42SH-SUBJECT — Subject

version #

ASSISTANTS C/UNIX 2024 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

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

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 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 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 eventually, 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.

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

3.2 The POSIX standard

POSIX stands for Portable Operating System Interface. The most recent POSIX specification defines a standard interface and environment that can be used by an OS to provide access to POSIX-compliant applications. This standard also defines a command interpreter, shell, and common utility programs. POSIX supports application portability at the source code level so they can be built to run on any POSIX-compliant OS.

A brief history of POSIX:

- 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, the XPG4 standard was integrated into POSIX. This standard, published the same year by the X/Open company, included an attempt to define how the UNIX shell[2] language works.

POSIX, as most other standards of the kind, evolves over time. That is 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.

By making sure your project is as POSIX-compliant as possible, you can ensure that your 42sh is portable and usable on any POSIX-compliant OS. To check the compliance of your project with the POSIX standard, make sure to compare its behavior with that of the POSIX mode of Bash, which can be started with `bash --posix`. `bash --posix` is not always POSIX-compliant, but it is one of the most POSIX-compliant shells. To learn more about what is and what is not POSIX-compliant in `bash --posix`, you can take a look at the documentation.

Be careful!

When implementing features, prioritize the specifications of the subject over other sources, even if it means not exactly matching the behavior of the real features.

3.3 UNIX Shell and standardization

The first release of UNIX in 1971 came with a shell known as the Thompson Shell. In the following 20 years, UNIX derivatives developed 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 was published.[3] The `--posix` option of Bash was later added to make it POSIX-compliant. Most of Bash's non-standard features are still available even with this option.

Later, other shells such as Dash (1997) were written with POSIX compliance in mind. However, even Dash deviates from the POSIX specification by a small amount, by including some features that are not part of the specification, and by missing features that are part of it.

The goal of this project is to write a POSIX shell[4], which is why the subject will often refer you to the

[2] A shell is a user interface. In the UNIX world, shell mostly means "command line user interface".
[3] XPG4 was not published until 1992, so its specification of the UNIX Shell Command Language (SCL) was not integrated into POSIX yet.
[4] You won't have time to implement a full POSIX shell, and this subject also contains a few non-standard (but ubiquitous) features.

POSIX specification. More specifically, you will often have to refer to the *Shell Command Language* chapter of the Shell and Utilites (XCU) volume of the POSIX specification, which you are highly encouraged to read, as it is essential to the understanding of the project.

4 Guide

This guide touches on notions that are required in order to prepare you for the project.

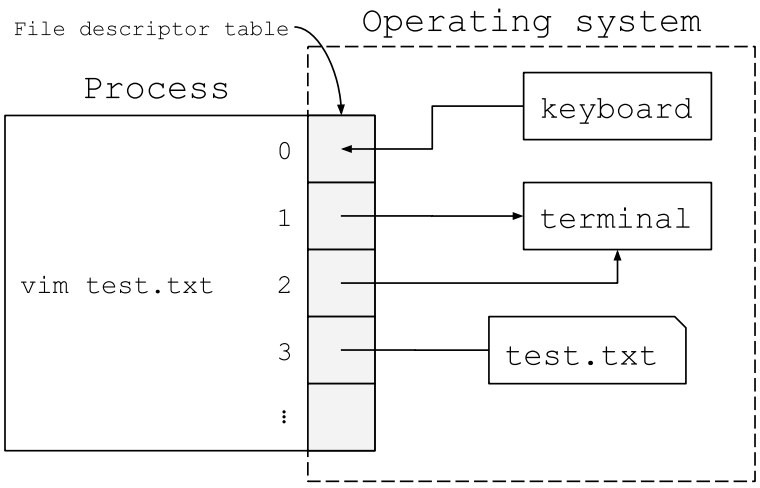
4.1 Prerequisites

4.1.1 File descriptors

Processes need to deal with files, but cannot do it directly. Writing to files requires access to an actual storage medium, such as a hard disk: you would not like any process to be able to wipe your disk, would you? Because of this reason and many others, operating systems manage files for processes.

When a process asks to open a file, the operating system returns a special identifier called a file descriptor.

On the operating system side, each process has an array of open files, called the file descriptor table. A **file descriptor**, often abbreviated to fd, is the index of a file in this table.



When a new file is opened, the first unused cell of the array is filled by a pointer to the open file. The operating system returns the file descriptor of this cell to the process, which uses it to interact with the file.

The process can then `read(2)` and `write(2)` to the file using the file descriptor. Once the process stops working with the file, it can call `close(2)` to release it.

Tips

`stdin`, `stdout` and `stderr` have fixed file descriptor numbers:

- `stdin` is 0
- `stdout` is 1
- `stderr` is 2

4.1.2 An animated example

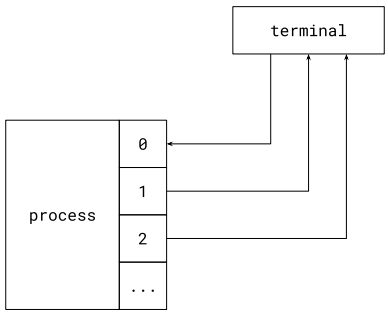


Fig. 1: A process was just started in a terminal. It reads input from stdin (file descriptor 0), writes to stdout (file descriptor 1) and in case of errors, to stderr (file descriptor 2).

