From: sahayman@iuvax.cs.indiana.edu (Steve Hayman)
Newsgroups: comp.unix.questions,comp.unix.wizards
Subject: Frequently Asked Questions about Unix - with Answers [Monthly posting]
Message-ID: <FAQ2.9@iuvax.cs.indiana.edu>
Date: 1 Dec 89 19:54:14 GMT
Expires: 1 Jan 90 05:00:00 GMT
Followup-To: comp.unix.questions
Organization: Computer Science Department, Indiana University
Lines: 865
Supersedes: <FAQ2.8@iuvax.cs.indiana.edu>

[Last changed: $Date: 89/12/01 14:50:10 $ by $Author: sahayman $]

This article contains the answers to some Frequently Asked Questions
often seen in comp.unix.questions and comp.unix.wizards. Please don't
ask these questions again, they've been answered plenty of times
already - and please don't flame someone just because they may not have
read this particular posting. Thank you.

This article includes answers to:

While these are all legitimate questions, they seem to crop up in comp.unix.questions on an annual basis, usually followed by plenty of replies (only some of which are correct) and then a period of griping about how the same questions keep coming up. You may also like to read the monthly article "Answers to Frequently Asked Questions" in the newsgroup "news.announce.newusers", which will tell you what "UNIX" stands for.

With the variety of Unix systems in the world, it's hard to guarantee that these answers will work everywhere. Read your local manual pages before trying anything suggested here. If you have suggestions or corrections for any of these answers, please send them to to sahayman@iuvax.cs.indiana.edu or iuvax!sahayman.

1) How do I remove a file whose name begins with a "-" ?

    Figure out some way to name the file so that it doesn't begin with a dash. The simplest answer is to use

        rm ./-filename

    (assuming "-filename" is in the current directory, of course.) This method of avoiding the interpretation of the "-" works with other commands too.

    Many commands, particularly those that have been written to use the "getopt(3)" argument parsing routine, accept a "--" argument which means "this is the last option, anything after this is not an option", so your version of rm might handle "rm -- -filename". Some versions of rm that don't use getopt() treat a single "-" in the same way, so you can also try "rm - -filename".

2) How do I remove a file with funny characters in the filename ?

    The classic answers are

        rm -i some*pattern*that*matches*only*the*file*you*want

        which asks you whether you want to remove each file matching the indicated pattern; depending on your shell, this may not work if the filename has a character with the 8th bit set (the shell may strip that off);

    and

        rm -ri .

which asks you whether to remove each file in the directory,
answer "y" to the problem file and "n" to everything else.,
and which, unfortunately, doesn't work with many versions of rm;
(always take a deep breath and think about what you're doing
and double check what you typed when you use rm's "-r" flag)

and

```
find . -type f ... -ok rm '{}' \;
```

where "..." is a group of predicates that uniquely identify the
file. One possibility is to figure out the inode number
of the problem file (use "ls -i .") and then use

```
find . -inum 12345 -ok rm '{}' \;
```

or

```
find . -inum 12345 -ok mv '{}' new-file-name \;
```

"-ok" is a safety check - it will prompt you for confirmation of the
command it's about to execute. You can use "-exec" instead to avoid
the prompting, if you want to live dangerously, or if you suspect
that the filename may contain a funny character sequence that will mess
up your screen when printed.

If none of these work, find your system manager.

3) How do I get a recursive directory listing?

One of the following may do what you want:

```
ls -R                   (not all versions of "ls" have -R)
find . -print           (should work everywhere)
du -a .                 (shows you both the name and size)
```

If you're looking for a wildcard pattern that will match
all ".c" files in this directory and below, you won't find one,
but you can use

```
% some-command `find . -name '*.c' -print`
```

"find" is a powerful program. Learn about it.

4) How do I get the current directory into my prompt?

It depends which shell you are using. It's easy with some shells,
hard or impossible with others.

