Pediluve — Tutorial D3

version **#cc457c03a97aa05a2569d8326567fa158565d2e7**

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 Globbing

1.1 Definition

Sometimes, you need to apply the same treatment to a large number of files. It would be very tedious to list all the files by hand. *Globbing* allows you to express filenames in the form of *patterns*. A pattern is a string of characters, and describes some characteristics of a filename. For this, we use *metacharacters* and *wildcard* characters. These are special characters in the context of globbing, representing one or more other characters.

We are going to go through each *metacharacter* and *wildcard* character and explain what they do.

1.1.1 '?'

First of all, the '?'. It matches 1 character exactly.

```
42sh$ ls # We have 3 files named a, aa and aaa
a aa aaa
42sh$ echo ??
aa
```

As you can see in this example, the `echo` command will not print out two question marks but will expand the '?' to 'match one character'. As we tried to echo two of them, it will try to expand it on the filenames currently inside the directory. `echo ??` can be thus seen as "Print all the filenames that are exactly 2 characters long".

1.1.2 '*'

The '*' character matches 0 or more characters.

```
42sh$ echo * # Similar to ls
glob2.sh glob.c glob.md glob.sh
42sh$ echo *.sh
glob2.sh glob.sh
42sh$ echo *.java
*.java
42sh$ echo ?
?
```

Here you can see several things:

- `echo *` will print all the files in the current directory. It makes sense since the '*' expands to 0 or more characters. Thus, it can match every name.
- `echo *.sh` will print only `glob2.sh` and `glob.sh`. Indeed both these files can be seen as some characters ("glob2" or "glob" for us, that are represented with '*') followed by ".sh".
- In the last two examples you see that globbing does not apply. Actually it does apply, but finds nothing, so it prints out literally.

Be careful!

- Globbing does not apply to hidden files. Hence, if you want to match `.bashrc` for instance, you will have to explicitly express the dot (`echo .*rc` for example).
- The character '/' that separates the elements of a file's *path* is not considered part of the file's *name*, so it will not be matched by '`*`' or '`?`' *wildcard* and will not be included in ranges.

1.1.3 '`[ ]`'

The brackets will match any of the characters between them, but only once.

```
42sh$ ls
a.sh b.sh e.sh y.sh
42sh$ echo [aeiouy].sh
a.sh e.sh y.sh
42sh$ echo [a-e].sh
a.sh b.sh e.sh
42sh$ echo [!abe].sh
y.sh
42sh$ echo [!a-e].sh
y.sh
```

A lot of new possibilities here:

- `echo [aeiouy].sh` will match `a.sh`, `e.sh`, `i.sh`, `o.sh`, `u.sh`, `y.sh`. It will not find `b.sh` so it will not print it.
- `echo [a-e].sh` will match `a.sh`, `b.sh`, `c.sh`, `d.sh` and `e.sh`. The '-' symbol tells the shell to match all characters between the left bound and the right bound (both included).
- `echo [!abe].sh` will match every file that starts with one letter that is **not** 'a', 'b' or 'e', followed by ".sh".
- `echo [!a-e].sh` will match every file that starts with one letter out of the range "a-e", followed by ".sh".

1.1.4 Summary

Here is a list of all *metacharacters* and *wildcard* characters and their meanings:

?	1 character
*	0 or more characters
[aeiouy]	1 of the characters between the brackets
[a-t]	1 of the characters between a and t, limits are included
[!abc]	1 character other than those mentioned

1.2 Disabling globbing

What if now you really want to `echo *`? You need a way to disable globbing. As we saw earlier, if nothing is found, globbing does not apply. But this is not enough.

