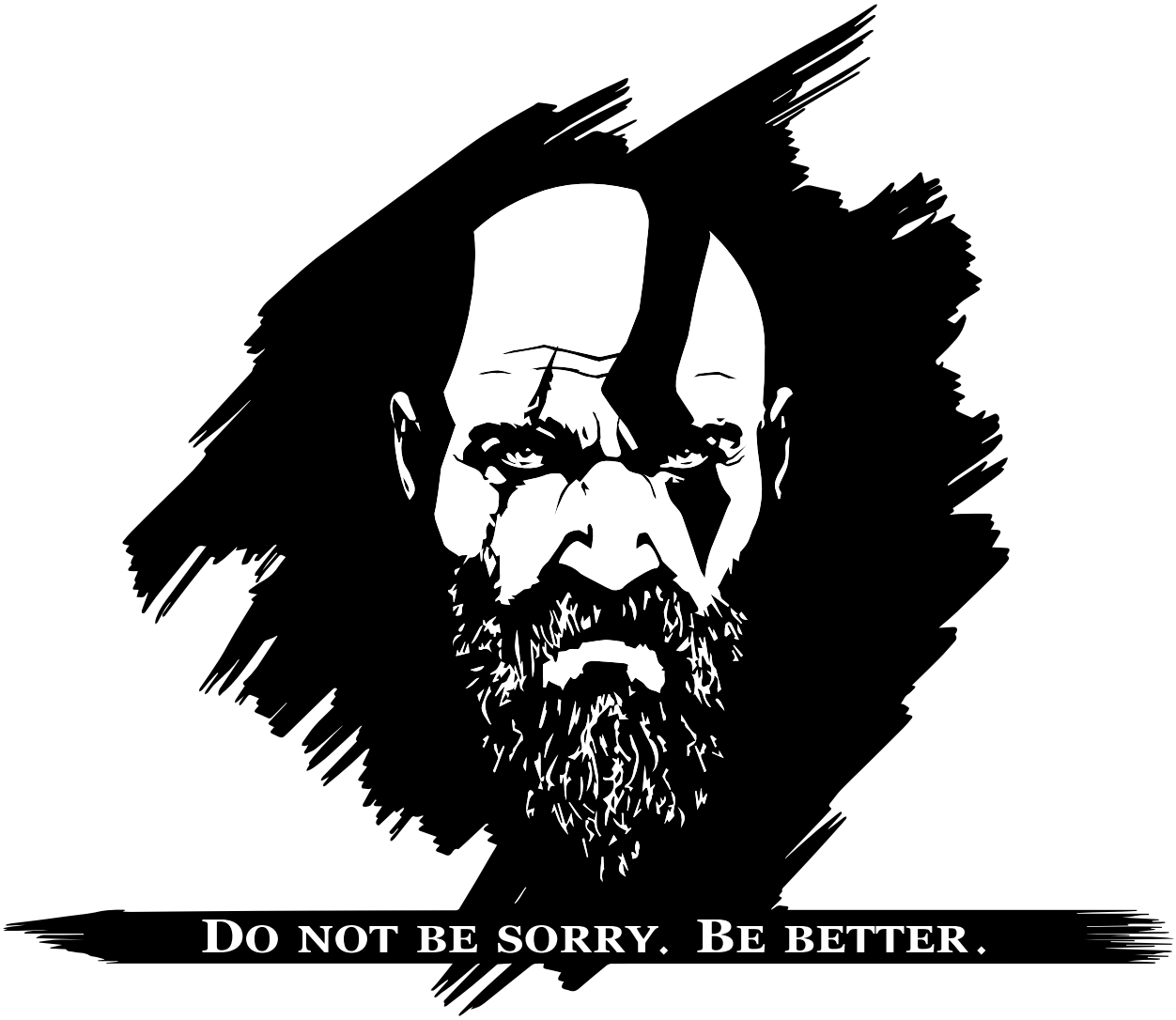
42sh — Subject

version **#1.7-6-gd57cef8**



Assistants C/Unix 2019 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

1 42sh **5**

2 Prologue **5**
- 2.1 The Ultimate Answer . . . 5
- 2.2 The reaction . . . 5

3 Preamble **5**
- 3.1 Why the 42sh? . . . 5
- 3.2 Instructions . . . 6
- 3.3 Technical constraints . . . 6
- 3.4 Shell Command Language . . . 7
- 3.5 Hand in . . . 7
- 3.6 Grading system . . . 7
- 3.7 Documentation . . . 8
- 3.8 Compilation . . . 8