C Shell (csh):

Put this in your .cshrc - customize the prompt variable the way you want.

```
alias setprompt 'set prompt="${cwd}% "'
setprompt           # to set the initial prompt
alias cd 'chdir \!* && setprompt'
```

If you use pushd and popd, you'll also need

```
alias pushd 'pushd \!* && setprompt'
alias popd  'popd  \!* && setprompt'
```

Some C shells don't keep a $cwd variable - you can use `pwd` instead.

If you just want the last component of the current directory in your prompt ("mail% " instead of "/usr/spool/mail% ") you can use

```
alias setprompt 'set prompt="$cwd:t% "'
```

Some older csh's get the meaning of && and || reversed. Try doing:

```
false && echo bug
```

If it prints "bug", you need to switch && and || (and get a better version of csh.)

Bourne Shell (sh):

If you have a newer version of the Bourne Shell (SVR2 or newer) you can use a shell function to make your own command, "xcd" say:

```
xcd() { cd $* ; PS1="`pwd` $ "; }
```

If you have an older Bourne shell, it's complicated but not impossible. Here's one way. Add this to your .profile file:

```
LOGIN_SHELL=$$ export LOGIN_SHELL
CMDFILE=/tmp/cd.$$ export CMDFILE
PROMPTSIG=16 export PROMPTSIG
trap '. $CMDFILE' $PROMPTSIG
```

and then put this executable script (without the indentation!), let's call it "xcd", somewhere in your PATH

```
: xcd directory - change directory and set prompt
: by signalling the login shell to read a command file
cat >${CMDFILE?"not set"} <<EOF
cd $1
PS1="\`pwd\`$ "
EOF
kill -${PROMPTSIG?"not set"} ${LOGIN_SHELL?"not set"}
```

Now change directories with "xcd /some/dir".

Korn Shell (ksh):

Put this in your .profile file:

```
PS1='$PWD $ '
```

If you just want the last component of the directory, use

```
PS1='${PWD##*/} $ '
```

5) How do I read characters from a terminal without requiring the user to hit RETURN?

Check out cbreak mode in BSD, ~ICANON mode in SysV.

If you don't want to tackle setting the terminal parameters yourself (using the "ioctl(2)" system call) you can let the stty program do the work - but this is slow and inefficient, and you should change the code to do it right some time:

```
main()
{
        int c;

        printf("Hit any character to continue\n");
        /*
         * ioctl() would be better here; only lazy
         * programmers do it this way:
         */
        system("/bin/stty cbreak");
        c = getchar();
        system("/bin/stty -cbreak");
        printf("Thank you for typing %c.\n", c);

        exit(0);
```

```
}
```

6) How do I read characters from the terminal in a shell script?

In sh, use read. It is most common to use a loop like

```
while read line
do
        ...
done
```

In csh, use $< like this:

```
while ( 1 )
    set line = "$<"
    if ( "$line" == "" ) break
    ...
end
```

Unfortunately csh has no way of distinguishing between a blank line and an end-of-file.

If you're using sh and want to read a *single* character from the terminal, you can try something like

```
echo -n "Enter a character: "
stty cbreak
readchar=`dd if=/dev/tty bs=1 count=1 2>/dev/null`
stty -cbreak

echo "Thank you for typing a $readchar ."
```

7) How do I check to see if there are characters to be read without actually reading?

Certain versions of UNIX provide ways to check whether characters are currently available to be read from a file descriptor. In BSD, you can use select(2). You can also use the FIONREAD ioctl (see tty(4)), which returns the number of characters waiting to be read, but only works on terminals, pipes and sockets. In System V Release 3, you can use poll(2), but that only works on streams. In Xenix - and therefore Unix SysV r3.2 and later - the rdchk() system call reports whether a read() call on a given file descriptor will block.

There is no way to check whether characters are available to be read from a FILE pointer. (Well, there is no *good* way. You could

poke around inside stdio data structures to see if the input buffer is nonempty but this is a bad idea, forget about it.)

