Pediluve — Tutorial D2 AM

version **#dirty**

Assistants C/Unix 2024 <assistants@tickets.assistants.epita.fr>

Copyright

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

Contents

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

1 Shell script

1.1 What is a shell script?

A shell script is simply a file in which you write commands that can be read by your shell. Commands will be executed one after another, thus you can consider a shell script as an interpreted program.

1.2 Example

Given this file, named `simple_script.sh`:

```
echo Hello
ls
cd ..
ls
cd -
ls
```

You can execute it with your shell (`sh`) by doing:

```
42sh$ sh simple_script.sh
```

Which will output something like:

```
42sh$ tree
.
   directory_1
       simple_script.sh
2 directories, 1 file
42sh$ cd directory_1
42sh$ sh simple_script.sh
Hello
file_1 file_2
directory_1
file_1 file_2
```

2 Comments

In shell, the character used to write a comment is the '#'. It must be preceded by a whitespace in order to be considered as such. It is possible to comment out a whole line, by putting the '#' at the beginning of the line. It can also follow a line of code that will still be executed.

```
42sh$ # This is a comment and this won't be executed
42sh$ # echo "This is a comment as well"; This echo won't print
42sh$ echo "You see this line" # echo "You don't see this one"
You see this line
42sh$ echo "You see this line"#"You will also see this"
You see this line#You will also see this
```

(continues on next page)

(continued from previous page)

```
42sh$ echo "#" # This will print '#' as is it surrounded by quotes
#
```

The '#' character has other uses when quoted or escaped and can be used to manipulate strings as well.

3 Shebang

3.1 Explanation

There is one special comment that you need to know about. It is called the "Shebang" (abbreviation for Sharp Bang '#!'). It must be on the first line of the script and it tells the interpreter which program to use when running the script.

```
#!/bin/sh
#! /bin/sh
#!/usr/bin/env -S sed -f
```

The above lines, if placed at the very top of the file are valid shebangs. It means that when calling `./script.sh`, it will call the program specified after the shebang (`/bin/sh script.sh` for instance).

You can bypass the shebang by explicitly giving the program to execute the script (`bash script.sh` if you want to use `bash`), and in this case the shebang is just a basic comment that is ignored.

It is a common practice to use `#!/bin/sh` as a shebang for shell scripts. This is fine, however, the best pratice is to invoke `#!/usr/bin/env`. For example, `#!/bin/sh` would be written as `#!/usr/bin/env sh`. Actually, `/bin/sh` does not refer to the `sh` shell[1], but is a symbolic link to another shell, usually `bash`, with *POSIX* mode activated.

Be careful!

Some shells may not fully respect *POSIX* even when *POSIX* mode is activated. To be sure that your script is *POSIX* compliant, you may run your script with `dash`. As a result, we advise you to test your script with `dash` when you are aiming to be *POSIX* compliant (i.e. when using `#!/bin/sh`). Example:

```
42sh$ dash script.sh
```

Going further...

POSIX (Portable Operating System Interface) is a family of standards which purpose is to homogenize the API used by UNIX Operating Systems. The goal is to make compliant *POSIX* programs really easy to port on other Operating Systems than the one they were written on (mostly Linux). *POSIX* defines several things including, among others:

- *C* API
- *Shell*[2]
- Filenames

[1] fully named `Bourne shell`

- Regular expressions

[2] The last version of the Shell command language can be found here: https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html

3.2 Other Shebangs

The shebang does not **have** to be a shell, it can be any other program. In order to run a script with a specific program, you can use `#!/path/to/program`.

Sadly there is no real way to know where a specific program is installed on a system. To address that, the usual way is to use `env(1)` program located at `/usr/bin/env`. The shebang looks like: `#!/usr/bin/env $program`.

If you want your script to be executed with `bash`, write `#!/usr/bin/env bash`. Same goes for Python: `#!/usr/bin/env python` and you can specify a version with `#!/usr/bin/env python3.8`.