4 Version 0.3 **8**
- 4.1 Manual . . . 8
- 4.2 Options parser . . . 9
- 4.3 Shell parser . . . 9
- 4.4 AST printer . . . 11
- 4.5 Autotools/CMake . . . 11
- 4.6 Tests suite . . . 12
  - 4.6.1 Test program . . . 12
  - 4.6.2 Advanced features . . . 12
- 4.7 Execution . . . 13
  - 4.7.1 Simple commands . . . 13
  - 4.7.2 "If" commands . . . 13
  - 4.7.3 "While" and "until" commands . . . 13
  - 4.7.4 "For" commands . . . 13

*https://intra.assistants.epita.fr

5 Version 0.5 **14**
5.1 Prompt . . . 14
5.2 Tests suite . . . 14
5.2.1 Tests format . . . 14
5.2.2 Options format . . . 15
5.2.3 Timeout management . . . 15
5.3 Execution . . . 15
5.3.1 Operators . . . 15
5.3.2 Pipes . . . 16
5.3.3 Functions . . . 16
5.3.4 “Case” commands . . . 16
5.4 Resource files . . . 16
5.5 History . . . 17

6 Version 0.8 **17**
6.1 Tilde expansion . . . 17
6.2 Built-in commands . . . 17
6.2.1 “exit” . . . 17
6.2.2 “cd” . . . 17
6.2.3 “shopt” . . . 18
6.2.4 “export” . . . 18
6.2.5 “alias” . . . 18
6.2.6 “unalias” . . . 19
6.2.7 “echo” . . . 19
6.2.8 “continue” . . . 19
6.2.9 “break” . . . 19
6.2.10 “source” . . . 19
6.2.11 “history” . . . 20
6.3 Advanced prompt . . . 20
6.4 Variables . . . 21
6.5 Quoting . . . 21
6.6 Path expansion . . . 22
6.7 Arithmetic expansion . . . 22
6.8 Readline . . . 23
6.9 Aliases . . . 23
6.10 Path expansion, challenge mode . . . 23

7 Version 0.9 **23**
7.1 Completion . . . 23
7.2 Command Substitution . . . 24
7.3 Job Control . . . 24

8 Version 1.0 **25**
8.1 Git sanity . . . 25

9 Bibliography **25**

10 Epilogue **26**
10.1 The search for the Ultimate Question . . . 26
10.2 Douglas Adams’ view . . . 26

Project data

Instructors:

- Loic Reyreaud <reyrea_l@assistants.epita.fr>
- Clement Galland <gallan_c@assistants.epita.fr>

Dedicated newsgroup: assistants.projets with [42SH]

Members per team: 4

1 42sh

Files to submit:

- `./doc/*`
- `./autogen.sh`
- `./CMakeLists.txt`
- `./src/*`
- `./tests/*`
- `./README`
- `./TODO`

Forbidden functions: glob(3), regexec(3), wordexp(3), system(3), popen(3), syscall(2)

Allowed libraries: readline, history

Makefile: The generated makefile should define at least the following targets:

- **all**: Produces the 42sh binary

Forbidden functions: You can use all the functions of the standard C library except:

- **Unclassified functions**: glob, regexec, wordexp, system, popen, syscall

2 Prologue

2.1 The Ultimate Answer

According to *The Hitchhiker's Guide to the Galaxy*, researchers from a pan-dimensional, hyper-intelligent race of beings constructed the second greatest computer in all of time and space, *Deep Thought*, to calculate the Ultimate Answer to Life, the Universe, and Everything. After seven and a half million years of pondering the question, *Deep Thought* provides the answer: **"forty-two"**.

2.2 The reaction

> "Forty-two! Is that all you've got to show for seven and a half million years work?" yelled Loonquawl. "I checked it very thoroughly," said the computer, "and that quite definitely is the answer. I think the problem, to be quite honest with you, is that you've never actually known what the question is."
>
> —Douglas Adams

3 Preamble

3.1 Why the 42sh?

The 42sh is a *big* project that will teach you more than you can now imagine. The first aspect is *human*, because four students will try to work together during about four weeks. You will discover the joys of

working into a group of motivated people... or not. Thus, you will (hopefully) acquire some wisdom, patience, self-mastery and become sociable.

With the different projects you have done since the beginning of your ING1, you have gained an incredible experience; don't forget to use every lesson you learnt so far in order to achieve a *great* 42sh.

The 42sh is also a mean to learn new languages and technologies. You will write your test suite with a high-level programming language, such as Python or Ruby, and your 42sh in C99. You will see how to write a manpage. You will use Doxygen and collaborative development tools (like Git) on a large-scale project. You will control a terminal cursor. You will also have to use Autotools or CMake as a build system.