Sometimes people ask this question with the intention of writing

```
        if (characters available from fd)
                read(fd, buf, sizeof buf);
```

in order to get the effect of a nonblocking read. This is not the best way to do this, because it is possible that characters will be available when you test for availability, but will no longer be available when you call read. Instead, set the O_NDELAY flag (which is also called FNDELAY under BSD) using the F_SETFL option of fcntl(2). Older systems (Version 7, 4.1 BSD) don't have O_NDELAY; on these systems the closest you can get to a nonblocking read is to use alarm(2) to time out the read.

8) How do I find the name of an open file?

In general, this is too difficult. The file descriptor may be attached to a pipe or pty, in which case it has no name. It may be attached to a file that has been removed. It may have multiple names, due to either hard or symbolic links.

If you really need to do this, and be sure you think long and hard about it and have decided that you have no choice, you can use find with the -inum and possibly -xdev option, or you can use ncheck, or you can recreate the functionality of one of these within your program. Just realize that searching a 600 megabyte filesystem for a file that may not even exist is going to take some time.

9) How do I rename "*.foo" to "*.bar", or change file names to lowercase?

Why doesn't "mv *.foo *.bar" work? Think about how the shell expands wildcards. "*.foo" "*.bar" are expanded before the mv command ever sees the arguments. Depending on your shell, this can fail in a couple of ways. CSH prints "No match." because it can't match "*.bar". SH executes "mv a.foo b.foo c.foo *.bar", which will only succeed if you happen to have a single directory named "*.bar", which is very unlikely and almost certainly not what you had in mind.

Depending on your shell, you can do it with a loop to "mv" each file individually. If your system has "basename", you can use:

C Shell:

```
    foreach f ( *.foo )
```

```
        set base=`basename $f .foo`
        mv $f $base.bar
    end

Bourne Shell:
    for f in *.foo; do
        base=`basename $f .foo`
        mv $f $base.bar
    done
```

Some shells have their own variable substitution features, so instead of using "basename", you can use simpler loops like:

C Shell:

```
    foreach f ( *.foo )
        mv $f $f:r.bar
    end
```

Korn Shell:

```
    for f in *.foo; do
        mv $f ${f%foo}bar
    done
```

If you don't have "basename" or want to do something like renaming foo.* to bar.*, you can use something like "sed" to strip apart the original file name in other ways, but the general looping idea is the same.

A program called "ren" that does this job nicely was posted to comp.sources.unix some time ago. It lets you use

```
    ren '*.foo' '#1.bar'
```

Shell loops like the above can also be used to translate file names from upper to lower case or vice versa. You could use something like this to rename uppercase files to lowercase:

```
    C Shell:
        foreach f ( * )
            mv $f `echo $f | tr A-Z a-z`
        end
    Bourne Shell:
        for f in *; do
            mv $f `echo $f | tr A-Z a-z`
        done
```

If you wanted to be really thorough and handle files with
`funny' names (embedded blanks or whatever) you'd need to use

Bourne Shell:

```
for f in *; do
    eval mv '"$i"' \"`echo "$i" | tr '[A-Z]' '[a-z]'`\"
done
```

If you have the "perl" language installed, you may find this rename
script by Larry Wall very useful. It can be used to accomplish a
wide variety of filename changes.

```
#!/usr/bin/perl
#
# rename script examples from lwall:
#       rename 's/\.orig$//' *.orig
#       rename 'y/A-Z/a-z/ unless /^Make/' *
#       rename '$_ .= ".bad"' *.f
#       rename 'print "$_: "; s/foo/bar/ if <stdin> =~ /^y/i' *

$op = shift;
for (@ARGV) {
    $was = $_;
    eval $op;
    die $@ if $@;
    rename($was,$_) unless $was eq $_;
}
```

10) Why do I get [some strange error message] when I "rsh host command" ?

