scripts

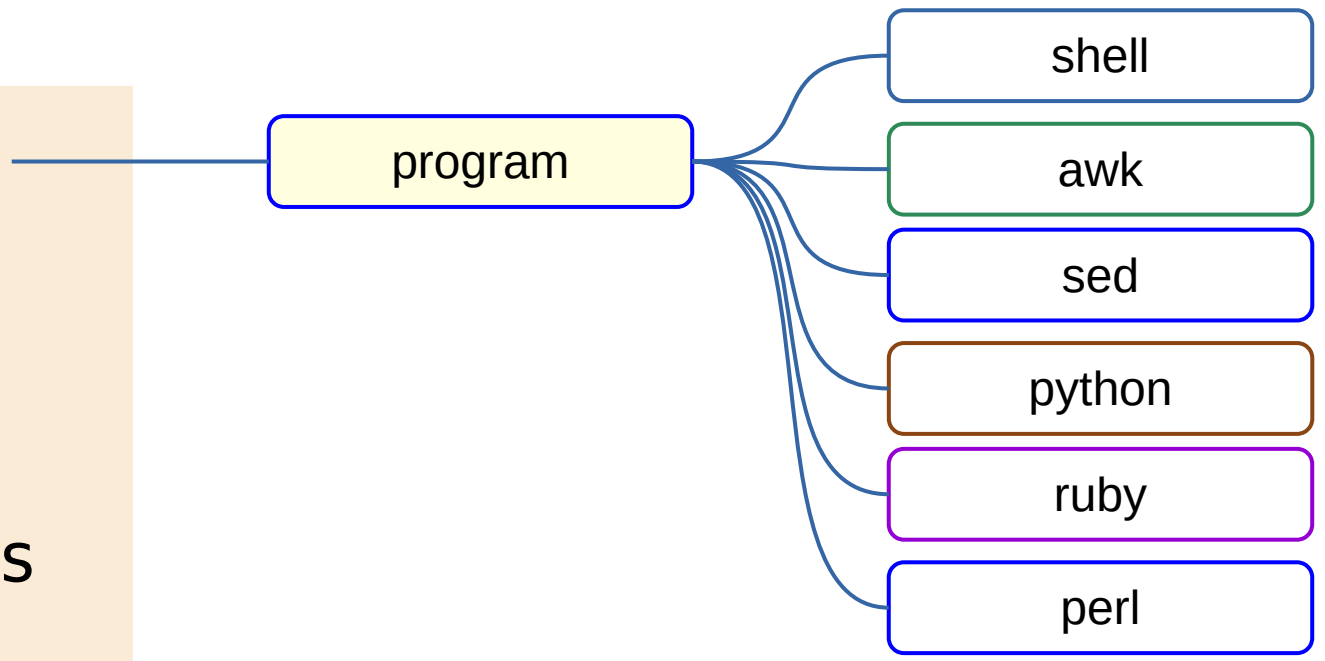
creating your own commands

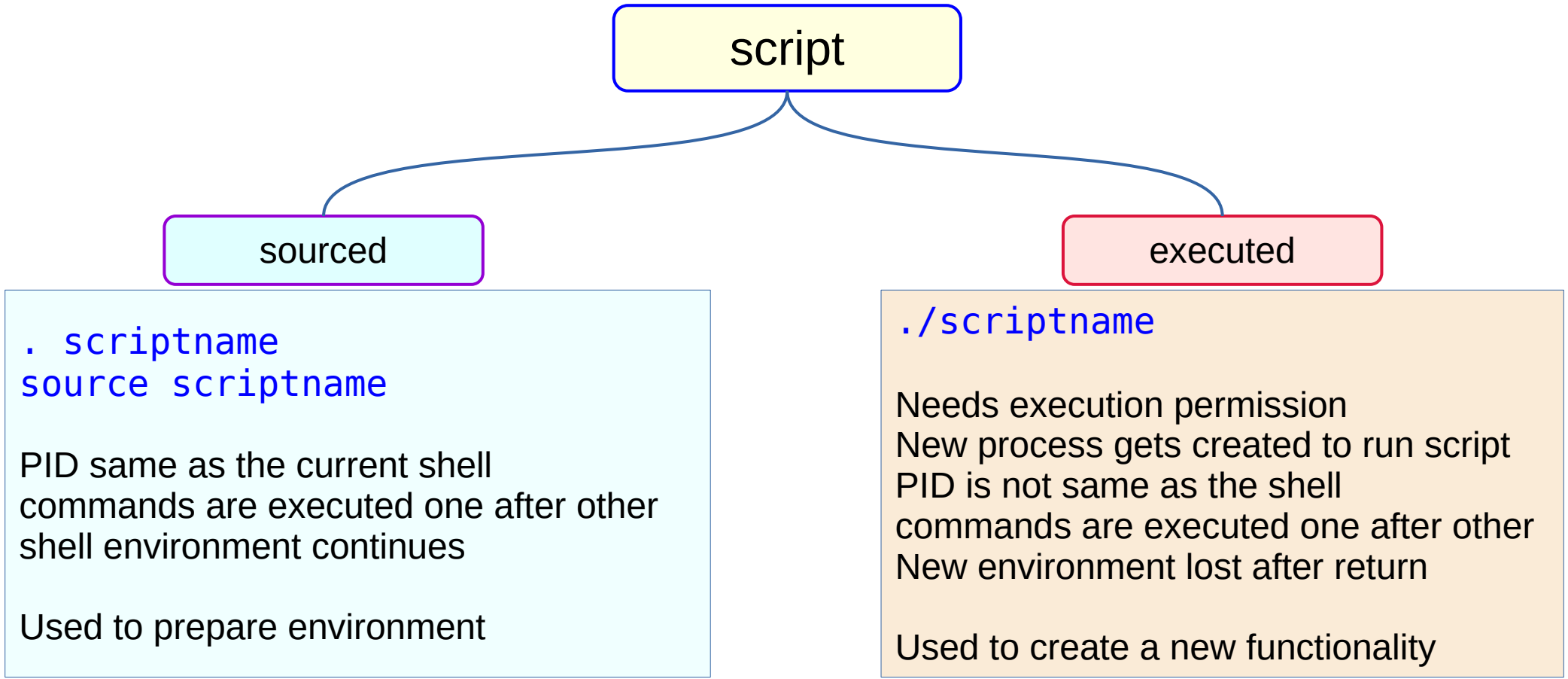
Software Tools Principles

- Do one thing well
- Process lines of text, not binary
- Use regular expressions
- Default to standard I/O
- Don’t be chatty
- Generate same output format accepted as input
- Let someone else do the hard part
- Detour to build specialized tools

Ref: Classic Shell Scripting – Arnold Robbins & Nelson H.F. Beebe

```
#! interpreter
# comments
commands
loops
variables
case statements
functions
```




script
sourced
executed
. scriptname
source scriptname
PID same as the current shell
commands are executed one after other
shell environment continues
Used to prepare environment
./scriptname
Needs execution permission
New process gets created to run script
PID is not same as the shell
commands are executed one after other
New environment lost after return
Used to create a new functionality

Script location

- Use absolute path or relative path while executing the script
- Keep the script in folder listed in $PATH
- Watch out for the sequence of directories in $PATH

bash environment

Login shell

```
/etc/profile
~/.bash_profile
~/.bash_login
~/.profile
```

Non-login shell

```
/etc/bash.bashrc
~/.bashrc
```

Output from shell scripts

- `echo`
  simple
  terminates with a newline if `-n` option not given

  ```
  echo My home is $HOME
  ```

- `printf`
  supports format specifiers like in C

  ```
  printf “My home is %s\n” $HOME
  ```

Input to shell scripts

- `read var`

  string read from command line is stored in `$var`

Shell Script arguments

- `$0` name of the shell program
- `$#` number of arguments passed
- `$1` or `${1}` first argument
- `${11}` eleventh argument
- `$*` or `$@` all arguments at once
- `"$*"` all argument as a single string
- `"$@"` all argument as a separate strings

```
./myscript.sh -l arg2 -v arg4
```

Command substitution

```
var=`command`

var=$(command)
```

command is executed and the output is substituted.