After the 42sh, you will never use your shell like before, spending four weeks on a project that consists in writing a shell is the best way to become a crustacean expert. Beyond understanding how to speak to a shell, you will be able to *visualize how shells "think"*. You will read in the matrix!

3.2 Instructions

Your mission, should you decide to accept it (and you really should), is to implement a shell. To complete this mission you have formed a crew of four students and chosen a leader among you. Every administrative contact must be established through him (or her), and only him (or her). If this student deserts or dies, the whole crew must announce the new leader in the assistants' laboratory: the show must go on!

To succeed in your initiative you must follow *exactly* every rule given in this subject. Do not hesitate to read it twice or more. You must also frequently read the dedicated newsgroup, where your subject managers can give amendments about the subject, announce oral examinations and more.

Your 42sh must follow the directives given in the SCL document, but the reference program is Bash with its *posix* option. Note, however, that you have to follow this subject *even if* it conflicts with the SCL or `bash --posix`.

A decent project always comes with a reasonable test suite. Yours must be written in an advanced scripting language. You also have to comment your code with Doxygen and document your 42sh in a manual page.

Your program must run correctly, i.e. without any memory leak or segmentation fault. If you fail to respect one of these points, your crew's final mark will decrease.

3.3 Technical constraints

- You must respect the "C99" standard of the C programming language.
- You must respect the EPITA coding style.
- Your project must compile and run correctly on Arch Linux.
- Your 42sh must free all its allocated memory[1] and close all its opened file descriptors.
- You must not strip your program, nor link it statically.
- You must use the Git versioning tool and fill a changelog correctly.
- You must comment your code for Doxygen.

[1] This will be checked with Valgrind.

- You must not include any code *not* written by one of your team members.
- Your git repository must not contain extraneous authors, that is to say `git shortlog -s` must display the name of each team member once.
- Your *42sh* binary **must** be generated in the same directory as the *Makefile*.

3.4 Shell Command Language

SUS (*Single Unix Specification*) is a set of standards that describe how a Unix operating system should be. The SUS is cut into four main parts:

- ***Base definition* (XBD)** which lists definitions and conventions used in the specifications, and C header files which must be provided by compliant systems
- ***Shell and utilities* (XCU)** which lists utilities and a description of the shell
- ***System interface* (XSH)** which lists available C system calls
- ***Rationale* (XRAT)** which explains the standard

XCU chapter 2 describes the *Shell Command Language* (SCL for short). This is a reference about shells: anybody who wants to write a shell should read it first. So, before starting to write any line of your 42sh, you must read **all** this chapter!

To assess your knowledge about the SCL, you will have an **exam**. We will never provide any correction about this MCQ, simply because all answers are in XCU chapter 2.

3.5 Hand in

You should hand in four versions of your 42sh. For each of them, there are mandatory and advanced modules. To validate a version, all mandatory modules must be implemented. Of course, in order to avoid any attempt of cheating, we will set a threshold to determine if a module is implemented or not. Implementing only mandatory modules **will not** guarantee you a good mark, you have to implement some of the advanced features as well.

For each hand in, every module under the current version will be checked again, so when you hand in the next-to-last version of your project, all modules will be checked. You should consider this version as a “release candidate”.

The last hand in is the most important, because all your modules will be validated on this version. **Don’t fail this delivery!** See the “Grading system” section for more information.

3.6 Grading system

Your grade will be attributed according to a savant recipe and you will not know it beforehand; but you should know how to get points.

Points come from:

- MCQ about *Shell Command Language* (these marks are different for each student)
- oral examinations (these marks are different for each student)
- modules validation (these marks are common to the entire crew)

- time validation: if a feature is validated for the first time in the first version that requires it, you win the *bonus time* (these marks are common to the entire crew)
- negative points: these points are negative and are distributed to students or groups that don't respect the rules or are not serious.

42sh is a very important project, therefore it has got a strong coefficient. **Don't cheat, don't play with fire!**

3.7 Documentation

The `doc/` repository located in your project's Git root must contain documentation about your 42sh: manual page and Doxygen files should be in this directory.

3.8 Compilation

Your 42sh will be compiled using the following commands:

- if you use `autotools`:
  - `./autogen.sh`
  - `./configure`
  - `make`
- if you use `cmake`:
  - `mkdir build`
  - `cd build`
  - `cmake ..`
  - `make`

All files that can be generated by `autotools` or `cmake` are considered as trash files.

