De l'apprentissage à l'évaluation : `grader`

Floréal Morandat

Univ. Bordeaux - Enseirb-Matmeca, Bordeaux INP - LaBRI - UMR CNRS 5800
fmorandat@enseirb-matmeca.fr

Abstract

Cet article présente `grader`, une bibliothèque de test en C. Elle est spécialement conçue pour les étudiants débutants qui ne peuvent pas encore utiliser les outils de déboguage existants. Elle propose des messages d'erreur les plus clairs possibles pour l'étudiant, tout en minimisant les efforts de l'enseignant pour les obtenir. Cette bibliothèque est utilisée dans le contexte d'une plate-forme d'exercices en ligne ainsi que pour l'évaluation des matières de programmation de première année.

1 Introduction

Débuter en programmation est difficile et ce quel que soit le langage de programmation considéré. Notre école a fait le choix d'utiliser le C comme langage d'initiation à la programmation et à divers modules connexes. Ce choix, qui peut être contesté, présente néanmoins l'avantage de disposer d'un langage unique, rigoureux, simple — 32 mots clefs, aucun comportement implicite — et complexe en même temps — typage statique, gestion manuelle de la mémoire, chaîne de compilation non triviale. Bien que l'écosystème du C soit riche — il dispose d'excellents outils de deboguage — il est difficile d'apprendre aux étudiants à les utiliser en même temps que de leur apprendre à programmer.

De la même manière que nous avons appris à compter avant de nous servir d'une calculatrice, nous pensons qu'un IDE trop complet empêche les étudiants de réfléchir aux problèmes, et qu'ils se limitent à faire disparaître le rouge sans réellement chercher à comprendre. Il nous paraît aussi important qu'ils intègrent rapidement la réalité de la programmation : les messages d'erreurs ne sont pas toujours rattachés à la vraie cause de l'erreur et le programme possède des comportements erronés (boucle infinie, arrêt soudain, ...). L'objectif de l'étudiant est alors d'atteindre précisément un comportement spécifié.

Pour toutes ces raisons nous avons fait le choix d'utiliser un éditeur de texte, un terminal et un compilateur standard (`gcc` ou `clang`). De plus, nous avons décidé que pour les premières matières de programmation, leur évaluation se déroulerait sur ordinateur. Ce choix est à double tranchant, ils disposent du manuel, des retours du compilateur et de l'exécution. Par contre, ils n'ont pas le droit à *l'à peu près*. Ceci permet aussi de corriger les épreuves impartialement, puisqu'elles seront tout naturellement corrigées par un oracle : l'exécution de tests unitaires.

Dans ce cadre, nos objectifs sont :

- **Remonter des messages d'erreurs pertinents** sans trop altérer la réalité, sachant que i) l'étudiant ne sait pas utiliser un débogueur, ii) qu'un programme C s'exécute jusqu'au moment où il s'arrête brutalement (débordement de tableau qui provoque un écrasement de la pile), iii) que l'étudiant n'a pas forcément un accès ou une compréhension des tests.
- Autoriser les étudiants à **traiter les questions de manière indépendantes**, tout en permettant des dépendances entres les fonctions. Dans tous les cas, le programme doit compiler et s'exécuter normalement.
- **Vérifier des propriétés non triviales** qui vont au delà des simples vérifications du compilateur, par exemple : une fonction est-elle récursive, l'étudiant a-t-il géré correctement la mémoire, utilise-il une fonction interdite ?

– **Soutenir le travail de l'enseignant** dans l'écriture des exercices, en lui permettant de se focaliser sur ce qu'il veut tester.

Nous proposons une bibliothèque, `grader`, qui prend en charge l'ensemble de ces préoccupations tout en étant peu intrusive pour l'étudiant. Pour assister l'enseignant dans l'écriture des tests, elle fournit des assertions de haut niveau (e.g., vérifier que tous les éléments d'un tableau de structures sont présents et ce dans n'importe quel ordre). Elle permet de définir rapidement des afficheurs pour les structures complexes. Pour simplifier l'écriture et la maintenance des exercices, les énoncés sont directement extraits des solutions et les solutions sont utilisables par l'étudiant de manière transparente s'il n'arrive pas à écrire une fonction. Enfin cette bibliothèque se décline en deux variantes, une version en ligne, type plate-forme d'exercices, et une version autonome pour réaliser des épreuves machines hors ligne.