As a result, a good practice is to only have shebangs like `#!/bin/sh` or `#!/usr/bin/env $program`

Another example, you can create a self reader script, by specifying a pager after the shebang.

```
42sh$ cat self_reader.sh
#!/usr/bin/env less
Hello,
These lines will be displayed thanks to less
42sh$ ./self_reader.sh # Will launch less with the script
```

4 Builtins

4.1 Introduction

The commands you use every day can either be *programs*, in which case they do not depend on the shell you are using, or they can be *builtins*. Builtins are commands that are directly implemented inside the shell. Builtins are needed for two reasons. First of all, a builtin will execute faster than a normal command (because external commands launch other processes). Second, a builtin can change values of internal shell variables, which a normal command does not have access to since children cannot modify their parent's environment.

In order to know if a command is a builtin you can use `type` or `command -v`.

```
42sh$ type cat
cat is /bin/cat
42sh$ type cd
cd is a shell builtin
42sh$ command -v cat # Not a builtin
/bin/cat
42sh$ command -v cd # A builtin
cd
```

The commands you are going to see now are **not** the only `bash` builtins. Actually, builtins depend on the shell you are using and you can have the complete list of them by looking at the `man` or `info` pages of this shell or typing `help` in `bash`.

4.2 `echo`

`echo` simply displays lines of text it is given as parameter.

```
42sh$ echo 'This is a line of text'
This is a line of text
```

4.3 `cd`

You have probably already used `cd`. This builtin changes the current directory by moving into the one given in parameter. It modifies the internal variables `$PWD` and `$OLDPWD`.

Going further...

When moving inside your filesystem, you might face situations where you want to save your navigation folders. The `cd -` command changes the current directory to the last one visited.

4.4 `true` and `false`

These builtins respectively have a return code of 0 (for `true`) and something else than 0, usually 1 (for `false`). They are not doing anything besides returning those values.

4.5 `source`

The `source` builtin executes commands from a filename provided as parameter. Positional parameters can be passed to the script. `source` will execute the script in your current shell, so every environment modification will persist. It is especially useful when you modified your `.bashrc` and you want to apply those modifications to your current shell (instead of closing your shell and opening a new one).

```
42sh$ cat script.sh
#!/bin/sh

VARIABLE=value
42sh$ ./script.sh
42sh$ echo $VARIABLE

42sh$ source script.sh
42sh$ echo $VARIABLE
value
```

Going further...

`.` is an alias to `source`. This means `. script.sh` is equivalent to `source script.sh`. However, it

is important to note that `source` is **not** *POSIX* standard, while `.` is.

4.6 `help`

This builtin prints the list of all builtins with their options and parameters. If you type `help name` you will have more information about the builtin `name`.

```
42sh$ help true
true: true
    Return a successful result.

    Exit Status:
    Always succeeds.
```

4.7 Exercise

4.7.1 Goal

It is now your turn to write your own shell script. Your script will be named `hello.sh`. It will simply write "Hello World!" on the standard output. It must be callable with `./hello.sh` (which must call `/bin/sh hello.sh`). Do not forget to set the correct permissions in order for your script to be executable.

Tips

Using `<command> | cat -e` will print a `$` where there are linefeeds in the output of `<command>`. This will help you make sure that your format is correct.

4.7.2 Example

```
42sh$ ./hello.sh
Hello World!
42sh$ ./hello.sh | cat -e
Hello World!$
```

Going further...

What do you think will happen when you run `./script.sh` but no shebang has been specified? For example on `bash` or `ksh`, the script will be called with the shell itself, but with `tcsh` or `csh`, the script will be called with `/bin/sh`.

5 Display and comparison

5.1 Display commands

5.1.1 `echo`

The `echo` builtin is one of the simplest and yet a very useful command of the shell. It displays the text you pass as argument and then prints a line break. Use the `printf` or `echo -n` to avoid printing the line break.

```
42sh$ echo -n toto
toto42sh$ echo toto
toto
42sh$
```

5.1.2 `cat`

The `cat` command can be used to display the content of the files you pass as arguments. `cat` comes from *conCATenate*, as the content of the files is effectively concatenated on the display.

Without argument, `cat` will read on the standard input.

Here are some useful options:

- '`-n`': display the absolute line number in front of each line.
- '`-e`': display non-printable and extended characters alternatively, such as the line break which is represented by the '$' character.

5.1.3 `less` and `more`

The `less` and `more` commands are **pagers**: they display their input nicely on the screen and allow the user to browse through screens of text.

If you do not pass the files to display to the pager, they will try to paginate the standard input. For now your standard input is interactive (it is your keyboard), so they will refuse to paginate it.

