres restreints exécutés :
5 ### SCORE: 1806 PASSED / 1806 TESTS ###
6
7 Tests de vérification d'échec en cas de mauvaise valeur pour la
      quantité de registres :
8 ### SCORE: 6 PASSED / 6 TESTS ###
```

2 Lancement des scripts de test

Le nom des scripts est choisi comme le plus simple et le plus clair possible pour comprendre la partie testée. On les retrouve dans le dossier /src/test/script/.

Selon le type de test à lancer :
- le script basic-lex.sh vérifie le bon fonctionnement de la partie lexème du compilateur
- le script basic-synt.sh vérifie le bon fonctionnement de la partie syntaxique du compilateur
- le script basic-context.sh vérifie le bon fonctionnement de la partie contextuelle
- le script basic-decac.sh vérifie le bon fonctionnement de la compilation du code
- le script basic-gencode.sh vérifie le bon fonctionnement de la génération du code assembleur

La commande "mvn test" permet de lancer à la suite l'ensemble des scripts ci-dessus.

3 Gestion des risques et des rendus

3.1 Analyse et gestion des risques

3.1.1 Risques liés à la gestion du projet

Cette section comprend les différents risques pouvant être liés à une mauvaise gestion de projets notemment au niveau des dates de rendus.

Mauvaise planification des dates de rendus : oublier les dates de rendu ou ne pas les prendre en compte dans la planification des tâches.
Risque : rendre un compilateur complet et fonctionnel en retard par rapport aux dates de rendu annoncées ou rendre un compilateur incomplet voire non fonctionnel
Effet : perte de crédibilité auprès du client
Action mise en oeuvre : rappels récurrents et automatiques des échéances 2 jours avant les dates de rendu (voir partie 3.2).

Mauvaise gestion des user-story : mauvaise distribution des tâches aux membres du groupe possiblement trop conséquentes
Risque : surcharge de travail pour certains membres et sous-utilisation des compétences d'autres membres
Effet : rendu du projet possiblement retardé
Action mise en oeuvre : attribution des tâches selon les compétences, la disponibilité et les intérêts de chaque membre afin d'optimiser la productivité du groupe

3.1.2 Risques internes au travail en équipe

Le travail en équipe (en particulier grande équipe) nécessite une bonne organisation mais aussi une bonne communication et entente générale entre tout les membres du groupe. C'est un composant important tant pour les individus que pour l'efficacité globale de l'équipe.

Manque de communication : la communication est essentielle pour coordonner les efforts et partager les informations
Risque : malentendus, retards et diminution de l'efficacité globale
Effet : rendu du projet possiblement retardé suite à un mauvais travail d'équipe
Action mise en oeuvre : des réunions régulières sont organisées à 9h30 tout les matins et l'outil de collaboration en ligne Trello nous permet de coordonner le travail d'équipe

Risque de dépendance à une membre clé : si le groupe repose sur une seule personne ayant une expertise technique ou des connaissances spécifiques
Risque : il y a un risque élevé en cas d'indisponibilité de cette personne
Effet : retards importants sur le rendu
Action mise en oeuvre : partage des connaissances entre les membres du groupe et collaboration étroite afin que plusieurs personnes soient capables de prendre en charge

différentes responsabilités

Conflits : des conflits au sein d'un groupe peuvent entrainer une perte de confiance entre ses membres
Risque : diminution de la productivité et divergence dans le choix d'approche des tâches ou du code
Effet : prise de retard sur les écheances de rendu et perte de crédibilité auprès du client et de futurs collabotateurs
Action mise en oeuvre : les reunions quotidiennes permettent de régler les conflits pouvant apparaître au sein du groupe et la communication entre les membres doit rester un élément clé dans la gestion du projet et de l'équipe

3.1.3 Risques techniques liés à la programmation informatique

Le développement d'un projet informatique en équipe est un processus complexe qui peut être freiné par de nombreux facteurs. Il est important de reconnaître ces éléments et de mettre en place des pratiques de développement solides.