2 Grader DSL

L'idée d'écrire des tests pour évaluer — évaluer n'est pas noter — du code n'est pas nouvelle, elle est cependant peu appliquée à cause de nombreuses difficultés. Écrire des tests est long et verbeux, surtout en C, il est donc facile d'oublier des détails. Avoir un taux de couverture des spécifications élevé nécessite de nombreux cas de test. Dans les langages impératifs, les effets de bords nécessitent des traitements spéciaux ne serait-ce que pour donner des messages corrects — les valeurs avec lesquelles une fonction est appelée ne sont pas forcément les mêmes après l'appel. Un effort conséquent doit être fourni pour avoir des messages d'erreur clairs, sachant que l'étudiant n'a pas forcément accès au code.

Pour toutes ces raisons, nous proposons un DSL construit en pré-processeur C, sans dépendances externes, qui s'intègre naturellement dans le *workflow* de l'étudiant. Un test se construit autour de trois entités : une fonction à tester (le code de l'étudiant), une fonction de test (TEST_FUNCTION) et des données de tests (TEST_CASES). Ces entités sont (par défaut) automatiquement liées grâce à nom de la fonction. Le listing 1, montre le test intégral pour la fonction index_of.

Listing 1: Un exemple de fonction de test écrit avec `grader`

```
1  TEST_CASES(index_of,
2      int expected, const char *needle, int n, const char *haystack[10])
3  {
4    {0, "Bonjour", 1, {"Bonjour"}}, // Premier test
5    {1, "Bonjour", 2, {"Hello", "Bonjour"}}, // Second test
6    // ...
7  }
8
9  TEST_FUNCTION(index_of)
10 {
11   FOR_EACH_TEST {
12     int actual = DESCRIBE(index_of(_.needle, _.n, _.haystack),
13         "index_of(\"%s\", %d, %S)", _.needle, _.n, _.haystack);
14     ASSERT_INT(_.expected, actual, AS_RETURN);
15   }
16 }
```

Le test de la fonction ne se déclenche que si l'étudiant essaye d'écrire la fonction index_of. Les données de tests définissent les arguments qui seront fournis à la fonction de test — dans cet exemple, il y a deux cas à tester. L'itération sur les cas de test se fait au moyen de FOR_EACH_TEST, qui définit automatiquement dans son scope une variable sur le test courant (_). Le test à proprement parler est réalisé entre les lignes 12 et 14. L'appel au code de l'étudiant

est encadré par sa description (DESCRIBE) qui sera utilisé en cas d'erreur de segmentation ou plus généralement en cas d'échec du test. Cette étape, un peu redondante, est malheureusement obligatoire en l'absence de réflexion. On peut remarquer la présence d'un format non standard dans cet appel, %S. Ce format affiche un tableau de chaînes de caractère (ici _.haystack contenant _.n éléments) séparées par des virgules et entourées d'accolades. Finalement la ligne 14, vérifie que la valeur attendue (_.expected) est égale à la valeur (actual) retournée (AS_RETURN) par l'étudiant.

Bien que cet exemple soit relativement court et simple, il n'en demeure pas moins complet. Le listing 2 montre un message d'erreur typique que renverrait ce test. Sans être parfait le message est relativement précis pour un effort moindre.

Listing 2: Un message d'erreur renvoyé pour le listing 1

```
In test_index_of: Expecting as return value '1' but found '0'. Called by:
  index_of("Bonjour", 2, {"Hello", "Bonjour"})
```

3 Structure d'un exercice et chaîne de compilation

Écrire un exercice est une tâche complexe et itérative. L'énoncé est changé régulièrement pour le rendre plus clair, on change une signature ou une spécification pour simplifier une question à venir, on change une fonction de test pour ajouter un peu de liberté aux réponses correctes. Dès lors, il est difficilement concevable d'écrire les tests, les corrections et l'énoncé dans un ordre unique. Cela soulève le même genre de problèmes que la maintenance de code. Enfin même si le code n'est pas amené a évoluer grandement après son déploiement, il faudra tout de même le maintenir, en cas de changement dans l'API de notre bibliothèque, de découverte d'un bogue dans un test ou une correction.