`more` is quite primitive, it does not allow scrolling back, and has limited search functionality. When all the content of the input has been displayed, it quits, without cleaning the screen.

`less` is an advanced pager, allowing forward and backward movement and has much more features compared to `more`. `less` share some similarities with Vim, such as some of the ex commands and movement keys.

You can search in the input using the '/' key on both pagers.

5.2 Comparison commands

When you work with text files, you may want to do regular backups or keep history. You may then get to a point where your last modifications to the files break something or your program stops working. At this point it would be very interesting for you to view the differences between your files and a set of reference files, in order to identify what may have caused the bug.

5.2.1 `diff`

The `diff` command compares the two files passed as arguments and displays the lines that have been removed, inserted and modified. The default output is not the most commonly used since it is not easily readable, we thus usually use the '`-u`' option to use the unified diff format.

5.3 Exercise

1. Display the `/etc/passwd` file using the `cat` command, enable the line numbers and non-printable characters display.
2. What is the difference between `cmp` and `diff`?

6 Variables

6.1 Introduction

Variables are a way to store data into a named container. The container's name will be used to create, access and modify the value inside the variable. The syntax to declare a variable is `name=value`.

```
my_var=hello # Valid variable
var10=9 # Valid again
1var=hello # Not valid since its name begins with a digit
var =hello # Not valid, it will complain that var is not found
this_var= hello # Not valid, it will complain that hello is not found
```

Be careful!

There is no space on either side of the '=' sign. The identifier must begin with an uppercase or lowercase alphabetic character or an underscore '_' followed by 0 or more alphanumeric characters or underscores '_'.

Be careful!

In shell, **everything** is a string.

In order to access the value of the variable, we use the `$name` or `${name}` syntax in order to avoid confusion. The '$' tells the shell to look for the value stored inside `name`.

```
var=Best
echo $var # Will print Best
```

Variable's value can be changed. This can be done the same way as declaring them.

```
var=value1
echo $var # Print value1
var=value2
echo $var # Print value2
```

6.1.1 Exercise

Goal

For this exercise, every script **must** be executable.

Create a script named *create.sh* that prints "My frais is " followed by their name. To do so, you will create a variable named `my_local_frais`, set its value to "Javotte" and print it.

```
42sh$ MY_ENV_FRAIS=Fripouille ./create.sh | cat -e
My frais is Javotte$
```

Now that you have created your first variable, you can create another script named *use.sh* that will only print the variable `MY_ENV_FRAIS` after "My frais is ". If you want to print your frais' name, you can execute the script like this:

```
42sh$ MY_ENV_FRAIS=Fripouille ./use.sh | cat -e
My frais is Fripouille$

42sh$ ./use.sh | cat -e
My frais is $
```

Finally, create a script named *edit.sh*. It will first print your frais' name the same way as *create.sh*, then it will print another line containing "My frais is now " followed by "Pulpa". You should not create another variable, but edit the value of `my_local_frais`.

```
42sh$ ./edit.sh | cat -e
My frais is Javotte$
My frais is now Pulpa$
```

6.2 Special variables

These variables are inside your shell and cannot be modified using the `name=value` syntax. Their value only depend on the context. Here are some examples:

- `$?`: In shell, every command returns a value. `$?` contains the last command's return value.
- `$$`: Every process has an ID (PID: Process ID). `$$` corresponds to the PID of the current shell.
- `$#`: You can pass arguments to your program via the syntax `./script.sh arg1 arg2`. `$#` contains the number of parameters passed to the script.
- `$*`: This is the list of all the parameters passed to the script.
- `$@`: Same as above but has a special unique behavior when quoted (you will see that later on).

- `$n`: This holds the value of the n^{th} parameter of the script (1 <= n <= 9).
- `$0`: This is the script's name.
- `${n}`: This holds the value of the n^{th} parameter (1 <= n, brackets are mandatory when n >= 10).

6.2.1 Exercise

Goal

For this exercise, every script **must** be executable.

Write a script named *print.sh* that prints:

- The number of parameters passed to it
- The list of all the parameters
- The first parameter
- The second parameter
- The third parameter
- The thirteenth parameter
- The name of the script

Each element should be printed on a new line.