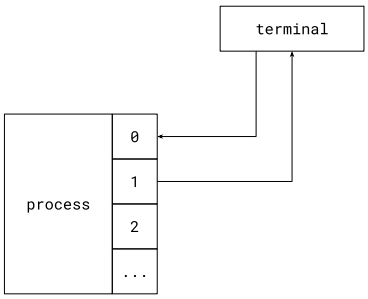


Fig. 2: After running `close(2)`

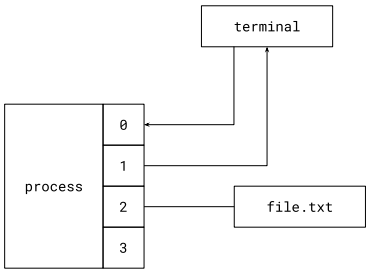


Fig. 3: After calling `open("file.txt")`. The operating system used the first available file number.

4.1.3 dup

The `dup(2)` function takes an open file descriptor as argument, and returns a new file descriptor referring to the same file (hence its name: it **dup**licates the given file descriptor).

This means that any operation performed on the newly created file descriptor will actually be done on the file descriptor referred to by the file descriptor which was given to `dup(2)` in the first place.

Example:

```c
#include <unistd.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
    const char *first_msg = "I will be printed on stdout!\n";
    size_t first_msg_size = strlen(first_msg);
    write(STDOUT_FILENO, first_msg, first_msg_size);

    int stdout_dup = dup(STDOUT_FILENO);
    printf("created a copy of stdout: %d\n", stdout_dup);

    const char *second_msg = "I also will be printed on stdout!\n";
    size_t second_msg_size = strlen(second_msg);
    write(stdout_dup, second_msg, second_msg_size);

    close(stdout_dup);

    return 0;
}
```

This program uses `dup(2)` to duplicate the standard output's file descriptor, writes a message to the standard output's file descriptor, and then writes another message to the duplicated file descriptor. Running this code writes both messages to the terminal's standard output.

`dup(2)` always uses the lowest available file descriptor number. For example, if all your file descriptors below 10 point to something and you `close(2)` two of them, the one with the lowest index will be reused.

Be careful!

Do not forget to `close(2)` the file descriptor created with `dup(2)` after you are done using it.

4.1.4 dup2

Duplicating file descriptors is nice, but what we actually want to perform is a redirection. How do we make this happen?

The answer is located in the same manual page as `dup(2)`, one line below: the `dup2(2)` function.

`dup2(2)` takes two arguments: an "old" file descriptor and a "new" file descriptor. `dup2(2)` makes a copy of the "old" file descriptor, but instead of using the lowest available number, it uses the file descriptor number in "new".

You can think of it as making a copy of “old” and storing it into “new”.

If “new” already points to something, the previous file descriptor is automatically closed.

For instance:

```c
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <err.h>

int main(void)
{
    int file_fd = open("/tmp/dup2_test", O_CREAT | O_TRUNC | O_WRONLY, 0755);
    if (file_fd == -1)
        err(1, "open failed");

    if (dup2(file_fd, STDOUT_FILENO) == -1)
        err(1, "dup2 failed");

    puts("This will be written to /tmp/dup2_test");
    return 0;
}
```

Going further...

`puts(3)` and `printf(3)` internally use a `write(2)` operation on `STDOUT_FILENO`. This is why the message printed by `puts` is redirected to the test file.

4.1.5 Saving and restoring file descriptors

Consider the example of the previous section. Can we print a message to stdout after performing the redirection to the file? No, we cannot, as `dup2(2)` closed `STDOUT_FILENO` by copying `file_fd` into it.

Well, we could do something about it if we bothered to save stdout before performing the redirection. This way, we can reverse the redirection once it is not useful anymore.

We can make a save of the file descriptor using `dup(2)`, and restore it using `dup2(2)`.

```c
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <err.h>

int main(void)
{
    int stdout_dup = dup(STDOUT_FILENO);

    int file_fd = open("/tmp/dup2_test", O_CREAT | O_TRUNC | O_WRONLY, 0755);
    if (file_fd == -1)
        err(1, "open failed");

    if (dup2(file_fd, STDOUT_FILENO) == -1)
        err(1, "dup2 failed");
```

(continues on next page)

(continued from previous page)

```
    puts("This will be written to /tmp/dup2_test");

    // Ensure the message is printed before dup2 is performed
    fflush(stdout);

    dup2(stdout_dup, STDOUT_FILENO);
    close(stdout_dup);

    puts("This will be printed on standard output!");

    return 0;
}
```

4.1.6 Executing commands

To execute a command, you must:

- call `fork(2)` to create a child process. This creates a copy of the current process.
- from the child process, call a function of the `exec(3)` family to start the given command inside the process.

From the parent process, you then **always** have to wait for the child process to terminate using `waitpid(2)`.

