File commands

ls	Directory listing
ls -al	Formatted listing with hidden files
cd dir	Change directory to dir
cd	Change to home
pwd	Show current directory
mkdir dir	Create a directory dir
rm file	Delete file
rm –r dir	Delete directory dir
rm –f file	Force remove fire
rm –rf dir	For remove directory dir
cp file1 file2	Copy file1 to file2
cp –r dir1 dir2	Copy dir1 to dir2; create dir2 if it doesn't exit
mv file1 file2	Rename or move file1 to file2. If tiled 2 is an existing directory, moves file1 into directory file2
ln –s file link	Create symbolic link link to file
touch file	Create or update file
cat > file	Places standard input into file
more file	Output the contents of file
head file	Output the firest 10 lines of file
tail file	Output the last 10 lines of file
tail –f file	Output the contents of file as it grows, starting with the last 10 lines

Process Management

ps	display all currently active processes
top	display all running processes
kill pid	kill process id pid
killall proc	kill all processes named proc *
bg	lists stopped or background jobs; resume a stopped job in the background
fg	Brings the most recent job to the foreground
fg a	brings job a to the foreground

File Permissions

chmod octal file	change the permissions of file to octal, which can be found separately for user, group, and world by adding: • 4 – read (r) • 2 – write (w) • 1 – execute (x) **Examples:** ***chmod 777*** *– read, write, execute for all* ***chmod 755*** *– rwx for owner, rx for group and world. For more options, see* **man chmod.**

SSH

ssh user@host	connect to host as user
ssh -p port user@host	connect to host on port port as user
ssh-copy-id user@host	add your key to host for user to enable a keyed or passwordless login

Searching

grep pattern files	search for pattern in files
grep -r pattern dir	search recursively for pattern in dir
command | grep pattern	search for pattern in the output of command
locate file	find all instances of file

System Info

date	show the current date and time
cal	show this month's calendar
uptime	show current uptime
w	display who is online
whoami	who you are logged in as
finger user	display information about user
uname -a	show kernel information
cat /proc /cpuinfo	cpu information
cat /proc /meminfo	memory information
man command	show the manual for command
df	show disk usage
du	show directory space usage
free	show memory and swap usage
whereis app	show possible locations of app
which app	show which app will be run by default

Compression

tar cf file.tar files	create a tar named file.tar containing files
tar xf file.tar	extract the files from file.tar
tar czf file.tar.gz files	create a tar with Gzip compression
tar xzf file.tar.gz	extract a tar using Gzip
tar cjf file.tar.bz2	create a tar with Bzip2 compression
tar xjf file.tar.bz2	extract a tar using Bzip2
gzip file	compresses file and renames it to file.gz
gzip -d file.gz	decompresses file.gz back to file

Network

ping host	ping host and output results
whois domain	get whois information for domain
dig domain	get DNS information for domain
dig -x host	reverse lookup host
wget file	download file
wget -c file	continue a stopped download

Installation

Install from source:	
./configure	
make	
make install	
dpkg -i pkg.deb	install a package (Debian)
rpm -Uvh pkg.rpm	install a package (RPM)

Shortcuts

Ctrl+C	halts the current command
Ctrl+Z	stops the current command, resume with fg in the foreground or bg in the background
Ctrl+D	log out of current session, similar to exit
Ctrl+W	erases one word in the current line
Ctrl+U	erases the whole line
Ctrl+R	type to bring up a recent command
!!	repeats the last command
exit	log out of current session
*	use with extreme caution

GNU Emacs Reference Card

(for version 27)

Starting Emacs

To enter GNU Emacs 27, just type its name: `emacs`

Leaving Emacs

suspend Emacs (or iconify it under X)	`C-z`
exit Emacs permanently	`C-x C-c`

Files

read a file into Emacs	`C-x C-f`
save a file back to disk	`C-x C-s`
save **all** files	`C-x s`
insert contents of another file into this buffer	`C-x i`
replace this file with the file you really want	`C-x C-v`
write buffer to a specified file	`C-x C-w`
toggle read-only status of buffer	`C-x C-q`

Getting Help

The help system is simple. Type `C-h` (or `F1`) and follow the directions. If you are a first-time user, type `C-h t` for a **tutorial**.

remove help window	`C-x 1`
scroll help window	`C-M-v`
apropos: show commands matching a string	`C-h a`
describe the function a key runs	`C-h k`
describe a function	`C-h f`
get mode-specific information	`C-h m`

Error Recovery

abort partially typed or executing command	`C-g`
recover files lost by a system crash	`M-x recover-session`
undo an unwanted change	`C-x u`, `C-_` or `C-/`
restore a buffer to its original contents	`M-x revert-buffer`
redraw garbaged screen	`C-l`

Incremental Search

search forward	`C-s`
search backward	`C-r`
regular expression search	`C-M-s`
reverse regular expression search	`C-M-r`
select previous search string	`M-p`
select next later search string	`M-n`
exit incremental search	`RET`
undo effect of last character	`DEL`
abort current search	`C-g`

Use `C-s` or `C-r` again to repeat the search in either direction. If Emacs is still searching, `C-g` cancels only the part not matched.