(We're talking about the remote shell program "rsh" or sometimes "remsh";
on some machines, there is a restricted shell called "rsh", which
is a different thing.)

If your remote account uses the C shell, the remote host will
fire up a C shell to execute 'command' for you, and that shell
will read your remote .cshrc file. Perhaps your .cshrc contains
a "stty", "biff" or some other command that isn't appropriate
for a non-interactive shell. The unexpected output or error
message from these commands can screw up your rsh in odd ways.

Fortunately, the fix is simple. There are, quite possibly, a whole
bunch of operations in your ".cshrc" (e.g., "set history=N") that are
simply not worth doing except in interactive shells. What you do is
surround them in your ".cshrc" with:

```
        if ( $?prompt ) then
                operations....
        endif
```

and, since in a non-interactive shell "prompt" won't be set, the operations in question will only be done in interactive shells.

You may also wish to move some commands to your .login file; if those commands only need to be done when a login session starts up (checking for new mail, unread news and so on) it's better to have them in the .login file.

11) How do I find out the creation time of a file?

You can't - it isn't stored anywhere. Files have a last-modified time (shown by "ls -l"), a last-accessed time (shown by "ls -lu") and an inode change time (shown by "ls -lc"). The latter is often referred to as the "creation time" - even in some man pages - but that's wrong; it's the time the file's status was last changed, either by writing or changing the inode (via mv or chmod, etc...).

The man page for "stat(2)" discusses this.

12) How do I use "rsh" without having the rsh hang around until the remote command has completed?

(See note in question 10 about what "rsh" we're talking about.)

The obvious answers fail:

```
        rsh machine command &
or      rsh machine 'command &'
```

The solution - if you use csh on the remote machine:

```
        rsh machine -n 'command >&/dev/null </dev/null &'
```

If you use sh on the remote machine:

```
        rsh machine -n 'command >/dev/null 2>&1 </dev/null &'
```

why? "-n" attaches rsh's stdin to /dev/null so you could run the complete rsh command in the background on the LOCAL machine. Thus "-n" is equivalent to another specific "< /dev/null". Furthermore, the input/output redirections on the REMOTE machine (inside the single quotes) ensure that rsh thinks the session can be terminated (there's no data flow any more.)

Note: on the remote machine, you needn't redirect to/from

/dev/null; any ordinary file will do.

In many cases, various parts of these complicated commands aren't necessary.

13) How do I truncate a file?

The BSD function ftruncate() sets the length of a file. Xenix - and therefore SysV r3.2 and later - has the chsize() system call. For other systems, the only kind of truncation you can do is truncation to length zero with creat() or open(..., O_TRUNC).

14) How do I {set an environment variable, change directory} inside a shell script and have that change affect my current shell?

You can't, unless you use a special command to run the script in the context of the current shell rather than in a child program. The process environment (including environment variables and current directory) is inherited by child programs but cannot be passed back to parent programs.

For instance, if you have a C shell script named "myscript":

```
cd /very/long/path
setenv PATH /something:/something-else
```

or the equivalent Bourne or Korn shell script

```
cd /very/long/path
PATH=/something:/something-else export PATH
```

and try to run "myscript" from your shell, your shell will fork and run the shell script in a subprocess. The subprocess is also running the shell; when it sees the "cd" command it changes *its* current directory, and when it sees the "setenv" command it changes *its* environment, but neither has any effect on the current directory of the shell at which you're typing (your login shell, let's say).

In order to get your login shell to execute the script (without forking) you have to use the "." command (for the Bourne or Korn shells) or the "source" command (for the C shell). I.e. you type

```
. myscript
```

to the Bourne or Korn shells, or

```
source myscript
```

to the C shell.

If all you are trying to do is change directory or set an environment variable, it will probably be simpler to use a C shell alias or Bourne/Korn shell function. See the "how do I get the current directory into my prompt" section of this article for some examples.

