A REPORT ABOUT HOW WE WILL DEAL WITH THIS LEGENDARY PROJECT

42sh v0.1

$ cat <<- EOF > AUTHORS

audebe_r - Rémi AUDEBERT
eddequ_n - Nassim EDDEQUIOUAQ
hervot_p - Paul HERVOT
pietri_a - Antoine PIETRI
schild_a - Adrien SCHILDKNECHT
EOF

"It is easier to port a shell than a shell script."

— Larry Wall, creator of Perl

4 novembre - 1 décembre
2013

Table des matières

1 Le groupe

Rémi Audebert Chef de projet. J'attendais le 42sh avec impatience. J'aime ce projet car il est complet et nous permet de mettre en œuvre des outils et des méthodes qui sont utilisés en entreprise.

Antoine Pietri Enthousiaste. J'espère que ce projet va me permettre de mettre en pratique les méthodologies agiles enseignées dans le cours de conduite de projet. Je frémis déjà à l'idée d'utiliser notre intégration continue dans le cloud qui s'avère être un excellent indicateur de qualité.

Paul Hervot Responsable readline. N'étant pas très bon en programmation, je comptais profiter de ce premier semestre pour me mettre à la pratique de façon sérieuse. L'objectif a été atteint, et je compte tout mettre en œuvre dans ce projet pour constater mes progrès.

Adrien Schildknecht J'aime Unix, j'aime le C, je vais aimer ce projet. Le *green IT* étant une de mes préoccupations je serai intransigeant sur les boucles infinies faisant fondre la banquise. J'espère que ce projet deviendra un jour le digne remplaçant de powershell.

Nassim Eddequiouaq السلام عليكم بسم الله الرّحمن الرّحيم. J'espère que mon équipe me rejoindra sur ce point. Pour ce qui est du reste, je suis impatient de réaliser ce projet et d'apprendre beaucoup de choses.

"It's 5.50 a.m...Do you know where your stack pointer is ?"

Write in C, they said

2 Aperçu du project

2.1 42sh

42sh est un shell compatible POSIX écrit en C99.

2.2 Build system

Nous utiliserons CMake comme *build system*. Les raisons de ce choix sont les suivantes :

- Moderne
- Propre
- Configuration relativement facile

2.3 Suite de tests

La suite de tests sera écrite en python3.3. Elle étendra le module `unittest` de la bibliothèque standard pour l'adapter à nos besoins :

- charger des tests depuis des fichiers ;
- sélectionner la catégorie de tests à effectuer ;
- afficher les résultats des tests pour chaque catégorie.

```
1        42sh$ make check
2        cd check && ./run_tests.py
3
4        Category: basic
5
6        basic/00_hello ... ok
7        basic/01_hello ... ok
8
9        Success: 100%
10
11       Final results:
12       Success: 100%
```

Listing 1: Exemple de sortie de la suite de tests.

2.4 Communication

Les développeurs utiliseront IRC pour les discutions quotidiennes.

2.5 Intégration continue

Chaque commit déclenchera un système charger de vérifier l'absence de régressions.

- Compilation avec gcc et clang
- Suite de tests
- Analyse valgrind (fd + mémoire)

Le projet sera automatiquement compilé sur différents systèmes (variations d'architecture, système d'exploitation, libc, etc.). Puis testé par le système décrit précedemment.

Actuellement les systèmes suivants sont disponibles :

- Archlinux
- FreeBSD 9.1

Un bot IRC annoncera chaque commit et le résultat des tests.

3 Découpage du projet

Le projet peut être découpé en 5 grosses parties :

3.1 Interface utilisateur

Cette partie s'occupe des différents moyens d'interagir avec le shell. Elle doit être capable de lire des commandes depuis un fichier, un argument ou depuis un mode interactif. Celui-ci est le plus important car il interagit directement avec l'utilisateur, c'est pourquoi nous apporterons un soin tout particulier au confort d'utilisation (historique, déplacement, complétion, ...)

Afin de donner la possibilité à l'utilisateur de manipuler la ligne de commande, le module readline devra être capable d'interprêter les combinaisons de touches telles que « flèche gauche », « Alt-W », « Ctrl-W », etc, et de reproduire le comportement attendu (déplacement du curseur, effacement de caractères). Nous utiliserons pour cela les fonctions de l'API termcap et les terminfos.