Motion

entity to move over	**backward**	**forward**
character	`C-b`	`C-f`
word	`M-b`	`M-f`
line	`C-p`	`C-n`
go to line beginning (or end)	`C-a`	`C-e`
sentence	`M-a`	`M-e`
paragraph	`M-{`	`M-}`
page	`C-x [`	`C-x ]`
sexp	`C-M-b`	`C-M-f`
function	`C-M-a`	`C-M-e`
go to buffer beginning (or end)	`M-<`	`M->`

scroll to next screen	`C-v`
scroll to previous screen	`M-v`
scroll left	`C-x <`
scroll right	`C-x >`
scroll current line to center, top, bottom	`C-l`
goto line	`M-g g`
goto char	`M-g c`
back to indentation	`M-m`

Killing and Deleting

entity to kill	**backward**	**forward**
character (delete, not kill)	`DEL`	`C-d`
word	`M-DEL`	`M-d`
line (to end of)	`M-0 C-k`	`C-k`
sentence	`C-x DEL`	`M-k`
sexp	`M-- C-M-k`	`C-M-k`

kill **region**	`C-w`
copy region to kill ring	`M-w`
kill through next occurrence of *char*	`M-z` *char*
yank back last thing killed	`C-y`
replace last yank with previous kill	`M-y`

Marking

set mark here	`C-@` or `C-SPC`
exchange point and mark	`C-x C-x`
set mark *arg* **words** away	`M-@`
mark **paragraph**	`M-h`
mark **page**	`C-x C-p`
mark **sexp**	`C-M-@`
mark **function**	`C-M-h`
mark entire **buffer**	`C-x h`

Query Replace

interactively replace a text string	`M-%`
using regular expressions	`M-x query-replace-regexp`

Valid responses in query-replace mode are

replace this one, go on to next	`SPC` or `y`
replace this one, don't move	`,`
skip to next without replacing	`DEL` or `n`
replace all remaining matches	`!`
back up to the previous match	`^`
exit query-replace	`RET`
enter recursive edit (`C-M-c` to exit)	`C-r`

Multiple Windows

When two commands are shown, the second is a similar command for a frame instead of a window.

delete all other windows	`C-x 1`	`C-x 5 1`
split window, above and below	`C-x 2`	`C-x 5 2`
delete this window	`C-x 0`	`C-x 5 0`
split window, side by side	`C-x 3`	
scroll other window	`C-M-v`	
switch cursor to another window	`C-x o`	`C-x 5 o`
select buffer in other window	`C-x 4 b`	`C-x 5 b`
display buffer in other window	`C-x 4 C-o`	`C-x 5 C-o`
find file in other window	`C-x 4 f`	`C-x 5 f`
find file read-only in other window	`C-x 4 r`	`C-x 5 r`
run Dired in other window	`C-x 4 d`	`C-x 5 d`
find tag in other window	`C-x 4 .`	`C-x 5 .`
grow window taller	`C-x ^`	
shrink window narrower	`C-x {`	
grow window wider	`C-x }`	

Formatting

indent current **line** (mode-dependent)	`TAB`
indent **region** (mode-dependent)	`C-M-\`
indent **sexp** (mode-dependent)	`C-M-q`
indent region rigidly *arg* columns	`C-x TAB`
indent for comment	`M-;`
insert newline after point	`C-o`
move rest of line vertically down	`C-M-o`
delete blank lines around point	`C-x C-o`
join line with previous (with arg, next)	`M-^`
delete all white space around point	`M-\`
put exactly one space at point	`M-SPC`
fill paragraph	`M-q`
set fill column to *arg*	`C-x f`
set prefix each line starts with	`C-x .`
set face	`M-o`

Case Change

uppercase word	`M-u`
lowercase word	`M-l`
capitalize word	`M-c`
uppercase region	`C-x C-u`
lowercase region	`C-x C-l`

The Minibuffer

The following keys are defined in the minibuffer.

complete as much as possible	`TAB`
complete up to one word	`SPC`
complete and execute	`RET`
show possible completions	`?`
fetch previous minibuffer input	`M-p`
fetch later minibuffer input or default	`M-n`
regexp search backward through history	`M-r`
regexp search forward through history	`M-s`
abort command	`C-g`

Type `C-x ESC ESC` to edit and repeat the last command that used the minibuffer. Type `F10` to activate menu bar items on text terminals.

GNU Emacs Reference Card

Buffers

select another buffer	`C-x b`
list all buffers	`C-x C-b`
kill a buffer	`C-x k`

Transposing

transpose **characters**	`C-t`
transpose **words**	`M-t`
transpose **lines**	`C-x C-t`
transpose **sexps**	`C-M-t`

Spelling Check

check spelling of current word	`M-$`
check spelling of all words in region	`M-x ispell-region`
check spelling of entire buffer	`M-x ispell-buffer`
toggle on-the-fly spell checking	`M-x flyspell-mode`

Tags

find a tag (a definition)	`M-.`
find next occurrence of tag	`C-u M-.`
specify a new tags file	`M-x visit-tags-table`
regexp search on all files in tags table	`M-x tags-search`
run query-replace on all the files	`M-x tags-query-replace`
continue last tags search or query-replace	`M-,`

Shells

execute a shell command	`M-!`
execute a shell command asynchronously	`M-&`
run a shell command on the region	`M-|`
filter region through a shell command	`C-u M-|`
start a shell in window `*shell*`	`M-x shell`

Rectangles

copy rectangle to register	`C-x r r`
kill rectangle	`C-x r k`
yank rectangle	`C-x r y`
open rectangle, shifting text right	`C-x r o`
blank out rectangle	`C-x r c`
prefix each line with a string	`C-x r t`

Abbrevs

add global abbrev	`C-x a g`
add mode-local abbrev	`C-x a l`
add global expansion for this abbrev	`C-x a i g`
add mode-local expansion for this abbrev	`C-x a i l`
explicitly expand abbrev	`C-x a e`
expand previous word dynamically	`M-/`

Miscellaneous

numeric argument	`C-u` *num*
negative argument	`M--`
quoted insert	`C-q` *char*

Regular Expressions

any single character except a newline	`.` (dot)
zero or more repeats	`*`
one or more repeats	`+`
zero or one repeat	`?`
quote special characters	`\`
quote regular expression special character *c*	`\`*c*
alternative ("or")	`|`
grouping	`\( ... \)`
shy grouping	`\(:? ... \)`
explicit numbered grouping	`\(:NUM ... \)`
same text as *n*th group	`\`*n*
at word break	`\b`
not at word break	`\B`

entity	**match start**	**match end**
line	`^`	`$`
word	`\<`	`\>`
symbol	`\_<`	`\_>`
buffer	`\'`	`\'`

class of characters	**match these**	**match others**
explicit set	`[ ... ]`	`[^ ... ]`
word-syntax character	`\w`	`\W`
character with syntax *c*	`\s`*c*	`\S`*c*
character with category *c*	`\c`*c*	`\C`*c*