When forking and using an exec function, file descriptors are not reset: the child process inherits its parent's file descriptors.

4.1.7 Fork

The `fork(2)` syscall duplicates the current process to make an almost identical copy. Everything within the process is duplicated, including file descriptors.

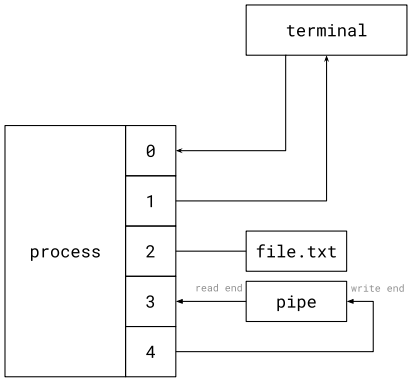


Fig. 4: Before `fork(2)`

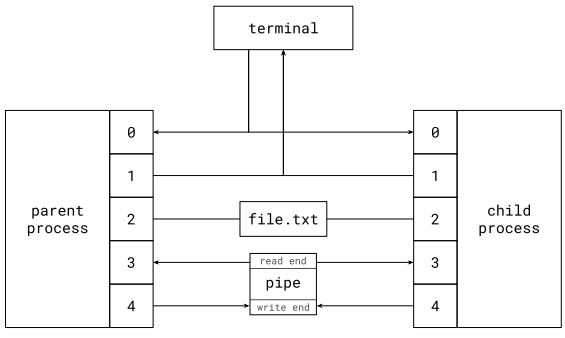


Fig. 5: After `fork(2)`

4.1.8 exec

The `exec(3)` family of functions[1] all execute a program, with a twist: instead of creating a new process, the process which calls `exec(3)` is re-used, and the current program is replaced.

File descriptors remain open across an `exec(3)` call[2].

Consider the program `myprogram.c`:

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <err.h>

int main(int argc, char *argv[])
{
    int pid = fork();
    if (pid == 0)
    {
        printf("[PID %d] %s \tI am a new process\n", getpid(), argv[0]);
        char *argv[] = { "mychild", "Hello Child!\n", NULL };
        execve("mychild", argv, NULL);
        err(1, "failed to exec mychild");
    }
    else
    {
        printf("[PID %d] %s \t"
               "I am the parent process, waiting for its child to exit\n",
               getpid(), argv[0]);
        waitpid(pid, NULL, 0);
    }

    return 0;
}
```

As well as `mychild.c`:

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    printf("[PID %d] %s \tI am the new process, but with a different program\n",
           getpid(), argv[0]);
    return 0;
}
```

[1] Some variants of `exec(3)` look for the program in the folders listed with the `PATH` variable, and some take variadic arguments instead of an array. Check out `man 3 exec`.

[2] You can explicitly tell the OS to close some file descriptors when exec is called using `fcntl(fd, F_SETFD, FD_CLOEXEC)`.

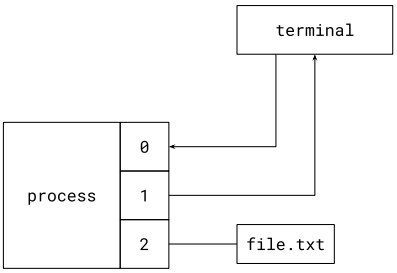


Fig. 6: Before `fork(2)`

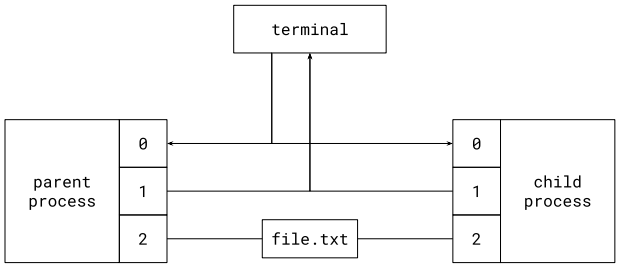


Fig. 7: After `fork(2)`

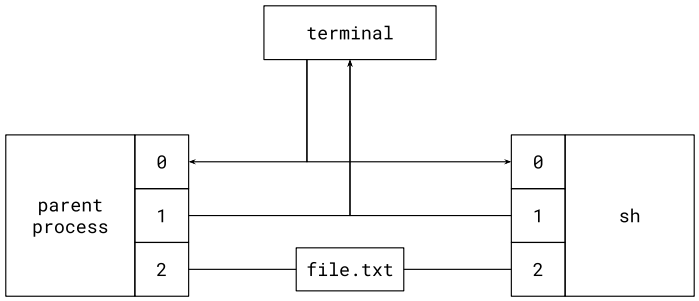


Fig. 8: After running :

```
execve("sh", [ "sh", "-c", "echo", "Hello World!", NULL ], NULL);
```

Let us compile both programs and execute `myprogram`:

```
42sh$ make myprogram mychild
42sh$ ./myprogram
[PID 673713] ./myprogram    I am the parent process, waiting for its child to exit
[PID 673714] ./myprogram    I am a new process
[PID 673714] mychild        I am the new process, but with a different program
```

4.1.9 pipe

Pipes are a data queue provided by the operating system: you can write data on one end, and read the same data on the other end.

