42sh — Subject

version #

Assistants C/Unix 2022 <assistants@tickets.assistants.epita.fr>

Copyright

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

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

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

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 **explicitely** 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.

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: The coding-style needs to be respected at all times.

Obligation #6: Global variables are forbidden, unless they are **explicitly** authorized

Obligation #7: Anything that is not **explicitly** allowed is **disallowed**.

Obligation #8: Your code must compile with the flags:

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

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>** otherwise.
- ▷ In examples, `42sh$` is our prompt: use it as a reference point.
- ▷ Do **not** wait for the last minute to start your project!

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

Be careful!

A guide is provided alongside the subject. Please read all of it before jumping in.

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 The UNIX wars

When command line operating systems were first written, every single one had its own kind of command line interface and language, each fairly different from the others.

One amongst many, UNIX was developed by Bell Labs (a research center) for internal use by AT&T (a telephone company).

In the late 1970s, AT&T sold UNIX licenses to academics, which included access to the source code. UNIX variants became popular among academics[1], and eventualy, spread into the computer system business.

By the early 1980s, a number of slightly incompatible UNIX variants were competing. Users could port their programs over from an UNIX system to another without much effort.

Soon, some variants became more popular than others. To avoid becoming irrelevant, smaller vendors decided to build common standards their products would meet (those products were then dubbed "Open systems").

Despite initial competition between standards, most of these vanished or merged, and today, only one truly remains: POSIX.

A brief history of POSIX:

[1] Some of the descendants of these operating systems are still in use today, such as OpenBSD, FreeBSD, or NetBSD

- in 1988, the first version of POSIX was published by the IEEE. Amongst many things, it defined how much of the C standard library shall work.
- in 1992, XPG4 was integrated into POSIX. This formerly separate standard was published the same year by the X/Open company. It includes an attempt to define how the UNIX shell[2] programming language works.

POSIX, and most other standards of the kind, evolve over time. That's why when programming in C, you may have to `#define` either `_POSIX_C_SOURCE` or `_XOPEN_SOURCE` to declare what revision of what standard you expect headers to be compliant to.

2.2 UNIX Shell and standardization

The first release of UNIX, in 1971, came with a shell.

For the next 20 years, UNIX derivatives developped their own, better (and sometimes slightly incompatible) shell programs.

Bash, the most used shell in the world nowadays, was first released in 1989, before any specification of the shell programming language.[3]

Eventualy, in 1992, the first specification of the UNIX Shell Command Language was published by X/Open. The IEEE integrated it into POSIX the same year.

Later, other shells, such as `dash` (1997), were written with POSIX compliance in mind. `dash` in particular also has few features outside the scope of the POSIX specification.

This project is about writing a POSIX Shell[4]. Thus, this subject will often provide links the the POSIX Shell Command Language specification.

3 Instructions

- *You only have to implement what's required by the subject, if you are unsure, ask*
- You can test your programs using `dash`, but beware: you do not have to implement everything it can do. When in doubt, ask.
- You must regularly checkout the `assistants.projets` newsgroup, where your project managers can amend the subject, announce conferences and events.
- Everything your program allocates must be freed
- Your program must not crash or exhibit unreliable behavior
- You must not strip your program, nor link it statically (it's properly done by default with meson, please don't change it).

[2] A shell is an user interface. In the UNIX world, shell mostly means "command line user interface".

[3] XPG4 wasn't published yet, and its description of the shell language didn't reach POSIX. Bash later added a `--posix` option, which when enabled, makes it POSIX compatible. Most of Bash's nonstandard features are still available even with this option.

[4] You won't have time to implement a full POSIX shell, and this subject also contains a few non-standard (but ubiquitous) features.

4 42sh

Files to submit:

- ./meson.build
- ./meson_options.txt
- ./README.md
- ./src/*
- ./tests/*

Makefile: Your makefile should define at least the following targets:

- 42sh: Produce the 42sh binary
- check: Runs your testsuite
- clean: Deletes everything produced by make

Coding Style Change:

- The number of non-function exported symbol (such as a global variable) MUST be at most 1 per source file.