15) Why doesn't find's "{}" symbol do what I want?

"find" has a -exec option that will execute a particular command on all the selected files. Find will replace any "{}" it sees with the name of the file currently under consideration.

So, some day you might try to use "find" to run a command on every file, one directory at a time. You might try this:

```
find /path -type d -exec command {}/\* \;
```

hoping that find will execute, in turn

```
command directory1/*
command directory2/*
...
```

Unfortunately, find only expands the "{}" token when it appears by itself. Find will leave anything else like "{}/*" alone, so instead of doing what you want, it will do

```
command {}/*
command {}/*
...
```

once for each directory. This might be a bug, it might be a feature but we're stuck with the current behaviour.

So how do you get around this? One way would be to write a trivial little shell script, let's say "./doit", that consists of

```
command "$1"/*
```

You could then use

```
find /path -type d -exec ./doit {} \;
```

If all you're trying to do is cut down on the number of times that "command" is executed, you should see if your system has the "xargs" command. Xargs reads arguments one line at a time from the standard input and assembles as many of them as will fit into one command line. You could use

```
find /path -print | xargs command
```

which would result in

```
command file1 file2 file3 file4 dir1/file1 dir1/file2
```

Unfortunately this is not a perfectly robust or secure solution. Xargs expects its input lines to be terminated with newlines, so it will be confused by files with odd characters such as newlines in their names.

16) How do I redirect stdout and stderr separately in csh?

In csh, you can redirect stdout with ">", or stdout and stderr together with ">&" but there is no direct way to redirect stderr only. The best you can do is

```
( command >stdout_file ) >&stderr_file
```

which runs "command" in a subshell; stdout is redirected inside the subshell to stdout_file, and both stdout and stderr from the subshell are redirected to stderr_file, but by this point stdout has already been redirected so only stderr actually winds up in stderr_file.

17) How do I set the permissions on a symbolic link?

Permissions on a symbolic link don't really mean anything. The only permissions that count are the permissions on the file that the link points to.

18) What does {awk,grep,fgrep,egrep,biff,cat,gecos,nroff,troff,tee,bss} stand for?

awk = "Aho Weinberger and Kernighan"

This language was named by its authors, Al Aho, Peter Weinberger and Brian Kernighan.

grep = "Global Regular Expression Print"

    grep comes from the ed command to print all lines matching a
    certain pattern

                g/re/p

    where "re" is a "regular expression".

fgrep = "Fixed Grep".

    fgrep searches for fixed strings only. The "f" does not
    stand for "fast" - in fact, "fgrep foobar *.c" is usually slower
    than "egrep foobar *.c" (yes, this is kind of surprising. Try it.)

    Fgrep still has its uses though, and may be useful when searching
    a file for a larger number of strings than egrep can handle.

egrep = "Extended Grep"

    egrep uses fancier regular expressions than grep.
    Many people use egrep all the time, since it has some more
    sophisticated internal algorithms than grep or fgrep,
    and is usually the fastest of the three programs.

cat = "catenate"

    catenate is an obscure word meaning "to connect in a series",
    which is what the "cat" command does to one or more files.
    Not to be confused with C/A/T, the Computer Aided Typesetter.

gecos = "General Electric Comprehensive Operating System"

    When GE's large systems division was sold to Honeywell,
    Honeywell dropped the "E" from "GECOS".

