Document de validation

GL 03

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

12 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

Cette documentation concerne uniquement Deca sans objet pour le rendu intermédiaire 2

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 chacunes 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 vérifient à la fois l'implémentation des opérateurs multiplication, division, addition, soustraction et modulo.

1.2 Tests de la partie objet

// A faire

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.

```
----- Test de decac -r  pour le fichier src/test/deca/context/valid/hello-world.deca :  -----
----- OK -----

----- Test de decac -r 17 pour le fichier src/test/deca/context/valid/hello-world.deca :  -----
Error during option parsing:
L'option du compilateur "-r" ne prend en charge que les entiers positifs <= 16 et >= 2
Exception in thread "main" java.lang.UnsupportedOperationException: not yet implemented
        at fr.ensimag.deca.CompilerOptions.displayUsage(CompilerOptions.java:133)
        at fr.ensimag.deca.DecacMain.main(DecacMain.java:25)
```

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 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 }
```

1.5 Objectif des tests

1.5.1 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 diviser).

1.5.2 Conclusion des tests

// A faire

2 Lancement des scripts de test

Le nom des scripts est choisis comme le plus simple et le plus clair possible pour comprendre la partie testée.

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

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

3.2 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 choisis 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.

4 Résultats de la méthode Jacoco

// A faire

5 Autres méthodes : relecture et validation

5.1 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 en vert 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).

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 relue 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).