4 Version 0.3

4.1 Manual

To validate this module, the crew must write a manual page (in English) for the 42sh. This manual should present all the features of your current 42sh. It must at least contain the `NAME`, `SYNOPSIS`, `DESCRIPTION` and `AUTHORS` sections. You can have more sections and this will be appreciated during the evaluation. This page must be stored in the file `./doc/42sh.1`, where `./` is the project root directory.

Validation: mandatory

Evaluation: human

Level: easy

4.2 Options parser

To validate this module, your 42sh must be able to handle the command line it was invoked with correctly. If an invalid option is detected, you must print an error message and a usage message, both on the error output, and return a proper code.

The command line syntax is: `42sh [GNU long option] [option] [file]`

Your 42sh must accept at least the following options:

- `-c <command>`: follow the `bash` behavior for this option.
- `[-+]O`: follow the `bash` behavior for this option.

  Only `shopt` variables that are required by 42sh must be accepted.
- `--norc`: this option deactivates the resource reader.
- `--ast-print`: this option activates the AST printer.
- `--version`: this option prints the current 42sh version on the standard output and exits.

  Use the following syntax: `Version x.x\n`.

Example:

```
42sh$ ./42sh --version
Version 1.0
```

If an option is activated for a mode you don't have implemented yet, simply check the syntax and act accordingly. Don't forget to update your option parser later though!

Validation: mandatory

Evaluation: computer

Level: easy

4.3 Shell parser

To validate this module, your 42sh must be able to parse the shell language and build a corresponding AST.

The SCL gives a LALR grammar, but it's difficult to implement a LALR parser by hand. For this reason, we provide you with a **LL grammar** for your 42sh. You must follow this grammar, *not* the behavior of `bash` (which is much laxer).

```
input:      list '\n'
        |   list EOF
        |   '\n'
        |   EOF

list:       and_or ((';'|'&') and_or)* [';'|'&']

and_or:     pipeline (('&&'|'||') ('\n')* pipeline)*
```

(continues on next page)

(continued from previous page)

```
pipeline:   ['!'] command ('|' ('\n')* command)*

command:    simple_command
        |   shell_command  (redirection)*
        |   funcdec (redirection)*

simple_command: (prefix)+
        |   (prefix)* (element)+

shell_command:  '{' compound_list '}'
        |   '(' compound_list ')'
        |   rule_for
        |   rule_while
        |   rule_until
        |   rule_case
        |   rule_if

funcdec:    ['function'] WORD '(' ')' ('\n')* shell_command

redirection:    [IONUMBER] '>' WORD
        |   [IONUMBER] '<' WORD
        |   [IONUMBER] '>>' WORD
        |   [IONUMBER] '<<' HEREDOC
        |   [IONUMBER] '<<-' HEREDOC
        |   [IONUMBER] '>&' WORD
        |   [IONUMBER] '<&' WORD
        |   [IONUMBER] '>|' WORD
        |   [IONUMBER] '<>' WORD

prefix:     ASSIGMENT_WORD
        |   redirection

element:    WORD
        |   redirection

compound_list:
  ('\n')* and_or ((';'|'&'|'\n') ('\n')* and_or)* [('&'|';'|'\n') ('\n')*]

rule_for:
  'for' WORD ([';']|[('\n')* 'in' (WORD)* (';'|'\n')]) ('\n')* do_group

rule_while: 'while' compound_list do_group

rule_until: 'until' compound_list do_group

rule_case:  'case' WORD ('\n')* 'in' ('\n')* [case_clause] 'esac'
```

(continues on next page)

(continued from previous page)

```
rule_if:    'if' compound_list 'then' compound_list [else_clause] 'fi'

else_clause:    'else' compound_list
        |   'elif' compound_list 'then' compound_list [else_clause]

do_group:   'do' compound_list 'done'

case_clause:    case_item (';;' ('\n')* case_item)* [;;] ('\n')*

case_item:  ['('] WORD ('|' WORD)* ')' ('\n')*  [ compound_list ]
```

Don't forget to correctly parse comments!

Validation: mandatory

Evaluation: human

Level: easy

4.4 AST printer

To validate this module, your 42sh must be able to print a DOT file representing the **AST** generated by the parser.

You are free to choose how your 42sh will generate this DOT file, but you must be able to show an image of your AST during your oral examination. During this examination, we will use the `dot` program to create a PNG (*Portable Network Graphics*) picture of your AST. Furthermore, you will be asked about your AST design.

This module is activated by the `--ast-print` option on the command line or set by `shopt`. By default, this option is deactivated.

This module is a mandatory module and very important to help you debug other modules of your 42sh. We might refuse to help you if this module is not implemented.