    Unix's password file has a "pw_gecos" field. The name is
    a real holdover from the early days. Dennis Ritchie
    has reported:

        "Sometimes we sent printer output or batch jobs
         to the GCOS machine. The gcos field in the
         password file was a place to stash the information
         for the $IDENT card. Not elegant."

nroff = "New ROFF"
troff = "Typesetter ROFF"

These are descendants of "roff", which was a re-implementation of the Multics "runoff" program.

tee = T

From plumbing terminology for a T-shaped pipe splitter.

bss = "Block Started by Symbol"

Dennis Ritchie says:

> Actually the acronym (in the sense we took it up; it may have other credible etymologies) is "Block Started by Symbol." It was a pseudo-op in FAP (Fortran Assembly [-er?] Program), an assembler for the IBM 704-709-7090-7094 machines. It defined its label and set aside space for a given number of words. There was another pseudo-op, BES, "Block Ended by Symbol" that did the same except that the label was defined by the last assigned word + 1. (On these machines Fortran arrays were stored backwards in storage and were 1-origin.)
>
> The usage is reasonably appropriate, because just as with standard Unix loaders, the space assigned didn't have to be punched literally into the object deck but was represented by a count somewhere.

biff = "biff"

This command, which turns on asynchronous mail notification, was actually named after a dog at Berkeley.

> I can confirm the origin of biff, if you're interested. Biff was Heidi Stettner's dog, back when Heidi (and I, and Bill Joy) were all grad students at U.C. Berkeley and the early versions of BSD were being developed. Biff was popular among the residents of Evans Hall, and was known for barking at the mailman, hence the name of the command.

Confirmation courtesy of Eric Cooper, Carnegie Mellon University

Don Libes' book "Life with Unix" contains lots more of these tidbits.

19) How do I pronounce "vi" , or "!", or "/*", or ...?

You can start a very long and pointless discussion by wondering about this topic on the net. Some people say "vye", some say

"vee-eye" (the vi manual suggests this) and some Roman numerologists say "six". How you pronounce "vi" has nothing to do with whether or not you are a true Unix wizard.

Similarly, you'll find that some people pronounce "char" as "care", and that there are lots of ways to say "#" or "/*" or "!" or "tty" or "/etc". No one pronunciation is correct - enjoy the regional dialects and accents.

Since this topic keeps coming up on the net, here is a comprehensive pronunciation list that has made the rounds in the past. Origin unknown - please let me know if you know where it came from, and I'll attribute it properly.

Names derived from UNIX are marked with *, names derived from C are marked with +, and names deserving futher explanation are marked with a #. The explanations will be given at the very end.

-- SINGLE CHARACTERS --

SPACE, blank

! EXCLAMATION POINT, exclamation mark, exclamation, exclam, excl, clam, bang#, shout, yell, shriek, pling, factorial, ball-bat, smash, cuss, wow, hey, boing

" QUOTATION MARK, quote, double quote, dirk, literal mark, rabbit ears, double ping, double glitch

CROSSHATCH, pound, pound sign, number, number sign, sharp, octothorpe#, hash, fence, crunch, mesh, hex, flash, grid, pig-pen, tictactoe, scratch, scratch mark, gardengate, gate, hak, oof, rake, sink

$ DOLLAR SIGN, dollar, cash, currency symbol, buck, string#, escape#, ding, big-money

% PERCENT SIGN, percent, mod+, shift-5, double-oh-seven, grapes

& AMPERSAND, and, amper, address+, shift-7, andpersand, snowman, bitand+, donald duck#, daemon