Non respect de la nomenclature ou des conventions : utilisation de noms incohérents, ambigus ou non descriptifs pour les fichiers, les variables ou les fonctions
Risque : code difficile à comprendre et à maintenir entrainant des difficultés à reprendre le code pour un autre développeur
Effet : prise de retard non prévue sur le développement du compilateur
Action mise en oeuvre : les règles à respecter sont rappelées sur le GitLab commun dans l'onglet Wiki et sont connues par tout les membres

Manque de compétences techniques : les membres du groupe ne possèdent pas les compétences techniques nécessaires pour mener à bien le projet
Risque : erreurs, retards et mauvaise qualité de travail
Effet : perte de crédibilité auprès du client
Action mise en oeuvre : évaluation des compétences des membres du groupe avant de commencer le projet puis remise à niveau notemment en travaillant le TD2

Ne pas mettre à jour le git quotidiennement : si un membre ne push pas son travail sur la plateforme GitLab commune à l'équipe avant la fin de la journée ou au moins le plus souvent possible
Risque : reprise du code par un autre membre de l'équipe impossible
Effet : prise de retard possible dû à l'attente d'un push de la part du développeur concerné
Action mise en oeuvre : rappel récurrent de push le travail sur GitLab par le TechLead

3.1.4 Risques techniques liés aux tests

Il est important de mettre en place des tests de non-régression approfondis pour chacune des étapes afin d'identifier et atténuer les risques que le compilateur soit défaillant ou ne répondent pas à toutes les attentes du client.

Mauvaise couverture des tests : les tests ne couvrent pas tous les scénarios possibles
Risque : des bogues pourraient ne pas être détéctés ou des fonctionnalités ne pas être entièrement implémentées
Effet : perte de crédibilité auprès du client suite au rendu d'un compilateur non fonctionnel
Action mise en oeuvre : un membre a été assigné spécifiquement à la rédaction des tests avec pour tâche de couvrir le plus de cas possibles

Incompréhension des scripts de test : certains tests sont codés en langage shell qui n'est pas un langage compris par tout les membres du groupes
Risque : toute l'équipe ne peut pas travailler sur les tests à moins de former les membres non compétents
Effet : prise de retard dans le développement des tests dû à un besoin de formation pour certains membres
Action mise en oeuvre : l'écriture des tests en shell a été assigné aux membres les plus compétents en langage shell

3.1.5 Risques techniques liés à l'intégration du code

Une mauvaise gestion de l'intégration du code des divers developpeurs vers le dépôt principal du projet peut nuire à la collaboration et à la productivité globale de l'équipe.

Intégration dans les branches principales du dépôt git : la branche principale correspond à l'état actuel du projet, ainsi toute modification merge sur cette branche amène un danger si la modification amène un bug
Risque : push de bugs vers la branche principale qu'il va donc falloir débuguer
Effet : perte de temps pour un problème facilement évitable
Action mise en oeuvre : aucun push ne doit être effectué directement vers la branche principale et les merge request sont vérifiées et testées avant d'être acceptées ou non

Gérer les "merge" : une mauvaise gestion des fusions (merges) peut entraîner des problèmes dans le développement collaboratif
Risque : conflits lors des merges, perte de données ou intégration incomplète du code développé
Effet : perte de travail ou de temps
Action mise en oeuvre : vérification du code avant de merge, "git pull origin master" fréquents depuis les autres branches pour éviter un maximum de conflits avec la branche principale

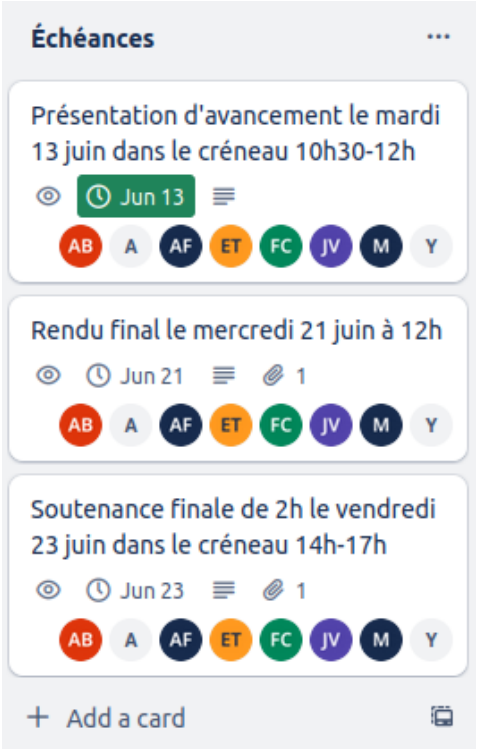
3.2 Gestion des rendus

La gestion des rendus permet de s'assurer que le projet est livré à temps. En définissant des échéances claires, en suivant les progrès et en effectuant des ajustements si nécessaire, on évite les retards et les dépassements de délais de rendu.

Une gestion des rendus efficace intégre beaucoup de communication notemment sur les prochaines échéances et l'avancée du projet. En effet, en communiquant régulièrement sur l'avancement du projet, on assure l'implication de tout les membres du groupe dans la suite du travail à effectuer.

