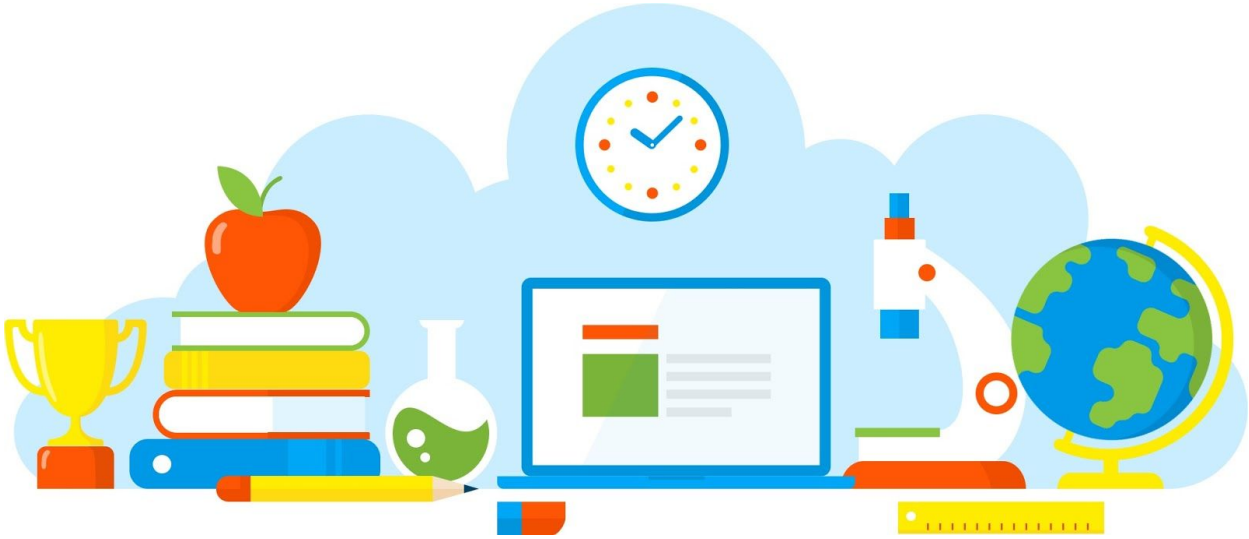


Rapport de Projet Système

Tesh, an Awesome Shell For an Incredible World!

Jean Calvet & Clément Martinez
2017

Introduction
I - Déroulement du projet
II - Découpage de l'application
III - Déroulement du développement
IV - Outils
V - Démonstration
VI - Répartition du temps
VII - Conclusions personnelles

Introduction

Rappelons tout d'abord les objectifs du module de rs et du projet en lui même :

"L'objectif du module de Système est d'apprendre à devenir un utilisateur avancé du système et maîtriser la programmation système. Il ne s'agit pas de comprendre comment le système fonctionne (ce qui constitue l'objectif du module de RSA), mais plutôt comment tirer le maximum du système."

- *https://members.loria.fr/lnussbaum/rs.html*

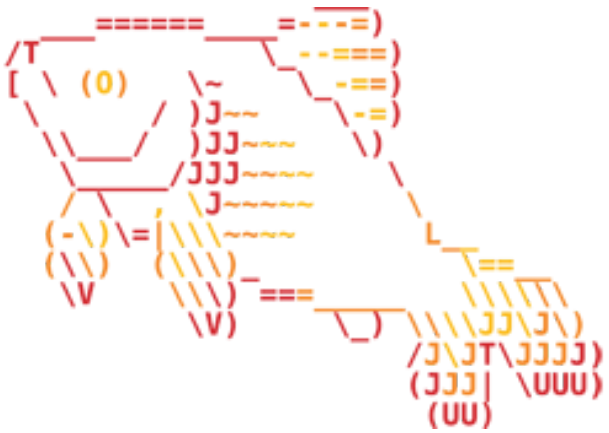
"L'objectif de ce projet est de réaliser un shell (comme bash, tcsh, zsh ou dash). D'un point de vue pédagogique, il permettra de revoir (et de mettre en pratique) les différents concepts vus pendant la partie système du module de RS (manipulation de processus et de fichiers avec l'API POSIX, notamment)."

- *https://members.loria.fr/lnussbaum/RS/projet-1718-tesh.pdf*

Le but était donc simple : mettre la main à la pâte, pour voir en profondeur ce qu'avait à proposer un shell et s'amuser à décortiquer des fonctionnalités avancées et parfois largement sous utilisées.