Pipes only work one way: one side can only be written to, and the other side can only be read from.

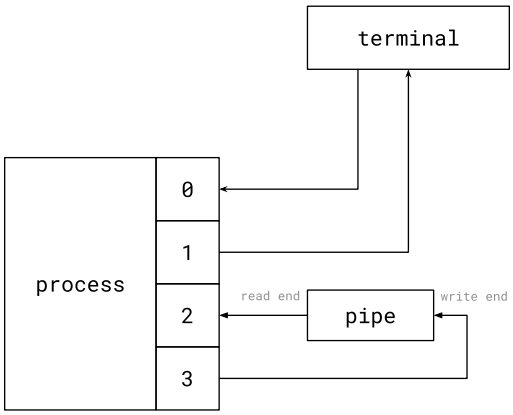
The `pipe(2)` syscall takes a pointer to an array of two integers. Calling `pipe(pipefds)` fills `pipefds` with two file descriptors:

- `pipefds[0]` is the read end (the side processes read from)
- `pipefds[1]` is the write end (the side processes write to)

Pipes are mainly used to exchange data between processes: the read end of the pipe is in a process which consumes data, and the write end of the pipe is in a process which writes data.

Be careful!

When two processes exchange data this way, the process which reads data has an easy way of



telling whether the process which writes data has completed its task, and will no longer write to the pipe.

When all the file descriptors to the write side are closed, reading on the other side returns an `EOF`.

It **only** works if all the file descriptors to the write side are **closed**.

```c
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <err.h>
#include <assert.h>

int main(void)
{
    // this program has no error checks
    // your code must check for errors
    int pipefds[2];
    pipe(pipefds);

    write(pipefds[1], "Hello", 5);
    write(pipefds[1], "World", 5);
    write(pipefds[1], "!", 1);

    char buffer[10];
    read(pipefds[0], buffer, 10);
    printf("%.10s\n", buffer);
    read(pipefds[0], buffer, 1);
    printf("%.1s\n", buffer);

    close(pipefds[0]);
    close(pipefds[1]);
    return 0;
}
```

Pipes have a limited size: if you try to write some data into a pipe, and it happens to be full, your process will be blocked until something reads on the other side of the pipe.

If there is no process reading data on the other end, your program will get stuck.

This *could* happen in the above program: if the operating system's pipes were not large enough to hold a full "HelloWorld!", the process would get stuck trying to write it in the pipe.

Never write data in a pipe unless you are sure another process will read it immediately.

4.1.10 Using pipes across processes

When a process calls `fork(2)`, the child process can still use the exact same file descriptors as the parent.

For that reason, if a pipe is created in a process which then calls `fork(2)` to create a new process, the new process will still be able to read from and write to the pipe.

Be careful!

The read-end of the pipe will receive an `EOF` when the write-end is closed and if there is no data left to read. However, in order for a file descriptor to be closed, *every process* has to close it. Otherwise, the reader will never receive the `EOF`.

```
Writer  : fds[1] -------+----> fds[0]
                        \
                         X
                        /
Receiver: fds[1] -------+----> fds[0]
```

In the example above, if the writer closes `fds[1]`, the reader will never get an `EOF` since it is opened on its side. This is why unused pipe ends must be closed.

```
Writer  : [opened fds[1]] ----      [closed fds[0]]
                                \
Receiver: [closed fds[1]]       ---->[opened fds[0]]
```

Now, when the writer is done writing on the pipe, it can close it and the receiver will be notified with an `EOF` since no other process has this file descriptor opened.

Below is a code example of what has been seen above:

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <err.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(int argc, char *argv[])
{
    if (argc != 2)
        errx(1, "Usage: %s MESSAGE", argv[0]);

    int fds[2];

    if (pipe(fds) == -1)
        errx(1, "Failed to create pipe file descriptors.");

    pid_t pid = fork();
```

(continues on next page)

(continued from previous page)

```
    if (pid < 0)
        errx(1, "Failed to fork.");

    // if we are inside the child process,
    // read the data from end pipe and write it to stdout
    if (pid == 0)
    {
        // we NEED to close the write side: otherwise, we cannot
        // know when the parent is done writing
        close(fds[1]);

        char buf[10];
        size_t nb_read = 0;
        while ((nb_read = read(fds[0], buf, 10)))
            fwrite(buf, 1, nb_read, stdout);

        close(fds[0]);
        return 0;
    }
    // in the parent, write the data the the pipe
    else
    {
        close(fds[0]);

        size_t data_len = strlen(argv[1]) + 1;
        size_t nb_written = 0;
        while (nb_written < data_len)
            nb_written += write(fds[1],
                                argv[1] + nb_written,
                                data_len - nb_written);
        close(fds[1]);
        int status;
        waitpid(pid, &status, 0);
        return WEXITSTATUS(status);
    }
}
```