Pour proposer des fonctionnalités avancées et pratiques pour l'utilisateur, le module readline devra communiquer avec les différentes parties du projet et avec l'environement dans lequel 42sh aura été appelé. L'auto complétion par exemple, dans sa forme basique, listera les fichiers disponibles dans un dossier pour proposer une completion de nom de fichier, mais elle pourrait par la suite communiquer avec le parseur pour connaître le prochain type de token attendu et proposer l'auto complétion des commandes ou des mot-clés. Elle pourrait aussi demander à l'exécution d'expandre le mot courant si le parseur le détecte comme étant un token valide à expandre.

3.2 Lexer, parser, AST

Le shell doit essayer de comprendre ce que veux faire l'utilisateur. En se basant sur une grammaire LL(1), le shell doit construire une suite d'actions logiques sous la forme d'un arbre (AST). Les entrées provenant d'un humain, il doit être assez robuste pour résister aux multiples erreurs : « never trust user input ».

3.3 Gestions des options du shell

Cette partie permet de définir les réactions du shell vis-à-vis de différentes actions. L'utilisateur peut donc changer le comportement du glob-

bing ou de l'extension à travers des arguments qui seront gérés ici.

3.4 Évaluation de l'AST

Il s'agit ici d'interpréter l'AST que le parser a produit pour effectuer les différentes actions (structures de contrôles, commandes, ...). Tout en gérant l'expansions des différentes différentes composantes (variables, globbing, ...).

3.5 Builtins, commandes, gestion des processus

Il faut coder certaines commandes du shell appellées « builtins ». Si l'utilisateur utilise une commande externe, cette partie s'assurera de trouver cette commande et de suivre son activité (code de retour, redirections, ...).

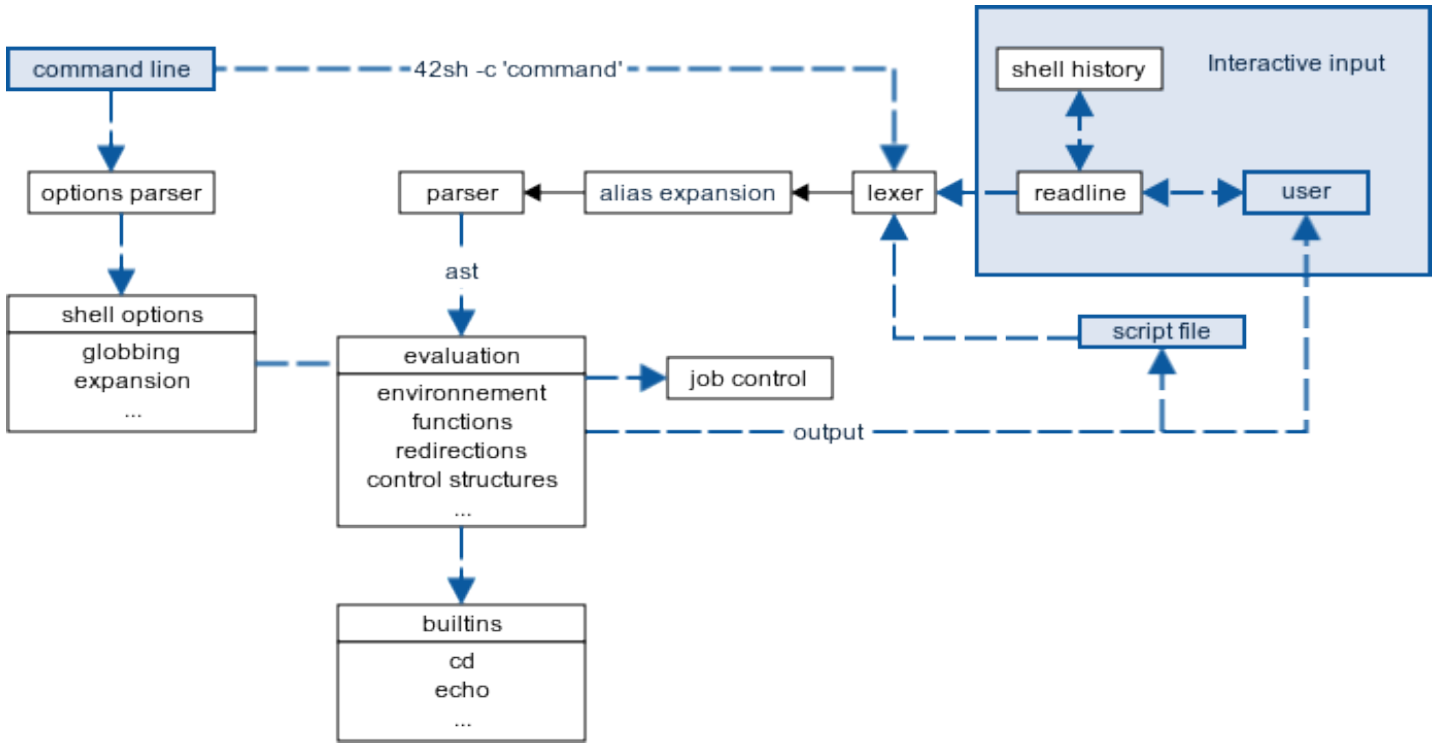
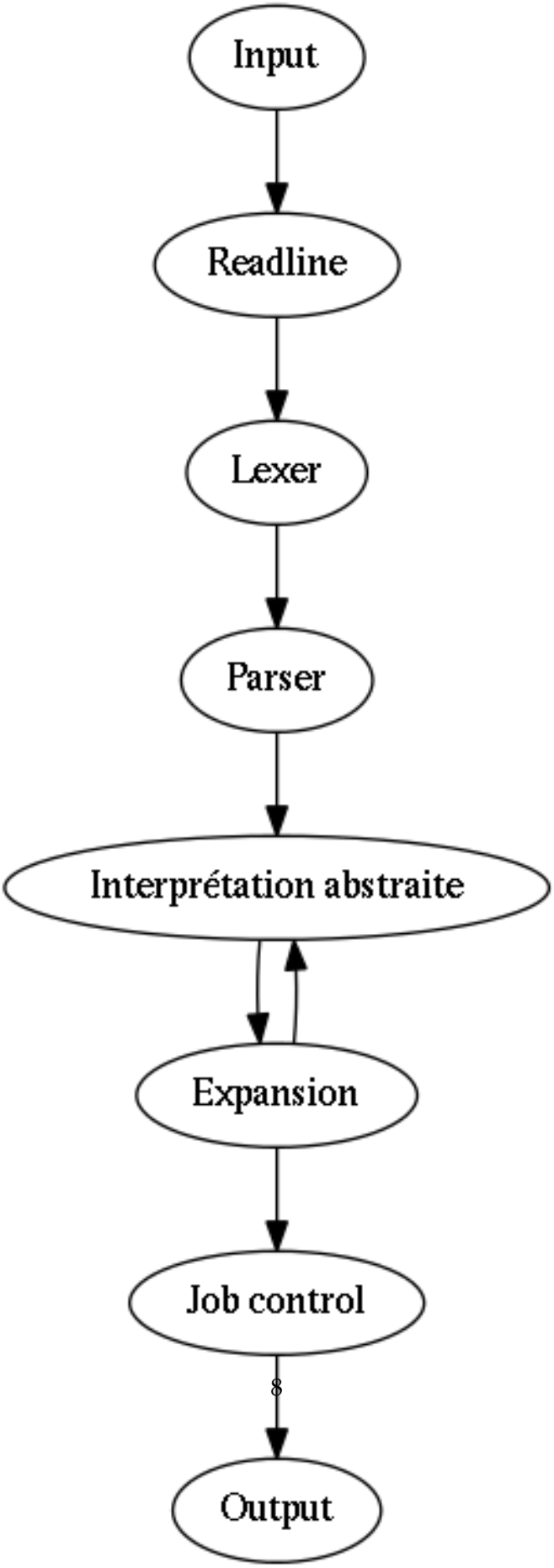