If you use weak (or strong) quoting, globbing will not apply and you will be able to print wildcard characters.

```
42sh$ echo "*"
*
42sh$ echo '?'
?
```

You can also escape the wildcard characters if you want to avoid quoting but still disable globbing. Escaping a special character deactivates its special meaning.

```
42sh$ echo \*
*
42sh$ echo \?
?
```

1.3 Character classes

Globbing provides us with a way to match classes of characters. They correspond to some often used patterns and automatically adapt to the current `LOCALE` (language related environment variable). Here are a few examples[1]:

- `[[:digit:]]` → Same as `[0-9]`
- `[[:upper:]]` → Same as `[A-Z]`
- `[[:lower:]]` → Same as `[a-z]`

Going further...

You can use globbing in order to generate the `list` in a `for` loop.

```
42sh$ ls
glob_loop.sh tmp1.md tmp1.txt tmp2.md tmp2.txt tmp3.md tmp3.txt
42sh$ cat glob_loop.sh
#!/bin/sh
for i in *.txt; do
    echo "$i"
done
42sh$ ./glob_loop.sh
tmp1.txt
tmp2.txt
tmp3.txt
```

Going further...

[1] See `man 7 glob` for more information.

Globbing makes `case` very powerful if used as a `case` variable:

```
#!/bin/sh

case "$1" in
    *.c)
        echo 'It is a C file'
        ;;
    *.sh)
        echo 'It is a shell script'
        ;;
    *.com)
        echo 'It is a website'
        ;;
    *)
        echo 'Unknown extension'
        ;;
esac
```

```
42sh$ ./extension.sh hello.c
It is a C file
42sh$ ./extension.sh hello.sh
It is shell script
42sh$ ./extension.sh hello.zip
Unknown extension
```

1.4 Exercises

1.4.1 Easy Globbing

Goal

Write a script that matches all files in the directory passed as argument containing a two characters extension. Also note that this extension shouldn't contain any number.

```
42sh$ ls dir
del.sh fact.cs irish macro.m4 todo.txt
42sh$ ./glob_easy.sh dir
dir/del.sh dir/fact.cs
```

1.4.2 Unmatched extension copying

Goal

Write a script that takes some parameters (single characters) and move all the files which extensions are not the ones specified in parameters inside the trash directory. We only consider one character extensions.

```
42sh$ ls
allo.a baba.b bobo.c cp.sh dede.u ele.x ff.z trash
42sh$ ./cp.sh a b c
42sh$ ls
allo.a baba.b bobo.c cp.sh trash
42sh$ ls trash
dede.u ele.x ff.z
```

1.4.3 Extension based removal

Goal

Now that you know how test constructs work, you will be able to write scripts that are based on testing conditions. Write a script that will delete all the files that have the extension specified as the first parameter of the script. If no argument is given, the default extension is "txt". If there is no file to delete, your program must return 1.

Example

```
42sh$ ls
a.c e.txt c.sh glob_remove_shell.sh
42sh$ ./glob_remove_shell.sh
42sh$ ls
a.c c.sh glob_remove_shell.sh
42sh$ ./glob_remove_shell.sh c
42sh$ ls
c.sh glob_remove_shell.sh
```

2 Symbolic links

2.1 Concept

A symbolic link (or *symlink*) is a file that points to another file. This file does not contain the data of the target file but a reference to the location of where the actual data is.

For example, on the PIE, everything stored outside of your AFS is deleted when you shutdown your computer. But your configuration files that need to be in your home, hence outside of your AFS, keep the changes you make. This is because they are symbolic links created at login to files that are in your AFS. If you look in the directory `afs/.confs` you should see the real files.

To create symlinks on Unix you can use the command `ln`:

```
ln -s target_file link_name
```

Tips

- `target_file`: the existing file you want to create a link to.
- `link_name`: the name of the symbolic link you want to create.

You can also do it with the `cp` command by giving it the argument `-s`.

When you delete the symlink, it will not delete the target file. Use the command `ls` with the parameter `-l` to see symbolic links and where they point to. If there are symbolic links you should see something like this:

```
lrwxr-xr-x  1 xavier.login  staff       5 aug 26 12:14 link_name -> target
```

2.2 Exercise

1. Create a symlink named `tmp` in your home that points to `/tmp`.
2. `cd` into the newly created directory.
3. What is your current path?
4. Remove the symlink using its absolute path.
5. What is your current path?
6. `cd` to the current directory (e.g. `cd .`).
7. What is your current absolute path?