```
42sh$ make test2
cc test2.c -o test2
42sh$ ./test2 "This is a test."
This is a test.
```

Be careful!

In order to avoid leaks, always close the file descriptors in both processes.

For further information, you can read the man page of `pipe(2)`.

4.2 The shell syntax

Be careful!

You do not have to implement everything the shell grammar encompasses. If your project does not support a feature yet, it makes little sense to parse it. However, keep in mind the big picture when implementing parts of it, as it will lower the chances of having to start again from scratch.

The syntax of shell is defined in a layered fashion:

- first, characters are grouped together into tokens
- then, a grammar establishes the way tokens can be organized to make valid shell programs

To get a good overview of the steps the input goes through to get to an AST representation, we recommend the excellent Crafting Interpreters online book (at least "Scanning", "Representing Code" and "Parsing Expressions" chapters).

4.2.1 Token recognition

What are tokens and how to get from characters to tokens is pretty well defined by the SCL.

It even defines an algorithm!

4.2.2 Grammar

A grammar is a document which describes what inputs, once delimited into tokens, are part of a language. If an input is not part of the language, then it is invalid and will trigger a syntax error.

The SCL defines a grammar in a format that is fairly difficult to read and implement. It was transcribed into another form, which is meant to be easier to work with.

The format we use to write grammars is called EBNF. Here is a quick overview of this notation:

- the grammar is a set of rules, which all define a set of valid inputs
- each rule starts like so: `rule_name =` and ends with a `;`
- the `input` rule is the start of the grammar
- `|` represents alternative options
- if something is between `[` these brackets `]`, it is optional
- if something is between `{` these curly brackets `}`, it may be repeated 0 to infinite times
- if something is between `(` these parentheses `)`, it is grouped
- `'a'` is the literal character `a`, and `'abc'` the literal string `abc`
- `EOF` is the end of the input
- rules can appear inside other rules (and thus may be recursive)

You do not have to implement everything in this grammar, you can stick to what you can execute.

The grammar is designed this way to make the `AST (Abstract Syntax Tree)` building process easier, but you may add custom rules to factor repetitions of token sequences.

```
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 }
  | { prefix } WORD { element }
  ;

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

funcdec = WORD '(' ')' {'\n'} shell_command ;

redirection =
    [IONUMBER] ( '>' | '<' | '>>' | '>&' | '<&' | '>|' | '<>' ) WORD
  | [IONUMBER] ( '<<' | '<<-' ) HEREDOC
  ;

prefix =
    ASSIGNMENT_WORD
  | redirection
  ;

element =
    WORD
  | redirection
  ;

compound_list =
    {'\n'} and_or { ( ';' | '&' | '\n' ) {'\n'} and_or } [ ';' | '&' ] {'\n'} ;
```

(continues on next page)

(continued from previous page)

```
rule_for =
    'for' WORD ( [';'] | [ {'\n'} 'in' { WORD } ( ';' | '\n' ) ] ) {'\n'} 'do' compound_list
↪'done' ;

rule_while = 'while' compound_list 'do' compound_list 'done' ;

rule_until = 'until' compound_list 'do' compound_list 'done' ;

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 ]
  ;

case_clause = case_item { ';;' {'\n'} case_item } [';;'] {'\n'} ;

case_item = ['('] WORD { '|' WORD } ')' {'\n'}  [compound_list] ;
```

Reserved words

Since the separator at the end of the `compound_list` rule is optional, the grammar may appear to accept commands such as:

```
if echo foo then echo bar fi
```

However, you must bear in mind that reserved words, such as `if`, `then` and `fi` are only recognized at the start of a command. As such, the places where reserved words can be detected would be:

```
if echo foo then echo bar fi
^  ^
```

Indeed, every word after the first echo is part of its arguments. Thus, `then` and `fi` are not special and does not end the condition.

However, adding a `;` ends the current command and starts another one, which creates an opportunity for reserved words to appear:

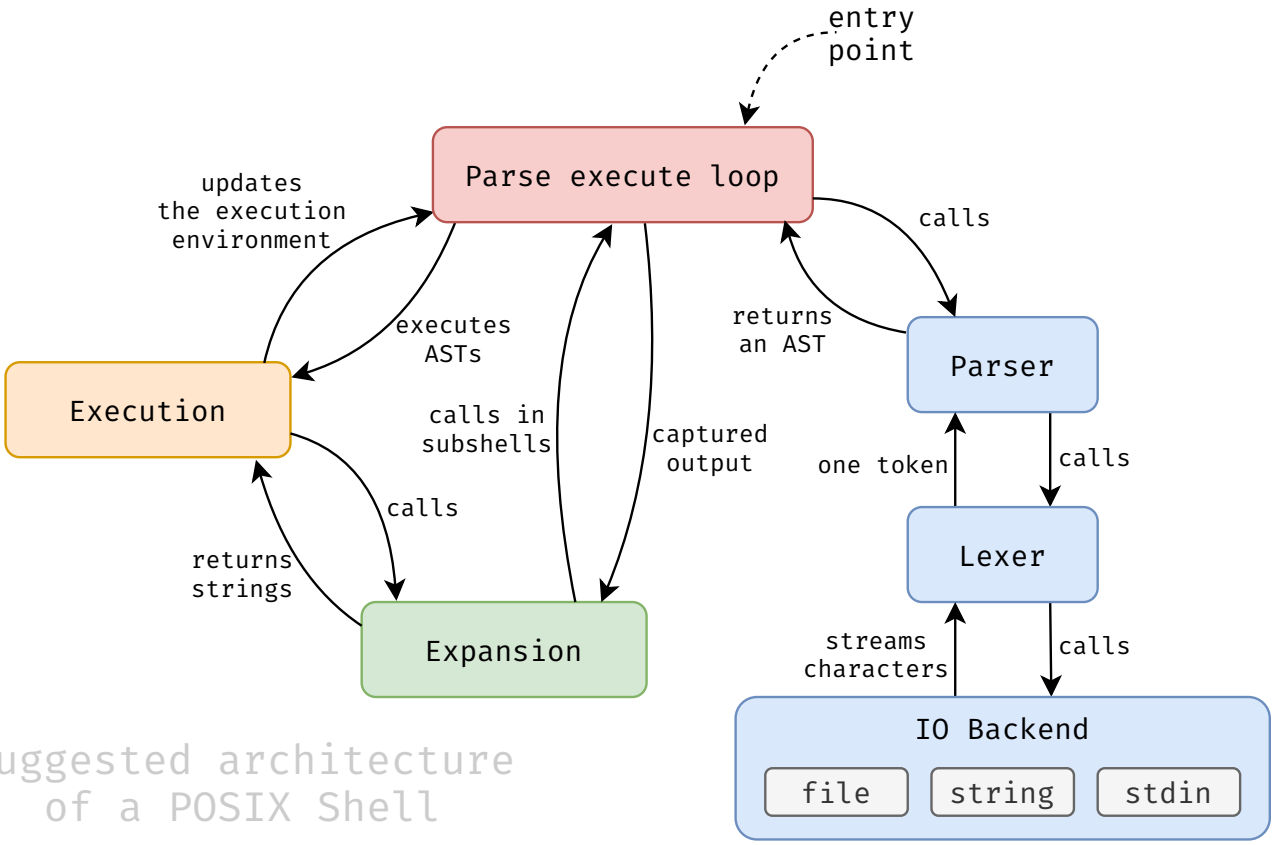
```
if echo foo; then echo bar; fi
^  ^         ^    ^         ^
```

4.2.3 When to implement parts

Every feature mentioned in the assignments part of the subject will refer to parts of this grammar to list every new token and grammar rule involved in the feature. However, keep in mind that you will eventually handle a lot of this grammar; so, to avoid regression, take into account rules in their full recursive hierarchy, and do not deviate from the grammar to facilitate a case.

4.3 Shell anatomy

4.3.1 Architecture



Suggested architecture
of a POSIX Shell

5 Instructions

- You only have to implement what is required by the subject, if you are unsure, ask.
- You can test your programs using `bash --posix`, but beware: you do not have to implement everything it can do. When in doubt, ask.
- You must **not** use `_GNU_SOURCE` extensions.
- You must regularly checkout the `Assistants ING - 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.
- When submitting your program, you must remove all debugging and logging messages displayed on standard outputs or you will be penalized.

Some slight changes apply compared to the default EPITA coding style:

- You are allowed to use the `goto` keyword for error handling only.
- You are allowed to use explicit casts. **BUT** beware that it can cause issues and difficult errors to debug. Use them wisely.

5.1 42sh

File Tree

```
./
├── Makefile.am  (to submit)
├── configure.ac  (to submit)
├── src/
│   ├── **/
│   │   ├── *.{c,h}  (to submit)
│   │   └── Makefile.am  (to submit)
│   └── Makefile.am  (to submit)
└── tests/
    └── *  (to submit)
```

Compilation : Your code must compile with the following flags

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

Autotools

- all: Produces the src/42sh binary

- check: Runs the testsuite
- clean: Deletes 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)

5.2 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(3) 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(3) functions to execute any of the builtin commands this subject mandates.

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

5.3 Compilation

For your project you will need to compile with the GNU Autotools build system. Have a look at the documentation or watch the conference replay.

An article is available on the trove to help you configure your build system.

With Autotools, your 42sh will be compiled using the following commands:

```
42sh$ autoreconf --install
...
42sh$ ./configure
...
42sh$ make
```

Be careful!

Make sure that no building dependency is necessary to build your project (ex: Criterion) or else you risk having a failed submission.

Consider that the environment in which your project will be tested has nothing.

6 Assignment

6.1 Step 1

6.1.1 Getting started

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

Lexer / Parser

For this step, 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 cannot afford to wait until you have a fully functional lexer to start writing your parser. Likewise you cannot 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 step. Even if it means rewriting a better lexer and parser later.

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

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

6.1.2 Usage

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

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

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

- It must 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 must 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 through stdin
42sh$ cat script.sh | ./42sh
Input through stdin
```

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 `--pretty-print` option, introduced elsewhere in this subject.

Tips

Consider using `fmemopen(3)` to handle string inputs as FILEs.

Pitfalls:

- Implement all three ways of reading input as we will test them.