Validation: mandatory

Evaluation: human

Level: easy

4.5 Autotools/CMake

To validate this module, your crew must use the Autotools or CMake build system.

Validation: mandatory

Evaluation: human

Level: easy

4.6 Tests suite

You should realize how important it is to *test* in this kind of project. Therefore you will have to provide us with a test suite which must:

- have all the needed functions to really test your program
- prevent you from *regressions*
- give you the possibility to follow the progress you made

4.6.1 Test program

To validate this module, you must write a test program in Python or Ruby. Other script languages, such as Perl, can be allowed if asked for by news.

Display is done in the terminal, the format is free but part of the grade depends on its quality. Nevertheless you must follow several rules:

- You must issue only one line per test.
- We must be able to clearly understand the result of the test. This means that at least failure or success must be printed. But it is desirable to display the cause of a failure: standard output, error output, exit value or those three reasons.
- Remember that tests are grouped into categories. Therefore, we must be able to clearly identify the category of the running test. Before testing a category, you must display its name followed by a blank line; after the tests, a blank line and the result of the category (with a percentage or else, see option `-n`).
- You must display the global result of your tests after the execution.

Validation: mandatory

Evaluation: human

Level: easy

4.6.2 Advanced features

Your test program is evaluated during your defenses. You can come with any feature you can imagine. Here is a short list of what you can try to add to your test program:

- colors (red for failures, green for success, etc.)
- tests generation
- cool output file (like graphs)

But remember that your test program is not your priority during this project. Do not take too much time working on it. It's better to have a good 42sh with a decent test suite than to have an absolutely amazing test program with a poor 42sh.

Validation: advanced

Evaluation: human

4.7 Execution

After that, your 42sh will be able to execute some of the shell grammar. The following modules are only there for marking; at the end, all the grammar must be correctly executed. For the version 0.3, you don't have to implement *expansion* during execution. These modules are the **most important** of your project. **Implement them seriously!**

4.7.1 Simple commands

To validate this module, your 42sh must be able to execute commands. Those commands can be prefixed, redirected and, if they are builtins, you must not *fork* if you have implemented them! You also have to manage variable assignments at this level.

Validation: mandatory

Evaluation: computer

Level: hard

4.7.2 "If" commands

To validate this module, your 42sh must be able to execute `if` statements. Of course, redirections must work correctly with this command.

Validation: mandatory

Evaluation: computer

Level: easy

4.7.3 "While" and "until" commands

To validate this module, your 42sh must be able to execute `while` and `until` loops. Of course, redirections must work correctly with these commands.

Validation: mandatory

Evaluation: computer

Level: easy

4.7.4 "For" commands

To validate this module, your 42sh must be able to execute `for` loops. Of course, redirections must work correctly with this command.

Validation: mandatory

Evaluation: computer

Level: medium

5 Version 0.5

5.1 Prompt

To validate this module, your 42sh must show a prompt if the shell is in interactive mode. You have to implement only PS1 and PS2 prompts as they are described in the SCL. For the moment, the PS1 prompt can be the basename[2] of your program followed by "$" and a space, and the PS2 prompt can be ">" and a space. This module will only be tested if it is possible to launch your 42sh in interactive mode.

Validation: advanced

Evaluation: human

Level: easy

5.2 Tests suite

5.2.1 Tests format

To validate this module, your test format must respect the following rules. The tests are distributed into categories, with the name of the category matching the name of the directory that contains the tests. A category represents a whole set of tests aiming at evaluating a particular part of the 42sh.

For instance:

```
42sh$ ls -la
total 20K
drwx------   5 login_x epita 4,0K 2016-10-18 15:01 ./
drwx------  24 login_x epita 4,0K 2016-10-18 15:00 ../
drwx------   2 login_x epita 4,0K 2016-10-18 15:01 echo/
drwx------   2 login_x epita 4,0K 2016-10-18 15:01 globbing/
drwx------   2 login_x epita 4,0K 2016-10-18 15:01 pipes/
```

The tests format is left free in order to develop your creativity and make you think by yourself. However, in order to ensure a minimum of cohesion and not to lose too much time during the correction, you must have only one file per test. This file must at least contain:

- a test description
- the input
- the standard output which can be the expected text or a *diff* with something from `bash`
- the error output, just like the standard output, can be a text or a *diff* with `bash`[3]

Validation: mandatory

Evaluation: human

Level: easy

[2] See `basename` manual, section 3, for more information.

[3] Bash prints its basename when an error happens. To *diff* your program with Bash you must rename it into `bash`. Of course you must have correctly implemented your error messages.