L'outil de gestion de projet que nous avons choisi est Trello. Tout les membres du groupe ont accés à un projet commun sur cet outil où l'on retrouvre chaque user-story (rédigée par le Product Owner) et leur avancée. Les dates et heures de rendu y sont rappelées et un rappel automatique par mail des prochains éléments à rendre est lancé 2 jours avant la prochaine échéance.

Enfin, les prochaines échéances sont réguliérement rappelées lors des réunions quotidiennes à 9h30 tout les matins.



Avant chaque rendu, en plus de la réunion Daily quotidienne où sont rappelés le travail déjà effectué et les objectifs de la journée, l'équipe de gestion se réunit pour vérifier à plusieurs le contenu de la documentation. La documentation est aussi relue par le Product Owner qui a une vision globale du projet et par les développeurs qui maîtrisent au mieux les sujets abordés par les documents.

L'équipe de développement se réunit aussi pour vérifier que l'entiéreté du code à implémenter a bel et bien été implémentée. Toute modifictaion non implémentée doit être notifiée et notée dans les limites du compilateur dans le document correspondant au manuel utilisateur.

La relecture est un point important du processus avant un rendu car aucune faute ne doit être laissée dans les documents rendu au client.

De même, le code est de nouveau testé avant le rendu afin de ne laisser aucun bug dans le projet final.

Si un bug est trop important pour être corrigé avant l'échéance de rendu et que la fonctionnalité qui y correspond n'est pas purement nécessaire, il a été décidé qu'il valait mieux retirer la fonctionnalité et la lister dans les limites du compilateur.

Une fonctionnalité présentant un disfonctionnement mais étant trop importante sera laissée dans le projet final seulement après correction entraînant un retard dans le rendu. Cette situation est à éviter impérativement (voir les risques partie 3.1).

Pour rappel les échéances propre au projet sont les suivantes :

— Lundi 12 Juin à 12h : Rendu intermédiaire 2
— Mercredi 21 Juin à 12h : Rendu final

Et les présentations d'avancement du projet sont les suivantes :

— Mercredi 7 Juin dans le créneau 9h00-10h30 : Présentation chartes de travail en grandes équipes
— Mardi 13 Juin dans le créneau 10h30-12h : Présentation d'avancement
— Vendredi 23 Juin dans le créneau 14h-17h : Soutenance finale de 2h

4 Résultats de la méthode Jacoco

Jacoco (Java Code Coverage) est un outil de mesure de la couverture du code pour une application Java. Il permet de déterminer quelle partie du code est exécutée lors de l'exécution de tests automatisés (voir documentation du projet GL).

Après exécution des commandes Jacoco, le résultat obtenu est le suivant :

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		58%		44%	486	626	874	1,954	276	363	16	48
fr.ensimag.deca.tree		81%		80%	114	469	188	1,045	74	346	5	66
fr.ensimag.deca.context		47%		40%	74	107	103	177	71	102	7	21
fr.ensimag.deca		72%		71%	24	75	62	222	7	44	1	5
fr.ensimag.ima.pseudocode		76%		80%	30	84	45	179	26	74	5	26
fr.ensimag.ima.pseudocode.instructions		62%		n/a	25	63	42	113	25	63	20	54
fr.ensimag.deca.tools		96%		100%	2	26	4	71	2	23	0	4
Total	5,448 of 16,150	66%	337 of 814	58%	755	1,450	1,318	3,761	481	1,015	54	224

Le if n'est couvert qu'à moitié étant donné que la branche else n'est pas couverte. De même, la boucle while n'est couverte qu'à moitié. Les catch et le return ne sont pas couverts dans toutes les configurations possibles non plus.

fr.ensimag.deca.syntax

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Cl
DecaParser		64%		44%	216	293	532	1,423	30	64	0	

```
952.        catch (RecognitionException re) {
953.            _localctx.exception = re;
954.            _errHandler.reportError(this, re);
955.            _errHandler.recover(this, re);
956.        }
957.        finally {
958.            exitRule();
959.        }
960.        return _localctx;
961.    }
962.
963.    @SuppressWarnings("CheckReturnValue")
964.    public static class List_exprContext extends ParserRuleContext {
965.        public ListExpr tree;
966.        public ExprContext e1;
967.        public ExprContext e2;
968.        public List<ExprContext> expr() {
969.            return getRuleContexts(ExprContext.class);
970.        }
```

Au final, la couverture générale d'après Jacoco est de 66% ce qui représente une mauvaise couverture en soit. Ce mauvais pourcentage s'explique par les limites du compilateur mais aussi par le fait que Jacoco teste la couverture d'éléments qu'il ne fait pas sens de tester comme les instructions en asm (NEW, BSP, SUBSP, FMA...).