Examples

```
42sh$ ./print.sh 1 2 3 4 5 6 7 8 9 10 11 12 13 | cat -e
13$
1 2 3 4 5 6 7 8 9 10 11 12 13$
1$
2$
3$
13$
./print.sh$
```

Going further...

What happens when you try to print the third parameter and only pass two to the script?

6.3 The `shift` command

The `shift` command allows you to interact with the positional parameters. It can take an argument, which default value when not specified is 1. The `shift` command will move the positional parameters (`$1`, `$2`, `$3`, ...) to the left by the argument given to `shift`. If you have 5 parameters, and you do `shift 2`:

- `$1` takes the value of `$3`.
- `$2` takes the value of `$4`.
- `$3` takes the value of `$5`.
- The old values of `$1` and `$2` are thrown away.
- `shift` will also decrease the value of `$#`.

```
#!/bin/sh

echo First parameter: $1
echo Number of parameters: $#
shift
echo First parameter : $1
echo Number of parameters: $#
```

```
42sh$ ./script.sh 1 2 3
First parameter: 1
Number of parameters: 3
First parameter : 2
Number of parameters: 2
```

Going further...

What happens when you try to pass a number greater than `$#` to `shift`? What happens when you pass a negative number to `shift`?

6.4 Internal variables

Your shell also has what we call *environment variables*. These variables provide you with some useful information about your current environment/system. You can even change their value in order to configure your shell. Here are some examples:

- `$PATH`: this variable contains a list of directories in which the shell looks for commands.
- `$USER`: name of the current user.
- `$PS1`: the prompt configuration. The prompt is what you see on your shell even though you have not typed anything yet (`42sh$` in our examples). It can be relevant to add information in it like the current directory, the username, or even the return code of your last command.
- `$PWD`: the current working directory.
- `$EDITOR`: the user's preferred editor (`vim`/`emacs`).
- `$SECONDS`: this variable auto increments every second and stores the number of seconds the script has been running. You can set it to 0 and it will still auto increment.

Going further...

Environment variables used by bash are explained in `man 1 bash` in the section *Shell Variables*.

6.5 Using environment variables

Now it is your time to play. First of all, set the value of `$SECONDS` to 0. Once you have done that, print your `$PATH` and see what it contains.

Your `$PS1` is currently pretty basic... Try to see what `\w`, `\W`, `\h` and `\u` do inside your `$PS1`.

Going further...

The `\w`, `\W` and the others are special and belong to the PS1. See `man 1 bash` and look for the *PROMPTING* part for more of these.

You might already have chosen your favorite editor between `vim` and `emacs`. You can set your favorite editor inside the variable `$EDITOR` so that every time your system wants to open something with your editor, it picks the one you like the most.

Now, print the value of `$SECONDS`. This is the time you have spent playing with those different variables.

6.6 Passing variables to commands

When you want to set a variable just for a command or a script, you can use the syntax `var=value cmd`.

```
#!/bin/sh

echo "$var"
```

```
42sh$ echo "$var"

42sh$ ./script.sh

42sh$ var=hello ./script.sh
hello
42sh$ echo "$var"

42sh$
```

6.7 Exporting variables

Now that you have set all your variables, try to spawn a new shell by simply typing `bash`. You will see that the variables you have set earlier have lost their values. Why?

Let us say you reboot your computer and start a new shell. The variable would be lost because the shell has no way of storing the value you put inside the variable. This is why, if you want to save the value of special variables, you must write `export name=value` in a file called `.bashrc`. This configuration file is read every time a shell is started. Putting your environment variables inside allows them to be accessible everywhere.

7 Quoting

7.1 Full quoting

Also called *strong quoting*, full quoting preserves the literal value of each character between two single quotes.

```
42sh$ var=10
42sh$ echo '$var'
$var
42sh$ echo '\$var' # escaping will not change a thing
\$var
```

Note that you cannot nest single quotes either.

7.2 Partial quoting

Also called *weak quoting*, behaves the same way as strong quoting with the exception of some special characters: `$`, `` ` `` and `\`.