3 Processes

3.1 Definition

A process is an instance of a program in execution. It contains the program state (including its code and data). When you launch a program, a process that contains this information is started. Processes are like cells in some way: they are forked by their parent process, they have their own life, they optionally generate one or more child processes, and eventually, they die. Each process also has a unique number called the Process Identifier, or *PID*[1], associated with it.

3.2 Process lifecycle

Processes on a computer form a tree hierarchy: each process has a parent process and, optionally, child processes. The root of the **process tree** is the process with the PID *1*, called `systemd`, which is launched by the kernel when the machine boots. A process whose parent dies before itself becomes the child of `systemd`. You can view the process tree with the command `pstree(1)`:

[1] `credentials(7)` is a great source of information about PIDs.

```
42sh$ pstree -T
systemd-+-.salt-minion-wr---.salt-minion-wr
    |-.sddm-wrapped-+-.sddm-helper-wr---53d9kn2p11hkfvf---i3
    |               `-X
    |-afsd
    |-agetty
    |-alacritty---bash---pstree
    |-aria2-start---aria2c
    |-bluetoothd
    |-dbus-daemon
    |-dhcpcd-+-dhcpcd---dhcpcd
    |        `-2*[dhcpcd]
    |-i3bar---i3status
    |-lldpd---lldpd
    |-machine-state-s---.machine-state-
    |-node_exporter
    |-nsncd
    |-sshd
    |-sssd-+-sssd_be
    |      |-sssd_nss
    |      |-sssd_pam
    |      `-sssd_ssh
    |-systemd-+-(sd-pam)
    |         |-.redshift-wrapp
    |         `-ssh-agent
    |-systemd-journal
    |-systemd-logind
    |-systemd-oomd
    |-systemd-timesyn
    `-systemd-udevd
```

Of course, the output of `pstree` might be very different on your machine. Notice that `systemd` is indeed at the root of the tree. Once a process dies, it does not get removed from the process tree immediately. To remove a dead process from the process tree, the process must have its termination information read by its parent. A *dead process* whose state has not been read by its parent becomes a *zombie*[2].

Going further...

Note that `systemd` will always read its child's termination information.

3.2.1 Daemons

Some processes can run without direct user interaction, as background processes. These are called *daemons*. To communicate with a daemon, you can send it signals as it is not interactive. A daemon is a process whose parent died and is attached to the first user process: `systemd`.

[2] A *zombie* still has an entry in the process tree but has finished its execution. As such it can be considered dead and alive at the same time, hence the *zombie* name.

3.3 Get PID

When you want to manipulate the process running on your computer, you will have to know their *PID* (mostly to kill them). For this you can use different program. `ps aux`, `top` or `htop`. They will all show you every process on your computer. The difference is that `ps` will show you only the process at one moment, when the other will show you the process over time.

3.4 Kill(1)

During your life as engineer, you will sometimes encounter programs that run infinitly. If you want to stop them, you can use one of the commands seen previously to get their *PID* and then use the `kill` command. This command does more than just kill a process, it can also launch signals. Once you have the *PID*, you can stop them by sending the "terminate" signal to them with `kill`: simply write `kill` followed by the *PID*.

4 Upload/Download

Until now, when you were downloading or uploading files on the internet, you were always using some GUI tools (such as your web browser). But as always, you can also do it directly from the command line in your shell. For this, two command exist:

- `wget(1)`: a program that allows you to download from an URL non interactively. It means that it can be run in background task or while being logged out, unlike how you were usually doing with a web browser.
- `curl(1)`: a program that can upload or download data with a server. It supports more protocols than wget. As such it is the only one of the two that can be used to send data.

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