International Character Sets

specify principal language	`C-x RET l`
show all input methods	`M-x list-input-methods`
enable or disable input method	`C-\`
set coding system for next command	`C-x RET c`
show all coding systems	`M-x list-coding-systems`
choose preferred coding system	`M-x prefer-coding-system`

Info

enter the Info documentation reader	`C-h i`
find specified function or variable in Info	`C-h S`

Moving within a node:

scroll forward	`SPC`
scroll reverse	`DEL`
beginning of node	`b`

Moving between nodes:

next node	`n`
previous node	`p`
move **up**	`u`
select menu item by name	`m`
select *n*th menu item by number (1–9)	*n*
follow cross reference (return with `l`)	`f`
return to last node you saw	`l`
return to directory node	`d`
go to top node of Info file	`t`
go to any node by name	`g`

Other:

run Info **tutorial**	`h`
look up a subject in the indices	`i`
search nodes for regexp	`s`
quit Info	`q`

Registers

save region in register	`C-x r s`
insert register contents into buffer	`C-x r i`
save value of point in register	`C-x r SPC`
jump to point saved in register	`C-x r j`

Keyboard Macros

start defining a keyboard macro	`C-x (`
end keyboard macro definition	`C-x )`
execute last-defined keyboard macro	`C-x e`
append to last keyboard macro	`C-u C-x (`
name last keyboard macro	`M-x name-last-kbd-macro`
insert Lisp definition in buffer	`M-x insert-kbd-macro`

Commands Dealing with Emacs Lisp

eval **sexp** before point	`C-x C-e`
eval current **defun**	`C-M-x`
eval **region**	`M-x eval-region`
read and eval minibuffer	`M-:`
load a Lisp library from **load-path**	`M-x load-library`

Simple Customization

customize variables and faces	`M-x customize`

Making global key bindings in Emacs Lisp (example):

```
(global-set-key (kbd "C-c g") 'search-forward)
(global-set-key (kbd "M-#") 'query-replace-regexp)
```

Writing Commands

```
(defun command-name (args)
  "documentation" (interactive "template")
  body)
```

An example:

```
(defun this-line-to-top-of-window (line)
  "Reposition current line to top of window.
With prefix argument LINE, put point on LINE."
  (interactive "P")
  (recenter (if (null line)
                0
              (prefix-numeric-value line))))
```

The `interactive` spec says how to read arguments interactively. Type `C-h f interactive RET` for more details.

For GNU Emacs version 27
Designed by Stephen Gildea

For more Emacs documentation, and the TEX source for this card, see the Emacs distribution, or `https://www.gnu.org/software/emacs`

v1.3

steve-parker.org

Sept 2012

Shell Scripting Cheat Sheet for Unix and Linux

Online: http://steve-parker.org/sh/sh.shtml
Book: http://steve-parker.org/shellscripting

File Redirection

> file	create (overwrite) file
>> file	append to file
< file	read from file
a | b	Pipe 'a' as input to 'b'

Common Constructs

$ while read f > do > echo "Line is $f" > done < file	read text file line by line note: "$" prompt becomes ">"
$ grep foo myfile afoo foo foobar	find lines in myfile containing the text "foo"
$ cut -d: -f5 /etc/passwd Steve Parker	get 5th field delimited by colon
$ cmd1 || cmd2	run cmd1; if fails, run cmd2
$ cmd1 && cmd2	run cmd1; if it works, run cmd2
case $foo in a) echo "foo is A" ;; b) echo "foo is B" ;; *) echo "foo is not A or B" ;; esac	act upon the value of a variable note that ";;" is required at the end of each section
myvar=`ls`	get output of ls into variable

doubleit() {
expr $1 * 2
}
doubleit 3 # returns 6
	function declaration and syntax for calling it

Test Operators

if ["$x" -lt "$y"]; then # do something fi

Numeric Tests

lt	less than
gt	greater than
eq	equal to
ne	not equal
ge	greater or equal
le	less or equal

File Tests

nt	newer than
d	is a directory
f	is a file
x	executable
r	readable
w	writeable

String Tests

=	equal to
z	zero length
n	not zero length

Logical Tests

&&	logical AND
||	logical OR
!	logical NOT

Arguments

$0	program name
$1	1st argument
$2	2nd argument
...	...
$#	no. of arguments
$*	all arguments

Variable Substitution

${V:-default}	$V, or "default" if unset
${V:=default}	$V (set to "default" if unset)
${V:?err}	$V, or "err" if unset

Conditional Execution

cmd1 || cmd2	run cmd1; if fails, run cmd2
cmd1 && cmd2	run cmd1; if ok, run cmd2

Files

mv /src /dest	move /src into /dest
ls a*	list files beginning with "a"
ls *a	list files ending with "a"
ls -ltr	list oldest first, newest last
ls -lSr	list smallest first, biggest last
ls -a	list all files, including hidden
find /src -print \ | cpio -pudvm	copy /src into current directory, preserving links, special devices, etc.

Preset Variables

$SHELL	what shell am I running?
$RANDOM	provides random numbers
$$	PID of current process
$?	return code from last cmd
$!	PID of last background cmd

Generally Useful Commands

file /etc/hosts	determine file type
basename /bin/ls	strip directory name (ls)
dirname /bin/ls	get directory name (/bin)
ifconfig -a	show all network adapters
netstat -r	show routers
netstat -a	show open ports
date +%Y%m%d	Year, Month, Day
date +%H%M	Hours, Minutes
wc -l	count number of lines
pwd	present working directory

Misc Useful Commands and Tools

egrep "(foo|bar)" file	find "foo" or "bar" in file
awk '{ print $5 }' file	print the 5th word of each line
cal 3 1973	March 1973
df -h	show disk mounts
three=`expr 1 + 2`	simple maths
echo "scale = 5 ; \ 5121 / 1024" | bc	better maths (5.00097)
time cmd	stopwatch on cmd
touch file	create blank file
alias ll='ls -l'	alias for ls -l
unalias ls	unset existing alias

find . -size 10k -print	files over 10Kb
find . -name "*.txt" -print	find text files
find /foo -type d -ls	list all directories under /foo
less file	display file page by page
sed s/foo/bar/g file	replace "foo" with "bar"
sed -i s/foo/bar/g file	in file (-i: update file)
strace -tfp PID	trace system calls for PID
tar cvf archive.tar file1 file 2 file3	create tar archive
ssh user@host	log in to host as user
scp file.txt user@host:	copy file.txt to host as user
scp user@host:/tmp/file.txt /var/tmp	copy /tmp/file.txt from user at host to /var/tmp locally
cd -	return to previous directory

Elisp Reference Sheet

Everything is a list!