In other words, it allows *variable expansion*:

```
42sh$ var=10
42sh$ echo "$var"
10
42sh$ echo "\$var" # here escaping cancels the behavior of special chars
$var
42sh$ echo "Partial quoting escaping \
> can also cancel newlines"
Partial quoting escaping can also cancel newlines
42sh$ echo "See
> that ?"
See
that ?
```

7.3 Why should all variables be quoted?

```
42sh$ filename='I love bash'
42sh$ touch $filename
42sh$ ls -l
-rw-rw-r--. 1 login login 0 Aug  3 06:21 bash
-rw-rw-r--. 1 login login 0 Aug  3 06:21 I
-rw-rw-r--. 1 login login 0 Aug  3 06:21 love
42sh$ touch "$filename"
42sh$ ls -l
total 0
-rw-rw-r--. 1 login login 0 Aug  3 06:21 bash
-rw-rw-r--. 1 login login 0 Aug  3 06:21 I
-rw-rw-r--. 1 login login 0 Aug  3 06:57 I love bash
-rw-rw-r--. 1 login login 0 Aug  3 06:21 love
```

As you can see without partial quoting the `filename` variable is split and `touch` is executed with three different arguments.

Always quote your variables, this will avoid you many bugs and security issues[1].

8 Conditions

8.1 `if` syntax

In shell, `if` is a keyword used to test an expression. If it evaluates as `true`, a particular statements is executed. Otherwise, it is skipped. The condition to test is a command and its return value is checked:

- 0 being `true`.
- Everything else being `false`.

```
if command1
then
    do_something
elif command2
then
    do_something
else
    do_something
fi

# cleaner format

if command1; then
    do_something
elif command2; then
    do_something
else
    do_something
fi
```

8.2 Test commands

Two builtins are often used in conjunction with the `if` construct: the `[` and `test` commands. These commands have exactly the same behavior but not the same syntax.

Be careful!

It may sound strange, but contrary to what you may think, `[` **is** a builtin command, not a piece of syntax! Thus, if you forget the whitespace, it will not work.

```
42sh$ toto=42
42sh$ [ toto -a toto
[: ']' expected
```

[1] See this discussion for more information about security issues with non quoted variables.

You may wonder what is `]` if `[` is a command. It is an argument, and it is **necessary** if you use the `[` command.

As all commands, they take parameters:

Options	Meaning
`-n $var`	Check that the variable is not empty (**N**ot zero length).
`-z $var`	Check if the string is empty (**Z**ero length).
`-e path`	Check whether `path` **e**xists.
`-d path`	Check whether `path` exists and is a **d**irectory.
`-f path`	Check whether `path` exists and is a regular **f**ile.
`-x path`	Check whether `path` exists and is e**x**ecutable.
`str1 = str2`	Test if the strings are identical.
`str1 != str2`	Test if the strings are different.
`int1 -eq int2`	Test if two integers have the same value (**Eq**ual).
`int1 -ne int2`	Test if two integers have different value (**N**ot **E**qual).
`int1 -lt int2`	Test if `int1` is **l**ower **t**han `int2`.
`int1 -gt int2`	Test if `int1` is **g**reater **t**han `int2`.
`int1 -ge int2`	Test if `int1` is **g**reater than or **e**qual to `int2`.

There are many other comparison operators, see `man test` for additional information.

Going further...

If you ever need to consult this table, check out the man page for test by typing `man test` in your terminal.

Also note that the `!` negates the test operation.

Here are a few examples of these tests:

```
var='hello'

if [ -n "$var" ]; then
    echo 'var is not null'
fi

if [ 2 -eq 3 ]; then
    echo 'Math is broken'
else
    echo 'As expected'
fi

if test ! -e file ; then
    echo 'file does not exist'
fi
```

8.2.1 Exercise

Goal

This script must take exactly one parameter. If more or less is given, a message should be printed and the script should exit with 1. When the argument is given, it has to print the inside of the file if it is a regular file, and exit with 2 if it is not. The error messages must be as in the example.

Example

```
42sh$ ls
file1 file2 file3 inside.sh
42sh$ ./inside.sh
Sorry, expected 1 argument but 0 were passed
42sh$ echo $?
1
42sh$ ./inside.sh file1 file2
Sorry, expected 1 argument but 2 were passed
42sh$ ./inside.sh no
no:
    is not a valid file
42sh$ echo $?
2
42sh$ cat file2
Content
42sh$ ./inside.sh file2
Content
42sh$ echo $?
0
```

Be careful!