FIGURE 1 – Schéma logique du projet


Input
Readline
Lexer
Parser
Interprétation abstraite
Expansion
Job control
Output

4 Répartitions des tâches par personne

4.1 Organisation générale

Rémi Audebert a accepté d'être le chef de ce groupe pour le 42sh. Afin d'éviter de bloquer sur des parties importantes du projet nous avons, dans la mesure du possible, mis 2 personnes par tâches. L'avantage étant de pouvoir confronté deux avis différents sur l'implémentation d'un module. Il est également possible de faire du « pair programming ».

4.2 Affectation des tâches

4.2.1 `hervot_p` - Paul HERVOT

- Readline
  - lire l'entrée standard
  - déplacement et éditions de la ligne
- Auto-completion (nom et path)
- Gestion de l'historique de commandes

4.2.2 `audebe_r` - Rémi AUDEBERT

- Suite de tests et système de build
- Lexer et parser

4.2.3 `eddequ_n` - Nassim EDDEQUIOUAQ

- Gestion des redirections
- Evaluation de commandes simples
- Implémentation des built-ins
- Implémentation du contrôle de processus (jobs, fg, bg, wait)

4.2.4 `schild_a` - Adrien SCHILDKNECHT

- Lexer et parser
- Expansions des variables
- Getopt
  - Options « simples » avec « - »
  - Options du shell « +/- »
  - Stockage et accès