' APOSTROPHE, single quote, quote, tick, prime, irk, pop, spark, glitch

* ASTERISK, star, splat, spider, aster, times, wildcard*, gear, dingle, (Nathan) Hale#, bug, gem, twinkle

```
()   PARENTHESES, parens, round brackets, bananas, ears, bowlegs,
       parenthesee (singular only), weapons
(    LEFT PARENTHESIS,  paren,  so,      wax,  parenthesee,   open,  sad
)    RIGHT PARENTHESIS, thesis, already, wane, unparenthesee, close, happy

+    PLUS SIGN, plus, add, cross, and, intersection, and

,    COMMA, tail

-    HYPHEN, minus, minus sign, dash, dak, option, flag, negative,
       negative sign, worm, bithorpe#

.    PERIOD, dot, decimal, decimal point, radix point, point, spot, full stop,
       put#, floor

/    SLASH, stroke, virgule, solidus, slant, diagonal, over, slat, slak,
       across#, compress#, spare

:    COLON, two-spot, double dot, dots

;    SEMICOLON, semi, hybrid

<>   ANGLE BRACKETS, angles, funnels, brokets
<    LESS THAN,    less, read from*, from*,        in*,  comesfrom*, crunch,
       sucks
>    GREATER THAN, more, write to*,  into/toward*, out*, gazinta*,   zap,
       blows

=    EQUAL SIGN, equals, equal, gets, quadrathorpe#, half-mesh

?    QUESTION MARK, question, query, whatmark, what, wildchar*, huh, ques,
       kwes, quiz, quark, hook

@    AT SIGN, at, each, vortex, whorl, whirlpool, cyclone, snail, ape, cat,
       snable-a#, trunk-a#, rose, cabbage, Mercantile symbol

[]   BRACKETS, square brackets, U-turns, edged parentheses, mimics
[    LEFT BRACKET,  bracket,   bra, square,   opensquare
]    RIGHT BRACKET, unbracket, ket, unsquare, close

\    BACKSLASH, reversed virgule, bash, backslant, backwhack, backslat,
       escape*, backslak, bak, reduce#

^    CIRCUMFLEX, caret, carrot, hat, cap, uphat, party hat, housetop,
       up arrow, control, boink, chevron, hiccup, to-the, fang, sharkfin,
       and#, xor+, wok, trap

_    UNDERSCORE, underline, underbar, under, score, backarrow, flatworm, blank
```

` GRAVE, grave accent, accent, backquote, left/open quote, backprime, unapostrophe, backspark, birk, blugle, backtick, push, backglitch, backping

{} BRACES, curly braces, squiggly braces, curly brackets, squiggle brackets, Tuborgs#, ponds
{ LEFT BRACE, brace, curly, leftit, embrace, openbrace, begin+
} RIGHT BRACE, unbrace, uncurly, rytit, bracelet, close, end+

| VERTICAL BAR, pipe*, pipe to*, vertical line, broken line#, bar, or+, bitor+, vert, v-bar, spike, to*, gazinta*, thru*, pipesinta*, tube, mark, whack, gutter, wall

~ TILDE, twiddle, tilda, tildee, wave, squiggle, swung dash, approx, wiggle, enyay#, home*, worm

-- MULTIPLE CHARACTER STRINGS --

!?	interrobang (one overlapped character)
/*	slashterix+
*/	asterslash+
>>	appends*, cat-astrophe
->	arrow+, pointer to+, hiccup+
#!	sh'bang, wallop
\!*	bash-bang-splat
()	nil#
&&	and+, amper-amper, succeeds-then*
||	or+, fails-then*

-- NOTES --

! bang	comes from old card punch phenom where punching ! code made a loud noise
# octothorpe	from Bell System
$ string	from BASIC
$ escape	from TOPS-10
& donald duck	from the Danish "Anders And", which means "Donald Duck"
* splat	from DEC "spider" glyph
* Nathan Hale	"I have but one asterisk for my country."
= quadrathorpe	half an octothorpe
- bithorpe	half a quadrathorpe (So what's a monothorpe?)
. put	Victor Borge on Electric Company
/ across	APL
/ compress	APL
@ snable-a	from Danish; may translate as "trunk-a"

```
@ trunk-a       "trunk" = "elephant nose"
^ and           from formal logic
\ reduce        APL
{} Tuborgs      from advertizing for well-known Danish beverage
| broken line   EBCDIC has two vertical bars, one solid and one broken.
~ enyay         from the Spanish n-tilde
() nil          LISP
```

--
Steve Hayman Workstation Manager Computer Science Department Indiana U.
sahayman@iuvax.cs.indiana.edu iuvax!sahayman (812) 855-6984