Here, the variable *var* will be assigned with that output.

for do loop

```
for var in list
do
  commands
done
```

commands are executed once for each item in the *list*

space is the field delimiters

set `IFS` if required

case statement

```
case var in
pattern1)
    commands
    ;;
pattern2)
    commands
    ;;
esac
```

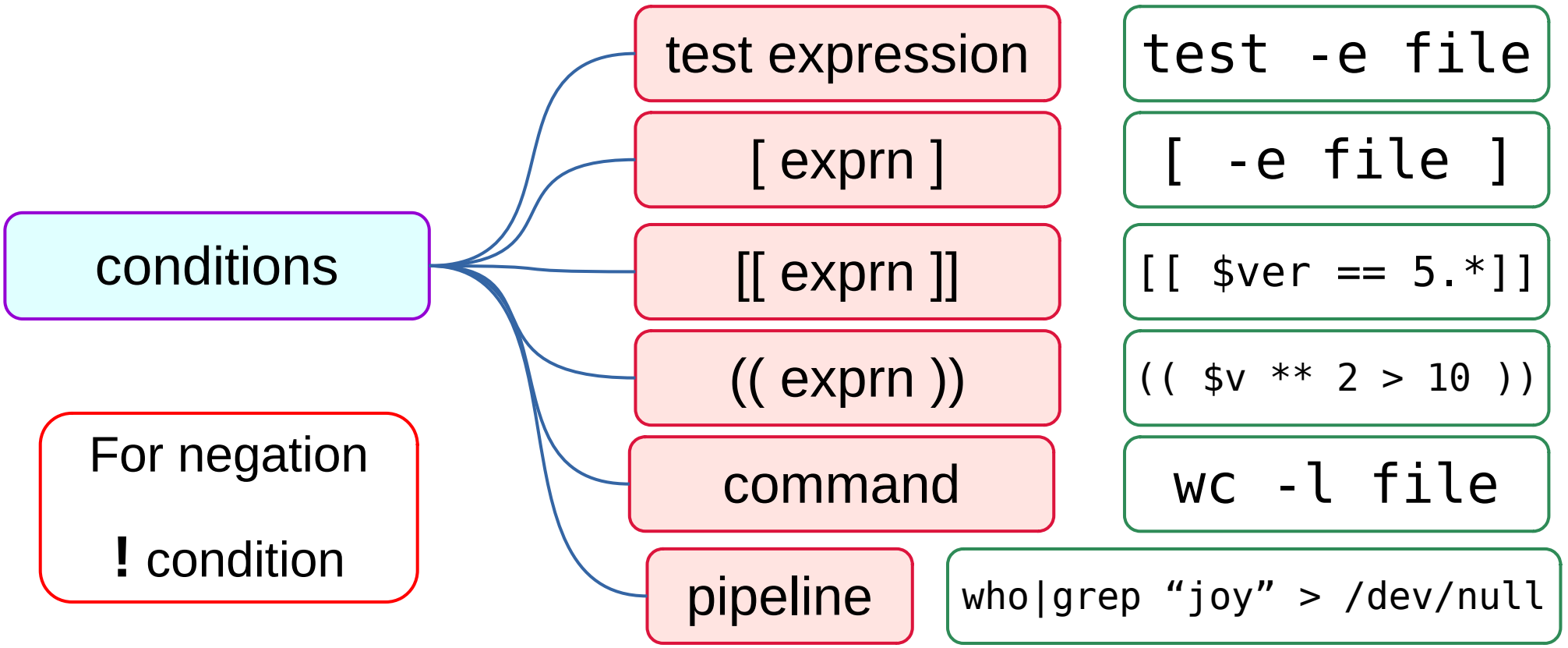
commands are executed each pattern matched for var in the options

