Table of Contents

The Berkeley Utilities are a set of *UNIX*[1] like utilities for *MS-DOS*[2]. They have been developed by P.M.C. (a software company from Paris, France) for its internal use, because there was no available package covering the same needs. It is not as complete a set of *UNIX* commands as can be found in some other packages (e.g. MKS) but it contains some useful utilities you don't find elsewhere (e.g. `cb`, `xstr`). At the time they were written, there was no port to *MS-DOS* of the GNU utilities either. *The Berkeley Utilities* have been maintained because we make constant use of them (for instance they now work with long filenames under *WIN95/98*, and `df` can understand multi-gigabyte partitions). Some functional advantages over other sets of utilities are described in the next paragraph. Compared to the GNU utilities, our utilities have the advantage that they are much smaller (20K on average instead of 100K). Compared to MKS utilities, they have the advantage that each utility is self contained (it can run separately without any other support file, and each utility contains an help screen).

The particular advantages of our utilities come from our design goals:

- Since we had to rewrite for *MS-DOS* the *UNIX* utilities, we decided to do them right: you will find any useful option you ever had on any *UNIX* system, and often new options which make sense and increase the power of the package. People who have been using our package, when going (or coming back to) *UNIX* often wish our extra options would work there (we are considering alleviating their suffering by porting our commands to *UNIX* !). If you are a *UNIX* user, look at our extra options on `cp`, `mv`, `ls` and others; you will see what we mean!

- We also decided to use the advantages of working in *MS-DOS* when they exist, e.g. the use of video attributes to make displays clearer.

- Since we believe that using combinations of *UNIX* commands to do small (or big jobs) is a powerful way to work, which we wish to teach to others, we also are aiming our package to all PC users, and made a special effort to provide on-line help and tutorial information. You will learn a new way to work, and it will be an ever useful knowledge (at least until *UNIX* dies, which won't be tomorrow).

[1] *UNIX* is a trademark of **Novell**

[2] *MS-DOS and WINDOWS* are trademarks of **Microsoft Corporation**

• We wanted our utilities to be self-contained, and able to be used individually without going through any installation procedure. Each utility is self-contained without any extra files needed, and has a help screen (usually enough to get by, excepted for the more ambitious utilities such as `awk` and `make`).

At this stage, if you are new to *UNIX*, we recommend that you go through our User's Manual, and use our integrated help. We recommend the following books to help you along:

- "The *UNIX* programming environment", by Kernighan and Plauger.
- For `awk` "The AWK programming language", by Aho, Weinberger and Kernighan.

If you are a *UNIX* wizard look in our `man` pages, try everything and enjoy (`man` refers to the reference manual, which in *UNIX* is available on-line by running a program called man).

Jean MICHEL

Command line handling:

All *The Berkeley Utilities* can apply transformations to the command line, doing their best to emulate the behavior of `shell`s under *UNIX*, insofar as *MS-DOS* allows it:

- Arguments prefixed with '`-`' are options. An option is defined by the character following the '`-`' and may or may not take a parameter. As in *UNIX*, the options are case sensitive, which means that the command `ls -t` has a different meaning than the command `ls -T`. When the option takes a parameter, the parameter may be mandatory, in which case it may be separated from the option by spaces, or it may be optional, in which case it must follow the option immediately. An option standing alone may be followed by (bundled with) another option without separating space and '`-`' character, e.g., if `-` s and `-` x are options without parameters, those options may be bundled as: `-sx`. Some options are "boolean flags" (that is, it makes sense to turn them on or off). These options can then usually be turned off by giving them followed by a '`-`', e.g., `-s-x` means take option `x` but turn off `s` (this is especially useful in connection with an initial command line — see below).
- Options `-?`, `-H`, and, in case it has no other significance, option `-h`, are taken to be a call for help: a message describing the utility usage syntax is sent to standard output with a short explanation of the semantics of the arguments and of the other options.
- An option of "`--`" is taken to end the list of options; any further arguments beginning with '`-`' will not be interpreted as options. Contrary to usual unix behaviour, options may appear anywhere on the command line (they do not have to be grouped at its beginning). This behaviour may be changed by giving the option '`-!`'; then the first non-option argument will end the options (this is particularly useful in conjunction with setting an initial command line; see below).
- "`-`" standing alone is not usually taken to be an option, but to be an argument standing for standard input *stdin* (we adopt standard *UNIX* and *MS-DOS* terminology: a program takes its input from *stdin* (usually the terminal, but it may be redirected with '<'), sends its normal output to *stdout* (which can be redirected with '>'), and sends its error output to *stderr* (which cannot be redirected under *MS-DOS*, unless you use our utility `rederr`)).
- Arguments starting with a "`$`" are assumed to be environment variables, and are replaced by their value: e.g., if the `autoexec.bat` file included

the assignment:

```
set INCLUDE=c:\include
```

"`c:\include`" will be substituted for "`$INCLUDE`". If the variable is immediately followed by characters allowed in identifiers, curly brackets must be used: if the environment variable "`MAKEFLAGS`" has been assigned the value "`ei`", "`eid`" will be substituted for "`${MAKEFLAGS}d`". The following variations are also understood:

- `${x-z}` stands for the value of environment variable `x` if `x` has been defined, else for the string `z`. E.g., by anticipation on the following paragraph on command substitution, `${HOME-‘cd‘}` stands for the value of environment variable `HOME` if `HOME` has been defined, and else for the name of the current directory.
- `${x=z}` is like the previous case, but in addition if `x` had not been defined, it is now assigned value `z` until the utility ends.
- `${x?z}` stands for the value `y` of environment variable `x` if `x` has been defined, else message `z` is sent to standard output, and the utility aborts.
- `${x+z}` stands for `z` if `x` has been defined, else for the empty string.

- Another substitution mechanism is applied to arguments surrounded by backquotes "‘", called *command substitution*. First, the text inside the "‘"s is executed as a command, then its standard output is inserted in the command line, after substituting spaces for imbedded newlines, and after stripping trailing newlines.
- Finally, an argument including a "`*`", a "`?`" or a "`[`" is taken to be a file specification pattern, and *expansion* is applied: its place is taken by the list of actual files whose name matches the pattern according to the following rules:
  - The star "`*`" stands for any number (0 included) of characters except "`\`", "`/`", "`.`".
  - The question mark "`?`" stands for exactly one character not in "`\`", "`/`", "`.`".
  - One or more characters surrounded by "`[ ]`" stand for exactly one of the surrounded characters. If the character following "`[`" is "`!`", the form stands for exactly one character *not* in the set of the characters between the "`!`" and the "`]`". Inside the "`[]`", after the possible initial "`!`", a sequence like *a*`-`*p* stands for the set of characters whose ASCII code is between those of *a* and *p*.
  - A "`.`" ending the pattern is ignored.
  - "`/`" et "`\`" are both acceptable as delimiting directories in a path name.

- Finally, the sequence "`//`" matches any number of consecutive directory names in a pathname.

note. The sequence "`\\`" was also used in earlier versions of the *The Berkeley Utilities* for the same purpose. This has been made obsolete, in order to avoid conflict with the names of network drives.

For example,

```
[c-e]://[!a]*.c
```

means all the files whose filename start with a letter different from `a`, which have an extension of c, and which are anywhere ("`//`" means "any number of directories below the root") in hard disk `c:`, `d:` or `e:`.

NB1: Those patterns are similar in some ways to *regular expressions*. But there is no closure operator, "`?`" plays the part of "`.`", "`*`" is equivalent to "`[^.\]*`", in character classes the complement operator is "`!`" instead of "`^`".

NB2: Divergences from *UNIX* file specification patterns: under *UNIX*, "`.`" has no special properties except as the first character of a filename. Under *MS-DOS*, a file or directory name may contain at most one "`.`", and a name without a "`.`" designates the same file as the same name with the "`.`" appended. Moreover a name can have at most 8 characters before a "`.`" and 3 characters after it.

NB3: Divergences from *MS-DOS*: a name with less than 8 characters before an explicit or implicit "`.`" is not supposed to be completed to 8 characters by spaces. Thus the behavior of "`?`" differs from *MS-DOS* expansion where it sometimes ends up standing for one or no character. On the other hand, patterns where an initial "`*`" is followed by non-special characters are handled as you would expect (as in *UNIX*), whereas *MS-DOS* sees no difference from "`*`" alone. For instance, *The Berkeley Utilities* see `*A.*` as all the files whose filename ends with an A, while *MS-DOS* does not distinguish `*A.*` from `*.*`.

NB4: Be aware that patterns can only stand for existing files. Since the syntax of `cp` and `mv` are different from that of `copy` and `rename`, commands such as

```
copy *.bin *.obj
```

have no equivalent with *The Berkeley Utilities*. Nevertheless, an equivalent result may be obtained by a combination of some command repetition mechanism (such as the *MS-DOS* `for` command) and the `basename` utility:

```
for \%i in (*.bin) do cp \%i `basename \%i .bin=.com`
```

(note that `%i` must be replaced by `%%i` in a batch file).

Installation and Video Attributes:

The installation is an easy procedure: just copy the files from the distribution disks to a subdirectory on your hard disk, then place that directory on the path before the one which contains *MS-DOS*. In recent *WIN95/98* systems you may have trouble doing that since *WIN95/98* will automatically prepend its `COMMAND` subdirectory to the path. On such systems, to use the Berkeley `find` and `more` the best way is to rename the same-name commands in `WINDOWS/COMMAND` to a different name. Since `ECHO`, a built-in command, cannot be renamed, we named ours `ech`.

The Berkeley Utilities will work independently of each other and without any installation. They are easily configurable with the help of the supplied program `opts.exe`. This program sets an initial command line for any utility. For instance

```
opts rm "-i -r"
```

would set the initial command line of `rm` to `-i -r`, which means that on any future call to `rm`, the options `-i -r` will be prepended to the actual command line. This could be used to give default arguments, in addition to setting default options. The mechanism is such that the part of the command line thus given does not count in the DOS limit of 128 characters for the command line and may be arbitrarily long.

The Berkeley Utilities use different video attributes in order to highlight the key parts of their output. Most of *The Berkeley Utilities* use 3 attributes, but some (`more`, for instance) use many more. Those attributes may be selected globally for all utilities by assigning values to the environment variable "`VATTR`" in the following way:

`set VATTR=`$attribute_0$`-`$attribute_1$`-`$attribute_2$`-`$\ldots$ $attribute_n$

$attribute_i$ being the middle part of an ANSI 'Set Graphic Rendition' Escape-sequence, e.g., a sequence formed thus:

$\langle Esc \rangle$`[`p_1`;`p_2`;`$\ldots$ p_k`m`

stripped of initial "$\langle Esc \rangle$`[`" and of final "`m`". The parameters p_i are as described in *MS-DOS* reference manuals, for instance:

```
set VATTR=44;36;1-44;33;1-44;35;1-40;36;1-40;31;1-40;37;1-42;30-42;33;1
```

selects 8 attributes:
bright cyan on blue, bright yellow on blue, bright magenta on blue, bright cyan on black, bright red on black, white on black, black on green, bright yellow on green.

The following example:

```
set VATTR=0-1-4-7;1-7-7;4
```

selects 6 attributes for a monochrome screen (hard to find nowadays): normal on black, highlighted on black, underlined black, grey on white, black on white, inverse underlined. The attributes can be set for a single utility using the '`-@`' option which takes as argument a string following the same syntax as `VATTR`. The usual way to proceed would be to set this option in an initial command line via `opts`. The command '`opts -e`' may be used to edit interactively the initial command line and has support for selecting attributes from a menu.

When the utilities output is redirected to a file, attributes are normally *not* output. Nevertheless, if the option '`-&`' is given, output to a file and to the terminal is treated the same way. This is specially useful if the output is piped to a browser which can emulate `ansi.sys`, such as `more`. For instance, to look at leisure at strings found, type

```
grep -& thing *.c |more
```

The method of assigning the value `ANSI` to the environment variable `FATTR` which was in the version 1 of *The Berkeley Utilities* is now obsolete (the above method is better since it can be controlled on each use).

There is a way to tell the utilities not to use `ANSI` attributes: just do "`set VATTR=NO`".

Command spawning from *The Berkeley Utilities* **:**

In many cases, (command substitution, `make` *methods*, "`!`" commands of `ed` and `more`, `-exec` predicate of `find`, etc...). utilities have to spawn some other command. The normal way to do this is to spawn a `subshell`, where under *MS-DOS* the `shell` to be spawned is given by the value of the environment variable "`COMSPEC`". Actually, if it is possible, the commands are spawned directly without a `subshell` intervening. This is useful for the utilities which need to know the exit status of spawned commands (`make`, `find`, ...), since the standard *MS-DOS* `shell` (`command.com`) does not make this information available. A command can be spawned directly if it is not an internal *MS-DOS* command and does not use pipes (`|`). Otherwise the command is spawned via a subshell and will always be assumed to have succeeded.

String-processing language

Synopsis: `awk [-Fc] -f` *program* `[`*files*`]`

or `awk [-Fc] "`*program*`" [`*files*`]`

Description:

If you are not already familiar with `awk`, it is strongly recommended that you read the excellent 1988 Addison-Wesley book "The AWK Programming Language", by Aho, Kernighan and Weinberger who gave the language its "awk"ward name. The following is not intended to serve as a tutorial for the language.

The `awk`program to execute is in the file specified as argument to the `-f` option, or is the first argument on the command line if there is no `-f` option. The file arguments processed by the program are considered as a sequence of *records* separated by *record-separator* characters, each record itself being a sequence of *fields* separated by *field-separator* characters. By default, the *record-separator* is the newline, so records are consecutive lines of the file, and the *field-separator* is the space. These defaults may be changed as will be seen below. If the *field-separator* is the space, as a special convention the ⟨*Tab*⟩ and the newline are also *field-separators* (this is specific to the space). An "*awk*" program consists in a sequence of pairs "*condition* `{` *actions* `}`". For each record in each file argument which matches the *condition* the corresponding *actions* are executed. A missing *condition* is considered to match every record, and a missing *action* is equivalent to the action `{print}` which prints the current record.

The *actions* are written in a language whose syntax is similar to that of the language **C**, but whose semantics are quite different: Variables can hold numeric or string values, or be arrays, but there are no declarations. A variable may indifferently hold numeric or string values; the conversion between these is automatically performed in any context where it is necessary; numeric values are floating-point numbers. On the other hand, the first occurence of a variable decides if it will be an array or scalar (if it is indexed or not in this first occurence) and then its nature (scalar or array) will be the same for the rest of the program. Array indices may be any scalar value, which provides a kind of associative memory. Operators are those of the **C** language when they make sense. Structured programming constructs are

available as in **C** by using the keywords `for`, `while`, `if` and `else`. A variant of `for` is provided which loops over an associative array. The language contains a few built-in variables and functions. The *conditions* are built using boolean operators from relational expressions and regular expressions (look in the Appendix for a definition of regular expressions; the regular expressions currently do not have the alternation operator, they will be extended in a later version). In addition a condition may be a pair of conditions as described above, separated by commas. Such a condition holds between the first line satisfying the first condition and the next line satisfying the second condition, and again between such pairs of lines until the end of the file.

Formal Grammar of `awk`:

(In the documentation which follows, "**iff**" is an abbreviation for "**if and only if**").

⟨*program*⟩
:= ⟨*begin*⟩ ⟨*body*⟩ ⟨*end*⟩

⟨*begin*⟩
:= `BEGIN {` ⟨*actions*⟩ `}`

`BEGIN` is a special condition which declares ⟨*actions*⟩ to perform before starting to read the first file argument.

| *nothing*

i.e. no initial ⟨*actions*⟩.

⟨*body*⟩
:= ⟨*body*⟩ ⟨*action-condition*⟩

| ⟨*body*⟩ ⟨*action-condition*⟩ ⟨*terminator*⟩

| *nothing*

The ⟨*body*⟩ of the program is a sequence of ⟨*action-condition*⟩s, separated by "`;`" or newlines. The ⟨*body*⟩ is executed by applying successively each ⟨*action-condition*⟩ to each records of each file.