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

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

4.1 Builtins

During the project, you will have to implement several commands, referred to as "*builtins*".

Most of the time, when given a command, a shell performs a fork(2) and calls one of the exec(3p) functions to execute it. However, this is not the case for some commands, defined as "builtins": these are coded directly in the shell, so it does not have to fork and execute it.

The same goes for your 42sh: you *must not* call one of the exec(3p) function to execute any of the builtin commands we will ask you to implement.

You can find information about builtins in the bash(1) manual page.

4.2 Compilation

Your 42sh will be compiled using the following commands:

- `meson setup builddir`
- `ninja -C builddir`

4.3 Usage

There are three ways 42sh can read its input Shell program:

- It can read its input from a string, given using `-c`

```
42sh$ ./42sh -c "echo Input as string"
Input as string
```

- It can read from a file, directly given as a positional argument

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

- It can also read commands from standard input when no other source is provided.

```
42sh$ cat -e script.sh
echo Input through stdin$
42sh$ ./42sh < script.sh
Input throught stdin
42sh$ cat script.sh | ./42sh
Input through stdin
```

5 Assignment

5.1 Week 1

5.1.1 Getting started

This section does not give any assignment, but rather advice about how to properly start the project.

Lexer / Parser

For this week, you will already need to execute commands. To do so you will need an AST. But this AST is given by a parser which itself depends on a lexer.

You can't afford to wait until you have a fully functional lexer to start writing your parser. Likewise you can't wait for the parser to handle every rule to start writing execution functions.

You should first write a first temporary rudimentary lexer and parser to be able to execute something early in the week. Even if it means rewriting a better lexer and parser later.

This first lexer should only lex what is needed this week, that is:

- `if`
- `then`
- `elif`
- `else`
- `fi`
- `;`
- `\n`
- `'`
- words

Pretty printer

During development, the ASTs your parser outputs might contain mistakes. In order to uncover those mistakes, understand what part of your code is at fault and move on, you have to compare what you expected to parse with what you actually parsed. This process is a whole lot easier when you can display the output of your parser.

Pretty-printing is a very efficient way to diagnose these bugs. Pretty-printing is the process of writting your AST (or any data structure) as easy to read text. As an example, the following command:

```
if echo ok; then echo foobar > example.txt; fi
```

Could be pretty-printed as:

```
if { command "echo" "ok" }; then { redir ">example.txt" command "echo" "foobar"; }
```

The format is up to you, of course.

Directly seeing the shape and contents of your AST will save you a lot of time during debugging sessions on your own or with the assistants.

You should be able to enable this feature through a command-line option or an environment variable when calling your program.

For instance:

```
./42sh --pretty-print example.sh
```

or

```
PRETTY_PRINT=1 ./42sh example.sh
```

This feature doesn't take a lot of time to implement, and brings immense quality of life improvements during development. You should probably implement it as soon as you have an AST.

5.1.2 GitLab CI

You should set up a CI to run your testsuite. Also, you should always write enough tests to be confident your project works and have some tests ready for the next step.

5.1.3 Options parser

Your project needs to interpret its command line arguments. If an invalid option is detected, you must print an error message and a usage message, both on the error output, and exit with a an error[1].

The command line syntax is: `42sh [OPTIONS] [SCRIPT] [ARGUMENTS ...]`

Your 42sh must accept at least the following option:

- `-c [SCRIPT]` instead of reading the script from a file, directly interpret the argument as a shell script

There is no other mandatory option, but you are free to implement some if it can ease your development or debugging process.

For instance you could implement a `--verbose` option for logging, and the `--ast-print` option, introduced elsewhere in this subject.

Tips

You are allowed to use `getopt_long`, which will make option parsing a lot easier.

5.1.4 Simple commands

```
simple_command: WORD*
```

Implement execution of simple commands such as `ls /bin`. You will need:

- a lexer which produces `word` tokens
- a `simple_command` AST node
- an execution module which runs the `simple_command` in your AST, waits for its status code, and returns it

[1] An exit status is an error if it's not zero

Tips

Use the execvp(3) version of exec(3), it searches the location of the executable in the PATH for you.