5 Autres méthodes : git, relecture et validation

5.1 Nomenclature et bonnes pratiques à respecter

Pour faciliter la relecture et le travail en équipe sur un code et des platformes en communs, il est nécessaire de décider de plusieurs conventions et règles que les membres devront impérativement respecter.

5.1.1 Pratiques propre au code

Les pratiques propres au code du projet se basent sur les conventions communes à la plupart des projets informatiques Java. Les conventions à respecter sont donc les suivantes (beaucoup de CamelCase) :

- Le code est écris intégralement en anglais
- Les attributs sont au moins protected et accédés par des getters et des setters
- Ne pas oublier de commenter le code
- Le nom des variables, des methodes et des classes doit permettre de facilement retrouver ce qu'elles représenteent et doit être unique dans le programme où elles sont initialisées
- Le CamelCase des variables est le suivant : maVariable (la première lettre est minimuscule et chaque nouveau mot commence par une majuscule)
- Le CamelCase des méthodes est similaire à celui des variables : maMethode() (la première lettre est minimuscule et chaque nouveau mot commence par une majuscule)
- Le CamelCase des classes est le suivant : MaClasse (la première lettre est majuscule et chaque nouveau mot commence par une majuscule)

Prenons en exmemple le code suivant :

```
1 public class e {
2     public int elementx;
3 }
```

qui devient (en respectant la nomenclature et les conventions) :

```
1 public class ElementX {
2     private int elementX;
3     public int getElementX() {
4         return this.elementX;
5     }
6     public void setElementX(int newElement) {
7         this.elementX = newElement;
8     }
9 }
```

5.1.2 Description de la structure GitLab

Nous avons organisé notre travail en équipe depuis la plateforme GitLab. La branche principale est la branche master. Elle correspond à une implémentation du code du projet stable et validée par l'équipe de développement. Pour toutes les fonctionalités implémentées ou testées, une nouvelle branche est créée. Aucun push ne doit être effectué directement vers la branche principale. La première étape correspond à merge la branche correspondant au développement de la fonctionnalité sur la branche correspondant au développement des tests. Si les tests sont passés, le développeur peut demander à merge sur la branche principale. Le merge sera validé après relecture.(voir partie 5.2).

En rouge : la branche main
En vert et en rose : deux branches correspondant à deux fonctionnalités différentes à implémenter

La branche verte a été créée mais pas encore merge (il reste des fonctionnalités à implémenter ou les modifications qu'elle aporte n'ont pas encore été validées). La branche en rose a été entièrement implémentée et merge vers la branche main (où l'on retrouvera les fonctionnalités qu'elle implémente).

Enfin, lorsqu'un push est effectué, l'ensemble du projet est testé automatiquement (comme si la commande "mvn test" étais lancée) avant de valider le push. Cette étape peux ralentir le groupe mais permet de toujours push un travail testé et validé.

5.1.3 Toutes les conventions à respecter du GitLab commun

Travailler en équipe sur GitLab demande de décider de plusieurs règles à respecter (Tx est le titre de l'issue où x est le plus petit nombre n'ayant pas déjà d'issue à ce numéro) :

- Une branche par feature / fonctionnalité testée
- Nom des branches features : Feature/Tx_titre
- Nom des branches tests : Tests/Tx_titre
- Nom des branches bugs : Bugs/Tx_titre
- Nom des commit : [fix | feature | test | merge]#x Description avec x le numéro de l'issue gitlab
- Description claire et concise des commits
- Push à minima tous les jours pour ne pas perdre une journée de travail
- Dans les merge requests : laisser des commentaires / un petit résumé de ce que fait le code pour le reviewer
- Les branches de Tests et de Devs sont des branches indépendantes
- Une issue est déclarée pour chaque feature / bugs
- Aucun push ne doit être effectué directement sur la branch "master"

5.2 Validation du TechLead

Avant de merge toutes les modifications ou tout les ajouts sur la branche principale, un système de relecture a été mis en place au sein de l'équipe de développement.

Le TechLead (désigné meneur de l'équipe de développement au début du projet car membre le plus compétent du groupe) a pour tâche de relire le code de chacun des membres du groupe avant de valider ou non un merge vers la branche main.

Les modifications ou les ajouts apportés par le TechLead lorsque celui-ci implémente une fonctionnalité ou règle un bug dans le code sont elles-mêmes relues par un autre membre de l'équipe le plus compétent possible ou possédant le moins de lacunes possibles vis à vis de la fonctionnalité implémentée (test, parser, etc).

Si un membre est amené à relire du code, il doit être intransigeant et ne laisser passer aucune imperfection de nomenclature, de convention ou tout autre chose remarquée.