⟨*end*⟩
:= `END {` ⟨*actions*⟩ `}`

`END` is a special condition which declares the ⟨*actions*⟩ to perform after processing the last record of the last file.

| *nothing*

i.e. no final ⟨*actions*⟩.

⟨*action-condition*⟩
:= ⟨*pattern*⟩

Print each record which matches the ⟨*pattern*⟩.

| ⟨*pattern*⟩ `{` ⟨*block*⟩ `}`

For each record matching the ⟨*pattern*⟩, execute actions in ⟨*block*⟩.

| ⟨*pattern*⟩ `,` ⟨*pattern*⟩

Wait for a record matching the first ⟨*pattern*⟩, then print each record until the next record matching the second ⟨*pattern*⟩, and so on.

| ⟨*pattern*⟩ `,` ⟨*pattern*⟩ `{` ⟨*block*⟩ `}`

Wait for a record matching the first ⟨*pattern*⟩, then execute the ⟨*block*⟩ of actions for each record until the next record matching the second ⟨*pattern*⟩, and so on.

| `{` ⟨*block*⟩ `}`

For each record, execute actions in ⟨*block*⟩.

⟨*pattern*⟩
:= ⟨*regular-expression*⟩

A record matches the ⟨*pattern*⟩ iff it matches the ⟨*regular-expression*⟩.

| ⟨*match*⟩

| ⟨*relational-expression*⟩

| ⟨*composed-pattern*⟩

⟨*composed-pattern*⟩
:= ⟨*pattern*⟩ `||` ⟨*pattern*⟩

Alternation: a record matches the ⟨*composed-pattern*⟩ if it matches one of the two ⟨*pattern*⟩s.

| ⟨*pattern*⟩ `&&` ⟨*pattern*⟩

Conjunction: a record matches the ⟨*composed-pattern*⟩ if it matches both ⟨*pattern*⟩s.

| `!` ⟨*pattern*⟩

Negation: a record matches the ⟨*composed-pattern*⟩ if it does *not* match the ⟨*pattern*⟩.

| `(` ⟨*composed-pattern*⟩ `)`

Grouping.

⟨*block*⟩
:= ⟨*block*⟩ ⟨*statement*⟩

| *nothing*

⟨*block*⟩ is a sequence of ⟨*statement*⟩s, executed by successively executing each ⟨*statement*⟩. A `break`, `continue`, `next` or `exit` statement may stop execution before the end of the ⟨*block*⟩.

⟨*statement*⟩
:= ⟨*simple-statement*⟩ ⟨*terminator*⟩

| `if (` ⟨*condition*⟩ `)` ⟨*statement*⟩ `else` ⟨*statement*⟩

If the ⟨*condition*⟩ is **true** the first ⟨*statement*⟩ is executed, else the second one.

| `if (` ⟨*condition*⟩ `)` ⟨*statement*⟩

The ⟨*statement*⟩ is executed if the ⟨*condition*⟩ is **true**.

| `while (` ⟨*condition*⟩ `)` ⟨*statement*⟩

While the ⟨*condition*⟩ evaluates to **true**, execute the ⟨*statement*⟩.

| `for (` ⟨*variable*⟩ `in` ⟨*variable*⟩ `)` ⟨*statement*⟩

The second ⟨*variable*⟩ must be an array, and then for each element of that array the ⟨*statement*⟩ is executed, with the first ⟨*variable*⟩ set to the value of that element.

| `for (` ⟨*simple-statement*⟩ `;` ⟨*condition*⟩ `;` ⟨*simple-statement*⟩ `)` ⟨*statement*⟩

Execute the first ⟨*simple-statement*⟩, then loop on the sequence: evaluate the ⟨*condition*⟩, if **true** execute the ⟨*statement*⟩, then execute the second ⟨*simple-statement*⟩.

| `for (` ⟨*simple-statement*⟩ `;;` ⟨*simple-statement*⟩ `)` ⟨*statement*⟩

Identical to the above form except the condition is always **true**. This loop can be exited only by a `break`, `next`, or `exit`.

| `break` ⟨*terminator*⟩

Get out of the current loop (the innermost one if several loops are embedded).

| `continue` ⟨*terminator*⟩

Go directly to the next iteration through the current loop.

| `{` ⟨*block*⟩ `}`

Execute the ⟨*block*⟩ (see the definition of a ⟨*block*⟩ above).

| `next` ⟨*terminator*⟩

The `next` statement causes the current record to be abandoned, the next record to be read and execution to resume at the beginning of the program body.

| `exit` ⟨*expression*⟩ ⟨*terminator*⟩

| `exit` ⟨*terminator*⟩

The `exit` statement is equivalent to the end of the last file. If an expression follows `exit`, it is evaluated and its value is used as the return code from `awk`.

⟨*condition*⟩
:= ⟨*expression*⟩

As in the **C** language, the ⟨*condition*⟩ is true iff the ⟨*expression*⟩ evaluates to a non-zero value.

| ⟨*relational-expression*⟩

| ⟨*match*⟩

| ⟨*composed-condition*⟩

⟨*composed-condition*⟩
:= ⟨*condition*⟩ `||` ⟨*condition*⟩

| ⟨*condition*⟩ `&&` ⟨*condition*⟩

| `!` ⟨*condition*⟩

| `(` ⟨*composed-condition*⟩ `)`

The syntax of ⟨*condition*⟩s is very similar to that of the ⟨*pattern*⟩s. Note that, in contrast to **C**, an expression is meaningful as a condition but the converse is not true.

⟨*simple-statement*⟩
:= `print` ⟨*list*⟩ ⟨*redirection*⟩ ⟨*expression*⟩

The items of the ⟨*list*⟩ as well as the final ⟨*expression*⟩ are evaluated as character strings; then the items are printed, separated by the output field-separator (variable `OFS`) to the file whose name is the value of the final ⟨*expression*⟩ (this file is created if non-existent). If the file did exist, the text replaces its contents, except that if ⟨*redirection*⟩ is "`>>`", the text is appended to the file.

| `print` ⟨*list*⟩

Same as above, the output file being *stdout*.

| `print` ⟨*redirection*⟩ ⟨*expression*⟩

| `print`

If `print` has no arguments, `$0` (the current record) is printed.

| `printf` ⟨*list*⟩ ⟨*redirection*⟩ ⟨*expression*⟩

| `printf` ⟨*list*⟩

As `print`, but the first item in the list is interpreted as a character string to yield a format, which is used to print the other items, with the same conventions as in the **C** `printf` function.

| ⟨*expression*⟩

⟨*expression*⟩

:= ⟨*expression*⟩ ⟨*term*⟩

⟨*expression*⟩ and ⟨*term*⟩ are evaluated to character strings and catenated.

| ⟨*term*⟩

| ⟨*value*⟩ `=` ⟨*term*⟩

| ⟨*value*⟩ `+=` ⟨*term*⟩

| ⟨*value*⟩ `-=` ⟨*term*⟩

| ⟨*value*⟩ `*=` ⟨*term*⟩

| ⟨*value*⟩ `/=` ⟨*term*⟩

| ⟨*value*⟩ `%=` ⟨*term*⟩

Assignment operators, which have the same meaning as the corresponding operators in the **C** language.

⟨*term*⟩

:= ⟨*value*⟩

| `(` ⟨*expression*⟩ `)`

| ⟨*term*⟩ `+` ⟨*term*⟩

| ⟨*term*⟩ `-` ⟨*term*⟩

| ⟨*term*⟩ `*` ⟨*term*⟩

| ⟨*term*⟩ `/` ⟨*term*⟩

| ⟨*term*⟩ `%` ⟨*term*⟩

| `+` ⟨*term*⟩

| `-` ⟨*term*⟩

Dyadic and monadic operators, which have the same meaning and syntax as in **C**.

| `++` ⟨*value*⟩

| `--` ⟨*value*⟩

| ⟨*value*⟩ `++`

| ⟨*value*⟩ `--`

Pre and post-decrementation and incrementation, as in **C**.

| ⟨*function*⟩ `(` ⟨*expression*⟩ `)`

| ⟨*function*⟩ `()`

| ⟨*function*⟩

Where ⟨*function*⟩ is one of the intrinsic functions of `awk`(see below the list of these functions). If there is no argument, by default `$0` (the current record) is used.

| `getline`

`getline` reads the next record and returns it (`$0`) as its value, without breaking the program flow as `next` does.

| `sprintf` ⟨*list*⟩

The first item in ⟨*list*⟩ is taken to be a format string. Similar to the `sprintf` of the standard **C** library.

| `substr (` ⟨*expression*⟩ `,` ⟨*expression*⟩ `,` ⟨*expression*⟩ `)`

Returns the sub-string of the first ⟨*expression*⟩ which starts at the position specified by the second ⟨*expression*⟩, and whose length is at most the value of the third expression.

| `substr (` ⟨*expression*⟩ `,` ⟨*expression*⟩ `)`

Returns the terminal substring of the first ⟨*expression*⟩ which starts at the position specified by the second ⟨*expression*⟩.

| `split (` ⟨*expression*⟩ `,` ⟨*variable*⟩ `,` ⟨*expression*⟩ `)`

Sets ⟨*variable*⟩ to an array whose elements are the substrings obtained by splitting the first string ⟨*expression*⟩ at places where occurs the separator which is specified by the first character of the second string ⟨*expression*⟩, and returns as result the number of elements of that array.

| `split ( ⟨expression⟩ , ⟨variable⟩ )`

Like the previous form, but using as separator *field-separator* character specified by the built-in variable `FS`.

| `index ( ⟨expression⟩ , ⟨expression⟩ )`

returns an integer, the position of the first occurence of the second string ⟨*expression*⟩ as a substring of the first one; returns 0 if there is no occurence.

⟨*value*⟩

:= ⟨*variable*⟩

| ⟨*variable*⟩ `[` ⟨*expression*⟩ `]`

⟨*variable*⟩ must be an array, or must be mentioned here for the first time. ⟨*expression*⟩ must evaluate to a scalar value.

| ⟨*field*⟩

| *number*

A *number* is a floating-point number written as a sequence of digits, with an optional decimal point and exponent.

| *string*

A *string* constant is a sequence of characters between double quotes '`"`'. The "\" character may be used to quote the next character, allowing to specify characters impossible to put in the string otherwise:
`\\`: A \.
`\"`: A double quote ".
`\n`: A newline.
`\t`: A ⟨*Tab*⟩.

⟨*field*⟩

:= `$` ⟨*expression*⟩

⟨*expression*⟩ must evaluate to a non-negative integral value. `$0` is the current record, and cannot occur on the left of an assignment operator. `$`*n* where *n* `!= 0` represents the *n*th field, and can be assigned to as any other.

⟨*function*⟩

:= `length`

The function `length` gives back the length of its argument (`$0` by default) interpreted as a character string.

| `log`

logarithm function.

| `int`

floor function.

| `exp`

exponential function.

| `sqrt`

square root function.

These functions interpret their argument (`$0` by default) as numbers, and return what their name implies.

⟨*variable*⟩
:= `NF`

The variable `NF` holds the number of fields of the current record.

| `NR`

`NR` holds the ordinal number of the currently processed record.

| `FS`

`FS` holds the *field-separator* character (this character is taken from the first character of the value of `FS` interpreted as a string). By default this character is the space, unless the option `-F` has been given.

| `RS`

`RS` holds the *record-separator* character, which by default is the newline. If "`RS`" is an empty string, the records will be separated by an empty line.

| `OFS`

`OFS` holds the output *field-separator* character which, by default, is the space.

| `ORS`

`ORS` holds the output *record-separator* character which, by default, is the newline.

| `OFMT`

`OFMT` holds the default output format for numbers which, by default, is "`%.6g`".

| `FILENAME`

Holds the current filename.

| *identifier*

An identifier is a sequence of letters, digits and "_", not beginning with a digit, and not one of the names of built-in functions and variables. Variables are initialized to the empty string (i.e. this is the value they have when used before being assigned to).

⟨*regular-expression*⟩
:= `/`*re*`/`

Look at the Appendix for the syntax of regular expressions.

⟨*match*⟩
:= `(` ⟨*match*⟩ `)`

| ⟨*expression*⟩ `~` ⟨*regular-expression*⟩

True iff ⟨*expression*⟩ matches ⟨*regular-expression*⟩.

| ⟨*expression*⟩ `!~` ⟨*regular-expression*⟩

True iff ⟨*expression*⟩ does not match ⟨*regular-expression*⟩.

⟨*relational-expression*⟩
:= ⟨*expression*⟩ `==` ⟨*expression*⟩

| ⟨*expression*⟩ `!=` ⟨*expression*⟩

| ⟨*expression*⟩ `>=` ⟨*expression*⟩

| ⟨*expression*⟩ `<=` ⟨*expression*⟩

| ⟨*expression*⟩ `>` ⟨*expression*⟩

| ⟨*expression*⟩ `<` ⟨*expression*⟩

| `(` ⟨*relational-expression*⟩ `)`

These operators have the same meaning as in the **C** language.

⟨*list*⟩
:= `(` ⟨*list*⟩ `)`

| ⟨*list*⟩ `,` ⟨*expression*⟩

| ⟨*expression*⟩

⟨*redirection*⟩
:= `>`

| `>>`

⟨*terminator*⟩
:= ;
| *newline*

Newlines are not irrelevant as in **C**, since they can be used to mark the end of a statement, but they are allowed after `if(...)`, `else`, `while(...)`, and `for(...)`. Outside of character string constants or regular expressions, "`#`" signals the beginning of a comment, and the rest of the line is ignored.

Option:

The option `-F`*c* allows to change the default field-separator character to *c*. If *c* is "`t`", it is understood as ⟨*Tab*⟩.

Examples:

- To count the number of lines of a file (same as `wc -l file`):

```
awk "END{print NR}" file
```

- To print a file, each line prefixed with its line number:

```
awk "{print NR, "'$'"0}" file
```

or more reasonably, place the following line in a separate awk program:

```
{print NR, $0}
```

- To print all lines of a file which exceed 79 characters:

```
awk "length > 79" file
```

- To print all lines of a file containing december in French or English (equivalent to "`grep \<[Dd][eé]c file`"):

```
awk "/\<[Dd][e]c/" file
```

• To find files in the current directory dated between 21th and 31th of december:

```
ls -T | awk "$1 ~ /Dec/ && $2>20{print $4}"
```

Let us follow how example **5** works. First, it is equivalent to running awk on the output of `ls -T1` (the option `-1` of `ls` is implied in case of a *pipe*). A typical line of that file looks like:

```
Dec 25 21:07 c:\bin\awk.exe
```

So when processing the file, `$1` is the month (here "`Dec`"), `$2` is the day (here "`25`"), `$3` is the hour (or the year for files more than 6 months old), (here "`21:07`"), `$4` is the filename (here "`c:\bin\awk.exe`"). "`$1 ~ /Dec/`" selects lines for december, and "`$2 > 20`" selects amongst those the ones whose day is greater than 20 (the operator "`>`" forces the second field to be interpreted as a number). The action for selected lines is to print the fourth field, i.e. the filename.

• To count the number of files dated from each month (this example uses an associative array):

```
ls -T | awk -f count
```

where `count` contains

```
$1~/Jan/{n["January"]++}
$1~/Feb/{n["February"]++}
$1~/Mar/{n["March"]++}
$1~/Apr/{n["April"]++}
$1~/May/{n["May"]++}
$1~/Jun/{n["June"]++}
$1~/Jul/{n["July"]++}
$1~/Aug/{n["August"]++}
$1~/Sep/{n["September"]++}
$1~/Oct/{n["October"]++}
$1~/Nov/{n["November"]++}
$1~/Dec/{n["December"]++}
END{ for (m in n)
     { if (n[m] > 1) NUM="s"
       else NUM=""
       print m ":",n[m],"file" NUM
     }
   }
```

Error Messages:

`can't open 'xxx'`

The program file, or an argument file, or a redirection file could not be opened.

```
error in program
syntax error
lexical error
```

Errors found in the `awk`program.