5.1.5 Command lists

```
command:    simple_command
pipeline:   command
and_or:     pipeline
list:       and_or (';' and_or)* [';'];
```

At this stage of the project, `command`, `pipeline`, and `and_or` are the same as `simple_command`. As you implement more features, this will change.

Your shell has to be able to group commands together in a list. At this stage, you only have to handle command lists as follows:

```
# this line must be in a single AST
echo foo; echo bar

# note the semicolon at the end
echo foo; echo bar;
```

In order to handle command lists:

- your lexer must recognize `;` tokens
- you need a special AST node for command lists

5.1.6 "If" commands

```
command:    simple_command
        |   shell_command

shell_command:  rule_if

list:       command (';' command)* [';'];

rule_if:    If compound_list Then compound_list [else_clause] Fi

else_clause:    Else compound_list
             |  Elif compound_list Then compound_list [else_clause]
```

You have to handle `if` commands:

- your lexer must recognize `if`, `then`, `elif` and `else` as special token types
- your AST has to have a node for conditions
- your parser must handle the `if` token returned by the lexer, parse the condition, the `then` token, the true branch, a series of `elif` branches, the `else` branch, and finally, `fi`

Pitfalls:

- The condition and body of `if`s are *compound lists*. At first, you can parse simple commands, but eventualy, you will have to support these.

5.1.7 Compound lists

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

Compound lists are just like command lists, with a few tweaks:

- this variant only appears inside code blocks such as conditions or functions
- compound list can separate commands using newlines instead of `;`

Compound lists are what enables conditions like this:

```
if false; true; then
   echo a
   echo b; echo c;
fi
```

Or even:

```
if false
   true
then echo a;echo b; echo c; fi
```

In order to handle compound lists:

- your lexer must recognize newline tokens
- you don't need another AST type, as compound lists are executed just like lists

Tips

This tip only applies if you're writting a recursive descent parser

When you meet a keyword which ends a control flow structure (such as `then` or `fi`), you have to stop parsing your compound list, and the function which called `parse_compound_list` decides if this keyword is appropriate.

5.1.8 Single quotes

Implementing this feature takes two main changes:

- implement lexing of single quotes
- implement expansion of single quotes during execution

The behavior of single quotes is specified by the SCL.

5.1.9 The echo built-in

As a builtin, `echo` is parsed as a simple command, but isn't executed in the same manner: It is executed directly inside your shell, without requiring `fork` or `exec`.

The `echo` command prints its arguments separated by spaces, and prints a final newline.

Your implementation of this command does not have to comply with POSIX.

You have to handle the following options:

- `-n` inhibits printing a newline.
- `-e` interprets the `\n`, `\t` and `\\` escapes.

5.2 Week 2

5.2.1 Redirections

```
redirection:    [IONUMBER] '>' WORD
        |   [IONUMBER] '<' WORD
        |   [IONUMBER] '>&' WORD
        |   [IONUMBER] '<&' WORD
        |   [IONUMBER] '>>' WORD
        |   [IONUMBER] '<>' WORD
        |   [IONUMBER] '>|' WORD

prefix:     redirection
element:    WORD
        |   redirection

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

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

Implement the execution of redirections as described in the SCL. You do not have to handle Here-Documents in this module.

Of course, redirections must work correctly with any command.

Be careful with your file descriptors, and do not forget to test this part a lot, as there may be some corner cases you did not handle in your first implementation.

Going further...

You need to call `fflush(stdout)` after running builtins.

Pitfalls:

- Redirections must work for builtins and functions. Consider the following code:

```
echo tofile >file.txt
echo tostdout
```

This shell program only uses builtins: `fork` will not be called, no new processes will be created.

If your shell performs the `>file.txt` redirection for the first command and doesn't take action to reverse it, the second command will write to file.txt too.

You have to save the file descriptors you override, and restore those to their former value when undoing the redirection.

- File descriptors are a scarse resource. On many systems, you can only have 1024 open file descriptors at once. If your redirection code inadvertently leaves file descriptors open, you may run out and get an error.
- When a new process is created using `fork`, it gets a copy of all the file descriptors of its parent. By default, the same thing occurs with `exec`.

  When you run an external program like `ls`, it doesn't need the saves of file descriptors you made for redirections.

  You can configure a file descriptor to be automaticaly closed on `exec` using `fcntl(fd, F_SETFD, FD_CLOEXEC)` (you also close these by hand if you really wanted to).

Tips

To check which file descriptors are open for a given process, you can run `ls -l /proc/${PID_YOUR_PROCESS}/fd`. You can also run `ls -l /proc/self/fd` to get which file descriptors are open for `ls` itself.

It can be combined with a command which gets the PID of a process by name: `ls -l "/proc/$(pgrep -n 42sh)/fd"`

5.2.2 Pipelines

```
pipeline:   command ('|' ('\n')* command)*
```

Implement pipelines as specified by the SCL.

The exit status of the pipeline is the exit status of the last command:

- `true | false` exits 1
- `false | true` exits 0

Pitfalls:

- `waitpid` must be called on all started processes
- Please refer to the guide for detailed explanations of pipe related pitfalls.

5.2.3 Negation

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

Adding a `!` reverses the exit status of the pipeline (even if there's no pipe):

- `true` exits 0
- `! true` exits 1
- `false` exits 1
- `! false` exits 0

You'll need to:

- Handle `!` in your lexer
- Add a new AST node type
- Parse and execute it

Pitfalls:

- Even if negation appears inside pipelines in the grammar, it makes little sense to perform negation inside a `pipeline` AST node. You can just have a separate AST node and only create it when needed.

5.2.4 "While" and "until" commands

```
shell_command:  rule_while
        |   rule_until
        |   rule_if

rule_while: While compound_list do_group
do_group:   Do compound_list Done
```

Implement the execution of `while` and `until` loops, as specified by the SCL.

5.2.5 "For" commands

```
shell_command:  rule_for
        |   rule_while
        |   rule_until
        |   rule_if

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

Implement the execution of `for` loops, as specified by the SCL.

5.2.6 Operators

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

Implement the execution of the "`&&`" and "`||`" operators, as specified by the SCL.

5.2.7 Double Quotes and Escape Character

Implement the lexing and expansion of double quotes and escape characters, as described in the SCL.

Pitfalls:

- This part has complicated interactions with later features, such as subshells
- You have to make sure your expansion algorithm is the same as your lexing algorithm. If the two don't agree, weird bugs will ensue.

5.2.8 Variables

```
prefix:     ASSIGNMENT_WORD
        |   redirection
```

Implement **variable assignment** and simple **variable subsitutions**, as described by the SCL.

You don't have to implement expansion modifiers, only the ``$name`` and ``${name}`` formats will be tested

The following special variables must be properly expanded:

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

Tips

For this week, expansion is only ever done within double quotes, you do not have to handle IFS spliting yet.

Pitfalls:

- because of `$@`, expansion outputs an array of strings

5.3 Week 3

5.3.1 Built-in commands

"exit"

Implement the `exit` builtin.

For more information about `exit`, please refer to the SCL

Pitfalls:

- All resources should be released (allocated memory and file descriptors) before calling exit. The easiest way to handle this is to have a special kind of "error" which stops execution and exits normaly.

"cd"

Implement the `cd` builtin. You do not have to implement the `-L` and `-P` options, nor to follow the required behavior for the `CDPATH` variable. However, you have to implement `cd -`.

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

You must also update the `PWD` and `OLDPWD` environment variables.

Beware, the shell keeps track of the path which was taken through symlinks:

```
mkdir -p /tmp/test_dir
ln -s /tmp/test_dir /tmp/link
cd /tmp/link

# the shell knows the current directory is also known as /tmp/link
echo "$PWD"

# pwd doesn't, as it's an external command. env -i ensures the PWD
# environment variable isn't passed down, and avoids executing a potential
# builtin implementation of pwd
env -i pwd
```

"export"

Implement the `export` builtin. You do not have to handle printing all exported variables, only the `export NAME=VALUE` and `export NAME` uses will be tested.