- To find out more about `name` execute `(describe-symbol 'name)`!
  - After the closing parens invoke `C-x C-e` to evaluate.
- To find out more about a key press, execute `C-h k` then the key press.
- To find out more about the current mode you're in, execute `C-h m` or `describe-mode`. Essentially a comprehensive yet terse reference is provided.

Functions

- Function invocation: `(f x0 x1 ... xn)`. E.g., `(+ 3 4)` or `(message "hello")`.
  - After the closing parens invoke `C-x C-e` to execute them.
  - *Warning!* Arguments are evaluated **before** the function is executed.
  - Only prefix invocations means we can use `-,+,*` in *names* since `(f+*- a b)` is parsed as applying function `f+*-` to arguments `a, b`.
    E.g., `(1+ 42)` $\to$ `43` using function *named* `1+` —the 'successor function'.
- Function definition:

```
;; "de"fine "fun"ctions
(defun my-fun (arg0 arg1 ... argk)          ;; header, signature
  "This functions performs task ..."        ;; documentation, optional
  ...sequence of instructions to perform... ) ;; body
```

  - The return value of the function is the result of the last expression executed.
  - The documentation string may indicate the return type, among other things.
- Anonymous functions: `(lambda (arg0 ... argk) bodyHere)`.

```
;; make then way later invoke
(setq my-f (lambda (x y) (+ x y)))
(funcall my-f 1 2) ;; ⇒ 3
;; (my-f 1 2) ;; invalid!
(funcall my-f 1 2) ;; ⇒ 3
```

```
;; make and immediately invoke
(funcall (lambda (x y) (+ x y)) 1 2)

;; works, but is deprecated
((lambda (x y) (+ x y)) 1 2)
```

Functions are first-class values *but* variables and functions have **separate namespaces** —"Elisp is a Lisp-2 Language". The function represented by the name g is obtained by the call `(function g)`, which is also denoted `#'g`. The sharp quote behaves like the usual quote but causes its argument to be compiled. `lambda` is a macro that calls `function` and so there is rarely any need to quote lambdas. If `h` is a variable referring to a function, then `(funcall h x0 ... xn)` calls that function on arguments x_i.

`(apply 'g x0...xk '(xk...xn))` $\approx$ `(funcall #'g x0...xn)` $\approx$ `(g x0...xn)`

```
;; Recursion with the 'tri'angle numbers: tri n = Σ_{0≤i≤n} i.
(defun tri (f n) (if (<= n 0) 0 (+ (funcall f n) (tri f (- n 1)))))
(tri #'identity 100)              ;; ⇒ 5050
(tri (lambda (x) (/ x 2)) 100) ;; ⇒ 2500

;; Run "C-h o tri" to see TWO items! Location determines dispatch.
(setq tri 100) (tri #'identity tri)      ;; ⇒ 5050
(setq tri (lambda (x) x)) (tri tri 100)  ;; ⇒ 5050
```

→ Use `funcall` or `apply` to call functions bound to variables.

→ Refer to functions outside of function calls by using a sharp quote, `#'`.

We may have positional `optional` arguments, or optional but named arguments —for which position does not matter. Un-supplied optional arguments are bound to `nil`.

```
(cl-defun f (a &optional b (c 5))
  (format "%s %s %s" a b c))

(f 'a)      ;; ⇒ "a nil 5"
(f 'a 'b)   ;; ⇒ "a b 5"
(f 'a 'b 'c) ;; ⇒ "a b c"
```

```
(cl-defun g (a &key (b 'nice) c)
  (format "%s %s %s" a b c))

(g 1 :c 3 :b 2) ;; ⇒ "1 2 3"
(g 1 :c 3)      ;; ⇒ "1 nice 3"
```

Keywords begin with a colon, `:k` is a constant whose value is `:k`.

Quotes, Quasi-Quotes, and Unquotes

Quotes: `'x` refers to the *name* rather than the *value* of `x`.

- This is superficially similar to pointers: Given `int *x = ...`, `x` is the name (address) whereas `*x` is the value.
- The quote simply forbids evaluation; it means *take it literally as you see it* rather than looking up the definition and evaluating.
- Note: `'x` $\approx$ `(quote x)`.

Akin to English, quoting a word refers to the word and not what it denotes.
This lets us treat *code* as *data*! E.g., `'(+ 1 2)` evaluates to `(+ 1 2)`, a function call, not the value 3! Another example, `*` is code but `'*` is data, and so `(funcall '* 2 4)` yields 8.

Elisp expressions are either atoms or function application –nothing else!
'Atoms' are the simplest objects in Elisp: They evaluate to themselves;
e.g., `5, "a", 2.78, 'hello, [1 "two" three]`.

An English sentence is a list of words; if we want to make a sentence where some of the words are parameters, then we use a quasi-quote –it's like a quote, but allows us to evaluate data if we prefix it with a comma. It's usually the case that the quasi-quoted sentence happens to be a function call! In which case, we use `eval` which executes code that is in data form; i.e., is quoted.