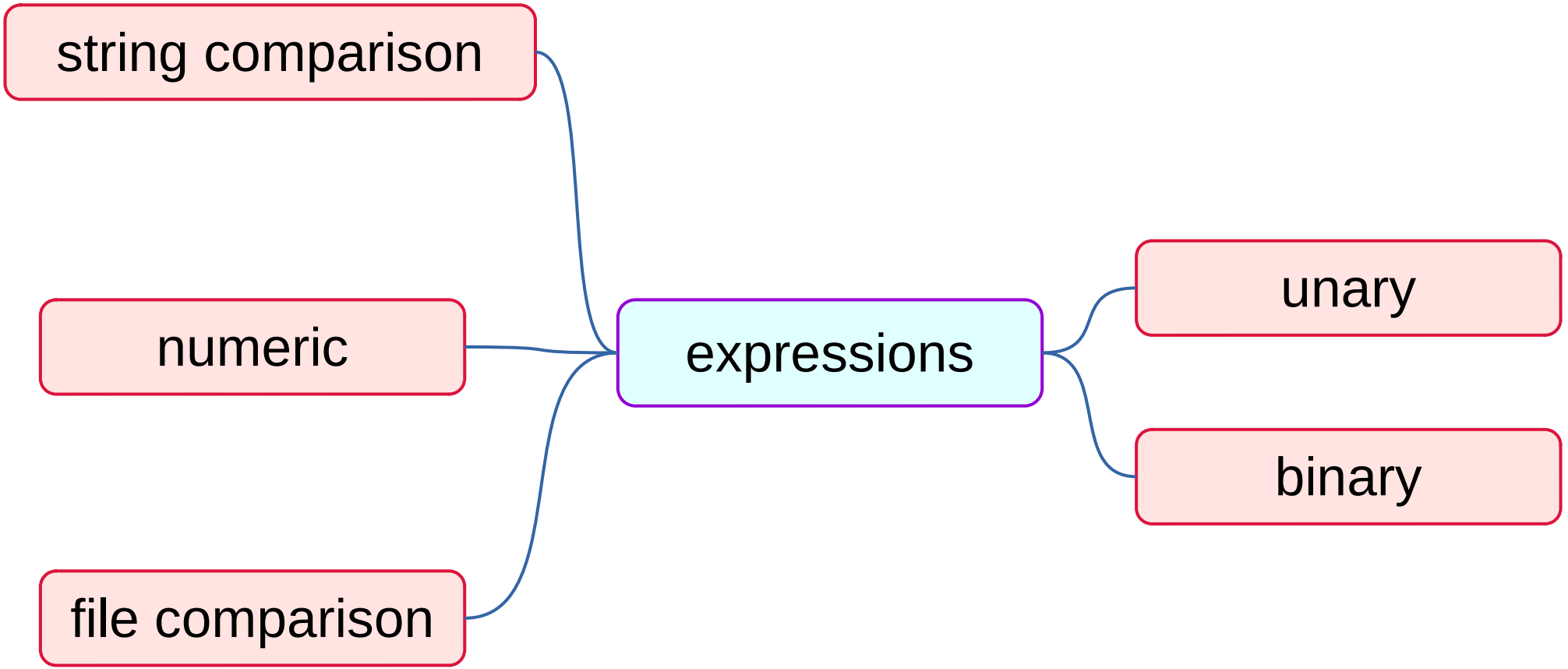
if loop

```
if condition
then
  commands
fi
```

commands are executed only if *condition* returns true

```
if condition; then
  commands
fi
```


conditions
test expression
[exprn]
[[exprn]]
((exprn))
command
pipeline
test -e file
[-e file]
[[$ver == 5.*]]
(($v ** 2 > 10))
wc -l file
who|grep “joy” > /dev/null
For negation
! condition


string comparison
numeric
file comparison
expressions
unary
binary

test numeric comparisons

`$n1 -eq $n2`	Check if n1 is equal to n2
`$n1 -ge $n2`	Check if n1 is greater than or equal to n2
`$n1 -gt $n2`	Check if n1 is greater than n2
`$n1 -le $n2`	Check if n1 is less than or equal to n2
`$n1 -lt $n2`	Check if n1 is less than n2
`$n1 -ne $n2`	Check if n1 is not equal to n2

test string comparisons

`$str1 = $str2`	Check if str1 is same as str2
`$str1 != $str2`	Check if str1 is not same as str2
`$str1 < $str2`	Check if str1 is less than str2
`$str1 > $str2`	Check if str1 is greater than str2
`-n $str2`	Check if str1 has length greater than zero
`-z $str2`	Check if str1 has length of zero

Unary file comparisons

`-e file`	Check if file exists
`-d file`	Check if file exists and is a directory
`-f file`	Check if file exists and is a file
`-r file`	Check if file exists and is readable
`-s file`	Check if file exists and is not empty
`-w file`	Check if file exists and is writable
`-x file`	Check if file exists and is executable
`-O file`	Check if file exists and is owned by current user
`-G file`	Check if file exists and default group is same as that of current user

Binary file comparisons

`file1 -nt file2`	Check if file1 is newer than file2
`file1 -ot file2`	Check if file1 is older than file2

while do loop

```
while condition
do
  commands
done
```

commands are executed only if *condition* returns true

until do loop

```
until condition
do
  commands
done
```

commands are executed only if *condition* returns false

functions

definition

```
myfunc()
{
  commands
}
```

call

```
myfunc
```

commands are executed eachtime *myfunc* is called

Definitions must be before the calls

Explore with the commands you learnt till now

There are more features available !