Grenoble INP - Ensimag

Documentation De Validation

PROJET-GL GROUPE 49

BENTAIBI Oussama
AÏT HAMMOU Driss
AJJA Hamza
DAOUD YOUSSEF
LEACHOURI Khalil

2^{nd} Year - 29 janvier 2021

Table des matières

I Introduction :

Le projet GL est un projet d'une grande ampleur. Pour un travail informatique si volumineux, il ne faut jamais sous-estimer l'étape de la validation. Bien au contraire, les tests doivent être au cœur du développement pendant toutes les phases du projet. Ce document va détailler donc notre stratégie de validation pour notre projet.

II Conception et réalisation des tests :

II.1 Convention d'écriture :

Le test commence par un commentaire préfixé de DESCRIPTION décrivant brièvement l'utilité et le contenu du test. Le résultat du test est indiqué selon le test, par exemple pour les tests invalides de contexte, on détaille l'erreur en sortie dans un autre commentaire préfixé par RESULTAT :

II.2 Tests pour l'étape A :

Nous utilisons le test ManueltestSynt pour construire un arbre non décoré correspondant au fichiers Deca de notre base de tests. On distingue ici deux répertoires : Le premier est le répertoire des tests valides dans lequel on vérifie si les arbres de nos programmes Deca valides syntaxiquement à la main. Il y'en a un deuxième répertoire de tests invalides qui doivent nous afficher des erreurs de syntaxe. Pour être sûr que les tests passent comme ils doivent passer, on repassait les tests à chaque fois que l'on arrivait à une étape de validation syntaxique. Comme ça, on se protège des regressions qui peuvent se passer lors du développement de notre compilateur. Au début, on a également créé un répertoire pour les tests de lexique, mais qui est devenu sans importance dans les phases les plus avancées du développement de notre projet.

II.3 Tests pour l'étape B :

L'architecture des tests et la façon de la validation pour cette étape était la même que l'étape précédente. On a essayé de couvrir tous les cas possibles pour le contexte surtout pour les exceptions des tests invalides. Vous allez donc trouver dans chaque test l'exception attendu après l'utilisation de l'exécutable "test context" sur lui.

II.4 Tests pour l'étape c :

On a créé deux types de tests : "valid" et "invalid". Dans le répertoire "valid", notre compilateur marche pour tous les tests. Par contre, pour les cas particuliers ou il y a encore des bug, on classe ces tests comme "invalid". Afin d'obtenir le résultat d'un fichier test.deca, on passe par deux étapes : Exécution : On génère le fichier assembleur ".ass" à partir de l'éxecutable "src/main/bin/deca". C'est une commande simple : "decac chamin/fichier.deca" dans le répertoire du projet. Interprétation : On interprète le fichier ".ass" généré en utilisant "ima". Il faut juste taper "ima chemin/fichier.ass" dans le terminal.

II.5 Scripts de tests :

Nous avons aussi réalisé un nombre de de scripts de test en langage shell permettant de faire tourner le compilateur (ou une partie précise de ce compilateur) sur un ensemble de tests en langage Deca. On a également automatisé les scripts shell valides qu'on a écrit et vous pouvez les lancer tous en utilisant la commande Maven test.

iter-lex.sh : Ce test utilise l'exécutable "test lex" pour lancer le test ManuelTestLex sur l'ensemble des tests de lexique. Il peut être lancé via la commande : ./src/test/script/iter lex.sh. Ce fichier ne vérifie pas si les fichiers valides sont corrects mais nous aide à retrouver des regressions très importantes.

iter-synt.sh : Ce test utilise l'exécutable "test synt" pour lancer le test ManuelTestSynt sur l'ensemble des tests de syntaxe. Il peut être lancé via la commande : ./src/test/script/iter-synt.sh. Ce fichier ne vérifie pas si les fichiers valides sont corrects mais nous aide à retrouver des regressions très importantes.

iter-context.sh : Ce test utilise l'exécutable "test context" pour lancer un test de vérification contextuelle sur l'ensemble des tests de contexte. Il peut être lancé via la commande : ./src/test/script/iter-context.sh. Ce fichier ne vérifie pas si les fichiers valides sont corrects mais nous aide à retrouver des regressions très importantes.