`xxx is not an array`

The variable after "`in`" in the 2nd form of a "`for`" loop is not an array.

`can't set $0`

`$0` has occured on the left of an (`= += -= *= /= %=`).

```
funny variable xxx
illegal arithmetic operator
illegal assignment operator
illegal boolean operator
illegal function type
illegal jump type
illegal relational operator
illegal statement
illegal transformation to statement
illegal reference to array xxx
```

An array as been referenced in a context where a normal variable was expected.

`newline in string`

A string constant started with "`"`" has not been closed before the end of the line.

`newline in regular expression`

A regular expression started with "`/`" has not been closed before the end of the line.

`regular expression: missing ']'`

A character class opened with "`[`" in a regular expression has not been closed before the end of the line.

`not enough arguments in printf(xxx)`

`printf` or `sprintf` does not have the number of argument corresponding to the format.

`trying to access field n`

The expression following a "`$`" has a value which does not correspond to the number of a field of the current record.

`unexpected break, continue or next`

A `break`, `continue`, or `next` has been found at the topmost program level.

`too many output files n`

The number of files to which output may be redirected is currently limited to 10.

`out of memory`
`format item xxx... too long`
`record 'xxx' has too many fields`
`record 'xxx' too long`
`string xxx... too long to print`
`string too long`
`yacc stack overflow`

Various resources have been exhausted.

Portability:

New features of awk introduced in *UNIX* version V.3 are not yet implemented.

give base part of a pathname

Synopsis: `basename` *file*

or `basename` *file* [... *file*] *suffix*

Extracts the 'filename' part from a full pathname.

Description:

In the first form, `basename` strips from a pathname logical unit and directory specifications. In the second form `basename` performs this operation on all its arguments excepted the last which is interpreted as a *suffix*, and stripped from filename arguments which end with it. If this *suffix* has the form s_1=s_2, all arguments ending with s_1 will have this final s_1 replaced by s_2.

Examples:

```
C:>basename c:\bin\abc.exe
abc.exe
C:>basename c:\bin\abc.exe c:\bin\other.bak .exe
abc
other.bak
C:>basename c:\bin\abc.exe c:\bin\other.bak .exe=.c
abc.c
other.bak.c
```

Notes:

`basename` is particularly useful in conjunction with the "command substitution" performed by *The Berkeley Utilities*.

For instance, to rename all files ending in `.bin` to `.com` you may use the `for` command of *MS-DOS* as follows:

```
for %i in (*.bin) do mv %i `basename %i .bin=.com`
```

And to move to directory `target` all **C** source files such that there exists an executable with the same name:

```
mv `basename *.exe .exe=.c` \target
```

See Also:

`find`.

Display the calendar for a month or a year

Synopsis: `cal [[`*month number*`] `*year number*`]`

Prints the calendar for a given month of a given year, or if the month is omitted, for all months of a given year; if given with no arguments, gives the calendar of current month.

Year may be between 1 and 9999; month must be between 1 and 12.

Notes:

To learn something about the history of England, try `cal 9 1752`.

Here is the output of `cal 7 1993`

```
      July   1993
Su Mo Tu We Th Fr Sa
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

concatenate files

Synopsis: `cat` *file* [... *file*]

Description:

`cat` writes the concatenation of all the argument files one after the other on *stdout*. If no argument has been given, or for each occurence of the argument "-", `cat` takes its input from *stdin*.

Examples:

- The following two lines are equivalent:

```
C:>cat abc
C:>type abc
```

- A way to add two lines to the beginning and one line to the end of a text file without using an editor:

```
C:>cat - autoexec.bat - >autoexec.new
set include=c:\msc\include;c:\msc\include\sys
set lib=c:\msc\lib
^Z
set temp=c:\tmp
^Z
C:>mv autoexec.new autoexec.bat
```

`^Z` represents ⟨*Control-Z*⟩ which informs *MS-DOS* that an end of file was entered from the console.

Bugs:

Since the same buffer is used for input and output, if one of the files being concatenated is also used as *stdout*, the contents of the file will be destroyed. In order to append `file2` at the end of `file1`, type:

```
cat file2 >> file1
```

See Also:

`cp`, `mv`, `more`.

C beautifier

Synopsis: `cb` [*options*] [*input file* [*output file*]]

`cb` takes as input a **C** source file, and rewrites it according to the options specified on the command line.

Description:

By default, `cb` works on *stdin* and *stdout*. `cb` *beautifies* a **C** source file according to your programming style, organizing especially the output of blocks. Preprocessor commands and declarations outside of a function are not changed.

Options:

The following options are available on the command line:

`-i`*n* *n* is an integer, value of the indentation used after keywords. By default, 2.

`-c` Same level of indentation for a closing curly bracket and its corresponding keyword. By default, the closing curly bracket is on the same level as the opening one.

`-s` The statement immediately following a keyword will appear on the same line. By default, it appears on the following line, beginning at the next level of indentation.

`-o` The opening curly bracket appears on the same line as its corresponding keyword. By default, it appears on the next line.

`-O` The opening curly bracket is on the line following the keyword, beginning at the next level of indentation. By default, it keeps the same level as the keyword.

`-n` An `else` or a `while` immediately following a closing curly bracket appears on the same line. By default, every closing curly bracket is followed by a newline.

`-r` The first statement inside of a block appears on the same line as the opening curly bracket.

`-R` The statements inside of a block appear on the same level of indentation as the opening curly bracket. By default, they begin at the next level.

Examples:

Let us have a look at the output of 3 `cb` options working on the same input file.

```
C:>cat fopenp.c
FILE *fopenp (path, fn, fnb, mode)
register char *path;char *fn, *fnb, *mode;
{   ...
    if  (*fn == '\\') if ( fd = fopen(fn, mode))
     { strcpy(fnb, fn);return (fd);}
    else return(NULL);
    do { while  ( *path != ';')*dst++ = *path++;
    if  (dst != fnb) *dst++ = '\\';
      while  (*dst++ = *src++);
    if  (fd = fopen (fnb, mode))return (fd);}
    while  (*path++);return (NULL);}
```

Default option:

```
C:>cb fopenp.c
```

gives as output:

```
FILE *fopenp (path, fn, fnb, mode)
register char *path;char *fn, *fnb, *mode;
{   ...
    if  (*fn == '\\')
      if  ( fd = fopen(fn, mode))
      {
        strcpy(fnb, fn);
        return (fd);
      }
      else
        return(NULL);
    do
    {
      while  ( *path != ';')
        *dst++ = *path++;
      if  (dst != fnb)
        *dst++ = '\\';
```

```
      while  (*dst++ = *src++);
      if  (fd = fopen (fnb, mode))
        return (fd);
    }
    while  (*path++);
    return (NULL);
}
```

Options `-o` and `-n` joined:

```
C:>cb -on fopenp.c
```

gives as output:

```
FILE *fopenp (path, fn, fnb, mode)
register char *path;char *fn, *fnb, *mode;
{   ...
    if (*fn == '\\')
      if ( fd = fopen(fn, mode)) {
        strcpy(fnb, fn);
        return (fd);
      } else
          return(NULL);
    do {
      while ( *path != ';')
        *dst++ = *path++;
      if (dst != fnb)
        *dst++ = '\\';
      while (*dst++ = *src++);
      if (fd = fopen (fnb, mode))
        return (fd);
    } while (*path++);
    return (NULL);
}
```

Options `-r` and `-s` joined:

```
C:>cb -rs fopenp.c
```

gives as output:

```
FILE *fopenp (path, fn, fnb, mode)
register char *path;char *fn, *fnb, *mode;
{   ...
    if  (*fn == '\\') if  ( fd = fopen(fn, mode))
    { strcpy(fnb, fn);
      return (fd);
    }
    else  return(NULL);
    do
    { while  ( *path != ';') *dst++ = *path++;
      if  (dst != fnb) *dst++ = '\\';
      while  (*dst++ = *src++);
      if  (fd = fopen (fnb, mode)) return (fd);
    }
    while  (*path++);
    return (NULL);
}
```

Error Messages:

```
else not following an if
unbalanced curly brackets
```

Can be printed in case of a syntax error. But be careful — `cb` is not a syntax analyzer !

Bugs:

If you ask for an output with both the options `-r` and `-o`, you won't get exactly what you expect: if there are nested blocks, the shift to the right of the output would very soon get unreadable.

Portability:

All options are enhancements (*UNIX* version cannot be configured).

compare binary files

Synopsis: `cmp [`*options*`] `*file1 file2*` [`*offset1*`] [`*offset2*`]`

`cmp` compares file1 (starting at byte offset1 if given) to file2 (starting at byte offset2 if given). A file given as '-' is taken to be standard input. Wildcards can be used to specify two files. By default, nothing is printed if files are identical; byte and line number of first difference are given otherwise. If one file is identical to some initial part of the other, it is reported.

Options:

`-l` long mode: all differences are reported (not only the first one).

`-s` silent mode, nothing reported in any case. Only the exit status indicates the result of the comparison:
`0` for identical arguments `1` for differences `2` for errors.

See Also:

`diff`.

Look for common lines in two files

Synopsis: `comm` [*options*] $file_1$ $file_2$

`comm` works on two already sorted files, and writes its result to *stdout*.

Description:

The default is to give the lines common to the two files.

Options:

Two options are allowed on the command line:

`-1` Asks `comm` to give as output the lines which are only in $file_1$.

`-2` Asks `comm` to give as output the lines which are only in $file_2$.

Examples:

```
C:\>ls -1 \util\src >files.c
C:\>ls -1 a:util\src >files.a
C:\>comm files.c files.a
```

Since, by default, the output of `ls` is sorted alphabetically, `comm` gives the list of files which belong to both subdirectories.

```
C:\>comm -2 files.a files.c
```

lists the files which appear only in the subdirectory `C:\util\src` .

See Also:

`diff`, `sort`, `uniq`.

Copy files and directories

Synopsis: `cp [`*options*`]` source target

or `cp [`*options*`]` file|dir ... file|dir dir

`cp` copies files or directories matched by the pathnames given as argument.

Description:

There are two forms of the command:

- short form: there are only two arguments, and furthermore both arguments consist of one file, or both of one directory, or the second argument is a new name. The first argument is copied over the second (target).
- long form: the last argument is a directory (target) and all other arguments are copied to that target directory.

Watch out when using wild cards (like `file.*`), as the target (the last argument) must expand to at most one name.

Options:

The possible options on the command line are:

`-r` Allows `cp` to copy (and possibly overwrite) non-empty directories (if not given, only empty directories are copied or overwritten).

`-m` When copying directories, merges the source with the target (instead of overwriting the target).

`-v` Gives on *stdout* a report on copied files.

`-f` Do not ask confirmation before overwriting read-only files (by default the authorization of the user is asked).

`-i` Asks confirmation before overwriting any file or directory.

`-I` Asks confirmation before copying any file or directory. This option implies the `-i` option.

When the options `-i` or `-I` are given, the only answers allowed are:

`n`: continue, do not overwrite or copy.
`q`: leave.
`g`: (go) stop asking questions.
`y`: overwrite or copy.
`s`: answer valid only **for a directory**. Overwrite or copy without asking further confirmations for files or sub-directories of this directory.

Examples:

```
cp -rvm a:dbaseiv c:\
```

Add the contents of directory `dbaseiv` from diskette `a:` to the hard disk `c:` (and do not overwrite, if this directory already exists on `c:`, the files in it whose name does not conflict with a name in `a:dbaseiv`); inform on performed actions (option `-v`).

Notes:

MS-DOS's `COPY` is capable of preventing the copying of a file over itself in simple cases, but will fail in more complicated cases (and trash the file):

```
C:\>COPY top.map top.map
File cannot be copied onto itself
        0 file(s) copied
C:\>COPY t*.map top.map
1 file copied.
```

`cp` does not make that kind of mistake.

cut out selected fields

Synopsis: `cut -c`*list* [*files*]

or `cut -f`*list* [*options*] [*files*]

`cut`cuts out columns or fields from each line of the *files* entered as arguments according to the options specified by the user on the command line. If no *files* are specified or the file name `-`, `cut`works on *stdin*.

Description:

`cut`looks at every line of *files* and copies to the standard output only the fields (option `-f`) or characters (option `-c`) specified in the argument *list*. *list* must immediately follow the option (no space allowed). *list* is a comma-separated list of integers or integer ranges, given in increasing order. A range is specified by a `-` as in `8-12`. A `-` not preceded by a number makes `cut`consider that the range begins with the first character or field. A `-` not followed by a number means that the range ends at the end of the line, with the last character or field.

Options:

One of the two following options must appear on the command line:

`-c`*list* *list* represents character positions, each integer is the position of a character on the line: for instance, the list `-28` asks `cut`to copy the first 28 characters of every line of *files* to the standard output.

`-f`*list* *list* represents field positions, each integer is the position of a field on the line. Fields are delimited by a special character (see option `-d`). If no delimiting character appears on one line, this line will be copied just as it is to the standard output, unless option `-s` has been given.

`-d`*c* Take *c* as the delimiting character. By default the fields are delimited by tabs.

`-s` Do not output lines containing no delimiters.

Examples:

```
C:\>cat junk
apples 12 \kilos
raisins 14 \pounds
oranges 23 \units
C:\>cut -c3-10 junk
ples 12
isins 14
anges 23
C:\>cut -f2 -d\ junk
kilos
pounds
units
```

See Also:

paste.

Displays space left on various drives

Synopsis: `df [`*drive specifications*`]`

Shows statistics for the space used on all hard disk drives (default).

Optionally, a list of drives, including floppies, may be given as an argument. A drive may be specified as a single letter or as a single letter followed by ":". Ranges are allowed.

Examples:

- ask for space left on floppy `a:` and hard drive `e:`

```
df a e:
```

- ask for space left on `c:`, `d:`, and `e:`

```
df c:-e:
```

or

```
df c-e
```

Here is the output which might be given by the above command:

```
drive  total bytes  bytes used  (%)   bytes free  (%)  cluster size
  C:      33462272    24475648 73.1%     8986624 26.8%       2048
  D:     314613760   295624704 93.9%    18989056  6.0%       8192
  E:      83247104    81633280 98.0%     1613824  1.9%       8192
Total    431323136   401733632 93.1%    29589504  6.8%
```

Compare files or directories

Synopsis: `diff` [*options*] $f_1 \ldots f_n$

`diff` compares files or directories. If the argument `-` is given, *stdin* is used.

Description:

When there are two arguments: if they are binary files, `diff` just tells if they differ; if they are text files, `diff` reports in a format similar to an `ed` script which lines must be changed to make f_1 identical to f_2. `diff` gives no report when the files are identical, except if the option `-s` described below has been given. If f_1 and f_2 are both directories, they are first sorted, and then `diff` shows the files which appear in only one of them, and gives a report on files or subdirectories with the same name. If there are more than two arguments, the last one (f_n) must be a directory, and for each other argument f_i, `diff` compares f_i and $f_n \backslash f_i$. If the option `-r` described below has not been given, `diff` reports common subdirectories, even if they are equal.

Options:

The possible options are:

`-t` Consider all files as binary, i.e. just tell if they differ.

`-h` n(half-hearted) Use a faster algorithm which also requires less memory for big files, but which is less precise and may give spurious results or no result at all (while the usual algorithm is guaranteed to find the minimum necessary set of lines to change). The optional number n is the maximum number of lines that a single difference can be (resynchronisation is on 3 identical lines) — default n is 200; making it bigger may make `diff -h` work in more situations at the cost of slower excution.

`-r` Tells `diff` to work recursively on subdirectories.

`-s` Give also a report on identical files.

`-b` Ignore final whitespace (blanks and tabs) at the end of a line and consider as equal any other non-empty sequence of whitespace characters when comparing lines.

Only one of the following options may be given at once:

`-e` Gives a true `ed` script.

`-f` Give an inverted script.

`-c`*n* Give *n* lines of context around each difference. By default, 3 lines are given.

`-D`*name* Useful mostly when dealing with **C** source files. Gives on *stdout* a new file which has `#ifdef`'s such that it will compile as f_2 if headed with `#define` *name* and as f_1 otherwise. The result might not compile if there were already `#ifdef`'s within the differences.

Examples:

We show the output given by various options on the following two files:

```
C:>cat a.c
#define LINT_ARGS
#include <stdio.h>
main(){
printf("hello, world!");
}
C:>cat b.c
#include <stdio.h>
main(){
printf("hello, world!");
exit(0);
}
```

Default behavior:

```
C:>diff a.c b.c
<<< a.c  and b.c  differ >>>
1d0
< #define LINT_ARGS
4a4
> exit(0);
```

Option "conditional compilation":

```
C:>diff -Dx a.c b.c
#ifndef x
#define LINT_ARGS
#endif /* x */
#include <stdio.h>
main(){
printf("hello, world!");
#ifdef x
exit(0);
#endif /* x */
}
```

Option "ed script":

```
C:>diff -e a.c b.c
4a
exit(0);
.
1d
```

Bugs:

The number of lines per file is limited to about 15000 unless option `-h` is given, whence there is no limit.

Portability:

The options `-r`, `-s` and `-c` are only found in BSD 4.xx. Option `-t` is an enhancement.

See Also:

`comm`, `ed`, `sed`.

display tree structure of directories

Synopsis: `dtree [`*option*`] [`*pathname*`]`

Description:

`dtree` displays the tree structure formed by subdirectories of the directory given as argument on the command line. By default (when no argument has been given) `dtree` works on the current directory. Video attributes are used to enhance the display of different levels in the hierarchy.

Option:

The option `-a` also lists the files in each subdirectory.

Examples:

```
C:>dtree \
\windows\pif\
\msc\include\sys\
    lib\
\games\
\dos\
```

Here we have represented different video attributes by different fonts.

Portability:

The use of different video attributes to highlight key parts of the output is an enhancement.

See Also:

`ls -RM`

estimate file space usage

Synopsis: `du [`*option*`] [`*pathnames*`]`

Description:

`du` reports the disk space used by each argument, in kilobytes. By default (when no argument has been given) `du` reports on the current directory. Video attributes are used to enhance the display of different levels in the hierarchy.

When a directory is encountered, subdirectories within it are reported on recursively, and then a total printed for that directory.

Options:

`-l`*nn* Set the level of detail to *nn*; that is, only print on the report those directories whose distance to the top is less than *nn* (the space of all subdirectories is still accounted for).

`-s` Only print the grand total for each argument (equivalent to `-l0`).

Portability:

The option `-l` gives more control than *UNIX* versions of `du`. The use of different video attributes to highlight key parts of the output is an enhancement.

See Also:

`ls -RMU`

echo

Synopsis: `ech [-n]` arg_1 arg_2 ... arg_n

Description:

`ech` echoes its arguments, separated by a space, to *stdout* and adds an end-of-line after the last argument. `ech` may be used to find out how *The Berkeley Utilities* interpret command line arguments.

Option:

The option `-n` tells `ech` not to add an end-of-line (`\n`) character after the last argument.

Examples:

```
C:\>ech Hello
Hello
C:\>ech $PATH
\bin;\util;\dos
C:\>ls *.dat
C:\                    3 entries       123456 bytes
abc.dat     def.dat     ghi.dat
C:\>ech *.dat
abc.dat def.dat ghi.dat
C:\>cd \tc\include
C:\TC\INCLUDE>ech  .
c:\tc\include
C:\TC\INCLUDE>ech  ..
c:\tc
```

Portability:

This command is called `echo` in *UNIX* systems, but since `ECHO` is also an internal command of *MS-DOS*, we had to give it a different name.

Text editor

Synopsis: `ed [`*options*`] [`*file*`]`

`ed` edits *file* if given as argument; *file* becomes the currently remembered filename (see below; more precisely `ed` simulates the command "`e` *file*" described below). If no *file* argument has been given, the edited buffer starts empty with no current filename.

Description:

Regular expressions are used within `ed` to specify line addresses and to specify part of lines (in the `s` command). Please consult the Appendix for more information about regular expressions.

Options:

The possible command-line options are:

`-s` "Silent": suppresses printing of a character and line count for commands `e`, `r` and `w`, of diagnostics when using `e` and `q` on a modified buffer, and of the prompt `!` for the command `!`*command*. Keeps the inscrutable form of error messages of `UNIX`'s `ed`, that is error messages consisting of a simple "`?`".

`-p` *string* Specifies a prompt *string* that will be used by `ed` in command mode.

`-f` *file* Takes the `ed` script (the sequence of commands to be executed) from *file*.

Addresses:

Individual lines of the file to edit are specified by addresses built as follows:

`.` Represents the current line (which is usually the last line affected by a command).

`$` Represents the last line of the edited file.

n Represents the *n*th line of the file (*n* is an integer, counting starts from 1. As a special convention, 0 sometimes represents a place before the first line of the file).

`'`*x*	Represents the line addressed by the label *x*, where *x* is a lower-case letter (these labels are created by the command `k`; see below).
`/`*pattern*`/`	Represents the first line from '`.`' matched by *pattern* (a regular expression). The search goes forward in the file, and at the end of the file wraps back to the beginning, until a match is found or until the search goes back up to and including its starting line.
`?`*pattern*`?`	Like `/`*pattern*`/` excepted that the search goes backwards.
`+`*number* `-`*number*	An address followed by `+` or `-` followed by a decimal number means that the computed address must be increased (or decreased) by that number of lines. The `+` sign may be omitted if the preceding address was nonempty. An address starting with `+` or `-` is computed with respect to the current line. If no *number* is given after the `+` or `-` the number 1 is taken by default. In additions several `+` or `-` can be given. For instance '`++`' is the same as '`.+2`'.

Commands:

The `ed` commands take 0, 1, or 2 addresses. When 2 addresses are given, they are usually separated by a comma. When two addresses are separated by a semicolon, the current line ('`.`') is set to the first address and only then the second computed. Several such addresses can be given and then the last two are used for the command. For commands taking two addresses, the second address must always specify a line *after* the first one in the buffer, and the pair of addresses identifies the range of lines between the two addresses. A command which usually requires n addresses ($n = 1$ or 2) and has been given fewer addresses assumes *default addresses*. When one address has been given to a two-addresses command, that address is taken as default for the second address. Finally "`%`" is equivalent to the address pair `1,$`. All commands are given below preceded with the specification of their default addresses within brackets.

The first three commands below put `ed` in *insert* mode. In that mode any characters entered by the user are taken as text and no command is recognized excepted that the character '`.`' given as only character on its line exits insert mode to go back to command mode.

`[.] a`
⟨*text*⟩
`.`
	Appends entered ⟨*text*⟩ just after the addressed line. The address 0 means the beginning of the file. '`.`' is set to the last inserted line (to the addressed line if there was no ⟨*text*⟩ entered).

`[.,.] c`
⟨*text*⟩
`.`
	Deletes the addressed lines and replaces them by entered ⟨*text*⟩. '`.`' is set to the last entered line (to the next one if no line was entered).

`[.] i` ⟨*text*⟩ . — Inserts entered ⟨*text*⟩ just before the addressed line. '`.`' is set to the last entered line (to the addressed line if no line was inserted).

Other commands:

`[.,.] d` — Deletes the addressed lines. '`.`' is set to the line after the last deleted line (if the last line of the buffer was deleted then '`.`' is set to the new last line).

`e`*file*
`e!`*command* — In the first form, this command replaces the content of the edited buffer with those of *file*. '`.`' is set to the last line read. If no *file* name has been given, the currently remembered filename, if any, is read. Otherwise *file* becomes the remembered filename for future `e`, `r`, `w` and `f` commands. In the second form *command* is sent to *MS-DOS* to be executed, and its output (*stdout*) is read and replaces the contents of the buffer. In that case the remembered filename is not changed. In both forms, if the contents of the buffer have been modified since the last `w` command, the `e` command must be confirmed by repeating it.

`E`*file*
`E!`*command* — This command is just like `e`, except that no confirmation is asked in case of buffer modifications since the last `w` command.

`f`*file* — If *file* is given, this command changes to *file* the currently remembered filename. Otherwise `f` just prints on *stdout* the currently remembered filename.

`[1,$] g`/pattern/*list* — First, all lines containing an occurence of *pattern* are marked. Then '`.`' is successively set to each of these lines and the *list* of commands entered is executed. The *list* of commands may extend over several lines if each of them, excepted the last, ends with a \. The commands `a`, `c` and `i` are allowed and insert mode is escaped either by a solitary dot (.) or by a line not ending with \. Commands `g` and `v` are not allowed in the *list* of executed commands. An empty *list* is equivalent to the `p` command.

`[.,.+1] j` — Joins consecutive lines specified by the addresses (suppressing intervening newline characters).

`[.] k`*x* — "Labels" with *x* the addressed line. *x* must be a lowercase letter. '*x* can then be used to address that line. '`.`' is left unchanged.

`[.,.] l` — Prints "visibly" the addressed lines: that is, nonprintable characters such as 'tab' or 'newline' are represented as in **C** by mnemonics and other non-printable characters are represented by their octal code. In addition lines greater than screen width are folded. '`l`' may be added as a flag to any command excepted `e`, `f`, `r` and `w`, and has then the effect of printing the new '`.`' after execution of that command.

`[.,.] m`*a* — Moves addressed lines to just after the lines addressed by *a*. Address 0 is allowed for *a* meaning before the first line. '`.`' is set to the new position of the last moved line.

`[.,.] n` — Prints addressed lines, preceded by their line number and a tab. '`.`' is set to the last printed line. `n` may be added as a flag to any command other than `e`, `f`, `r` and `w`, and has then the effect of printing the new '`.`' after execution of that command.

`[.,.] p` — Prints addressed lines. '`.`' is set to the last line printed. `p` may be added as a flag to any command other than `e`, `f`, `r` and `w`, and has then the effect of printing the new '`.`' after execution of that command.

`P` — The prompt in command mode is set to `*` the first time this command is executed. The prompt is then flipped from `*` to empty on subsequent uses of `P`.

`q` — Leaves `ed` without saving the buffer. This command must be confirmed by giving it twice if the buffer has been modified since the last `w` command.

`Q` — Leaves `ed`; does not ask for confirmation even if the buffer has been changed.

`[$] r`*file*
`r!`*command*

In the first form, inserts the contents of *file* in the buffer just after the addressed line. If no *file* name has been given, the remembered filename is used. Otherwise, *file* becomes the currently remembered filename only if it was the first name given since entering `ed`. The address 0 is allowed, meaning before the first line. The number of read lines and characters is printed, and '`.`' is set to the last read line. In the second form, the *command* is sent to *MS-DOS* to be executed and its output (*stdout*) is read into the buffer. In that case the remembered filename is not changed.

`[.,.] s/`*pattern*`/`*repl*`/`
`s/`*pattern*`/`*repl*`/g`
`s/`*pattern*`/`*repl*`/`*n*

Does substitutions on addressed lines containing the *pattern*. Depending on the flags, the first occurence (no flags given), or all occurences (with the `g` flag) or the *n*th occurrence of the *pattern* in each of these lines will be replaced by the string *repl*. Any character other than a space or a newline can be used as a delimiter for the *pattern* and the *repl*acement. '`.`' is set to the last line where the substitution occured. Several characters have a special meaning in *repl*. `&` represents the part of the line which matched the *pattern*, and `\`*n* where *n* is a single digit represents the part of the line matched by the *n*th sub-regular expression (delimited in *pattern* by `\(` and `\)`). If *repl* consists only of the character '`%`', it is replaced by the value is had in the last `s` command. The special meaning of `&` and of `\` and `%` can be escaped by preceding them with another `\`. It is possible to replace a line by several lines by putting newlines in *repl*; each of these must be preceded by a `\`, so *repl* consists of several lines, all but the last ending in a `\`. This is not allowed within a `g` command.

`[.,.] t`*a*

This command copies addressed lines to just after the line addressed by *a*. '`.`' is set to the last copied line. Address 0 is allowed for *a*.

`u`

Undoes the last command which modified the buffer, i.e the last command amongst `a`, `c`, `d`, `g`, `i`, `j`, `m`, `r`, `s`, `t` and `v`.

`[1,$] v/`*pattern*`/`*list*

This command is just like `g`, excepted that the *list* of commands is effected on lines containing *no* match of the *pattern*.

`[1,$]`	`w`*file* `w!`*command*	In the first form, the addressed lines are written to *file*. If no *file* name was given, the currently remembered filename is used. Otherwise, *file* becomes the currently remembered filename only if it was the first name given since entering `ed`. The number of written lines and characters is printed. '`.`' is left unchanged. In the second form, *command* is sent to *MS-DOS* to be executed, its standard input *stdin* being a file consisting of the addressed lines. In that case the remembered filename is not changed.
`[$]`	`=`	Prints the line number of the addressed line. '`.`' is not changed.
	`!`*command*	Sends *command* to *MS-DOS* to be executed. If the first character of the *command* is `!`, it is replaced by the last *command* executed by another `!` command in `ed`. '`.`' is left unchanged.
`[.+1]`		An address alone on a line is equivalent to the command `p`. A ⟨*CR*⟩ alone on a line is equivalent to the command '`.+1p`'.

Expands tabs to blanks in character files

Synopsis: `expand [-`*tabsize*`] [-`tab_1`,`tab_2`,...]`*file(s)*

Expands tabs to blank characters in character files given as argument, prints the result to the console (*stdout*). If no file arguments are given or one of them is "-" the corresponding input is taken from the console (*stdin*).

Description:

By default tab stops are put every 8 characters. If the option `-`*tabsize* is given, they are put instead every *tabsize* characters.

If instead the option `-`tab_1`,`tab_2`,...` is given, tab stops are put at columns tab_1, tab_2, *etc...* (origin 0).

See Also:

`unexpand`.

find files with certain attributes and execute commands on each

Synopsis: `find` *pathname-list predicate*

`find` searches for files matching *predicate*, down in the directory hierarchy below each argument of the *pathname-list*, or by default, below the current directory.

Description:

predicate is made of *primary predicates*, which are keywords preceded by a `-` and followed by 0, 1 or more *arguments*, and combined with logical *operators*. The *operators* are, in order of increasing precedence:

- logical or, represented by the argument `-o` appearing between two predicates.
  For example, `-name *.bak -o -name *.tmp` is true for each file whose extension is `.bak` or `.tmp`.
- logical and, which is implicitly represented by the juxtaposition of two predicates.
  For example `-name *.bak -mtime 0` is true for each file whose extension is `.bak`, and which has been created or modified during the last 24 hours.
- the negation, which is represented by the argument `!`, preceding a predicate.
  For example `! -name *.bak` is true for each file whose extension is not `.bak`.
- Arguments consisting of parentheses are used to group predicates, changing the default order of precedence of the operators.
  For example `( -name *.bak -o -name *.tmp ) -mtime 0` is true for each file whose extension is `.bak` or `.tmp`, and which has been created or modified during the last 24 hours. (since parentheses are just normal arguments on the command line, they must be preceded and followed by at least one space).

By default, directories are looked at before their subdirectories and files. The end of *pathname-list* (i.e. the beginning of *predicate*) is indicated by the first argument beginning with "`-`" or "`(`".

Syntax of "*predicate*":

The syntax of *predicate* may be described by the following formal grammar (in the description, "iff" stands for "if and only if" and "`|`" stands for "or"):

⟨*predicate*⟩
:= ⟨*conjunctive*⟩

⟨*predicate*⟩ is true iff ⟨*conjunctive*⟩ is true.

| ⟨*conjunctive*⟩ `-o` ⟨*conjunctive*⟩

⟨*predicate*⟩ is true iff one of the two ⟨*conjunctives*⟩ is true.

⟨*conjunctive*⟩
:= ⟨*term*⟩

⟨*conjunctive*⟩ is true iff ⟨*term*⟩ is true.

| ⟨*term*⟩ ⟨*term*⟩

⟨*conjunctive*⟩ is true iff the two ⟨*terms*⟩ are true.

⟨*term*⟩
:= ⟨*primary predicate*⟩

⟨*term*⟩ is true iff ⟨*primary predicate*⟩ is true.

| `!` ⟨*primary predicate*⟩

⟨*term*⟩ is true iff ⟨*primary predicate*⟩ is not true.

⟨*primary predicate*⟩
:= `(` ⟨*predicate*⟩ `)`

⟨*primary predicate*⟩ is true iff ⟨*predicate*⟩ is true.

| ⟨*primary predicate*⟩

One of the following predicates defined by keywords:

⟨*primary predicate*⟩
:= `-name` *pattern*

⟨*primary predicate*⟩ is true iff the name of the current file is matched by *pattern*. *pattern* may contain wild-cards which are expanded according to the usual rules for filename argument (for a precise description, look in the general section of the documentation).

| `-perm` *permission*

⟨*primary predicate*⟩ is true iff the file has the given permission. Two values can be specified for *permission*:
`r`: true for a read-only file.
`w`: true for a writable file.

| `-type` *filetype*

⟨*primary predicate*⟩ is true iff the file is of the given type. Two values can be specified for *filetype*:

`f`: true for an ordinary file.
`d`: true for a directory.

| `-size` *value*

⟨*primary predicate*⟩ is true iff the size of the file (given in kilobytes) matches the given *value*. Three forms are recognized for *value*; *n* below is an integer:

n: true for files whose size is exactly *n* kilobytes.
+*n*: true for files whose size is more than *n* kilobytes.
-*n*: true for files whose size is less than *n* kilobytes.

| `-mtime` *value*

⟨*primary predicate*⟩ is true iff the file has been modified a number of days ago matching the given *value*. Three forms are recognized for *value*; *n* below is an integer:

n: true for files modified exactly *n* days ago.
+*n*: true for files modified more than *n* days ago.
-*n*: true for files modified less than *n* days ago.

| `-newer` *filename*

⟨*primary predicate*⟩ is true iff the current file has been created or modified more recently than *filename*.

| `-exec` *command*

The *command* is sent to *MS-DOS* to be executed, where *command* is a sequence of arguments ending with a "`;`". If one of the arguments is `{}`, this argument is replaced by the current filename. The resulting ⟨*primary predicate*⟩ is true iff the executed *command* returns an exit status of 0 (success). For example,

