



42SH — Subject

version #1.5



Copyright

This document is for internal use at EPITA (website) only.

Copyright © 2019-2020 Assistants <assistants@tickets.assistants.epita.fr>

The use of this document must abide by the following rules:

- ▷ You downloaded it from the assistants' intranet.*
- ▷ This document is strictly personal and must **not** be passed onto someone else.
- ▷ Non-compliance with these rules can lead to severe sanctions.

Contents

1	Prologue	6
1.1	The Ultimate Answer	6
1.2	The reaction	6
2	Preamble	6
2.1	Why the 42sh?	6
2.2	Instructions	6
3	42sh	7
3.1	Technical constraints	7
3.2	Shell Command Language	8
3.3	Shell parser	8
3.4	Grading system	10
3.5	Documentation	10
3.6	Compilation	11
4	Assignment	11
4.1	Week 1	11
4.1.1	Autotools/CMake	11
4.1.2	Options parser	12
4.1.3	Standard Input	12
4.1.4	AST printer	12
4.1.5	Execution	13
4.1.6	Interactive	13
4.1.7	Prompt	14
4.1.8	String and File Inputs	14
4.1.9	Resource files	14
4.1.10	Tests suite	15
4.1.11	Manual	17
4.2	Week 2	17
4.2.1	“While” and “until” commands	17

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

4.2.2	“For” commands	17
4.2.3	“Case” commands	18
4.2.4	Operators	18
4.2.5	Redirections	18
4.2.6	Pipes	18
4.2.7	Functions	19
4.2.8	History	19
4.2.9	“history” command	19
4.2.10	“shopt”	19
4.3	Week 3	20
4.3.1	Built-in commands	20
4.3.2	Variables	21
4.3.3	Quoting	22
4.3.4	Command Substitution	22
4.3.5	Tilde expansion	22
4.3.6	Path expansion	23
4.3.7	Path expansion, challenge mode	23
4.3.8	Advanced prompt	24
4.4	Week 4	25
4.4.1	“alias” command	25
4.4.2	“unalias”	25
4.4.3	Aliases	25
4.4.4	Arithmetic expansion	25
4.4.5	Completion	26
4.4.6	Job Control	26
4.5	Final Version	26
5	Bibliography	27
6	Epilogue	27
6.1	The search for the Ultimate Question	27
6.2	Douglas Adams’ view	28

Obligations

Obligations are **fundamental** rules shared by all subjects. They are non-negotiable and to not apply them means to face sanctions. Therefore, do not hesitate to ask for explanations if you do not understand one of these rules.

Obligation #0: Cheating, as well as sharing source code, tests, test tools or coding-style correction tools is **strictly forbidden** and penalized by not being graded, being flagged as a cheater and reported to the academic staff.

Obligation #1: If you do not submit your work before the deadline, it will not be graded.

Obligation #2: Your submission repository must be **clean**. Except for special cases, which (if any) are **explicitly** mentioned in this document, an *unclean* repository may contain:

- binary files;¹
- files with inappropriate privileges;
- forbidden files: `*~`, `*.swp`, `*.o`, `*.a`, `*.so`, `*.class`, `*.log`, `*.core`, etc.;
- a file tree that does not follow our specifications (see ??).

Obligation #3: All your files must be encoded in ASCII or UTF-8 without BOM.

Obligation #4: When examples demonstrate the use of an output format, you must follow it scrupulously.

Obligation #5: Your code must compile with the flags:

```
-std=c99 -pedantic -Werror -Wall -Wextra
```

Obligation #6: If a function, a command or a library is not *explicitly* authorized, it is **forbidden**. If you use a forbidden symbol, your work will not be graded.

Advice

- ▷ Read the *whole* subject.
- ▷ If the slightest project-related problem arise, you can get in touch with the assistants.
Post to the dedicated **newsgroup** (with the appropriate **tag**) for questions about this document, or send a **ticket** to [<assistants@tickets.assistants.epita.fr>](mailto:assistants@tickets.assistants.epita.fr) otherwise.
- ▷ In examples, `42sh$` is our prompt: use it as a reference point.
- ▷ Do **not** wait for the last minute to start your project!