Pour toutes ces raisons `grader` est pensé pour simplifier les opérations de maintenance, principalement entre un énoncé, sa correction et les tests qui en dépendent.

Un exercice est décomposé en trois parties les plus distinctes possibles : le sujet corrigé (`provided.c` dans la figure 1), les tests associés (`exercice.c`) et des méta-informations (`Makefile.mk`). Les méta-informations pour un exercice sont : sa description (pour la plateforme en ligne), les options de compilation, les bibliothèques supplémentaires nécessaires, le type de gestionnaire de mémoire utilisé, ainsi que la liste des fonctions qui ne peuvent pas être utilisées. Les tests dépendent du sujet car il faut au minimum connaître les signatures des fonctions à tester et avoir la même définition des types (structures, ...). Comme ces définitions sont amenées à changer en fonction des évolutions du sujet, le fichier d'en-tête est automatiquement extrait de la correction par un script. Enfin le sujet est lui aussi extrait de la correction par un script. Ce dernier peut, au choix, effacer le corps d'une fonction (fonction à faire), le remplacer par un autre (fonction à trou ou fonction à corriger), ou le laisser tel quel (à titre informatif pour l'étudiant). Les choix sont directement annotés dans le code source par un commentaire à la déclaration de fonction.

Bien que globalement tous les éléments d'un exercice soient du C standard, la chaîne de compilation présente quelques spécificités représentées par les nuages dans la figure 1. Nous voulons qu'un étudiant puisse se servir d'une fonction même s'il n'a pas réussi à l'écrire et ce de manière transparente. Pour cela lors de la compilation de la solution, toutes les fonctions sont renommées par leur noms préfixés par __, les définitions comme les appels. Les noms originaux sont ensuite ré-introduits en version *faible*, c'est-à-dire que lors d'une édition de lien, s'il y a un conflit, les versions faibles sont ignorées.

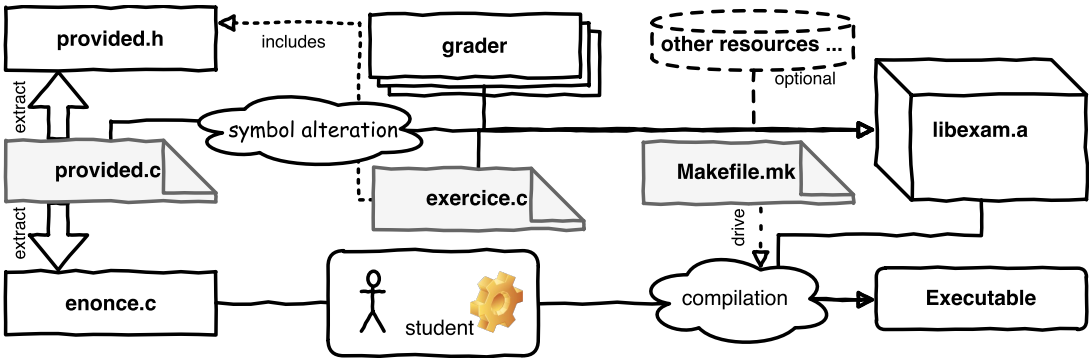

Figure 1: Chaîne de compilation d'un exercice.

Les symboles sont exploités une seconde fois lors de la compilation du code de l'étudiant afin de vérifier qu'il n'utilise pas une fonction interdite. Cette solution n'est pas infaillible mais résiste mieux aux attaques — volontaires ou non — qu'une simple recherche de motif dans le code source.

4 Fonctionnalités

Sans rentrer dans tous les détails de fonctionnement, cette section présente les fonctionnalités importantes ou techniquement compliquées à mettre en oeuvre.