Macros are essentially functions that return sentences, lists, which may happen to contain code.

```
;; Quotes / sentences / data
'(I am a sentence)
'(+ 1 (+ 1 1))

;; Executing data as code
(eval '(+ 1 (+ 1 1)))  ;; ⇒ 3
```

```
(setq name "Jasim")

;; Quasi-quotes: Sentences with a
;; computation, code, in them.
`(Hello ,name and welcome)
`(+ 1 ,(+ 1 1))  ;; ⇒ '(+ 1 2)
```

As the final example shows, Lisp treats data and code interchangeably. A language that uses the same structure to store data and code is called 'homoiconic'.

Reads

- How to Learn Emacs: A Hand-drawn One-pager for Beginners / A visual tutorial
- Learn Emacs Lisp in 15 minutes — `https://learnxinyminutes.com/`
- An Introduction to Programming in Emacs Lisp —also Land of Lisp
- GNU Emacs Lisp Reference Manual

Variables

- Global Variables, Create & Update: `(setq name value)`.
  - Generally: `(setq name0 value0 ··· namek valuek)`.

  Use `devfar` for global variables since it permits a documentation string –but updates must be performed with `setq`. E.g., `(defvar my-x 14 "my cool thing")`.

  `(setq x y)` $\approx$ `(set (quote x) y)`

  Variables are assigned with `set`, which takes a quoted identifier, so that it's not evaluated, and a value to associate to that variable. "set quoted", `setq`, avoids the hassle of quoting the name. More generally, `(set sym v)` assigns the *value of* `sym` to have the value `v`.
- Local Scope: `(let ((name0 val0) ... (namek valk)) bodyBlock)`.
  - `let*` permits later bindings to refer to earlier ones.
  - The simpler `let` indicates to the reader that there are no dependencies between the variables.
  - let $\approx$ parallel; let* $\approx$ sequential.
  - Local functions declared with `flet` and `flet*`; e.g., `(flet ((go (x) (+ 2 x))) (go 3))`.
- Any sequence of symbols is a valid identifier, including `x, x-y/z, --<==>--` and even ∀∃. Elisp names are case sensitive.
- Elisp is dynamically scoped: The caller's stack is accessible by default!

```
(defun woah ()
  "If any caller has a local 'work', they're in for a nasty bug
   from me! Moreover, they better have 'a' defined in scope!"
  (setq work (* a 111))) ;; Benefit: Variable-based scoped configuration.

(defun add-one (x)
  "Just adding one to input, innocently calling library method 'woah'."
  (let ((work (+ 1 x)) (a 6))
    (woah) ;; May change 'work' or access 'a'!
    work
  )
)

;; (add-one 2) ⇒ 666
```

Lists and List-Like Structures

- Produce a syntactic, un-evaluated list, we use the single quote: `'(1 2 3)`.
- Construction: `(cons 'x0 '(x1 ... xk))` $\to$ `(x0 x1 ... xk)`.
- Head, or *contents of the address part of the register*: `(car '(x0 x1 ... xk))` $\to$ `x0`.
- Tail, or *contents of the decrement part of the register*: `(cdr '(x0 x1 ... xk))` $\to$ `(x1 ... xk)`.

E.g., `(cons 1 (cons "a" (cons 'nice nil)))` $\approx$ `(list 1 "a" 'nice)` $\approx$ `'(1 "a" nice)`.

Since variables refer to literals and functions have lambdas as literals, we can produce forms that take functions as arguments. E.g., the standard `mapcar` may be construed:

```
(defun my-mapcar (f xs)
  (if (null xs) xs
   (cons (funcall f (car xs)) (my-mapcar f (cdr xs)))))

(my-mapcar (lambda (x) (* 2 x)) '(0 1 2 3 4 5)) ;; ⇒ (0 2 4 6 8 10)
(my-mapcar 'upcase '("a" "b" "cat")) ;; ⇒ ("A" "B" "CAT")
```

Pairs: `(x . y)` $\approx$ `(cons x y)`.
An association list, or alist, is a list formed of such pairs. They're useful for any changeable collection of key-value pairs. The `assoc` function takes a key and an alist and returns the first pair having that key. In the end, alists are just lists.

(Rose) Trees in lisp are easily formed as lists of lists where each inner list is of length 2: The first symbol is the parent node and the second is the list of children.

Lists are formed by chains of cons cells, so getting and setting are very slow; likewise for alists. If performance is desired, one uses arrays and hash tables, respectively, instead. In particular, the performance of arrays and hash tables always requires a constant amount of time whereas the performance of lists and alists grows in proportion with their lengths. However, the size of an array is fixed —it cannot change and thus grow— and hash tables have a lookup cost as well as issues with "hash collisions". Their use is worth it for large amounts of data, otherwise lists are the way to go.

An array is created like a list but using [only square brackets] with getter `(aref arr index)`.
A hash table is created with `(make-hash-table)` with getter `(gethash key table)`.

What if you look up a key and get `nil`, is there no value for that key or is the value `nil`? `gethash` takes a final, optional, argument which is the value to return when the key is not found; it is `nil` by default.

Generic Setters

Since everything is a list in lisp, if `G` is a way to get a value from variable `x`, then `(setf G e)` updates `x` so that the location `G` now refers to element `e`. Hence, once you have a getter `G` you freely obtain a setter `(setf G ···)`.

```
;; Element update
(setq x '(0 1 2 3))    ;; x ⇒ '(0 1 2 3)
(setf (nth 2 x) 'nice) ;; x ⇒ '(0 1 'nice 3)

;; Circular list
(setq y '(a b c))   ;; y ⇒ '(a b c)
(setf (cdddr y) y)  ;; y ⇒ '(a b c a b . #2)
;; "#2" means repeat from index 2.
(nth 99 y) ;; ⇒ a
```

Records

If we want to keep a list of related properties in a list, then we have to remember which position keeps track of which item and may write helper functions to keep track of this. Instead we could use a structure.

```
(defstruct X "Record with fields/slots f_i having defaults d_i"
  (f_0 d_0) ··· (f_k d_k))

;; Automatic constructor is "make-X" with keyword parameters for
;; initialising any subset of the fields!
;; Hence (expt 2 (1+ k)) kinds of possible constructor combinations!
(make-X :f_0 val_0 :f_1 val_1 ··· :f_k val_k) ;; Any, or all, f_i may be omitted

;; Automatic runtime predicate for the new type.
(X-p (make-X)) ;; ⇒ true
(X-p 'nope)    ;; ⇒ nil

;; Field accessors "X-f_i" take an X record and yield its value.

;; Field update: (setf (X-f_i x) val_i)

(defstruct book
  title  (year  0))

(setq ladm (make-book :title "Logical Approach to Discrete Math" :year 1993))
(book-title ladm) ;; ⇒ "Logical Approach to Discrete Math"
(setf (book-title ladm) "LADM")
(book-title ladm) ;; ⇒ "LADM"
```

Advanced OOP constructs can be found within the CLOS, Common Lisp Object System; which is also used as a research tool for studying OOP ideas.