¹If an executable file is required, please provide its sources **only**. We will compile it ourselves.

Project data

Instructors:

- FERDINAND LEMAIRE <lemair_i@assistants.epita.fr>
- JULIEN MOUNIER <mounie_a@assistants.epita.fr>

Dedicated newsgroup: assistants.projets with [42SH]

Members per team: 4

1 Prologue

1.1 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"**.

1.2 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

2 Preamble

2.1 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!

2.2 Instructions

To succeed in implementing this shell 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.

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

3 42sh

Files to submit:

- `./doc/*`
- `./autogen.sh`
- `./configure.ac`
- `./Makefile.am`
- `./CMakeLists.txt`
- `./src/*`
- `./tests/*`
- `./README`
- `./TODO`

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

- `all`: Produces the 42sh binary
- `check`: Run your testsuite with 42sh binary
- `clean`: Delete everything produced by make

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

- `glob(3)`
- `regexec(3)`
- `wordexp(3)`
- `popen(3)`
- `syscall(2)`
- `system(3)`

3.1 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 memory¹ and close all its opened file descriptors.
- You must not strip your program, nor link it statically.
- You must comment your code for Doxygen.
- 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*.

3.2 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.

3.3 Shell parser

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).

```
/* Tokens */

%token If Then Else Elif Fi Do Done /* ... */
/* 'if' 'then' 'else' 'elif' 'fi' 'do' 'done' /* ... */
**
** These tokens are only recognised when being the first token
** of a command. Otherwise, these type as WORD tokens.
*/
```

(continues on next page)

¹ This will be checked with Address Sanitizer and Valgrind.


```

/* --- */

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

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

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

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:     ASSIGNMENT_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

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!

Be careful!

The parser is *the most important* part of your project. As we probably have already told you, you shall not let only one person do this part, as it will define the success or failure of your 42sh. That is the reason we put this assignment before any other.

3.4 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
- modules validation (these marks are common to the entire team)
- 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 team)
- 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!**

3.5 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.

3.6 Compilation

Your 42sh will be compiled using the following commands:

- if you use autotools:
 - ./autogen.sh
 - ./configure
 - make
- if you use cmake:
 - cmake .
 - make

However, in order to avoid having trash files at your root, you can do:

- With autotools:
 - ./autogen.sh
 - mkdir build
 - cd build
 - ../configure
 - make
- With cmake:
 - mkdir build
 - cd build
 - cmake ..
 - make

But remember that it won't be tested like that.

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

4 Assignment

4.1 Week 1

4.1.1 Autotools/CMake

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

Validation: core

Evaluation: computer

Level: easy

4.1.2 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.
- `[-+]0`: 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.

Tips

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

Validation: core

Evaluation: computer

Level: easy

4.1.3 Standard Input

In order for us to evaluate your 42sh, we will use commands launch via the standard input directly. What it means is that your 42sh must be able to execute command that way.

Be careful!

Without this part, you will have 0 to your traces, as we won't be able to launch most tests.

How are we going to launch it:

```
42sh$ cat test.sh | ./42sh
```

Validation: core

Evaluation: computer

Level: easy

4.1.4 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.

Validation: bonus

Evaluation: human

Level: easy

4.1.5 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. You don't have to implement *expansion* during execution yet, but it will be needed before the end of the project. These modules are the **most important** of your project. **Implement them seriously!**

Tips

In this first week, we're not expecting you to handle the variables and redirections correctly, however, you have to implement them before the end of the project, in order to get a good grade.

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: core

Evaluation: computer

Level: medium

"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: core

Evaluation: computer

Level: easy

4.1.6 Interactive

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.

If your interactive receives a ctrl+c (a SIGINT), it must not be killed (meaning that you have to catch it). However, your shell must exit if it receives a ctrl+d.

Validation: advanced

Evaluation: human

Level: easy

4.1.7 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 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

4.1.8 String and File Inputs

To validate this module, your 42sh must work properly with input script provided as files and strings. The example below shows the different input types:

```
42sh$ ./42sh -c "echo Input as string"
Input as string
42sh$ cat -e script.sh
echo Input as file$
42sh$ ./42sh script.sh
Input as file
```

Tips

The workflow of your lexer and parser should not be modified to handle different kinds of input. Take a look at *fmemopen(3)*.

Validation: advanced

Evaluation: computer

Level: medium

4.1.9 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

4.1.10 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

Test program To validate this module, you must write a test program in Python. Other script languages, such as Ruby or Perl, can be allowed if asked for by news. Be careful, you may not have any help from the assistants for those languages.

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 the number of successful and failed tests).
- You must display the global result of your tests after the execution.

Validation: core

Evaluation: human

Level: easy

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/
```

(continues on next page)

```
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

You must compare the standard output and standard error of your tests with `bash --posix`, you don't need to store it in files, however, you can make your script generate the expected output into a file.

Validation: bonus

Evaluation: human

Level: easy

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: core

Evaluation: human

Level: medium

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

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: bonus

Evaluation: human

Level: hard

4.1.11 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: Bonus

Evaluation: human

Level: easy

4.2 Week 2

4.2.1 “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: core

Evaluation: computer

Level: easy

4.2.2 “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: core

Evaluation: computer

Level: medium

4.2.3 “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: core

Evaluation: computer

Level: hard

4.2.4 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: core

Evaluation: computer

Level: easy

4.2.5 Redirections

To validate this module, your 42sh must be able to execute properly every redirection described in the SCL. Be careful with your file descriptors, and do not forget to test this part a lot, as there may be some corner cases you didn’t handle in your first implementation.

Validation: core

Evaluation: computer

Level: hard

4.2.6 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: core

Evaluation: computer

Level: hard

4.2.7 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: advanced

Evaluation: computer

Level: medium

4.2.8 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: bonus

Evaluation: human

Level: medium

4.2.9 “history” command

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: bonus

Evaluation: human

Level: medium

4.2.10 “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 `[+-]0` option must correctly activate `shopt` variables.

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

Validation: bonus

Evaluation: computer

Level: easy

4.3 Week 3

4.3.1 Built-in commands

“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: core

Evaluation: computer

Level: easy

“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: core

Evaluation: computer

Level: easy

“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: core

Evaluation: computer

Level: easy

“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: core

Evaluation: computer

Level: medium

“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: core

Evaluation: computer

Level: medium

“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: core

Evaluation: computer

Level: medium

“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: core

Evaluation: computer

Level: medium

4.3.2 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` variable is a colon-separated list of enabled `set` options. You can find the others in the Bash manual.

Validation: core

Evaluation: computer

Level: hard

4.3.3 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: core

Evaluation: computer

Level: medium

4.3.4 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: core

Evaluation: computer

Level: Really hard

4.3.5 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: advanced

Evaluation: computer

Level: easy

4.3.6 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: advanced

Evaluation: computer

Level: hard

4.3.7 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: Really hard

4.3.8 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: bonus

Evaluation: human

Level: easy

4.4 Week 4

4.4.1 “alias” command

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: advanced

Evaluation: computer

Level: easy

4.4.2 “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: advanced

Evaluation: computer

Level: easy

4.4.3 Aliases

To validate this module, your 42sh must be able to make alias substitution in the lexer, 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: advanced

Evaluation: computer

Level: hard

4.4.4 Arithmetic expansion

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

Validation: advanced

Evaluation: computer

Level: Really hard

4.4.5 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

4.4.6 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: Really hard

4.5 Final Version

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.

5 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>
 - <https://shell.multun.net/>
- **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>

6 Epilogue

6.1 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

6.2 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

I never really sleep. Got one eye open, always.