5.2.2 Options format

To validate this module, your test program must be able to understand the following options:

- `-l` and `--list`: Display the list of test categories.
- `-c <category>` and `--category <category>`: Execute the test suite on the categories passed in argument *only*.
- `-s` and `--sanity` Execute the test suite with sanity checks enabled: e.g. valgrind. Any reported error must mark the test as failed.

Your test suite must be able to check against `bash --posix` in order to verify that your tests are valid.

These options are only available when you launch your testsuite manually (i.e., without `make check`).

Validation: mandatory

Evaluation: human

Level: medium

5.2.3 Timeout management

To validate this module, your test program must be able to implement a timeout during the testing process. Sometimes a program gets stuck in an infinite loop or is blocked in a child process. In order to avoid to block the testing process, it is interesting to be able to manage a *timeout*. Thus the test is regarded as failed and the test program continues its execution.

You must implement the `-t <time>` and `--timeout <time>` options which set `time` as a general timeout time (in seconds).

You should be able to specify a timeout in your test format in order to be able to set a different time for each test.

Validation: advanced

Evaluation: human

Level: medium

5.3 Execution

After that, your 42sh will be able to execute **all** the shell grammar. The following modules are only there for marking; at the end, all the grammar must be correctly executed. For the version 0.5 (and only this one), you don't have to implement *expansion* during execution. These modules are the **most important** of your project. **Implement them seriously!**

5.3.1 Operators

To validate this module, your 42sh must be able to execute properly the operators "`&&`", "`||`", "`;`" and "`&`". You don't have (yet) to run in background commands that end with "`&`". You can assume that "`&`" has the same behavior than "`;`".

Validation: mandatory

Evaluation: computer

Level: easy

5.3.2 Pipes

To validate this module, your 42sh must be able to execute *piped commands*. Pay particular attention to **zombies** or you could win some *malus* points!

Validation: mandatory

Evaluation: computer

Level: hard

5.3.3 Functions

To validate this module, your 42sh must be able to register and execute functions. This includes, of course, redirections to functions and argument transmission.

Validation: mandatory

Evaluation: computer

Level: hard

5.3.4 "Case" commands

To validate this module, your 42sh must be able to execute `case` statements. Of course, redirections must work correctly with this command.

Validation: mandatory

Evaluation: computer

Level: hard

5.4 Resource files

To validate this module, your 42sh must load the `/etc/42shrc` and `~/.42shrc` files (in this order) when it starts.

This behaviour must be deactivated by the `--norc` option. Don't forget to update your option parser!

Validation: advanced

Evaluation: computer

Level: medium

5.5 History

To validate this module, your 42sh must be able to write an history file named `~/.42sh_history`. This is not only a matter of copying a read line into a file: check what Bash writes in its own history file in order to provide the same behavior. This module will only be tested if it is possible to launch your 42sh in interactive mode.

Validation: advanced

Evaluation: human

Level: medium

6 Version 0.8

6.1 Tilde expansion

To validate this module, your 42sh must be able to expand "~", "~-" and "~+" as described in the SCL and in the Bash manual.

Validation: mandatory

Evaluation: computer

Level: easy

6.2 Built-in commands

6.2.1 "exit"

To validate this module, you must implement the `exit` builtin and all its options. For more information about it, please refer to the Bash manual.

Validation: mandatory

Evaluation: computer

Level: easy

6.2.2 "cd"

To validate this module, you must implement the `cd` builtin. You neither have to implement the `-L` and `-P` options, nor to follow the required behavior for the `CDPATH` variable. However, you have to implement all other behaviors.

For more information about `cd`, please refer to the Bash manual.

Validation: mandatory

Evaluation: computer

Level: easy

6.2.3 “shopt”

To validate this module, you must implement the `shopt` builtin with its `-s`, `-u` and `-q` options.

The following variables must be handled:

- `ast_print`
- `dotglob`
- `expand_aliases`
- `extglob`
- `nocaseglob`
- `nullglob`
- `sourcepath`
- `xpg_echo`

Don’t forget: the `[+-]O` option must correctly activate `shopt` variables.

For more information about `shopt`, please refer to the Bash manual.

Validation: mandatory

Evaluation: computer

Level: easy

6.2.4 “export”

To validate this module, you must implement the `export` builtin with its `-n` and `-p` options. You should use the `environ` global variable.

For more information about `export`, please refer to the Bash manual.

Validation: mandatory

Evaluation: computer

Level: easy

6.2.5 “alias”

To validate this module, you must implement the `alias` builtin with all its options. Note that this module is independent of the *aliases* module. You can implement it even if your *aliases* module is broken.