Block of Code

Use the `progn` function to treat multiple expressions as a single expression. E.g.,

```
(progn
  (message "hello")
  (setq x  (if (< 2 3) 'two-less-than-3))
  (sleep-for 0 500)
  (message (format "%s" x))
  (sleep-for 0 500)
  23    ;; Explicit return value
)
```

This' like curly-braces in C or Java. The difference is that the last expression is considered the 'return value' of the block.
Herein, a 'block' is a number of sequential expressions which needn't be wrapped with a `progn` form.

- ◇ Lazy conjunction and disjunction:
  - Perform multiple statements but stop when any of them fails, returns `nil`: `(and s_0 s_1 ... s_k)`.
    - ⋆ Maybe monad!
  - Perform multiple statements until one of them succeeds, returns non-`nil`: `(or s_0 s_1 ... s_k)`.

  We can coerce a statement s_i to returning non-`nil` as so: `(progn s_i t)`. Likewise, coerce failure by `(progn s_i nil)`.
- ◇ Jumps, Control-flow transfer: Perform multiple statements and decide when and where you would like to stop. This' akin to C's `goto`'s; declare a label with `catch` and goto it with `throw`.
  - `(catch 'my-jump bodyBlock)` where the body may contain `(throw 'my-jump returnValue)`;

  the value of the catch/throw is then `returnValue`.
  - Useful for when the `bodyBlock` is, say, a loop. Then we may have multiple `catch`'s with different labels according to the nesting of loops.
    - ⋆ Possibly informatively named throw symbol is `'break`.
  - Using name `'continue` for the throw symbol and having such a catch/throw as *the body of a loop* gives the impression of continue-statements from Java.
  - Using name `'return` for the throw symbol and having such a catch/throw as the body of a function definition gives the impression of, possibly multiple, return-statements from Java –as well as 'early exits'.
  - Simple law: `(catch 'it s_0 s_1 ... s_k (throw 'it r) s_{k+1} ··· s_{k+n})` ≈ `(progn s_0 s_1 ··· s_k r)`.
    - ⋆ Provided the s_i are simple function application forms.
- ◇ `and`, `or` can be thought of as instance of catch/throw, whence they are control flow first and Boolean operations second.

```
(and s_0 ··· s_n e)   ⇒  when all x_i are true, do e
(or  s_0 ··· s_n e)   ⇒  when no x_i is true, do e
```

Conditionals

- ◇ Booleans: `nil`, the empty list `()`, is considered *false*, all else is *true*.
  - Note: `nil` ≈ `()` ≈ `'()` ≈ `'nil`.
  - (Deep structural) equality: `(equal x y)`.
  - Comparisons: As expected; e.g., `(<= x y)` denotes $x \leq y$.
- ◇ `(if condition thenExpr optionalElseBlock)`
  - Note: `(if x y)` ≈ `(if x y nil)`;
    better: `(when c thenBlock)` ≈ `(if c (progn thenBlock))`.
  - Note the else-clause is a 'block': Everything after the then-clause is considered to be part of it.
  - `(if xs ···)` means "if xs is nonempty then ···" is akin to C style idioms on linked lists.

```
(cond
  (test_0
    actionBlock_0)
  (test_1
    actionBlock_1)
  ...
  (t                        ;; optional
    defaultActionBlock))
```

```
;; pattern matching on any type
(defun go (x)
  (pcase x
    ('bob 1972)
    (`(,a ,_ ,c) (+ a c))
    (otherwise "Shucks!")))

(go 'bob)      ;; ⇒ 1972
(go '(1 2 3)) ;; ⇒ 4
(go 'hallo)    ;; "Shucks!"
```

Avoid nested if-then-else clauses by using a `cond` statement –a (lazy) generalisation of switch statements: It sequentially evaluates the expressions `test`$_i$ and performs only the action of the first true test; yielding `nil` when no tests are true. Or use pattern matching; which even allows predicates in the case position —`C-h o` ;-)

Hint: If you write a predicate, think of what else you can return besides `t`; such as a witness to why you're returning truth –all non-nil values denote true after all. E.g., `(member e xs)` returns the sublist of `xs` that begins with `e`.

Loops

Let's sum the first 100 numbers in 3 ways.

```
(let ((n 100) (i 0) (sum 0))
  (while (<= i n)
    (incf sum i) (incf i))
    (message (format "sum: %s" sum)))
```

C	Elisp
`x += y`	`(incf x y)`
`x -= y`	`(decf x y)`

`y` is optional, and is 1 by default.

```
;; Repeat body n times, where i is current iteration.
(let ((result 0) (n 100))
  (dotimes (i (1+ n) result) (incf result i)))

;; A for-each loop: Iterate through the list [0..100].
(let ((result 0) (mylist (number-sequence 0 100)))
  (dolist (e mylist result) (incf result e)))
```

In both loops, `result` is optional and defaults to nil. It is the return value of the loop expression.