Forks. Chaque test est effectué dans son propre processus. Les bénéfices sont nombreux. Cela assure l'indépendance et l'isolation de chaque test, ce qui limite les problèmes liés aux effets de bords et les erreurs de segmentations. En contre partie, la bibliothèque n'est compatible qu'avec un système répondant à la norme POSIX. Cela rajoute aussi une difficulté pour faire remonter au processus principal l'état d'un test. La bibliothèque se sert du code de retour du processus fils. Il va de soit que les appels depuis le code utilisateur à `exit`, `abort` et `assert` sont reroutés et nécessitent des barrières spécifiques (fournies) pour être rattrapés.

Fixtures. Le DSL propose un système d'initialisation des tests au niveau global, local (pour chaque fonction de test) ainsi qu'au niveau de chaque cas de test (`FOR_EACH_TEST`). Il est principalement utilisé pour initialiser les afficheurs spéciaux ou allouer des tampons.

Chaînes de format non standard. Les chaînes de format des messages sont étendues aux tableaux d'entiers (`%D`), de chaînes de caractères (`%S`) et de structures (`%@`). Pour les structures, l'enseignant peut définir son propre afficheur.

```
1 struct some_struct { int x, y; char *label; };
2 PRINT_RENDERER(some_struct, s, "(%d, %d): %s", s->x, s->y, s->label);
```

Gestion de la mémoire. Nous proposons des exercices où l'étudiant doit allouer de la mémoire — et éventuellement la rendre. Pour cela l'allocateur standard (`malloc` et associés) est redéfini. L'enseignant peut vérifier que la mémoire est bien équilibrée (tout ce qui à été alloué est rendu) ou regarder la quantité de mémoire allouée au bout d'un pointeur.

Version enseignante des fonctions. Dans l'exemple 1, l'auteur a fait le choix de tester la valeur de retour explicitement (`_.expected` dans les données de tests). Une alternative serait de se servir de la version enseignante de `index_of` et de la comparer à la version étudiante. La version enseignante est toujours accessible avec son nom préfixé de `__`, par exemple : `int expected = __index_of(_.needle, _.n, _.haystack);`

Capture de la sortie standard. Pour les exercices réalisant des affichages, la bibliothèque possède un mécanisme de capture de la sortie standard. Au final, il suffit de comparer la chaîne obtenue avec une chaîne attendue (qui peut être obtenue par le même mécanisme).

Récursivité. Il nous paraît important de vérifier certaines propriétés non fonctionnelles, typiquement, savoir si une fonction est récursive ou itérative. Pour cela nous instrumentons automatiquement le code étudiant ce qui permet de savoir si une fonction est appelée et de connaitre la profondeur de la pile d'appel. Malheureusement pour cela nous utilisons une spécificité de `gcc`, un portage pour `clang` n'est cependant pas exclu.

Plate-forme en ligne. Depuis quelques années nous disposons d'une plate-forme d'exercices en ligne. Elle exécute chaque exercice dans son propre conteneur (`docker`). Elle utilise une API Rest pour ces requêtes et nous proposons deux *front-end*. Le premier, sous forme d'un `Makefile`, permet de développer à l'aide de son éditeur favori même sous Windows — seuls `make` et `curl` sont requis. Le second, plus récent, utilise ACE, un éditeur de texte en JavaScript, et ne nécessite qu'un navigateur internet.

5 Discussion

La solution que nous défendons n'est pas parfaite, nous essayons dans cette section de remonter les faits que nous avons pu observer.

Coût en temps d'un examen. Créer un partiel machine complet est une tâche fastidieuse. Globalement nous estimons que le temps de préparation est sensiblement équivalent au temps de correction d'un examen papier d'envergure comparable. Néanmoins, il y a un besoin de maturation du sujet qui doit être fait en amont, afin d'envisager toutes les interprétations possibles par les étudiants, un temps que nous estimons à une semaine.