common-tests.sh : C'est un fichier fourni par les professeurs avec un contenu un peu lisible... en hypothèse, il vérifie certaines fonctionnalités de bases mais importantes pour le langage Deca. Il peut être lancé par la commande : ./src/test/script/common-test.sh. iter-decac : Ce test devrait compiler les fichiers Deca de test et comparer le résultat obtenu avec un résultat attendu que l'on devrait écrire à la main. C'est un peu comme un test automatique qui fait la vérification de tout le compilateur. Malheureusement, on n'a pas eu le temps pour écrire les fichiers des résultats attendu et même de tester ce script...

iter-decac : Ce test devrait compiler les fichiers Deca de test et comparer le résultat obtenu avec un résultat attendu que l'on devrait écrire à la main. C'est un peu comme un test automatique qui fait la vérification de tout le compilateur. Malheureusement, on n'a pas eu le temps pour écrire les fichiers des résultats attendu et même de tester ce script...

III Gestion des risques :

risque	conséquence	solution	exécution
-Mal compréhension du travail à faire.	-Retard,erreurs lors de l'exécution des tests	-Contact des encadrants . -Réunion.	oui
-probleme de l'environnement du travail	-incapacité du travail pour certains membres du groupe.	-Travail à deux.	oui
-Absence d'un membre de l'équipe	-Retard.	-programmation par paire.	oui
-Manque de ponctualité de certain membres de l'équipe	-Tâches non réalisé -travail supplémentaire pour les autres membres	-un membre d'équipe (celui qui est en charge de l'organisation de travail) s'occupe de rappeler les membres de leurs tâches et les deadlines .	non
-déséquilibre de l'avancement(retard dans une partie)	-Retard	-Effort supllémentaire -Aide des autres membres si c'est possible.	oui
-erreurs lors des tests	-Compilateur faible	-Debugger la partie concernée	oui
-erreurs avec git	-perdre les heures de travail	-être prudent lors de la manipulation de git	oui

III.1 Risques de l'étape A :

En cas d'oubli de certaines règles dans le lexer ou le parser, le programme ne va pas reconnaître la signification des éléments correspondants.

III.2 Risques de l'étape B :

Oubli d'une règle : Dans ce cas, le programme ne produit pas d'erreur sur un fichier pourtant invalide, et donc il faudra compléter au maximum la base de test, notamment les tests invalides. Oubli de décoration d'un noeud : vérifier la décoration en lancer le "test context" sur les fichiers. Ajout d'une vérification qui impacte d'autres cas : faire tourner systématiquement la base de tests pour éviter les regressions.

III.3 Risques de l'étape C :

On a créé deux types de tests : "valid" et "invalid". Dans le répertoire "valid", notre compilateur marche pour tous les tests. Par contre, pour les cas particuliers ou il y a encore des bug, on classe ces tests comme "invalid". Afin d'obtenir le résultat d'un fichier test.deca, on passe par deux étapes : Exécution : On génère le fichier assembleur ".ass" à partir de l'éxecutable "src/main/bin/deca". C'est une commande simple : "decac chamin/fichier.deca" dans le répertoire du projet. Interprétation : On

interprète le fichier “.ass” généré en utilisant “ima”. Il faut juste taper “ima chemin/fichier.ass” dans le terminal.

IV Gestion des rendus :

Étapes par ordre	buts
-Regrouper l’ensemble des fichiers+docs dans un seul ordinateur.	-Indispensable.
-Exécuter tous les scripts des tests.	-Trouver de potentielles erreurs.
-Correction des erreurs trouvées.	-Réaliser un compilateur fonctionnel.
-Faire un git push en s’assurant que tous les documents existent dans le dépôt git(git add…).	-Envoyer le rendu dans le cloud
-Jouer le rôle du client en récupérant le dépôt git sur un autre ordinateur qui ne contient pas le répertoire Projet-GL.	-S’assurer que le dépôt git contient tous les fichiers nécessaires.
-Jouer le rôle du client en lançant une autre fois les tests.	-S’assurer que le client ne trouvera pas d’erreurs dans le compilateur .
-Faire un commit final.	-Indispensable.