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 writing 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. To visualize the AST you can use visual tools like Graphviz.

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

What you can do is printing an output you can reparse. By doing that, you can test the output by giving it as argument to your 42sh.

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 does not 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. Do not restrain yourself and feel free to add any information you want.

6.1.3 Exit Status and Errors

You have to handle errors as described in the corresponding section of the SCL.

Going further...

In 42sh, we use the exit code 2 for command line argument and grammar errors.

6.1.4 Simple commands

Tips

Most features have a few lines of grammar at the top, which describes what the minimum grammar requirements for this feature are.

Please read the section of the guide about shell syntax to learn more.

To avoid copying the entire shell grammar for every feature, every grammar block only shows the changes compared to the previous feature.

In case this evolution is unclear, the guide contains a full shell grammar.

```
input =
    list '\n'
  | list EOF
  | '\n'
  | EOF
  ;

(* Of course, many of these rules makes poor sense for now... *)
list = and_or ;
and_or = pipeline ;
pipeline = command ;
command = simple_command ;
```

(continues on next page)

(continued from previous page)

```
simple_command = WORD { element } ;
element = WORD ;
```

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

- a **lexer** which produces `word` tokens
- a **parser** that accept `word` tokens and produces at least a `simple_command` AST node
- an **execution module** which travels your AST and executes the `simple_command` node, waits for its status code, and returns it

Tips

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

6.1.5 Command lists

```
(* list makes more sense now, isn't it ? *)
list = and_or { ';' and_or } [ ';' ] ;
and_or = pipeline ;
pipeline = command ;
command = simple_command ;
```

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 represented in a single `command_list` AST node
echo foo; echo bar

# just like this one with a semicolon at the end
echo foo; echo bar;
```

In order to handle command lists:

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

6.1.6 "If" commands

```
(* command can now also be a shell_command *)
command =
    simple_command
  | shell_command
  ;

(* for the time being, it is limited to a single rule_if *)
```

(continues on next page)

(continued from previous page)

```
shell_command = rule_if ;

rule_if = 'if' compound_list 'then' compound_list [else_clause] 'fi' ;

else_clause =
    'else' compound_list
  | 'elif' compound_list 'then' compound_list [else_clause]
  ;

compound_list = and_or [';'] {'\n'} ;
```

You have to handle `if` commands, as in the SCL):

- 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`
- your execution module must evaluate the condition, and run the `then` or the `else` compound_list depending on its exit code

Pitfalls:

- The condition and body of `if`s are *compound lists*. We'll implement them in the next part, but note how we do not simply reduce the rule to `and_or`. We need separators to delimit commands and recognize `if`'s separator tokens such as `then` and `fi`.

6.1.7 Compound lists

```
compound_list =
    {'\n'} and_or { ( ';' | '\n' ) {'\n'} and_or } [';'] {'\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
```

(continues on next page)

(continued from previous page)

```
    echo b; echo c
fi
```

In order to handle compound lists:

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

Tips

This tip only applies if you are writing 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.

6.1.8 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.

6.1.9 True and False builtins

As builtins, `true` and `false` are parsed as simple commands, but are not executed in the same manner: they are directly executed inside your shell, without requiring `fork` or `exec`.

`true` (resp. `false`) must do nothing and return 0 (resp. 1).

6.1.10 The echo built-in

The `echo` builtin 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` enable the interpretation of `\n`, `\t` and `\\` escapes.
- `-E` disable the interpretation of `\n`, `\t` and `\\` escapes.

Going further...

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

6.1.11 Comments

Implement the recognition of comments. As specified by the SCL, a comment always start with a # and can only be on one line.

```
# This is a comment echo "Hello world"
# This is also a comment
```

Be careful!

The # character is not considered as the beginning of a comment if it is quoted, escaped or is not the first character of a word.

```
42sh$ echo \#escaped "#"quoted not#first #commented
#escaped #quoted not#first
```

6.2 Step 2

6.2.1 Redirections

```
command =
    simple_command
  | shell_command  { redirection }
  ;

simple_command =
    prefix { prefix }
  | { prefix } WORD { element }
  ;

prefix = redirection ;

redirection = [IONUMBER] ( '>' | '<' | '>>' | '>&' | '<&' | '>|' | '<>' ) WORD ;

element =
    WORD
  | 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.

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 does not 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 scarce 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 does not need the saves of file descriptors you made for redirections.

  You can configure a file descriptor to be automatically 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"`

6.2.2 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.

6.2.3 Negation

```
pipeline = ['!'] command { '|' {'\n'} command } ;
```

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

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

You will need to:

- Handle `!` as a reserved word (not an operator!) 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.

6.2.4 "While" and "until" commands

```
shell_command =
    rule_if
  | rule_while
  | rule_until
  ;

rule_while = 'while' compound_list 'do' compound_list 'done' ;
rule_until = 'until' compound_list 'do' compound_list 'done' ;
```

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

6.2.5 Operators

```
and_or = pipeline { ( '&&' | '||' ) {'\n'} pipeline } ;
```

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

6.2.6 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 disagree, weird bugs will ensue.