I - Déroulement du projet

Le projet a donc commencé pendant la première période des vacances de la deuxième année de TELECOM Nancy. Notre groupe ayant la chance de compter dans ses rangs un contributeur du shell “fish”, nous avons pu avancer rapidement sur les questions de conception générale et se tourner vers l’implémentation. Nous avons choisi de découper notre projet en classes comme nous l’aurions fait dans un langage orienté objet et de développer pour le premier tiers du projet uniquement en peer-programming, ou plutôt en coaching intensif.



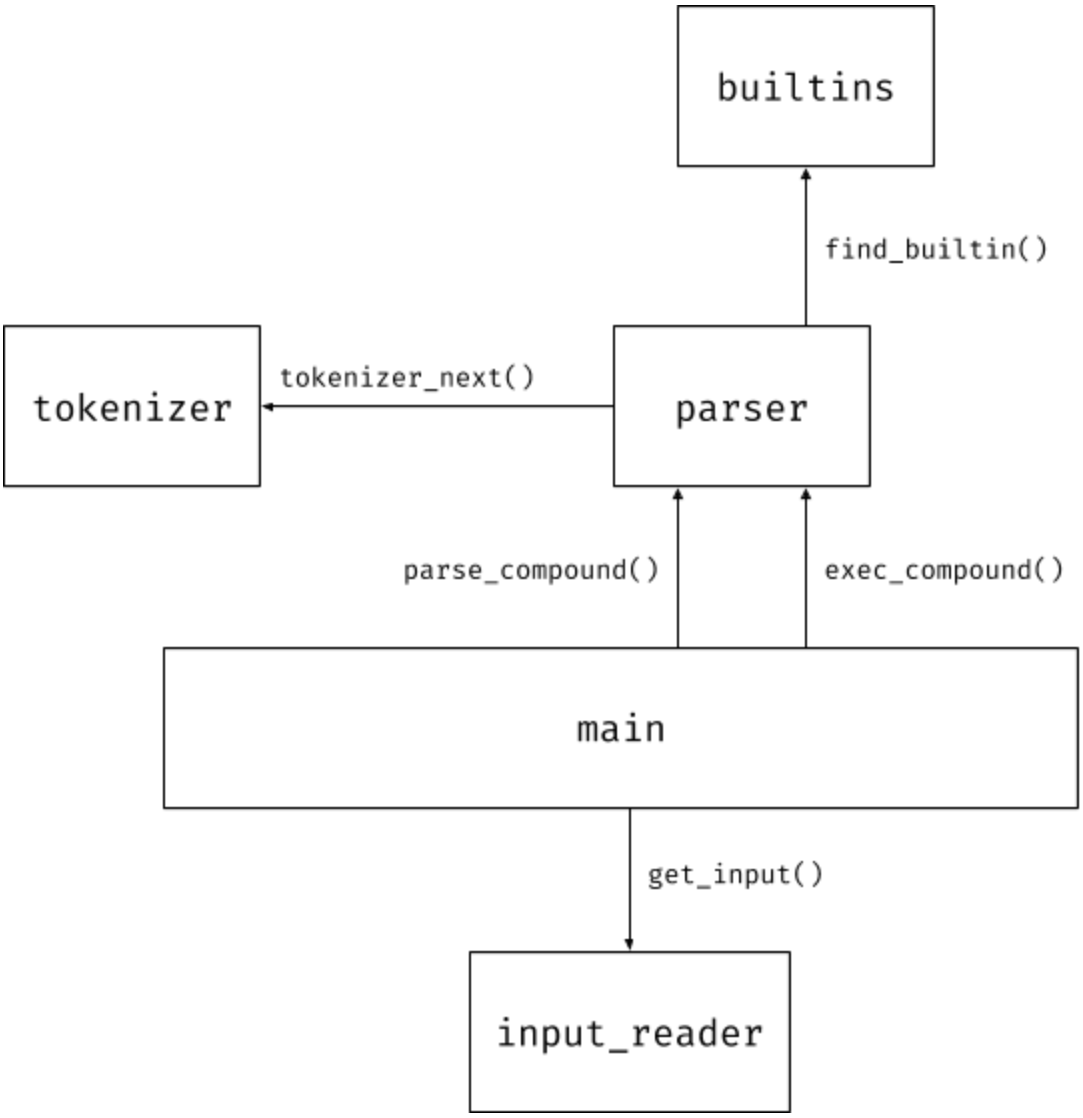
II - Découpage de l'application

Suivant les notions vues en module de traduction, nous avons choisi de découper le programme en 2 grandes parties :

- l'analyseur lexical, notre classe `tokenizer`,
- l'analyseur sémantique, notre classe `parser`.