For more information about `alias`, please refer to the Bash manual.

Validation: mandatory

Evaluation: computer

Level: easy

6.2.6 “unalias”

To validate this module, you must implement the `unalias` builtin with all its options. Again, note that this module is independent of the *aliases* module.

For more information about `unalias`, please refer to the Bash manual.

Validation: mandatory

Evaluation: computer

Level: easy

6.2.7 “echo”

To validate this module, you must implement the `echo` builtin with all its options.

For more information about `echo`, please refer to the Bash manual.

Validation: mandatory

Evaluation: computer

Level: medium

6.2.8 “continue”

To validate this module, you must implement the `continue` builtin with all its options.

For more information about `continue`, please refer to the Bash manual.

Validation: mandatory

Evaluation: computer

Level: medium

6.2.9 “break”

To validate this module, you must implement the `break` builtin with all its options.

For more information about `break`, please refer to the Bash manual.

Validation: mandatory

Evaluation: computer

Level: medium

6.2.10 “source”

To validate this module, you must implement the `source` and the `.` builtins.

For more information about them, please refer to the Bash manual.

Validation: mandatory

Evaluation: computer

Level: medium

6.2.11 "history"

To validate this module, you must implement the `history` builtin and its `-c` and `-r` options.

For more information about `history`, please refer to the Bash manual.

Validation: advanced

Evaluation: human

Level: medium

6.3 Advanced prompt

To validate this module, your 42sh must show a prompt when the shell is in interactive mode. You have to implement only PS1 and PS2 prompt as described in the SCL.

Your prompt must handle the following substitutions:

- `\a`
- `\d`
- `\D`
- `\e`
- `\h`
- `\H`
- `\n`
- `\r`
- `\s`
- `\u`
- `\w`
- `\W`
- `\$`
- `\nnn`
- `\\`
- `\[`
- `\]`

For the `\w` and `\W` substitutions, you must implement the same behavior for the paths as in the `cd` command. This module will only be tested if it is possible to launch your 42sh in interactive mode.

Validation: advanced

Evaluation: human

Level: medium

6.4 Variables

To validate this module, your 42sh must handle **variable substitution**. For the sake of simplicity there are *no read-only* variables and you can consider that valid variables are of the form `$name` and `${name}`.

The following special variables must be properly expanded:

- `$@`
- `$*`
- `$?`
- `$$`
- `$1` ... `$n`
- `$#`
- `$RANDOM`
- `$UID`
- `$OLDPWD`
- `$SHELLOPTS`
- `$IFS`

The `$SHELLOPTS` is colon-separated list of enabled `shopt` options. For others, have a look at the Bash manual.

Validation: mandatory

Evaluation: computer

Level: medium

6.5 Quoting

To validate this module, your 42sh must be able to interpret correctly the “quoting” as described in the SCL.

There are three types of quoting:

- escape character (`\`)
- single quotes (`'`)
- double quotes (`"`)

Validation: mandatory

Evaluation: computer

Level: medium

6.6 Path expansion

To validate this module, your 42sh must be able to expand properly the following special parameters (also known as *metacharacters* and *wildcards*):

- `*`
- `?`
- `[]`, with the special meaning of the "–" and "`!`" characters

You should also handle all globbing character classes (`[:alnum:]` etc.)

```
42sh$ find
.
./dir1
./dir1/dir11
./dir1/dir11/tota
./dir1/dir11/toti
./dir1/dir12
./dir1/dir12/toti
./dir1/dir12/totu
./dir2
./dir2/dir22
./dir2/dir22/toti
./dir2/dir22/totu
./dir2/dir21
./dir2/dir21/tota
./dir2/dir21/toti
42sh$ echo */*
dir1/dir11 dir1/dir12 dir2/dir21 dir2/dir22
42sh$ echo */*/////
dir1/dir11/ dir1/dir12/ dir2/dir21/ dir2/dir22/
42sh$ echo */*/*[ai]
dir1/dir11/tota dir1/dir11/toti dir1/dir12/toti
dir2/dir21/tota dir2/dir21/toti dir2/dir22/toti
42sh$
```

Validation: mandatory

Evaluation: computer

Level: medium

6.7 Arithmetic expansion

To validate this module, your 42sh must be able to expand arithmetic expressions, which are wrapped by `$(())`[4]. They can contain variables and the following operators: `-`, `+`, `*`, `/`, `**`, `&`, `|`, `^`, `&&`, `||`, `!` and `~`.

Validation: mandatory

Evaluation: computer

[4] You can consider that a `$((` always start an arithmetic expansion.