```
-exec grep -sw signal {} ;
```

is true for the files which contain at least one occurrence of the word `signal`.

| `-ok` *command*

Like "`-exec`", but *command* is echoed to the terminal before execution and the user is asked wether it should be executed. If the answer is negative, ⟨*primary predicate*⟩ is false. For example,

```
-ok cat {} ;
```

asks if the current file should be copied to the terminal; if the answer is positive, the following predicates will be applied on the current file

after its printing. If the answer was negative, `find` works on the next file.

| `-print`

This ⟨*primary predicate*⟩ is always true, and causes the current pathname to be printed on the standard output.

| `-depth`

This ⟨*primary predicate*⟩ is always true, and forces the directories to be looked at after their files or sub-directories.

Beware: operators (`!`, `-o`, `(`, `)`) must be separated by one or more spaces from the predicates, arguments and other operators.

Examples:

In order to delete the files older than a week whose extension is `.bak` or `.tmp`, under the directory `\applis` or its subdirectories:

```
find \applis ( -name *.bak -o -name *.tmp ) -mtime +7 -exec rm -i {} ;
```

Error Messages:

`predicate-list error`

The analysis has found the end of *predicates*, but the command line is not finished.

`unbalanced parentheses`

There is a missing closing parenthesis.

`predicate xxx unknown`

or

`xxx found when expecting predicate`

Another token was found at a place where a keyword predicate was expected.

`incomplete statement`

The argument of `-exec` or `-ok` doesn't end with a "`;`".

`can't access xxx`

The argument of `-newer` cannot be found or looked at.

`xxx: no match`

An argument in *pathname-list* doesn't exist.

Portability:

The `-depth` option is an enhancement.

See Also:

Command line expansion of "`//`" in the section entitled "Calling the Utilities".

Search for a text pattern in files

Synopsis: `grep [`*options*`] [`*pattern*`] [`*files*`]`

`grep` works on the given argument files. If no argument or the argument `-` has been given, `grep` takes its input from *stdin*.

Description:

`grep` searches for occurrences of a pattern (regular expression) in each of the argument files and gives on *stdout* the list of lines where the pattern has been found. Video attributes are used to show the part of the line which matches the specified pattern. To get more information on the syntax and usage of regular expressions, look at the Appendix.

Options:

The possible options on the command line are:

`-s` This option tells `grep` to give no output but to report the result of the search with a return code as follows:
`0` no match found. `1` a match found. `2` some error.

`-c` Only give a count of matched lines for each file.

`-l` Only give the names of the files containing a match.

`-h` Do not output in front of matched lines the name of the file where the line was found (the default is to output it).

`-n` Give line numbers of matched lines.

`-t` Stop the search at the first match in each file.

`-i` Do not take into account lower-case / upper-case distinction when searching.

`-w` Match only complete words.

`-x` Match only complete lines.

`-v` Instead of giving lines containing a match, give lines which *do not* contain a match.

`-e` *expr* Giving a pattern as argument to the `-e` option allows one to give a pattern beginning by the character `-`, and can also be used to look for several patterns simultaneously (if there are multiple `-e` options).

`-f` *file* This option also allows several patterns; the argument specifies a file containing patterns to look for, giving one per line.

`-V` Take the pattern *verbatim*, i.e. do not interpret any of the special regular expression operators.

Examples:

In the following examples we represent different video attributes by different fonts.

```
C:\TC\MCALC>grep video *.c

    mcdisply.c:/* Prints a string in video memory at a selected location */
```

The following example gives the names of all functions in a **C** program file whose name have less than 16 characters, as long as their name is given at the beginning of a line.

```
C:\>grep "^[a-zA-Z][0-9a-zA-Z]\{0,15\} \{0,1\}(" file.c

    cleanscreen()
    winnie (mess)
    show2 ()
    GetExp(c)
```

Notes:

`grep` was created around 1973; it was soon considered too slow. `fgrep` (fast `grep`) was then written — it can handle several words at once, each without metacharacters. `egrep` (extended `grep`) came later, incorporating every feature and adding many more (for instance `|`, which means "or", and is not yet implemented in our `grep`). `grep` should disappear on *UNIX*, but has not, and in fact is a nuisance, as many *UNIX* users will type `grep` when they should use `egrep`. Our `grep` is very close to `egrep`, and the use of option `-V` turns it into `fgrep`.

Portability:

The use of different video attributes to highlight key parts of the output is an enhancement.

Options `-w`, `-V` and the second part of option `-e` are also enhancements.

Display the head of one or several files

Synopsis: `head [-`*number*`] [`*file(s)*`]`

Copies on *stdout* the first lines of the argument(s) *file(s)*. If no files are given, `head` takes its input from the console (*stdin*).

Description:

By default, `head` copies the first 10 lines of each file to the output. If the option `-`*number* is given, it copies instead the first *number* lines.

If more than one file is given, each is listed preceded by a header of the form "`==>` *filename* `<==`". Thus for instance the command:

```
head -999 *.c
```

is a convenient way to list with headers all your short `.c` files.

relation join of two files

Synopsis: `join [`*options*`]` *file1 file2*

Description:

Does a *join* (cartesian product) of the two argument files. That is, lines of each file are interpreted as *records* divided into *fields* (separated by whitespace (a sequence of blanks or tabs) by default). Each file should be sorted for one of its fields (by default the first field) which represents the *join key*. For each couple of lines, one for each file, which have the same join field, a combined record is output which by default is the join field followed by the other fields in the first file and then the other fields in the second file.

Options:

The possible options are:

`-t`*c*	Use character *c* as field separator (default is whitespace).
`-a`*n*	Produce output for lines which do not match. If *n* is specified as 1 (resp. 2) then output only unpaired records from *file1* (resp. *file2*).
`-j`*n m*	Use field *m* as join field in *filen* (*n* absent means in both files).
`-e` *s*	On output, an empty field is replaced by string *s*.
`-o` *n.m* ...	This option specifies the fields to write to the output lines: each output line will have as many fields as arguments of the form *n*`.`*m* following this option (until the first argument not of this form which may be the end-of-options "`--`" specifier). An argument *n*`.`*m* specifies field *m* of *filen*.

list files and directories

Synopsis: `ls [`*options*`] [`*arguments*`]`

`ls` gives the list of files and directories matched by the arguments, first files, then directories and their contents, sorted by name. Used with the wild-card expansion built in *The Berkeley Utilities*, the various options of `ls` create a powerful way to look at the contents of a disk.

Description:

`ls` puts in front of the list (and of any sublist relative to a directory whose contents are listed) a header indicating the number of entries and the space taken by the files in the list.

Options:

list selection options:

`-a` List of all entries, including "hidden" files, system files, '`.`' and '`..`'.

`-A` Same list but omits '`.`' and '`..`'.

`-D` List only directories.

`-d` Do not list the content of the arguments which are directories, just give their names.

`-F` List only files.

`-R` Recursively list subdirectories. When combined with the option `-1`, names are given with their complete pathnames.

sort options:

The default is an alphabetic sort on filenames.

`-e` Alphabetic sort by extension.

`-t` Sort by time of last modification, the most recent time listed first.

`-L` Sort by decreasing size.

`-r` Reverse the specified sorting order.

`-f` Do not sort.

report options:

`-x` The list is laid out left to right, line by line (the default is top to bottom and left to right, column by column).

`-1` Give the list on a single column (this is the default if the output of `ls` is redirected to a file).

`-C` Multi-column report, with a header for each directory (this is the default when the output of `ls` is to a terminal).

`-M` Gives only the headers (useful with the option `-R` to make a survey of a subdirectory hierarchy).

`-m` List the entries with their full pathnames, separated by commas.

`-p` Decorate directory names by appending to their names a `\`.

`-s` Give the size in bytes of each entry.

`-T` Give last time modified for each entry.

`-l` Complete list: corresponds to options `-sT`, with in addition 3 indicators `xxx` where the first `x` is one of {`-dsc`} where `-` means ordinary file, `d` directory, `s` system file, `c` special character file (e.g. `con:` or `prn:`), the second `x` is one of {`-h`} meaning ordinary file or hidden file and the third is one of {`rw`} meaning read-only or read-and-write.

size options:

`-U[`*unit*`]` or `U`*number of bytes* Rounds up individual sizes to a multiple of the cluster size of *unit*. By default the disk unit of the first argument is taken. Free space left on the disk is also given. An explicit cluster size (in bytes) may given instead of a unit name. This option allows us to know the actual size taken by the files on the disk (the operating system always allocates an integral number of clusters to a file), and also to know the size they would take if they were transferred to the disk whose unit was given as argument to the `-U` option. It worth noting that a given set of files probably uses a lot more space on a hard disk than on a floppy because its clusters are usually larger.

Examples:

- List each of the subdirectories with its size (a very valuable piece of information when the disk is nearly full):

```
C:\>ls -RM
c:\                                              5 entries      115900 bytes
c:\dease\                                       15 entries      552217 bytes
c:\jc\                                          11 entries      146038 bytes
```

- List all the EXEcutable files with a filename starting in e, somewhere in or below the current directory:

```
D:\MSC>ls .\\e*.exe
.\\e*.exe                                        7 entries      102539 bytes
.\bin\errout.exe    .\bin\exepack.exe   .\exe2bin.exe       .\me\bin\exp.exe
.\bin\exemod.exe    .\errshow.exe       .\me\bin\ech.exe
```

- List the sidekick subdirectory and show the space it would take if transferred to a: (237056 bytes for the contents, 237568 bytes if the directory sidekick is also created on a:); we see that there is enough space left on a: to transfer it (there are 803840 bytes left):

```
D:\>ls -Ua \sidekick
\sidekick\                                      10 entries      237056 bytes
notes           read-me.sk      sk.hlp          skinst.com      skm.com
phone.dir       sk.com          skc.com         skinst.msg      skn.com
        === total: 237568 bytes= 20% of capacity of unit a:  ===
unit a: 512 bytes/sector 512 bytes/cluster 803840 bytes left (66.2%)
```

Portability:

The option -A is taken from the Berkeley Unix system.

The use of different video attributes to highlight key parts of the output is an enhancement.

The options -e, -L, -M, -T and -U are also enhancements.

See Also:

du, df; command line expansion (see section "Calling the Utilities").

update files

Synopsis: `make [`*options*`] [`*targets*`] [`*definitions*`]`

`make` finds out the minimum sequence of commands needed to update a program or a group of programs when some of the files they depend on have been modified (are more recent) and then executes that sequence of commands.

Description:

Among the arguments that are not options, those containing a "`=`" are treated separately (see macro definitions), the others are the *targets* to update. `make`reads one or more *description files* specified by an `-f` option (see below), and if there was no `-f` option, by default, the file "`makefile`" in the current directory. `make`interprets the contents of these files as a sequence of rules giving the dependencies between the *targets* (non-existent names or file names) and other files, and the actions to execute (*methods*) in order to create a target if the file doesn't exist, or to update it if it exists and is older than its dependents. `make`updates the *targets* specified on the command line or the first "real" *target* found in the first *description file* if no target was specified on the command line, or all *targets* if the `-a` option has been given. Rules and methods are applied recursively, i.e. if one *dependent* is not up to date, it will be updated before going on. Whether *description file* is explicitly given on the command line or not, `make`first loads the *built-in* rules except if the `-r` option is given. Those *built-in* rules are taken from the file `make.ini` if such a file can be found in the directories specified by the environment variable `path` (starting with the current directory), otherwise `make` will use only the few rules compiled into it. In order to send the *built-in* rules to the terminal, enter:

```
make -f nul -p
```

(see the `-p` option below).

Contents of *description files***:**

- "`#`" starts a comment and everything appearing between a "`#`" and the end of the line is ignored, the newline included.
- Blank lines are also ignored but may be used to terminate *entries*.
- Lines that do not begin with ⟨*Tab*⟩ and containing a "`=`" not preceded by a "`:`" are macro definitions.

- The following lines are grouped to form *entries*:
  - The first line of an *entry*, called a *rule*, must be a non-empty sequence of blank delimited *targets*, followed by "`:`" or "`::`", and followed by a *dependents* list that may be empty. *targets* and *dependents* are sequences of characters representing legal file specifications. Drive specifications (ex.: `a:`) are accepted, but bring a new constraint for the syntax of this line: if the *targets* and the *dependents* are separated by an single "`:`" and if the last target is a one-character name, this "`:`" must be followed by at least one blank. The *targets* beginning with a "`.`" and containing neither "\" nor "`/`" are called *pseudo-targets*.
  - The end of the line after an eventual "`;`" , and the following lines beginning with a ⟨*Tab*⟩, are the commands to execute in order to update *targets* if the updated *dependents* are newer. These commands are called *methods*. A line that doesn't begin with a ⟨*Tab*⟩ or a "`#`" terminates the entry. A *method* may be several lines long if every line except the last one ends with a "\". The ⟨*Tab*⟩ at the beginning of a *method* may be followed by one or both of the characters "`-`" and "`@`". "`-`" causes `make` to ignore an error status returned by this given *method* even if the `-i` option was not given, and "`@`" prevents displaying that *method* before execution except if the `-n` option was given (see those options below).
- A *target* may appear several times, with the following restrictions:
  - A *target* may not appear left of "`:`" and later left of "`::`".
  - If a *target* followed by "`:`" appears in several *entries*, only one of these *entries* may contain *methods*. If a *dependent* in any *entry* is newer than *target*, these *methods* are executed, and the *inference* rules are not examined. If no *method* was given, `make` looks for an inference rule to apply (see below).
  - If a *target* followed by "`::`" appears in several *entries*, several of these *entries* may contain lines of *methods*. In which case, if a *dependent* in an *entry* is newer than the *target*, the *methods* given for this *entry* are executed. The *inference* rules are also executed if applicable.
  - The *dependents* of each occurence of a *target* accumulate, except for the *pseudo-target* "`.SUFFIXES`".

Pseudo-targets:

The following *pseudo-targets* have a special meaning for `make`:

`.SUFFIXES` The dependents have a name beginning with a ".", usually conventional suffixes for files. They are used by the *inference rules* (see below). An entry "`.SUFFIXES`" without any *dependents* cancels all formerly declared suffixes.

`.DEFAULT` If `make`finds no rules, neither explicit ones nor inference rules giving methods for a *target*, and if an entry "`.DEFAULT`" is found, the methods following this entry will be applied.

`.PRECIOUS` If an user interrupt (`^C`) takes place during the updating of a file, this file is deleted unless it is a dependent of the *pseudo-target* "`.PRECIOUS`".

`.IGNORE` This *pseudo-target* forces the "ignore errors" mode, just as if the `-i` option had been given on the command line.

`.SILENT` This *pseudo-target* forces the "silent" mode just as if the `-s` option had been given on the command line.

The other *pseudo-targets* recognized by `make`are those built by catenating two *suffixes*, and are called *inference rules*. Typically, an entry for a *pseudo-target* "`.c.obj`" specifies a method to update the file *name*`.obj` from the file *name*`.c` if no explicit rule is found, i.e. if the *target name*`.obj` doesn't appear explicitly.

Examples:

Let us look at an example of a `makefile` :