En plus de ces deux grandes parties, nous utilisons d'autres objets, que voici :

- `vector` : outil principal permettant d'allouer dynamiquement des tableaux d'objets et de gérer la mémoire sans avoir à se soucier de l'allocation, il permet une flexibilité appréciable sans dégradation notable des performances. C'est un outil que Clément avait réalisé pour le projet de C de l'an dernier.
- `tools` & `test` : comme `vector`, ce sont des objets qui faisaient partie de la "boite à outils" de Clément et qui nous ont permis de gagner du temps dans nos différentes tâches. `tools` permet de compter les éléments d'un array de type quelconque et `test` de réaliser des tests unitaires simples en C, un peu comme JUnit (bien moins poussé, mais très utile pour notre utilisation).
- `get_input` : en absence de l'option readline, c'est l'objet qui nous permet de lire l'entrée standard (ou tout autre fichier) et charger caractère par caractère dans un buffer pour l'analyse. Il utilise le mode "cooked" du terminal pour récupérer une entrée.
- `built_in` : permet de centraliser le développement des builtins, de manière à pouvoir en ajouter facilement.



La fonction main se charge de faire le lien entre le fichier d'entrée (soit l'entrée du terminal, soit un fichier) et notre parseur qui va analyser le texte entré et l'exécuter.

Pour les trois premiers objets, nous avons aussi choisi de développer des classes de tests pour pouvoir avoir un retour de l'avancé du projet.

En effet, hormis les objets de la boite à outils qui existaient avant le projet, nous avons développé les objets dans l'ordre décrit plus haut. Ainsi, nous n'avons pas eu d'interaction avec le tesh en lui-même pendant la majeur partie du projet.

Une fois la classe `get_input` terminée, nous avons décidé de nous tourner vers des tests plus semblable à ceux effectués par les tests blancs, à savoir comparer la sortie de tesh et de bash sur un même fichier de commande d'entrée.

III - Déroulement du développement

Même si l'approche que nous avons suivi était maîtrisée, elle était un peu frustrante puisque nous pouvions voir nos camarades interagir avec leur tesh et de nouvelles fonctionnalités chaque semaine, alors que le nôtre a mis plus de temps avant de pouvoir être utilisable. Cependant, la réflexion préalable venant de l'expérience de Clément dans le domaine du scripting shell a fait que notre application, une fois les briques correctement implémentées, se sont emboîtées sans problème. On peut d'ailleurs le voir à la différence de test blanc réalisé par notre projet : nous sommes passés de 1 seul OK (la compilation) à quasiment 70% sur deux tests blancs successifs.

Durant ce projet, nous avons fait principalement du peer-programming. De plus, chaque modification de code était soumise à une revue avec pull request sur GitLab. Cela nous a permis non seulement de suivre l'avancement du code de l'un par rapport à l'autre, mais aussi de proposer des améliorations.

Nous ne nous sommes pas lancés dans l'aventure Flex/Bison (même si nous avons suivit de loin cette aventure de M. Carle & M. Aghenda, par curiosité scientifique et pour débattre des avantages et inconvénients de cette réalisation), nous avons pris plaisir à remplir une à une toutes les exigences fonctionnelles de ce projet, et à aller un peu plus loin.

IV - Outils

Dès le début, nous nous sommes mis d'accord sur le formatage du code qui est imposé par une configuration de `uncrustify`. Cela nous a permis de ne pas perdre de temps dans des considérations inutiles sur comment devrait être espacé le code.

Ensuite, nous avons revu la structure du dépôt. Les sources du programme ne sont pas à la racine du dépôt mais dans le répertoire `src`. Les binaires étant dans `build`.

Pour les tests, nous avons des binaires qui sont créés spécialement pour la tâche avec notre petit module C, `test`. Mais aussi des tests avec une entrée passée à l'exécutable dans `tests/`*`nom_du_test`*`_in` et la sortie attendue dans `tests/`*`nom_du_test`*`_out`. Ces tests sont lancés à chaque nouveau commit poussé sur GitLab grâce à l'intégration continue qui nous permet de voir rapidement si une fonctionnalité a été cassée. En particulier, avant de merger une PR (autant dire que c'est outil génial).

Comme éditeurs, chacun utilisait celui qu'il préférait. Jean utilisant Atom et Clément, Emacs (avec Spacemacs). Il n'y a eu aucun problème à ce niveau là grâce au formateur de code.

V - Démonstration

Ci-dessous un extrait de console issue de la branche de pre-release :

(à la 3ème ligne, l'autocomplétion de readline à été utilisé).