Level: hard

6.8 Readline

To validate this module, your 42sh must be able to manage the terminal in interactive mode. In a nutshell, when your 42sh is interactive, it must be able to understand special key features.

Of course, the `up` and `down` keys must be handled by the 42sh ***history*** module. You don't have to implement the `bind` builtin nor the completion.

Validation: advanced

Evaluation: human

Level: easy

6.9 Aliases

To validate this module, your 42sh must be able to make alias substitution in the parser, as the SCL describes it. You don't have to follow the behavior specified in the SCL when the alias ends with trailing spaces.

Validation: mandatory

Evaluation: computer

Level: hard

6.10 Path expansion, challenge mode

To validate this module, your 42sh must be able to follow the behavior of Bash about the following shell options:

- `dotglob`
- `extglob`
- `nocaseglob`
- `nullglob`

Validation: advanced

Evaluation: computer

Level: hard

7 Version 0.9

7.1 Completion

To validate this module, your 42sh must implement a simple completion feature in interactive mode.

```
42sh$ ls se (+tab)
secu-elf    seminair    seminaire   seminaire~
42sh$ ls sem (+ tab)
42sh$ ls seminair
42sh$ ls seminair (+tab)
seminair    seminaire   seminaire~
42sh$ ls .va (+tab)
42sh$ ls .vaisseau
```

Validation: advanced

Evaluation: human

Level: hard

7.2 Command Substitution

To validate this module, your 42sh must handle command substitution as described in the SCL. Command substitutions can be enclosed by "`" and "$(".

Validation: mandatory

Evaluation: computer

Level: hard

7.3 Job Control

To validate this module, your 42sh must be able to launch and manage more than one command at the same time. Your & operator must do its job correctly.

You must also implement the following builtin commands:

- `jobs` (with option `-l`)
- `fg [n]`
- `bg [n]`
- `wait [n]`

and the `$!` variable.

Switching between processes must be apparent. Of course, no zombie is tolerated.

Validation: advanced

Evaluation: human

Level: hard

8 Version 1.0

This is your *release* version. Your final mark depends mostly on it. **Do not fail to hand it in!** This version will be tortured to find memory leaks, segmentation faults, zombies, opened file descriptors, and other bad things. If any of those is found, your grade will be reduced by 25%.

Your source code won't be spared: it will be checked by the C99 coding style checker and, of course, cheating will be severely sanctioned.

Don't forget to update your 42sh manual page and to check that your code is properly Doxygen'ed.

You can also include extra bonuses in this version, but don't waste your time, there are few marks for this.

8.1 Git sanity

As the 1.0 version of your 42sh must not contain new major feature, we will not tolerate any *merge* commit.

9 Bibliography

- ***Unix general information:***
  - http://en.wikipedia.org/wiki/Unix_philosophy
  - http://opengroup.org/unix
  - http://www.yendor.com/programming/unix/apue/
- ***Shell information:***
  - http://www.gnu.org/software/bash/manual/
  - http://zsh.sourceforge.net/Doc/
  - http://www.kornshell.com/doc/
  - http://www.tcsh.org/Home
- ***Programming philosophies:***
  - http://programming-motherfucker.com/
  - http://www.extremeprogramming.org
  - http://agilemanifesto.org
  - http://en.wikipedia.org/wiki/Software_development_process
- ***Collaborative development tools:***
  - http://git-scm.com/
  - http://trac.edgewall.org
  - http://www.redmine.org

10 Epilogue

10.1 The search for the Ultimate Question

> Deep Thought informs the researchers that it will design a second and greater computer, incorporating living beings as part of its computational matrix, to tell them what the question is. That computer was called Earth and was so big that it was often mistaken for a planet. The researchers themselves took the apparent form of mice to run the program. The question was lost, five minutes before it was to have been produced, due to the Vogons demolition of the Earth, supposedly to build a hyperspace bypass. Later in the series, it is revealed that the Vogons had been hired to destroy the Earth by a consortium of philosophers and psychiatrists who feared for the loss of their jobs when the meaning of life became common knowledge.
>
> —Douglas Adams

10.2 Douglas Adams' view

Douglas Adams was asked many times during his career why he chose the number "forty-two". Many theories were proposed, but he rejected them all. On November 3, 1993, he gave an answer on `alt.fan.douglas-adams`:

> The answer to this is very simple. It was a joke. It had to be a number, an ordinary, smallish number, and I chose that one. Binary representations, base thirteen, Tibetan monks are all complete nonsense. I sat at my desk, stared into the garden and thought '42 will do'. I typed it out. End of story.
>
> —Wikipedia

Do not be sorry. Be better.