4.2.5 `pietri_a` - Antoine PIETRI

- Interprétation abstraite de l'AST
- Expansion des variables et aliases

5 Répartitions des tâches par projet

5.1 Suite de tests

- audebe_r - Rémi AUDEBERT

5.2 Interface utilisateur

- hervot_p - Paul HERVOT

5.3 Lexer et parser

- audebe_r - Rémi AUDEBERT
- schild_a - Adrien SCHILDKNECHT

5.4 Interprétation abstraite de l'AST

- pietri_a - Antoine PIETRI

5.5 Expansion des variables

- pietri_a - Antoine PIETRI
- schild_a - Adrien SCHILDKNECHT

5.6 Commandes, built-ins et contrôle des jobs

- eddequ_n - Nassim EDDEQUIOUAQ

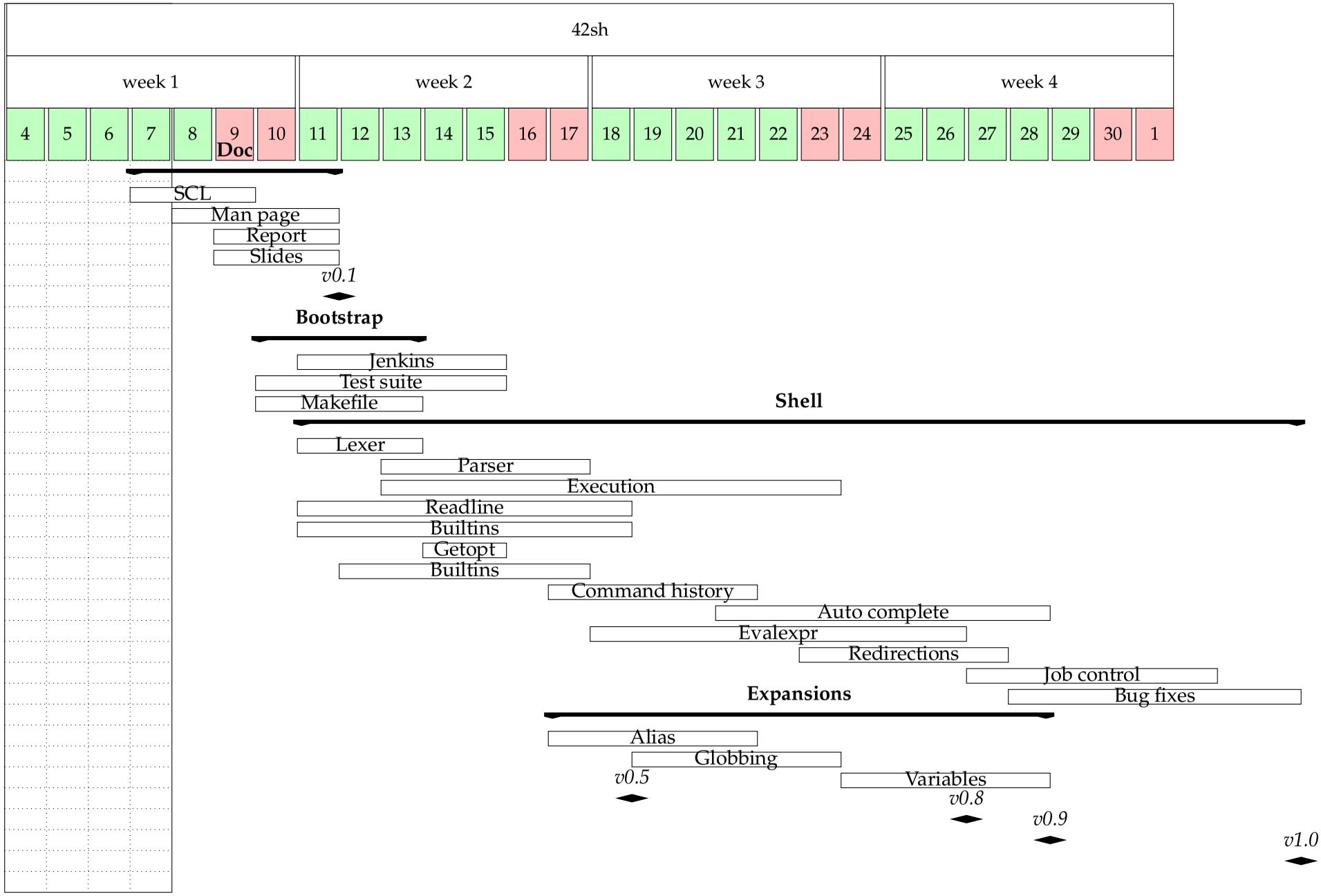


FIGURE 2 – Organisation