"is not a valide file" is preceded by a tabulation

8.3 Chaining commands

You can chain commands with operators `&&` and `||` which simply corresponds to the `AND` and `OR` logical operators.

These operators are evaluated **lazily**: it means that each term of the operation is evaluated only if it is needed.

```
[ -n "$var" ] && echo 'var is not null' # shorter version than with the if

[ 2 -eq 3 ] && echo 'Math is broken' || echo 'As expected'

[ -e file ] || echo 'file does not exist'
```

Be careful!

In the following example, this is what happens if the variable `var` is null.

```
42sh$ [ -n "$var" ] && echo toto
42sh$ [ -z "$var" ] && echo toto
toto
42sh$ [ -n $var ] && echo toto
toto
42sh$ [ -z $var ] && echo toto
toto
42sh$
```

It is because if the variable is not quoted, it is equivalent to `[ -n ]` or `[ -z ]` and thus the test does nothing and it goes to the next command in the chain.

Here is another example which will not work as expected. Consider that `script.sh` contains the following code:

```
#!/bin/sh
[ "$#" -eq 0 ] && echo 'Invalid arg' || # No argument provided
[ -x "$1" ] && "$1" # if the argument is executable then execute it
```

```
42sh$ ./script.sh /bin/ls # working as expected
file1 file2 ...
42sh$ ./script.sh # no argument provided
Invalid arg
./script.sh: line 3: : command not found
```

For the second part, the result is unexpected. We would have expected the output to be only `Invalid arg` and not to execute the rest of the code. However it still does. The issue here is that the `[ -x "$1" ]` is skipped because conditions are simply evaluated from left to right.

So the code with parentheses corresponds to:

```
( ( ( [ "$#" -eq 0 ] && echo 'Invalid arg' ) || [ -x "$1" ] ) && "$1" )
```

Going further...

```
42sh$ man bash
[...]
3.2.3 Lists of Commands

A list is a sequence of one or more pipelines separated by one of the
operators ‘;’, ‘&’, ‘&&’, or ‘||’, and optionally terminated by one of
‘;’, ‘&’, or a newline.

Of these list operators, ‘&&’ and ‘||’ have equal precedence, followed by
‘;’ and ‘&’, which have equal precedence.
[...]
```

To fix that, we can use code blocks.

8.4 Code blocks

Code blocks are similar to putting parentheses in mathematical operations to change their priority.

The syntax used is curly brackets: `{ echo hello;}`

Here is a simple way to fix the previous script:

```
#!/bin/sh

[ "$#" -eq 0 ] && echo 'Invalid arg' || # No argument provided
  {
    [ -x "$1" ] && "$1" # if the argument is executable then execute it
  }
```

```
42sh$ ./script.sh /bin/ls # working as expected
file1 file2 ...
42sh$ ./script.sh # no argument provided
Invalid arg
```

Working as expected!

8.4.1 Exercise

Goal

This script must take exactly one parameter, if more or less is given, a message should be printed and the script should exit with 1. When the argument is given, it has to print the inside of the file if it is a regular file, and exit with 2 if it is not. The error messages must be as in the example.

`if` constructs are not allowed. You can only use the operators `&&` and `||`.

Example

```
42sh$ ls
file1 file2 file3 inside.sh
42sh$ ./inside_noif.sh
Sorry, expected 1 argument but 0 were passed
42sh$ echo $?
1
42sh$ ./inside_noif.sh file1 file2
Sorry, expected 1 argument but 2 were passed
42sh$ ./inside_noif.sh no
no:
    is not a valid file
42sh$ echo $?
2
42sh$ cat file2
things again
42sh$ ./inside_noif.sh file2
things again
```

(continues on next page)

(continued from previous page)

```
42sh$ echo $?
0
```

Be careful!

"is not a valid file" is preceded by a tabulation

8.5 Case construct

Here is the basic syntax:

```
#!/bin/sh

case "$1" in
    1)
        echo 'one'
        ;;
    2)
        echo 'two'
        ;;
    3)
        echo 'three'
        ;;
    *)
        echo 'sun'
        ;;
esac
```

```
42sh$ ./number_to_litteral.sh 2
two
42sh$ ./number_to_litteral.sh 4
sun
```

In the example above, there is no real advantage of using a `case` over a simple `if`.

You will see later that the `case` construct shines with globbing.

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