Example of Above Constructs

```
(defun my/cool-function (N D)
  "Sum the numbers 0..N that are not divisible by D"
  (catch 'return
    (when (< N 0) (throw 'return 0)) ;; early exit
    (let ((counter 0) (sum 0))
      (catch 'break
    (while 'true
      (catch 'continue
        (incf counter)
        (cond ((equal counter N)       (throw 'break sum   ))
           ((zerop (% counter D))  (throw 'continue nil))
           ('otherwise             (incf sum counter   )) )))))))

(my/cool-function  100 3)  ;; ⇒ 3267
(my/cool-function  100 5)  ;; ⇒ 4000
(my/cool-function -100 7)  ;; ⇒ 0
```

The special `loop` construct provides immensely many options to form nearly any kind of imperative loop. E.g., Python-style 'downfrom' for-loops and Java do-while loops. I personally prefer functional programming, so wont look into this much.

Exception Handling

We can attempt a dangerous clause and catch a possible exceptional case –below we do not do so via `nil`– for which we have an associated handler.

```
(condition-case nil attemptClause (error recoveryBody))

  (ignore-errors attemptBody)
≈ (condition-case nil (progn attemptBody) (error nil))

(ignore-errors (+ 1 "nope")) ;; ⇒ nil
```

Types & Overloading

Since Lisp is dynamically typed, a variable can have any kind of data, possibly different kinds if data at different times in running a program. We can use `type-of` to get the type of a given value; suffixing that with `p` gives the associated predicate;
e.g., `function` ↦ `functionp`.

```
;; Difficult to maintain as more types are added.
(defun sad-add (a b)
  (if (and (numberp a) (numberp b))
      (+ a b)
      (format "%s + %s" a b))
)

(sad-add 2 3)       ;; ⇒ 5
(sad-add 'nice "3") ;; ⇒ "nice + 3"

;; Better: Seperation of concerns.
;;
(cl-defmethod add ((a number) (b number)) (+ a b))      ;; number types
(cl-defmethod add ((a t) (b t)) (format "%s + %s" a b)) ;; catchall types

(add 2 3)       ;; ⇒ 5
(add 'nice "3") ;; ⇒ "nice + 3"
```

While list specific functions like list-length and mapcar may be more efficient than generic functions, which require extra type checking, the generic ones are easier to remember. The following generic functions work on lists, arrays, and strings:

- ◇ `find-if`, gets first value satisfying a predicate.
- ◇ `count`, finds how often an element appears in a sequence
- ◇ `position`, finds the index of a given element.
- ◇ `some`, check if any element satisfies a given predicate
- ◇ `every`, check if every element satisfies the given predicate
- ◇ `reduce`, takes a binary operation and a sequence and mimics a for-loop. Use keyword `:initial-value` to specify the starting value, otherwise use head of sequence.
- ◇ `sum`, add all numbers; crash for strings.
- ◇ `length`, `subseq`, `sort`.

dash is a modern list library for Emacs that uses Haskell-like names for list operations ;-) Likewise, s is a useful Emacs string manipulation library.
In-fact, we can write Emacs extensions using Haskell directly.

Anchors

^	Start of string
\A	Start of string
$	End of string
\Z	End of string
\b	Word boundary
\B	Not word boundary
\<	Start of word
\>	End of word

Character Classes

\c	Control character
\s	White space
\S	Not white space
\d	Digit
\D	Not digit
\w	Word
\W	Not word
\x	Hexadecimal digit
\O	Octal digit

POSIX

[:upper:]	Upper case letters
[:lower:]	Lower case letters
[:alpha:]	All letters
[:alnum:]	Digits and letters
[:digit:]	Digits
[:xdigit:]	Hexadecimal digits
[:punct:]	Punctuation
[:blank:]	Space and tab
[:space:]	Blank characters
[:cntrl:]	Control characters
[:graph:]	Printed characters
[:print:]	Printed characters and spaces
[:word:]	Digits, letters and underscore

Assertions

?=	Lookahead assertion
?!	Negative lookahead
?<=	Lookbehind assertion
?!= or ?<!	Negative lookbehind
?>	Once-only Subexpression
?()	Condition [if then]
?()|	Condition [if then else]
?#	Comment

Available free from AddedBytes.com

Quantifiers

*	0 or more
+	1 or more
?	0 or 1
{3}	Exactly 3
{3,}	3 or more
{3,5}	3, 4 or 5

Quantifier Modifiers

"x" below represents a quantifier

x?	Ungreedy version of "x"

Escape Character

\	Escape Character

Metacharacters (must be escaped)

^	[	.
$	{	*
(	\	+
)	|	?
<	>	

Special Characters

\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\f	Form feed
\xxx	Octal character xxx
\xhh	Hex character hh

Groups and Ranges

.	Any character except new line (\n)
(a|b)	a or b
(...)	Group
(?:...)	Passive Group
[abc]	Range (a or b or c)
[^abc]	Not a or b or c
[a-q]	Letter between a and q
[A-Q]	Upper case letter between A and Q
[0-7]	Digit between 0 and 7
\n	nth group/subpattern

Note: Ranges are inclusive.

Pattern Modifiers

g	Global match
i	Case-insensitive
m	Multiple lines
s	Treat string as single line
x	Allow comments and white space in pattern
e	Evaluate replacement
U	Ungreedy pattern

String Replacement (Backreferences)

$n	nth non-passive group
$2	"xyz" in /^(abc(xyz))$/
$1	"xyz" in /^(?:abc)(xyz)$/
$`	Before matched string
$'	After matched string
$+	Last matched string
$&	Entire matched string

Sample Patterns

Pattern	*Will Match*
([A-Za-z0-9-]+)	Letters, numbers and hyphens
(\d{1,2}\/\d{1,2}\/\d{4})	Date (e.g. 21/3/2006)
([^\s]+(?=\.(jpg|gif|png))\.\2)	jpg, gif or png image
(^[1-9]{1}$\|^[1-4]{1}[0-9]{1}$|^50$)	Any number from 1 to 50 inclusive
(#?([A-Fa-f0-9]){3}(([A-Fa-f0-9]){3})?)	Valid hexadecimal colour code
((?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,15})	String with at least one upper case letter, one lower case letter, and one digit (useful for passwords).
(\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,6})	Email addresses
(\<(/?[^\>]+)\>)	HTML Tags

Note: These patterns are intended for reference purposes and have not been extensively tested. Please use with caution and test thoroughly before use.