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

“continue” and “break”

Implement the `continue` and `break` builtins.

For more information about these, please refer to the SCL:

- SCL specification for continue
- SCL specification for break

Pitfalls:

- Mind the corner cases of breaking / continuing out of more loops than are currently active. Continuing and breaking also has to work cross-functions.
- `continue` and `break` work across function boundaries.

“dot”

Implement the `.` builtin as specified by the SCL.

“unset”

Implement the `unset` builtin with all its options, as specified by the SCL.

5.3.2 Command blocks

```
shell_command:  '{' compound_list '}'
        |   rule_for
        |   rule_while
        |   rule_until
        |   rule_if
```

Command blocks are a way to explicitly create command lists. These are useful for making function bodies, as well as grouping commands together in redirections, but can be used anywhere.

```
{ echo a; echo b; } | tr b h

foo() { echo this is inside a command block; }
```

5.3.3 Functions

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

funcdec:    WORD '(' ')' ('\n')* shell_command
```

Implement function definition and execution. This includes, of course, redirections to functions and argument transmission.

Pitfalls:

- Functions have to get a reference to some part of your AST, which should otherwise be freed at the end of each command. It means that you either have to make a copy of part of your AST (the body of the function), or prevent it from being freed at the end of this "line". This can be accomplished fairly easily using reference counting.
- Functions can be defined in any command. This is valid:

```
foo() {
   bar() {
       echo foobar
   }
}

# defines bar
foo

# prints foobar
bar
```

5.3.4 Command Substitution

Implement command substitution as described by the SCL.

Pitfalls:

- At this point, your lexer needs to be recursive and remember what kind of context is currently active. The context is saved when entering a new context and restored when leaving a context.
- Your will have a hard time keeping your lexer and expansion in sync unless you create some kind of library.

5.3.5 Subshells

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

Subshells run commands in a new process.

```
echo "current shell pid: $$"
(echo "subshell pid: $$"; exit 42; echo never executed)
echo "subshell exited with status $?"
```

Please refer to the SCL specification.

5.4 Week 4

5.4.1 “Case” commands

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

Implement the `case` construct as specified by the SCL.

5.4.2 Aliases

Implement alias handling as specified by the SCL.

It requires:

- Implementing the `alias` and `unalias` builtins, also to SCL specification (for alias and for unalias).
- Using the alias list for substitutions inside the lexer

Substitutions are performed at the token level:

```
alias funcdec='foo('
funcdec) { echo ok; }
foo
```

Your have to lex, parse, and execute one line at a time. Otherwise, your lexer will not know about your aliases in time.

You don’t have to follow the behavior specified in the SCL when the alias ends with trailing spaces.

```
# this does not work, as the whole line is lexed and
# parsed as a whole (it matches the list grammar rule)
alias foo=ls; foo

# this works, as the bar alias was registered before bar was lexed
alias bar=ls
bar
```

5.4.3 Field Splitting

Implement Field spliting as specified by the SCL.

5.5 Advice

42sh is not an easy project to implement, let alone to debug. In this section, we will provide you with some ideas of features you might want to implement in order to ease both your development, and your debugging sessions with the assistants.

5.5.1 Testsuite

You might have realized how important a *strong* testsuite is for a project of this kind. We strongly advise you to build one which at least:

- implements all the needed functions to really test your program
- prevents *regressions*
- gives you the possibility to follow the progress you made

You can write unit tests to check the behaviour of some functions, but functional tests are handier to test this project. You should focus on them.

Here are some useful milestones to help you determine how advanced your testsuite is.

Test program

We recommand you to write a test program whose output would follow an easily readable format. Here are some tips:

- issue only one line per test.
- you must be able to clearly understand the result of the test. This means that at least failure or success should be printed. It is desirable to display the cause of a failure: standard output, error output, exit value or any combination among those three reasons.
- group tests into categories: you want to be able to clearly identify the category of the running test. Before testing a category, we advise you to display its name followed by a blank line, and after the tests, a blank line and the result of the category (with a percentage or the number of successful and failed tests).

- display the global result of your tests after the execution.

A category represents a whole set of tests aiming at evaluating a particular part of your 42sh.

Tests format

As your tests are distributed into categories you might want to have the tests of a category grouped in a directory which name matches that of the test category.

For instance:

```
42sh$ ls -la tests/categories
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/
```

We also advise you to have only one file per test. This file must at least contain:

- a test description
- the input

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

Options format

In order to have a better granularity concerning the use of your testsuite, we advise you to implement some of 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 make the test fail.

Only testing the features you want the way you want will save you a lot of time.

Timeout management

Sometimes a program gets stuck in an infinite loop or is blocked by a syscall. In order to avoid blocking 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.

An option you might want to implement would be a `-t <time>` or `--timeout <time>` options which set `time` as a general timeout time (in seconds).

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

5.6 Bonus features

5.6.1 Tilde expansion

Implement "~" expansion as specified by the SCL.

This only works in a limited number of contexts, so some indication has to be given to expansion to limit its scope.

```
echo ab~
echo a,~
echo a=~
echo a:~

foo=a:~
echo "$foo"

foo=a,~
echo "$foo"

foo=a=~
echo "$foo"
```

5.6.2 Path expansion

Implement the expansion of 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
```

(continues on next page)

(continued from previous page)

```
./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$
```

5.6.3 Arithmetic expansion

Implement the expansion of arithmetic expressions, which are wrapped by `$(())`.

You have to handle variables and the following opperators: `-`, `+`, `*`, `/`, `**`, `&`, `|`, `^`, `&&`, `||`, `!` and `~`. You don't need to handle other operators.

For more information, please refer to the SCL.

Pitfalls:

- Your lexer and expansion have do deal with an ambiguity for expansions beginning with `$((`. Please refer to the SCL.

5.6.4 Here-Document

Implement execution of Here-Documents as described in the SCL. Once again, be careful with your file descriptors.

Pitfalls:

- You have to both read and write to the same fd in the same process. The easier way to do that is to write the content of the heredoc in a temporary file, and read it afterwards. You can also do it with a pipe, as long as the size of the expanded variable doesn't exceed the capacity of the pipe (your process will block as soon as the pipe is full)

5.7 Going Further

If you implemented all previous modules and are confident about them you can consider doing the followings.

Be careful!

These features are not tested for, and you will not get any bonus points for implementing these.

You will only succeed in impressing friends, family, and teaching assistants.

5.7.1 Prompt

Your 42sh must show a prompt if the shell is in interactive mode. You only have to implement PS1 and PS2 prompts as they described by the SCL.

5.7.2 Job Control

Your 42sh must be able to launch and manage more than one command at the same time. You should add the `&` operator.

You shall also implement the following builtin commands:

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

and the `$!` variable.

Switching between processes must be apparent, and no zombie processes shall be left behind.

Tips

You can read this to get hints about how to do it: https://www.gnu.org/software/libc/manual/html_node/Implementing-a-Shell.html

6 Bibliography

- **Shell information:**
  - https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html
  - https://shell.multun.net/
  - https://en.wikipedia.org/wiki/Recursive_descent_parser
  - http://www.gnu.org/software/bash/manual/
  - http://zsh.sourceforge.net/Doc/
  - http://www.kornshell.com/doc/
  - http://www.tcsh.org/

- **Programming philosophies:**
  - https://martinfowler.com/agile.html
  - https://wiki.c2.com/?ExtremeProgrammingRoadmap
  - https://www.agilealliance.org/agile101/subway-map-to-agile-practices/
  - https://manifesto.softwarecraftsmanship.org/
- **Git and related tools:**
  - https://git-scm.com
  - https://learngitbranching.js.org/
  - https://docs.gitlab.com/ee/user/project/repository/repository_mirroring.html
  - https://docs.gitlab.com/ee/ci/
- **Build systems:**
  - https://mesonbuild.com/Manual.html
- **Documentation (man/Doxygen):**
  - https://www.doxygen.nl/index.html

7 Epilogue

7.1 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

7.2 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 3rd, 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

It is my job to make sure you do yours.