6.2.7 Variables

```
prefix =
    ASSIGNMENT_WORD
  | redirection
  ;
```

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

`ASSIGNMENT_WORD` is a special `WORD` as specified by the SCL.

You do not 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`

The following environment variables[1] must also be properly expanded:

- `$OLDPWD`
- `$PWD`
- `$IFS`

Tips

You do not have to handle IFS spliting yet as it will be a step4 feature.

Pitfalls:

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

[1] To learn more about environment variables, you can take a look at the SCL section about them.

6.2.8 “For” commands

```
shell_command =
    rule_if
  | rule_while
  | rule_until
  | rule_for
  ;

rule_for =
    'for' WORD ( [';'] | [ {'\n'} 'in' { WORD } ( ';' | '\n' ) ] ) {'\n'} 'do' compound_list
↪'done' ;
```

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

6.3 Step 3

6.3.1 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 normally.

“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 does not, as it is an external command. env -i ensures the PWD
# environment variable is not passed down, and avoids executing a potential
```

(continues on next page)

(continued from previous page)

```
# 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.

"dot"

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

"unset"

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

6.3.2 Command blocks

```
shell_command =
    '{' compound_list '}'
  | rule_if
  | rule_while
  | rule_until
  | rule_for
  ;
```

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; }
```

6.3.3 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. Don't forget to take a look at the SCL part on function definition.

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
```

6.3.4 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.

6.3.5 Subshells

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

Subshells run commands in a new process.

```
42sh$ a=sh; (a=42; echo -n $a);echo $a
42sh
```

Please refer to the SCL specification.

6.4 Step 4

6.4.1 “Case” commands

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

rule_case = 'case' WORD {'\n'} 'in' {'\n'} [case_clause] 'esac' ;

case_clause = case_item { ';;' {'\n'} case_item } [';;'] {'\n'} ;

case_item = ['('] WORD { '|' WORD } ')' {'\n'}  [compound_list] ;
```

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

6.4.2 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 do not have to follow the behavior specified in the SCL when the alias ends with trailing spaces, nor the alias listing (when called without argument).

```
# 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
```

6.4.3 Field Splitting

Implement Field spliting as specified by the SCL.

6.5 Advice

42sh is not an easy project to implement, let alone to debug. This section highlights some techniques and features meant to ease your development and debugging.

6.5.1 Test suite

You might have realized how important a *strong* test suite 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 test suite 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 whose name matches the test category.

For instance:

```
42sh$ ls -l tests/categories
total 20K
drwx------   2 login_x epita 4,0K 2023-01-03 18:42 echo/
drwx------   2 login_x epita 4,0K 2023-01-03 18:42 pipes/
drwx------   2 login_x epita 4,0K 2023-01-03 18:42 simple_commands/
```

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 do not need to store that of the latter in files. However you can make your script generate the expected output into a file.

6.5.2 Build Systems

Using a `build system` for 42sh **is not** a constraint. Use it wisely to integrate various development tools and coordinate everything in one place. Incorporate some rules into your build system to gain efficiency: generate documentation, bindings, lauch a test-suite, linting...

6.6 Bonus features

6.6.1 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.

6.6.2 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.)

6.6.3 Arithmetic expansion

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

You have to handle variables and the following opperators: `-`, `+`, `*`, `/`, `**`, `&`, `|`, `^`, `&&`, `||`, `!` and `~`. You do not 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.

6.6.4 Here-Document

```
redirection =
    [IONUMBER] ( '>' | '<' | '>>' | '>&' | '<&' | '>|' | '<>' ) WORD
  | [IONUMBER] ( '<<' | '<<-' ) HEREDOC
  ;
```

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 easiest 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 does not exceed the capacity of the pipe (your process will block as soon as the pipe is full).

6.7 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, and you will not get any bonus points for implementing these.

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

6.7.1 Prompt

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

6.7.2 Job Control

```
list = and_or { ( ';' | '&' ) and_or } [ ';' | '&' ] ;

compound_list = {'\n'} and_or { ( ';' | '&' | '\n' ) {'\n'} and_or } [ ';' | '&' ] {'\n'} ;
```

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

7 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
  - https://www.gnu.org/software/bash/manual/
  - https://zsh.sourceforge.io/Doc/
  - http://www.kornshell.com/doc/
  - https://www.tcsh.org/
- **Open Source Shells**
  - https://github.com/emersion/mrsh/
  - https://github.com/Swoorup/mysh/
  - https://github.com/brenns10/lsh
  - https://github.com/Fedjmike/tush (not POSIX)
- **Crafting Interpreters**
  - https://craftinginterpreters.com/contents.html
- **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://www.lrde.epita.fr/~adl/dl/autotools.pdf
  - https://www.gnu.org/savannah-checkouts/gnu/autoconf/manual/autoconf-2.71/html_node/index.html
  - https://www.gnu.org/software/automake/manual/automake.html
- **Documentation (man/Doxygen):**
  - https://www.doxygen.nl/index.html

8 Epilogue

8.1 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

8.2 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

I must not fear. Fear is the mind-killer.