```
# compiler options:
CFLAGS=/DLINT_ARGS
# linker options:
LFLAGS=/noi
#
# to cancel the eventual suffixes in "make.ini" ,
# and accept only the ".c" et ".obj" suffixes:
#
.SUFFIXES:
.SUFFIXES: .c .obj
# compile method:
.c.obj:
    msc $(CFLAGS) $*;
# list of all source files:
FILES=main.c sub1.c sub2.c
# main target, building the application:
main.exe: $(FILES:.c=.obj)
    link $(LFLAGS) $(FILES:.c);
#auxiliary target, for floppy backup:
backup:$(FILES)
    cp $? a:
    touch backup -f
```

The lines beginning with a `#` are ignored. The lines "`CFLAGS=...`" and "`LFLAGS=...`" are *macro definitions*, used to give a values to the compiler and linker options (which can be easily changed by arguments on the `make`command line or with environment variables; see below). The next lines cancel the suffixes possibly declared in the built-in rules and declare as only recognized suffixes `.c` and `.obj`; next the default rule to obtain an *object module* from a **C** *source* is given. Then a new macro is defined, and the *targets* `main.exe` and `backup` are successively declared, with their *dependents* and the *methods* to update them. The *target* `main.exe` is the main target, i.e. the default target if no *target* is given on the command line, because it is the first true *target* occuring in the *description file*.

Macro substitution:

In any line of a *description file*, the character "`$`" starts a substitution. Macro calls may take several forms:

1 `$(x)`, where x is a string containing neither "`(`" nor "`:`".

2 `$(x:y)`, where x contains no ")" and y contains neither ")" nor "=".
3 `$(x:y=z)`, where neither x nor y nor z contain any ")".

The parentheses may be replaced by curly brackets; in form 1, if `x` is a one-character string, they may be left out. In form 1, `x` is replaced by the "value" of the so called macro (by the empty string if the macro is not defined). In form 3, for each substitution, all non overlapping occurences of the string `y` in the value of `x` are replaced by `z`. Form 2 is similar to form 3 with `z` being an empty string.

For example, with the preceding `makefile` , after substitution, the entry of the file dealing with "`main.exe`" becomes:

```
main.exe: main.obj sub1.obj sub2.obj
    link /noi main sub1 sub2;
```

The part before the definition of "`FILES`" in `makefile` could be in `make.ini`.

Some *macros* are predefined in `make`, and can't be explicitly assigned to: The following *macro* always has the same value:

- `$$` always has the value `$`.

The following *macros* see their values changed according to the current rule:

- `$*` is meaningful only in the methods for an *inference* rule. Its value is the *dependent*'s name less the suffix, which is also the name of the *target* less the suffix.
- `$<` is meaningful only in a method for an *inference* rule or a `.DEFAULT` rule. Its value is the entire *dependent*'s name.
- `$@` is meaningful only in a method for an explicit rule. Its value is the entire *target*'s name.
- `$?` is meaningful only in a method for an explicit rule. Its value is the list of the dependents that should be updated.

The first three *macros* have two variants: if "`F`" is appended (e.g., `$(*F)`), the "directory" part of the name is stripped from the value. If "`D`" is appended (e.g., `$(@D)`), only the "directory" part is kept (`.\` if that part would be empty). The other *macros* may explicitly be given values in the following ways:

1 On the command line, through an argument of the form `x=v`; these definitions take effect before any other action and can't be modified by redefinitions in `make.ini` or a *description file*.
2 Indirectly, through environment variables. Any environment variable which has a value is considered as defining a macro with the same name and value.
3 By a line of the form `x=v` in the *description file*.
4 Through the execution of a method whose command is `set` (which thus defines an environment variable).

By default, 2 takes place before 3, and definitions in the *description files* supersede previous ones. The `-e` option changes that default but the environment variable `MAKEFLAGS` is always read first.

For example, with the preceding `makefile` , if the files `main.c` and `sub2.c` have been modified since the last `main.exe`, the command

```
make
```

will force the execution of:

```
msc /DLINT_ARGS main.c;
msc /DLINT_ARGS sub2.c;
link /noi main sub1 sub2
```

The command

```
make CFLAGS= LFLAGS=/noi/exepack
```

will force the execution of:

```
msc main.c;
msc sub2.c;
link /noi/exepack main sub1 sub2
```

and the command

```
make backup
```

will force the execution of:

```
cp main.c sub2.c a:
```

Another *macro* has a special meaning for `make`: The presence of `$(MAKE)` in a method forces the execution of that line even if the `-n` option is given. (see below).

> Beware : for the *macros* in the rules (up to an eventual "`;`"), the substitution takes place when *description file* is read. But in the lines of *methods*, the substitutions are computed again before each execution.

Options:

The following options are available on the command line:

`-a` Update every true *target* of the given *description files*.

`-p`*x* Print the information used by `make`on standard output; *x* may be a subset of:

- `m` Macros.
- `s` Suffixes
- `i` Inference rules.
- `e` Explicit rules.

If no *x* is given, "`msie`" is the default.

`-f` *file* *file* is the name of a *description file*. By default, `make` takes *makefile*. The `-` argument causes `make` to use *stdin*.

The following options invoke modes and may be forced if the corresponding letter is found in the environment variable "`MAKEFLAGS`". After their value is established, the new value of `MAKEFLAGS` is computed and exported to subshells.

`-d` Prints the information about the files and their dates on which `make`bases its decisions.

`-e` The values of the environment variables override the macro definitions of the *description files*.

`-i` Normally an exit status different from 0 returned by a *method* causes `make`to terminate execution. If this option is given, `make`will ignore error codes returned by the commands. This option is forced if the *pseudo-target* "`.IGNORE`" appears in the *description files*.

`-k` If the `-i` option is not given and if a command fails, `make`continues the execution of the entries not depending on the current *target*.

`-q` Checks if *target* is up to date: returns a status code of `0` if it is, `-1` otherwise.

`-n` Execute no commands: just display them on the terminal, including commands preceded by "`@`". Commands on *method* lines which contain the "`$(MAKE)`" macro will nevertheless be executed.

`-r` Do not use built-in rules.

`-s` "silent" mode: Do not display the commands before execution. This option is forced if the *pseudo-target* "`.SILENT`" appears in the *description files*.

`-t` "touch" (give the current date to) the *targets* without executing any command.

Error Messages:

The following errors are fatal:

`cannot open xxx`

Couldn't open a *description file* specified by an `-f` option.

`Bad character c (hex x), line d`

A lexical error has been found in a *description file*.

`syntax error`

A syntax error has been found in a *description file*.

`yacc stack overflow`

The *description file* is too complex for the syntax analyser.

`Must be a separator on rules line xxx`

Either a "`:`" in a rule line, or a ⟨*Tab*⟩ in a *method* line is missing.

`description file error`

An error has been detected in a *description file*.

`fatal error executing xxx`

A command returned a status different from 0.

`cannot execute xxx`

Couldn't execute a command: no executable with this name could be found in the directories specified by the environment variable `path`, or there was not enough memory available to execute it, or the command line length was more than 127 characters (an *MS-DOS* limitation). This last case may come from too long a macro expansion, especially `$?`.

`interrupted by user`

A `^C` has been sent by the user.

`Excessive macro nesting level`

A nested macro definition exceeds `make`'s capacity.

`symbol table overflow`

The number of targets and dependents exceed `make`'s capacity.

`line too long`

A line from a *description file* exceeds the size of the line input buffer (2500 characters).

`out of memory`

`make`needs more memory for this job.
The following error is fatal only if the `-k` option is not given.

`don't know how to make xxx`

The required rule couldn't be found.
The following messages are warnings:

`\$? list too long.`

Buffer overflow during expansion of `$?`, which is then truncated.

`file xxx does not exist.`

In case of a `-t` option, a target file needed to be created by "touch" before it could be set to the current date.

`Cannot touch xxx`

In case of a `-t` option, a file could not be set to the current date.

`Inconsistent rules lines for 'xxx'`

A target can't be followed once by "`:`" and later by "`::`".

`Multiple rules lines for 'xxx'`

A target followed by "`:`" is in two or more entries containing methods.

`nothing to make`

`make`couldn't find any target to update.

`no suffix list.`

No suffixes will be recognized by `make`.

`xxx removed`

Following a `^C`, the file being currently updated has been deleted.

Bugs:

The return status given by *MS-DOS* commands is not very consistent: use the `-i` option to overcome this difficulty.

Portability:

The ability to issue a command like `a:make .exe:` is an enhancement. So is the use of `make.ini`.

See Also:

`touch`.

Text files browser

Synopsis: `more [`*options*`] [`... *files*`]`

`more` works on the list of files you entered on the command line (wildcards are allowed), or directly on *stdin* if no argument was given, for instance in case of a pipe.

Description:

`more` is an interactive utility which enables the user to view one or more files on the screen.

Options:

The following options are available on the command line:

`-t`*n* — *n* is an integer, giving the tab size. By default, *n* is 8. If *n* is 0, tabs will not be interpreted.

`-f` — Lines longer than the screen width are not folded.

`-a` — ANSI escape sequences which specify screen attributes are not interpreted.

`-T` — Make tabs visible by giving them a different attribute (the fourth attribute in the environment variable `VATTR`).

`-E` — Make empty space (ie parts of the screen which do not correspond to any text in the file) visible by giving it a different attribute (the third attribute in the environment variable `VATTR`).

`-l`*n* — Gives an *n* lines display if your video card allows it. Currently the values 25, 43 if you have an EGA or VGA card, and 28 and 50 if you have a VGA card are accepted. By default the current number of lines of your display is used.

`+`*line* — Start the display at line number *line*. Lines are numbered beginning at 1.

`-e`*pattern* — Start the display at first occurence of this *pattern*. *pattern* is a regular expression in the style of "`ed`". For more information, consult specific documentation about regular expressions.

`-w` — Match only on complete words.

`-i` — No case significance in regular expression matching.

Most `more` commands are unechoed one-character commands with immediate effect. Some commands are longer, entered interactively on the command line (the last line of the screen, which also serves as a status line), and need a carriage-return (⟨*CR*⟩) before processing. The last screen line is reserved for various messages and statistics, and user input. It gives the name of the displayed file, the percentage currently read and receives the ⟨*CR*⟩ending commands (during input of such a command, you may edit it with the arrow keys, ⟨*backspace*⟩, the ⟨*Home*⟩ and ⟨*End*⟩ keys, the ⟨*Del*⟩ key, ⟨*Escape*⟩ which clears the whole command, and you may switch between insert and overwrite mode with the ⟨*Ins*⟩ key). The commands available when inside `more` may be sorted into different classes, according to their usage:

Getting on-line help:

`h` or `F1`	Open a "help window": all commands listed below are briefly described in the two pages of the help window.

Moving inside a file:

Most of these commands may be preceded by an integer argument.

⟨*space*⟩	Display the next screenful of the file.
`f`	Idem.
⟨*PgDn*⟩	Idem.
⟨*CR*⟩	Scroll forward one line.
↓	Idem.
`^U`	Scroll forward one half screen.
`b`	Skip backward and display the previous screenful of the file.
⟨*PgUp*⟩	Idem.
↑	Scroll backward one line.
`^D`	Scroll backward one half screen.
`G`	Go to line number n where n is the integer argument given before the command. This line is displayed as the first line on the screen if there are enough lines in the file to do so. If n is absent, goes to the end of the file.

The following commands take no argument:

⟨*Control Home*⟩	Go back to beginning of file.

⟨*Control End*⟩	Go to end of file.
← or →	Scroll laterally one column. This command is only allowed if the option `-f` was given on the command line, or the toggle `o` (see below) is not set, i.e. if long lines are not folded.
⟨*Home*⟩	Go back to the first column (in case of lateral scrolling).

Changing file:

`+`	If you asked to view several files, the `+` command closes the current file and displays the *n*th next file given on the command line. *n* is either the argument, or, by default the next file.
`:n`⟨*CR*⟩	Idem.
`-`	If you asked to view several files, the `-` command closes the current file and displays the *n*th previous file given on the command line.
`:p`⟨*CR*⟩	Idem.
`x`	Display the list of the files that you entered on the command line in a window. The name of the current file appears highlighted. You may use the following commands to move inside this window: arrow keys, ⟨*Home*⟩ and ⟨*End*⟩, ⟨*PgUp*⟩ and ⟨*PgDn*⟩. Hit ⟨*enter*⟩ to display the file whose name appears highlighted. If you change your mind and don't wish to switch files, just hit ⟨*Escape*⟩.
`:n`*list*	Be careful, the `:n` without any argument has a different meaning. This one permits you to change the list of the files that you wish to view. Just enter `:n` followed by the new list. The use of wildcards is allowed.
`q` or `:q`⟨*CR*⟩ `Q` or `:Q`⟨*CR*⟩	Quit `more`.

Searching for regular expressions:

In order to get the most out of the regular expressions feature, refer to the Appendix.

`/`*reg. exp.*⟨*CR*⟩	Search forward for *reg. exp.*.
`?`*reg. exp.*⟨*CR*⟩	Search backward for the *reg. exp.*.
`n`	Search for the next occurence of the last regular expression entered. This command keeps the same direction of search.

`N` Search for the the last regular expression entered in the reverse direction of search.

`i` Toggle case significance on searches.

`w` Toggle match on complete words only in searches.

If found, the expression is highlighted on screen with a different video attribute (the second attribute in the environment variable `VATTR`; if the regular expression has subexpressions, they are themselves highlighted with further attributes taken in sequence from `VATTR`).

Calling system interface:

`:!`⟨*CR*⟩ Give control to the `shell`.

`:!`*command*⟨*CR*⟩ Execute *command*.

`v` Call selected editor. This command is only available if you previously gave a value to the environment variable "`EDITOR`" through the *MS-DOS* command "`set`", for instance:

```
set EDITOR=VI.
```

You may make the editor start on the current line of the file, if there is such a start option on your editor, by placing a `%d` marker in `EDITOR`. For instance, `vi` starts on line 321 of `foo` if the command `vi +321 foo` is given; so if you use `vi`, set `EDITOR=vi +%d` to have it start on the current line of the file you are browsing.

Controlling presentation:

`t` Change tab size (to the numeric argument n) n absent or 0 means disable tab interpretation.

`T` Make tabs visible by giving them a different attribute (the fourth attribute in the environment variable `VATTR`).

`o` Toggle folding of long lines.

`a` Toggle interpretation of ANSI attribute sequences in the displayed file.

`E` Make empty space (ie parts of the screen which do not correspond to any text in the file) visible by giving it a different attribute (the third attribute in the environment variable `VATTR`).

Move files and directories

Synopsis: `mv [`*options*`]` source target

or `mv [`*options*`]` file|dir ... file|dir dir

`mv` moves files or directories matched by the pathnames given as argument.

Description:

There are two forms of the command:

- short form: there are only two arguments, and both arguments consist of one file, or both of one directory, or the second argument is a new name. The first argument is moved over the second (target).
- long form: the last argument is a directory (target) and all other arguments are moved to that target directory.

Watch out when using wild cards (like `file.*`), as the target must expand to at most one name.

Options:

The possible options on the command line are:

`-r` Allows `mv` to move (and possibly overwrite) non-empty directories (if not given, only empty directories are moved or overwritten).

`-m` When moving directories, merges the source with the target (instead of overwriting the target).

`-v` Gives on *stdout* a report on moved files.

`-f` Do not ask confirmation before overwriting read-only files (by default the authorization of the user is asked).

`-i` Asks confirmation before overwriting any file or directory.

`-I` Asks confirmation before moving any file or directory. This option implies the `-i` option.

When the options `-i` or `-I` are given, the only answers allowed are:

`n`: continue, do not overwrite or move.
`q`: leave.
`g`: (go) stop asking questions.
`y`: overwrite or move.
`s`: answer valid only **for a directory**. Overwrite or move without asking further confirmations for files or sub-directories of this directory.

Examples:

```
mv -I c:sources\*.*\*.bak a:
```

Moves all files `*.bak` in sub-directories of directory `sources` to diskette `a:`, asking confirmation for each file.

See Also:

`cp` and `rm`.

Portability:

The option `-r` is partly from the BSD version and partly an enhancement. The options `-m`, `-v`, `-i` and `-I` are enhancements.

"dump"

Synopsis: `od [`*options*`] [`*file ... file*`]`

Description:

`od` dumps the contents of its file argument(s) in one or several formats specified by the option arguments (by default, the options `-c` and `-h` are taken).

Options:

The possible options are:

`-c` Interprets consecutive bytes as ASCII characters. Non-printable characters are represented according to the same conventions as in the **C** language.

`-h` Interpret bytes in unsigned hexadecimal.

`-b` Interpret bytes in unsigned octal.

`-d` Interpret 2-byte words in unsigned decimal.

`-s` Interpret 2-byte words in signed decimal.

`-o` Interpret 2-byte words in unsigned octal.

`-x` Interpret 2-byte words in unsigned hexadecimal.

`-D` Interpret 4-byte words in unsigned decimal.

`-S` Interpret 4-byte words in signed decimal.

`-O` Interpret 4-byte words in unsigned octal.

`-X` Interpret 4-byte words in unsigned hexadecimal.

`+`*n* Starts dumping at the *n*th byte from the beginning of the file. The number *n* can be specified in octal (`0`*ddd*) or in hexadecimal (`0x`*ddd*) as well as in decimal. The running count of characters in the dump output will be given in the same format (decimal, octal or hexadecimal) as the format specified here.

`-r` Swap the 2 bytes before interpreting 2-byte words.

`-R` Swap the two 2-byte words before interpreting 4-byte words.

Portability:

The use of different video attributes to highlight key parts of the output is an enhancement.

The options `-D`, `-S`, `-O`, `-X`, `-r`, `-R` are also enhancements.

Set default options for *The Berkeley Utilities*

Synopsis: `opts [-a] [-e]` *pgm-name* `["`*new-opts*`"]`

Description:

`opts` sets or displays the initial command line, which is catenated to the actual command line upon execution, for the Berkeley utility *pgm-name*. It should be executed in the directory where the utility's `.exe` file resides. The most likely use of this program is to set default options for the utility.

If "*new-opts*" is not given, the current initial command line for *pgm-name* is just displayed. If "*new-opts*" is given, it replaces (or is appended to, if option `-a` was given) a previous initial command line.

Options:

The possible options are:

`-a` Append *new-opts* to a previous initial command line instead of replacing it.

`-e` Enter an interactive editing mode where the initial command line of *pgm-name* is displayed in a window ready to be edited. This mode provides support for choosing ANSI attribute escape sequences from a menu where their visual effect is displayed.

merge files as columns of a single file

Synopsis: `paste [-d`*list*`] [`*files*`]`

or `paste -s [-d`*list* `[`*file*`]]`

`paste`merges the lines of the *files* entered as arguments according to the options specified on the command line. If no *files* are specified or the file name `-`, `paste`works on the standard input.

Description:

`paste`merges corresponding lines of each *file* separating them in the result with a selected character, or concatenates subsequent lines of a single *file* (`-s` option). `paste`sends the result on the standard output. By default, the binding character is the tab character `\t`, unless option `-d` has been given (see the description of the `-d` option below). Whether the option `-d` has been given or not, the lines from the last file are always followed by the character `\n` (newline). When the option `-s` has been given, only the last line of the *file* is followed by a newline, unless a newline also appears in the *list* given with the `-d` option.

Option:

`-d`*list* *list* is a list of characters to use as the concatenation character. This list is used circularly, i.e. is reused when exhausted. The list is also restarted for every new cycle on the input files to give a line of output, if the option `-s` was not given. As mentioned above, the lines from the last file will be followed by a newline, not by a character from the *list*. The *list* may contain special characters encoded as follows: `\n` (newline), `\t` (tab), `\` (backslash) and `\0` (null). As usual, an argument containing characters which have a special meaning for DOS must be quoted by `"`'s.

Examples:

```
C:\>cat junk
love apples
hate raisins
eat  oranges
C:\>cat junk.bak
apples 12 \kilos
raisins 14 \pounds
oranges 23 \units
C:\>paste junk junk.bak
love apples     apples 12 \kilos
hate raisins    raisins 14 \pounds
eat  oranges    oranges 23 \units
```

See Also:

cut.

Redirects error output of commands

Synopsis: `rederr` *file command*

It is possible in *MS-DOS* to redirect the standard (*stdout*) output of commands with the > operator, but *MS-DOS* does not provide any way to redirect the error (*stderr*) output. This program solves the problem.

The first argument is the name of the *file* where you want to redirect the error output, and the following argument(s) is the *command* (given with its arguments, if it has any) whose error output you want to redirect.

Notes:

This command is not available in *UNIX*, but its function is. It is a *UNIX* standard that programs send their error messages to *stderr*. This standard is not followed by all *MS-DOS* programs.

Remove files and directories

Synopsis: `rm [`*options*`]` $file_1$ ... $file_n$

`rm` removes files or directories.

Description:

Wild-cards (like `file.*`) are allowed to specify the arguments. Before overwriting a read-only file, `rm` asks for confirmation; also `rm` will not remove a non-empty directory unless the option `-r` has been given.

Options:

The possible options on the command line are:

`-r` Recursively remove the contents of sub-directories.

`-v` Write on *stdout* a report on removed files.

`-f` Do not ask confirmation before overwriting read-only files or directories.

`-i` Ask confirmation before removing any file or directory.

The valid answers to questions are:
`n`: continue, do not remove.
`q`: leave.
`g`: (go) stop asking questions.
`y`: remove.
`s`: answer valid only **for directories**. Remove without asking further confirmations for files or sub-directories in this directory.

Examples:

```
C:>rm -i *.*
```

A safer way to clean up than "`del *.*`".

Portability:

The option `-r` comes from Berkeley's unix version, and the options `-v` and `-i` are enhancements.

See Also:

`ls`, `cp`, `mv`.

Stream editor

Synopsis: `sed [-n] [-e` *script*`] [-f` *script file*`] [`*files*`]`

`sed` copies the *files* entered as arguments to the standard output according to the commands given on the command line (option `-e`), or in a file script (option `-f`). If there is no `-f` option and only one `-e` option, the flag `-e` is optional.

Description:

Normally, `sed` cyclically reads a line of input and copies it into the pattern space. Every command is then executed, if its address matches the pattern space. At the end of the script, `sed` copies the pattern space on the standard output before deleting it.

Some commands also use a hold space, where they save all or part of the pattern space for a possible later use.

Regular expressions are used to specify addresses (lines) or, for some commands, parts of lines, as in the `s` command. If you want more information about the use of regular expressions, consult the Appendix.

Options:

The only available option on the command line is:

`-n` Suppress the default output of the pattern space at the end of each cycle.

Adresses:

An address is either an integer giving the number of the input line concerned or a "context" address, i.e. a regular expression in the style of `ed` but modified as follows:

The regular expression may be constructed as \?RE?, where ? represents any character. In order to get this character without its special meaning inside of the expression, you can just escape it with a \. This construction is exactly identical to /RE/.

A period (.) matches any character except the ending newline of the pattern space.

A command with no addresses selects every pattern space.

A command with one address selects every pattern space that matches this address.

A command with two addresses selects the range from the first pattern space that matches the first address through the pattern space that matches the second address, and again till the end of the last file.

The escape sequence '`\n`' matches a newline embedded in the pattern space.

Commands:

It is possible to group several commands under one address with a pair of curly brackets: `{}`. Commands are applied on the lines selected by the specified addresses. In order to have them applied to the lines *not* selected, just prefix the command with `!`. This is also allowed for groups of commands.

The following commands are preceded by their maximum number of permissible addresses in parentheseses.

The argument ⟨*text*⟩ of the first three commands may consist of several lines, in which case each line, except the last one, must end with `\`.

`(1) a\`
⟨*text*⟩ — Appends ⟨*text*⟩ after the pattern space. The result appears on the output just before reading the next input line.

`(2) c\`
⟨*text*⟩ — Deletes the contents of the pattern space, appends ⟨*text*⟩ and copies it to the output, after the pattern specified by zero or one address, or at the end of a two-addresses range. Starts the next cycle immediately.

`(1) i\`
⟨*text*⟩ — Places ⟨*text*⟩ immediately on the output.

The following commands take a file name as argument: this name must be preceded by exactly one blank and should terminate the line. There can't be more than ten files opened in write access at the same time and all such files are opened before processing.

`(2) r`*rfile* — Reads the contents of *rfile* and places them on the output just before reading the next input line.

`(2) w`*wfile* — Appends the contents of the pattern space to *wfile*.

The `s` command also has a file name as argument among its four optional flags:

(2) s/*Reg.Exp.*/*subst*/*flags* Replaces occurrences of the regular expressions in the pattern space with the string *subst*. You may use any character instead of /. One or more of the following *flags* may be added to the s command:

- *n* *n* is an integer in the range 1 - 512. The substitution will only occur on the *n*th occurrence of the regular expression.
- g Global: The substitution will occur on all occurrences of the regular expression.
- p Prints the pattern space only if a substitution occurred.
- w *wfile* Appends the contents of the pattern space to *wfile*, only if a substitution occurred.

Other commands:

(2) b *label* Branches to the : command bearing *label*. If no *label* is specified, branch to the end of the script.

(2) d Deletes the contents of the pattern space. Starts immediately the next cycle.

(2) D Deletes the contents of the pattern space, through the first newline. Starts immediately the next cycle.

(2) g Replaces the contents of the pattern space by the contents of the hold space.

(2) G Appends the contents of the hold space to the contents of the pattern space.

(2) h Replaces the contents of the hold space by the contents of the pattern space.

(2) H Appends the contents of the pattern space to the contents of the hold space.

(2) l Copies the contents of the pattern space to the standard output in an unambiguous form: control characters are represented by the conventional "caret-letter " sequence. Long lines are folded.

(2) n Prints the contents of the pattern space on the standard output. Deletes the pattern space and gets the next line of input.

(2) N Appends the next line of input to the pattern space with an embedded newline.

(2) `p`	Prints the contents of the pattern space on the standard output.
(2) `P`	Prints the contents of the pattern space through the first newline on the standard output.
(1) `q`	Quit `sed`.
(2) `t`*label*	Conditional branch to the `:` command bearing *label* if a substitution occurred since the most recent reading of input line or the last execution of a `t` command. If *label* is not specified, branch to the end of the script.
(2) `x`	Exchange the contents of the pattern space and the contents of the hold space.
(2) `y/`$string_1$`/`$string_2$`/`	Replaces all occurrences of characters that belong to $string_1$ with the corresponding character in $string_2$. Both strings must have the same length.
(1) `=`	Prints on the standard output the current input line number.
(0) `:`*label*	This command doesn't do anything: it only bears a label for the `b` or `t` commands.
(2) `{`	Execute all commands that follow `{` through the corresponding (balanced) `}` if the given address(es) select(s) the pattern space.

A script line beginning with a `#` is a comment line. If this line is the first line in the script and if the `#` is immediately followed by a `n`, the default output of the pattern space will be suppressed, as in the `-n` option of the command line.

See Also:

`ed`, `awk`, `diff`, `grep` and the Appendix on regular expressions.

Sort files

Synopsis: `sort [`*options*`] [+`*coldsc*`[-`*coldsc*`]] [`*file* ... *file*`]`

`sort` works on the files entered on the command line, sorts them line by line and sends the result to *stdout*. If no files arguments are given or if `-` was given as an argument, `sort` uses *stdin*.

Description:

`sort` works on the fields specified on the command line by the column descriptors (*coldsc*) `m.n`, where `m` represents a number of fields delimited by white space to skip from the beginning of the line and `n` a number of characters to skip further from the beginning of the field `m`. A *coldsc* preceded by `+` (`+m.n`) indicates the beginning of a range of columns that `sort` will use for comparisons and *coldsc* preceded by `-` (`-m.n`) indicates the end of the range. There may be several descriptor pairs `+m.n -m.n` specifying sorting for several ranges. The ranges given first are the most significant; if a `+`*coldsc* is given alone (without any `-`*coldsc*), the range goes to the end of the line. By default, `sort` sorts on the whole line. The following options are available on the command line:

`-c` Checks whether the file is already sorted. If it is, doesn't do anything, else sends a message.

`-m` Each argument file must be sorted, `sort` just merges them.

`-o`*file* *file* will be used instead of *stdout*. The *file* is allowed to be the same as one of the argument files.

`-T`*dir* The temporary files created by `sort` during its work will be written in the directory *dir*.

`-t`*x* Changes the fields delimiter to the character *x* (instead of the default: whitespace, i.e. blank, tab and newline).

Each of the seven following options may be added to a column descriptor and thus affect only the corresponding fields:

`-u` If several lines are found identical for the concerned fields, `sort` will output only one of them.

`-b` Ignore leading blanks when comparing fields.

`-d` Use "dictionary" order, i.e. only letters, digits and blanks will be significant in comparisons.

`-f` Use a collating sequence where each lower-case letter is immediately followed by the corresponding upper-case letter: `aA bB cC...`.

`-i` Lines with characters whose ASCII code doesn't belong to the range 32-126 will be appended to the output file.

`-n` A numeric string is sorted according to numerical ordering. This option implies option `-b`.

`-r` Reverse sorting order.

Examples:

Suppose we have the file "`fruits`" with the following contents:

```
C:>cat fruits
  apples          10 K
  apricots         3 K
  kiwis            2 K
  raspberries      5 K
  pears            7 K
  bananas          4 K
```

Let us have a look at the output of the following commands:

```
C:>sort fruits | tee fruits.t
  apricots         3 K
  bananas          4 K
  raspberries      5 K
  kiwis            2 K
  pears            7 K
  apples          10 K
C:>sort +1 -b fruits
  kiwis            2 K
  apricots         3 K
  bananas          4 K
  raspberries      5 K
  pears            7 K
  apples          10 K
```

Suppose a second file "vegetables" is already sorted:

```
C:>cat vegetables
  artichokes       2 K
  carrots          5 K
  leeks            1 K
C:>sort -m fruits.t vegetables
  apricots         3 K
  artichokes       2 K
  bananas          4 K
  carrots          5 K
  raspberries      5 K
  kiwis            2 K
  leeks            1 K
  pears            7 K
  apples          10 K
```

Notes:

This program is greatly superior to *MS-DOS*'s SORT.

Split a file into smaller pieces

Synopsis: `split [-` *number* `[`*file* `[`*name*`]]]`

`split` splits the given argument file in pieces which have the given *number* of lines specified on the command line. By default, `split` takes its input from *stdin* if no filename has been given and the maximum *number* of lines per piece is 1000 if no *number* has been given.

Description:

The names given to the pieces are built from the *name* given on the command line to which `split` adds the suffix `aa`, then `ab`, `ac`, ... and so on in alphabetical order. If no *name* has been specified, `split` uses by default the name `x`.

Display the end of a file

Synopsis: `tail [+|- [`*number*`]] [`*options*`] [`*file*`]`

`tail` writes on (*stdout*) the file entered as argument, starting at the specified location. If no file is specified, `tail` uses *stdin*.

Description:

Depending on the sign preceding *number*, the display will start at some distance from the beginning of the file (`+`*number*) or from the end of the file (`-`*number*). *number* represents an offset which by default is a number of lines.

If no *number* is given, the default is 10 lines, unless option `-r` is present, whence the default consists of displaying the whole file upside down.

Options:

`-c` makes the *number* offset represents a number of characters rather of a number of lines.

`-r` This option is special, and incompatible with the option `-c`. `tail -r` displays the lines in the opposite order to the initial file.

Bugs:

Cannot show a tail of more than 55K.

pipe connection and derivation

Synopsis: `tee [`*options*`] `*file*` [... `*file*`]`

`tee` diverts to all argument *files* the output of a pipe (as well as transmitting it faithfully to *stdout*).

Description:

The possible options are:

`-i` Ignore interrupts.

`-a` If one of the output *files* already exists, append data to the end of the file rather than overwrite the file.

the first option, which is useful under *UNIX*, does not make much sense under *MS-DOS*, since pipes are executed sequentially and not asynchronously.

Examples:

```
ls -tT *.c | tee abc | more
```

This redirects the output of `ls` to `more` where you can browse it, and simultaneously makes a copy in the file `abc`, which you can consult at your leisure later on.

updates files timestamp

Synopsis: `touch [`*options*`] [-`*date*`]` *file* `[...` *file*`]`

`touch` gives the current date and time to the files specified on the command line.

Description:

`touch` can work on empty files. Furthermore, if you ask `touch` to work on a non-existent file, it will ask you if you want to create this file.
If no *date* or *agefile* (see below) has been specified, `touch` updates its arguments with the current date and time, otherwise, `touch` interprets the given date according to the following format:

`[[[`*YY*`]`*MM*`]`*DD*`]`*hhmm*`[.`*ss*`]`

with the following meaning:

- *hh*: Hour, compulsory.
- *mm*: Minutes, compulsory.
- *ss*: Seconds, optional; by default: `0`.
- *DD*: Day, optional; by default: current day. If no day is given, no month or year can be specified.
- *MM*: Month (`1` = January, `2` = February, ... `12` = December), optional; by default: current month. If no month is given, no year can be specified.
- *YY*: Year (counted above `1900`), optional; by default: current year.

Options:

`-c`	Create a new file without asking for a confirmation.
`-f` *agefile*	Take the date as being equal to that of given *agefile*.
`-i`	Ask for a confirmation before touching any file.
`-r`	Allow `touch` to work recursively in directories.
`-v`	Write on *stdout* a report on touched files, including their initial date.

Notes:

UNIX internally counts seconds starting with 1970.
MS-DOS internally counts seconds starting with 1980.

Translate stdin to stdout

Synopsis: `tr [`*options*`] `*string*$_1$ `[`*string*$_2$`]`

`tr` takes its input from *stdin*, replaces characters occurring in *string*$_1$ by the corresponding characters in *string*$_2$, with possible variations, and writes the result on *stdout*. This is useful for some simple file transformations (see examples below).

Description:

string$_1$ and *string*$_2$ both specify a character set as follows:

- if *string*$_2$ is too short its last character is replicated.
- in either string character ranges in the form `a-z` are accepted (specifying characters between `a` and `z` in the ASCII order)
- to specify the characters `-` or `\` you must escape them by preceding them with a `\`.
- the following standard "C" notation is recognized:

  `\`*nnn* specifies the character of code octal *nnn*.
  `\x`*nn* specifies the character of code hexadecimal *nn*.
  `\n` specifies the character "newline".
  `\t` specifies the character "tab".
  `\b` specifies the character "backspace".
  `\r` specifies the character "carriage return".

Options:

There are three options which modify the translation:

`-d` Delete characters in *string*$_1$ (do not use *string*$_2$).

`-s` (squeeze) Output only one of a sequence of identical characters obtained from *string*$_2$ (which may correspond to one or several different characters from *string*$_1$).

`-c` *string*$_1$ is replaced by its complement set (amongst all characters with codes 0 to 255) taken in ascending order.

Examples:

- to look at the text in `file1` in uppercase, type

```
tr a-z A-Z <file1
```

- copy all words from `file1` to `file2`, one per line (here a word is a sequence of alphabetic characters):

```
tr -cs A-Za-z \n <file1 >file2
```

this works by translating all characters *not* (option `-c`) alphabetic to newlines and then by "squeezing" (option `-` s) consecutive newlines to one newline.

- display `file`, omitting from the display all control characters (codes 1 to 37 octal):

```
tr -d \001-\037 <file
```

Bugs:

Mimicking its *UNIX*counterpart, `tr` will delete all ASCII NULL (`\000`) from its input and will not handle them in *string1* or *string2*

See Also:

`ed`, `sed`.

Compresses runs of tabs and blanks in character files

Synopsis: `unexpand [`*options*`]`*file(s)*

Compresses initial runs of tabs and blank characters to optimal such sequences in each line of the character files given as argument and prints the result to the console (*stdout*). If no file arguments are given or one of them is "`-`" the corresponding input is taken from the console (*stdin*).

Options:

`-`*tabsize* By default tab stops are every 8 characters; if the option `-`*tabsize* is given, they are instead every *tabsize* characters.

`-a` By default only initial runs of blanks and spaces are optimized. If the option `-a` is given, all such runs are optimized even if they don't start a line.

See Also:

`expand`.

Weed out or find repeated lines

Synopsis: `uniq [`*options*`] [`*input file*`[`*output file*`]]`

Description:

`uniq` writes on the *output file* (default *stdout*) only one of a sequence of identical lines found in the *input file*.

Options:

The possible options are:

- `-u` Only output unique (non-repeated) lines.
- `-d` Only output repeated lines.
- `-c` Put in front of each output line the number of times it was repeated.
- `-`*n* Skip *n* whitespace-delimited fields at the beginning of lines before comparing them for identity.
- `+`*n* Skip *n* characters before comparing lines. This option can be combined with the previous one, fields are skipped before characters.

Examples:

```
tr -cs A-Za-z \n <document | sort | uniq -c
```

gives on *stdout* the list alphabetically sorted of all words in the file `document`, each given once preceded by its number of occurrences.

See Also:

`sort`, `comm`.

Count words and lines

Synopsis: `wc [`*options*`] `*file*` [... `*file*`]`

`wc` counts characters, words or lines of the file arguments.

Description:

If no file argument or the argument "`-`" has been given, `wc` works in *stdin*.

Options:

The possible options are:

`-c` Count characters.

`-w` Count words.

`-l` Count lines.

If none of these options has been given, all three are considered active. If there is more than one file argument, `wc` also gives a total for all the files.

Find which version of a program is active

Synopsis: `which [-a]` *name*

`which` *name* returns which version of *name* would be picked up by the operating system for execution (that is, if it is a `*.bat`, `*.com` or `*.exe` version and the complete pathname). We recall that *MS-DOS* scans the current directory, then all directories specified by the `PATH` variable, to find executable files; `which` does the same search. `which` prints nothing if no version was found.

Option:

`-a` Find all versions of *name* along path; tell which is active, and print complete information about all encountered files (in "`ls -l`" format).

Extract character strings from **C** programs

Synopsis: `xstr [`*options*`] `*file*` [... `*file*`]`

`xstr` works on the **C** source files given as arguments on the command line.

Description:

`xstr` reads a **C** source file, extracts all character strings from it and gathers them to a file named by default `x.h`. Each string appears in `x.h` only once, whatever its actual number of occurences in the **C** source files. Strings appear in `x.h` in lines of the form `#define S`*xxxx* `"..."` , where *xxxx* is a 4-digit integer. The **C** source file is rewritten to a file having the same name, but with a suffix `x` added to the extension (e.g., given a file `a.c`, `xstr` will write a file `a.cx`). In this new file, `xstr` will have replaced character strings by references of the form `S`*xxxx* and added the directive "`#include x.h`".

Options:

The possible options on the command line are:

`-c` Leave the character strings in comments besides `S`*xxxx* references in the new source files output by `xstr`.

`-o`*name* Use *name* instead of `x` as a prefix of the files `x.h`.

`-r` This "restring" option reinserts character strings in files output by a previous call to `xstr` reading them from `x.h`. Hence this option will only work if source files are the output of `xstr`. This option can be combined with the option `-c` in order to suppress comments from files output by a previous call to `xstr -c`. It can also be combined with `-o` in order to use another file than `x.h` to get the strings from.

`xstr` can be used to survey and normalize the set of strings appearing in a set of **C** source files, in particular to translate them into another language. It is useful anyway since it reduces the size of the object files by making sure that no string is repeated twice.

Examples:

```
C:>cat test.c
main(){printf("Hello, world!");}
C:>xstr test.c
writing test.cx ...
writing x.h
C:>cat test.cx
#include "x.h"
main(){printf(S0000);}
C:>cat x.h
#define S0000 "Hello, world!"
C:>ed -s x.h
s/Hello, world/Bonjour, le monde/
w
q
C:>xstr -r test.cx
reading x.h
writing test.cxx ...
C:>cat test.cxx
main(){printf("Bonjour, le monde!");}
```

This example shows how in principle xstr can be used to translate messages, particularly with repetitions. In addition, if the file `x.h` output by the first call to `xstr` is kept, the production of a French version can be largely automated as follows: Apply `xstr` to `test.c` as above, translate `x.h` to French and rename it `test.f`. Then write a `makefile` containing the methods to use to (re)-make a French version:

```
xstr test.c
cp test.f x.h
xstr -r test.cx
msc $(CFLAGS) test.cxx;
```

these will be enough for any modification of `test.c` which does not modify the number of strings or the order in which they appear. Otherwise, reexecute:

```
xstr test.c
```

And then compare `x.h` and `test.f`: most of the time, there will be but a few strings to insert and to delete in `test.f`. The string numbers will then be false, but can be corrected by executing:

```
awk -f renum test.f > x.f
mv x.f test.f
```

where `renum` is the following `awk` program:

```
{print substr($0,1,6) sprintf("%04d",NR-1) substr($0,11,length($0)-10)}
```

See Also:

`make`, `awk`.

Regular expressions are mostly used to look for character strings in text files. If you just want to find instances of the word "`search`", the corresponding regular expression is simply given as "`search`". But the use of special characters allows the simultaneous search of several different strings or of classes of strings described by a single regular expression.

In the current implementation, matches are recognized only within a line; there is no way to specify the character "newline" in a regular expression.

Special characters are used in regular expressions to specify regular expression operators. Actually, the list of characters having a special meaning depends on the utility where the regular expression is used (`ed`, `grep`, `awk`, ...) though the list of available operators is the same. This is done as follows: in each context there is a list of 'magic' characters which have special meaning, and other characters specifying operators must be quoted by preceding them by a `\` to be understood; and to specify the match of a 'magic' character you must quote it with a `\`.

For instance, the operator `+` (meaning "one or more repetitions") is invoked by `+` or `\+` depending on whether it is 'magic' or not. It is 'magic' in `awk` and `grep -E` ("egrep"), but not in the other utilities like `ed` and normal `grep`. So in `awk` you write `a+` to match one or more `a`'s, but you write `a\+` to match the string "`a+`"; while in `grep` you must write `a\+` to match one or more `a`'s and `a+` to match the string `a+`.

'`\`' is always 'magic', so you must quote the `\` itself, and specify it by '`\\`'. The following characters are 'magic':

`[].*\` always.

`^` at the beginning of an expression or as the first character encountered within brackets `[]`.

`$` at the end of an expression.

`()<>+?` in `awk` and in `grep -E`.

An exception to the above rules is that inside brackets `[]`, no character (except `^` as explained above) is magic.

Meaning of the special characters:

`[]` A set of characters within brackets `[]` is a regular expression which matches any of the characters within brackets. It is possible to specify a range of characters (in the ASCII collating sequence) using the `-` character: for instance, `[a-z]` matches any character whose ASCII code is between that of `a` and that of `z`, that is any lower-case letter. However the character `-` given as the first one after `[` or the last one before `]` is taken for itself. If the first character after `[` is `^` the match will be with characters which *are not* in the specified set (i.e. which are in the complement set in the ASCII table). To specify a `[` or a `]` it must appear as the first character after `[`. For instance `[]a-z]` matches '`]`' or any lower-case letter; `[^]a-z]` is also allowed and matches any character but `]` and lower-case letters.

`.` The dot is a regular expression matching *any* one character.

`^` We have seen its meaning within brackets. It has also a special meaning as the first character of a complete regular expression: it indicates that the expression will match a string only if that string starts at the beginning of a line.

`$` Similarly, the dollar has a special meaning as the last character of a complete regular expression, and indicates that the expression will match a string only if that string ends with the last character of a line.

It is of course possible to use both `^` and `$` to build an expression which matches only complete lines.

More complex regular expressions can be built from the previous ones using the following operators (special characters):

The concatenation of several regular expressions is an expression which matches the concatenation of the strings matched by each expression.

`*` A star immediately following an expression indicates repetition: the expression will match any number (0 or more) of consecutive occurences of the string matched by the expression which the '`*`' follows. When there are several possible such matches, the longest one is chosen.

`+` Is similar, but this time the match will occur for 1 or more occurences of the string matched by the expression which the '`+`' follows. As in the previous case, the longest possible match is chosen.

`?` Is again similar, but the match occurs for 0 or 1 occurence (the longest possible match is chosen).

`\{`m`\}` Also indicates repetition; m must be a positive integer less than 256. The match occurs for exactly m occurrences.

`\{`*m*`,\}`	Is similar; the match occurs for a number of occurences equal to or greater than *m* (the longest match is taken when there are several possible solutions).
`\{`*m*`,`*n*`\}`	*n* must also be a positive integer less than 256. The match occurs for a number of occurences between *m* and *n*. The longest match is again taken in case of doubt.
`\(`*expression*`\)`	The parentheses do not modify the meaning of the enclosed expressions; they are just used to capture a subexpression which gives meaning to the following form:
`\`*n*	Where *n* is a single digit matches exactly the same string as the one which has just been matched by the *n*th sub-regular expression enclosed within `(` and `)` in the same expression (the occurence of `\`*n* must of course be *after* the *n*th occurence of `(` in the expression).
`\<`	Word beginning. A match will occur only with a string which at the corresponding place does not contain a digit or a letter or is not immediately preceded by a digit or a letter (the character "`_`" is considered as a letter).
`\>`	Word end. A match will occur only with a string which at the corresponding place does not contain a digit or is not immediately followed by a digit or a letter (the character "`_`" is considered as a letter).

Examples:

The regular expression:

```
(<[a-zA-Z]{5,7}>).*\1
```

matches only 2 occurrences of the same 5 to 7 letter word, on the same line and separated by any number of characters. This is `awk` or `grep -E` ("egrep") syntax; the same expression must be written in `ed` or `grep` as

```
\(\<[a-zA-Z]\{5,7\}\>\).*\1
```