Problèmes de notation. Il faut d'une part écrire les bonnes signatures, types et commentaires qui guideront l'étudiant vers la solution, mais aussi que ces fonctions soient évaluables de manière non dichotomique. Pour l'évaluation, nous avons fini par ordonner les cas de tests de manière à ce que les cas triviaux soient testés tôt et les cas pathologiques à la fin, faisant souvent l'objet de test séparés. Pour l'attribution d'une note, la formule la plus satisfaisante que nous ayons à l'heure actuelle est : $\lceil 4 * \text{assertions passées/total assertions} \rceil / 4$. Empiriquement, cette formule repartit correctement les notes, généralement selon une loi normale — légèrement biaisée en faveur de l'étudiant.

Effet *code golf*. Pour des exercices à trous, pour lesquels le code est partiellement donné et partiellement faux, un biais apparaît. Comme l'étudiant a un retour immédiat du compilateur et de l'exécution, il lui est possible de passer l'exercice à force d'essais successifs. Ce phénomène est proche de ce que l'on peut voir lors d'oraux, où l'enseignant sait à quoi s'en tenir, ce qui n'est pas le cas de la machine.

Effet des signatures fournies. Le fait que les signatures et les types soient donnés peut être vu comme une limitation, l'étudiant n'a pas à les trouver lui même. Cependant nous ne considérons pas ça comme un problème en début d'apprentissage. Cela permet de montrer la décomposition fonctionnelle aux étudiants et leur permet, parfois, de se resservir de fonctions qu'ils n'ont pas écrites. Dans un cours de manipulation de structures de données, il ne nous paraît pas aberrant d'imposer les interfaces. Il est même possible aux étudiants de créer leurs propres types, dès lors qu'ils implémentent un constructeur. L'écriture des tests associés demande cependant plus de travail du côté de l'enseignant. Dans le même ordre d'idée, pour un examen, fournir les types aux étudiants une semaine en avance, leur permettrait de le préparer plus sereinement.

Limites de l'évaluation. D'une part il y a, comme dans le cadre du development logiciel classique, il n'est pas évident de couvrir l'ensemble des propriétés fonctionnelles que l'on cherche à évaluer. Il faut parfois ajouter des tests à posteriori. D'autre part, il ne faut pas oublier qu'il s'agit d'un processus automatique et qu'évaluer n'est pas noter. Une interpretation des résultats reste nécessaire pour les cas pathologiques.

Limitations de l'implémentation. Comme annoncé précédemment notre bibliothèque n'est compatible qu'avec l'univers POSIX. Du fait des choix de design, l'intégration d'outils comme `gdb` et `valgrind`, pose des problèmes techniques et demande encore de l'ingénierie. De plus le portage vers d'autres plate-formes, telles que MacOS, pose aussi des problèmes. En particulier du fait de l'absence de références faibles.

Actuellement nous cherchons à évaluer si la capacité des étudiants à apprendre à réellement été améliorée, ou si les améliorations constatées sont dues aux autres innovations que nous avons introduites en parallèle. Malheureusement pour évaluer ce point, nous manquons encore de recul.

6 Conclusion

Nous défendons l'idée que `grader` répond bien aux besoins des enseignements en C. Elle est le fruit de trois années d'expérimentations. Comparée à d'autres techniques d'examen machine, elle a permis de réduire drastiquement le temps de préparation des exercices, tout en augmentant la qualité du retour pour les étudiants. L'accueil des étudiants est dans l'ensemble très positif. Ils demandent à avoir généraliser ce type d'outils aux autres cours de programmation. Pour le moment la base d'exercices est limitée (une trentaine d'exercices) mais nous espérons trouver d'autres écoles/enseignants pour augmenter cette base. Les fonctionnalités actuelles devraient permettre aussi de répondre aux attentes de nouveaux modules, nous prévoyons prochainement de rajouter des exercices en programmation système et en programmation réseaux.

A Liens

Plate-forme d'exercice en ligne : `https://thor.enseirb-matmeca.fr:4443`

Code source de la bibliothèque : `https://github.com/morandat/grader/`

Documentation : `https://thor.enseirb-matmeca.fr:4443/doc/exercice`

Base d'exercice : GitHub privé accessible sur demande.