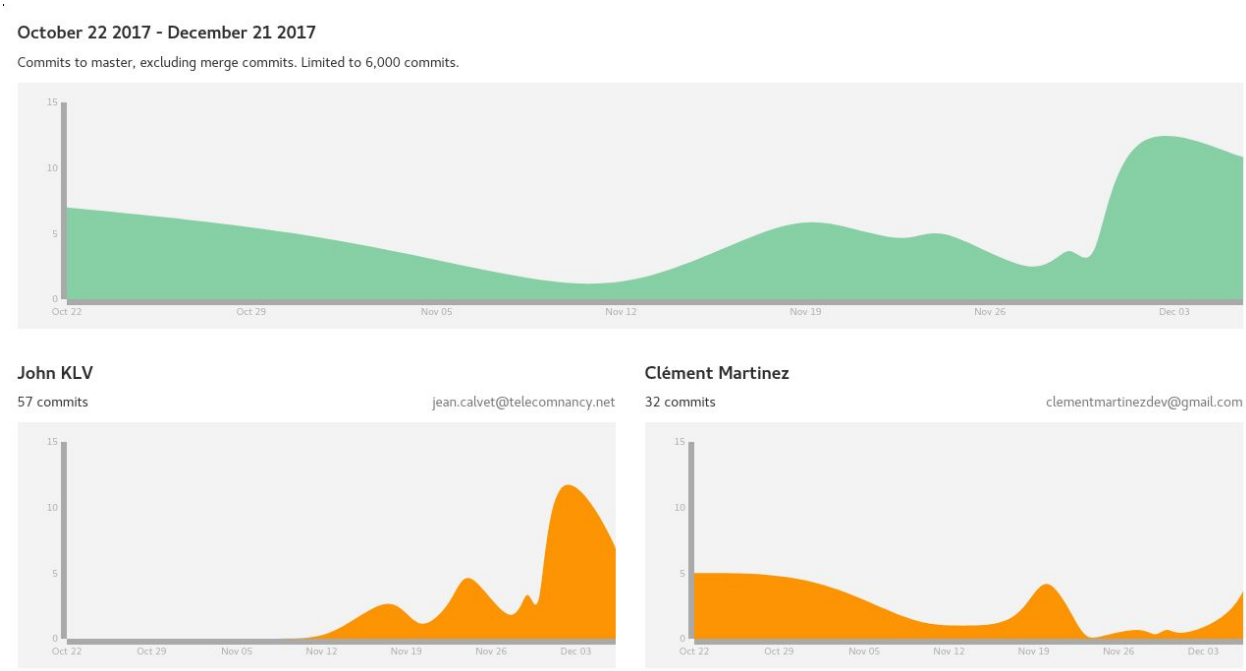
```
john@noah ~/G/rs2017-martinez-calvet (prepare_for_merge)> ./tesh -r
john@noah:/home/john/Github/rs2017-martinez-calvet$ cd ..
john@noah:/home/john/Github$ cd rs2017-martinez-calvet/
john@noah:/home/john/Github/rs2017-martinez-calvet$ ls
AUTHORS  build  built_in.o  doc  format.cfg  format.sh  Makefile  parser.o  README.md  src  tesh
tesh.o  tester.sh  tests  test_tokenizer.o  tokenizer.o
john@noah:/home/john/Github/rs2017-martinez-calvet$ echo "hahahahaahah \"la banana\" & les
minions en live !! "&
[3240]
john@noah:/home/john/Github/rs2017-martinez-calvet$ hahahahaahah "la banana" & les minions en
live !!
fg
[3240->0]
john@noah:/home/john/Github/rs2017-martinez-calvet$ false&&echo a &&echo b &&echo c &&echo
d||echo "Left Associativity of && & || respected"
Left Associativity of && & || respected
john@noah:/home/john/Github/rs2017-martinez-calvet$ ./tester.sh
[PASSED] tests/test_and_or_1
[PASSED] tests/test_and_or_2
[PASSED] tests/test_basic
[PASSED] tests/test_pipe
[PASSED] tests/test_redir_1
[PASSED] tests/test_redir_2
[PASSED] tests/test_semi_colon
[PASSED] tests/test_two_commands

8 succeeded.
0 failed.
john@noah:/home/john/Github/rs2017-martinez-calvet$
```

VI - Répartition du temps

Ayant commencé dès le début mais n'ayant pas compté les heures de travail, voici plutôt un résumé de notre utilisation de gitlab :

- ~ 100 commit, de manière régulière sur toute la période



- Répartis sur plus de 15 branches, supprimés au fur et à mesure de leur intégration dans la branche `master`
- Utilisation pour les merge de l'interface de merge de gitlab