Two types of RegEx: **Basic vs Extended Regular Expressions**

In basic regular expressions the meta-characters ?, +, {, |, (, and) lose their special meaning; instead use the backslashed versions \?, \+, \{, \|, \(, and \). Use grep -E flag to use Extended Regular Expressions (see notes as well!)

Remember, need to specify: !#/usr/bin/python at the top!

Document Outline

<!DOCTYPE>	Version of (X)HTML
<html>	HTML document
<head>	Page information
<body>	Page contents

Comments

<!-- Comment Text -->

Page Information

<base />	Base URL
<meta />	Meta data
<title>	Title
<link />	Relevant resource
<style>	Style resource
<script>	Script resource

Document Structure

<h[1-6]>	Heading
	Page section
	Inline section
<p>	Paragraph
 	Line break
<hr />	Horizontal rule

Links

<a href="">	Page link
<a href="mailto:">	Email link
<a name="name">	Anchor
<a href="#name">	Link to anchor

Text Markup

	Strong emphasis
	Emphasis
<blockquote>	Long quotation
<q>	Short quotation
<abbr>	Abbreviation
<acronym>	Acronym
<address>	Address
<pre>	Pre-formatted text
<dfn>	Definition
<code>	Code
<cite>	Citation
<del>	Deleted text
<ins>	Inserted text
<sub>	Subscript
<sup>	Superscript
<bdo>	Text direction

Lists

	Ordered list
	Unordered list
	List item
<dl>	Definition list
<dt>	Definition term
<dd>	Term description

Forms

<form>	Form
<fieldset>	Collection of fields
<legend>	Form legend
<label>	Input label
<input />	Form input
<select>	Drop-down box
<optgroup>	Group of options
<option>	Drop-down options
<textarea>	Large text input
<button>	Button

Tables

<table>	Table
<caption>	Caption
<thead>	Table header
<tbody>	Table body
<tfoot>	Table footer
<colgroup>	Column group
<col />	Column
<tr>	Table row
<th>	Header cell
<td>	Table cell

Images and Image Maps

<img />	Image
<map>	Image Map
<area />	Area of Image Map

Common Character Entities

"	"	Quotation mark
&	&	Ampersand
<	<	Less than
>	>	Greater than
@	@	"At" symbol
€	€	Euro
•	•	Small bullet
™	™	Trademark
£	£	Pound
		Non-breaking space
©	©	Copyright symbol

Objects

<object>	Object
<param />	Parameter

Empty Elements

<area />	<img />
<base />	<input />
 	<link />
<col />	<meta />
<hr />	<param />

Core Attributes

class	style
id	title

Note: Core Attributes may not be used in base, head, html, meta, param, script, style or title elements.

Language Attributes

dir	lang

Note: Language Attributes may not be used in base, br, frame, frameset, hr, iframe, param or script elements.

Keyboard Attributes

accesskey	tabindex

Window Events

onLoad	onUnload

Form Events

onBlur	onReset
onChange	onSelect
onFocus	onSubmit

Keyboard Events

onKeydown	onKeyup
onKeypress	

Mouse Events

onClick	onMouseout
onDblclick	onMouseover
onMousedown	onMouseup
onMousemove	

JS CheatSheet

Hide comments

If - Else

```
if ((age >= 14) && (age < 19)) {        // logical condi
    status = "Eligible.";               // executed if c
} else {                                // else block is
    status = "Not eligible.";           // executed if c
}
```

Switch Statement

```
switch (new Date().getDay()) {      // input is current
    case 6:                         // if (day == 6)
        text = "Saturday";
        break;
    case 0:                         // if (day == 0)
        text = "Sunday";
        break;
    default:                        // else...
        text = "Whatever";
}
```

Variables

```
var a;                          // variable
var b = "init";                 // string
var c = "Hi" + " " + "Joe";     // = "Hi Joe"
var d = 1 + 2 + "3";            // = "33"
var e = [2,3,5,8];              // array
var f = false;                  // boolean
var g = /()/;                   // RegEx
var h = function(){};           // function object
const PI = 3.14;                // constant
var a = 1, b = 2, c = a + b;    // one line
let z = 'zzz';                  // block scope local var
```

Basics

On page script

```
<script type="text/javascript">  ...
</script>
```

Include external JS file

```
<script src="filename.js"></script>
```

Functions

```
function addNumbers(a, b) {
    return a + b; ;
}
x = addNumbers(1, 2);
```

Edit DOM element

```
document.getElementById("elementID").innerHTML = "Hello
```

Output

```
console.log(a);             // write to the browser cons
document.write(a);          // write to the HTML
alert(a);                   // output in an alert box
confirm("Really?");         // yes/no dialog, returns tr
prompt("Your age?","0");    // input dialog. Second argu
```

Comments

```
/* Multi line
   comment */
// One line
```

Data Types

```
var age = 18;                           // number
var name = "Jane";                      // string
var name = {first:"Jane", last:"Doe"};  // object
```

Loops

For Loop

```
for (var i = 0; i < 10; i++) {
    document.write(i + ": " + i*3 + "<br />");
}
var sum = 0;
for (var i = 0; i < a.length; i++) {
    sum + = a[i];
}               // parsing an array
```

While Loop

```
var i = 1;                      // initialize
while (i < 100) {               // enters the cycle if s
    i *= 2;                     // increment to avoid in
    document.write(i + ", ");   // output
}
```

Do While Loop

```
var i = 1;                      // initialize
do {                            // enters cycle at least
    i *= 2;                     // increment to avoid in
    document.write(i + ", ");   // output
} while (i < 100)               // repeats cycle if stat
```

Break

```
for (var i = 0; i < 10; i++) {
    if (i == 5) { break; }          // stops and exits t
    document.write(i + ", ");       // last output numbe
}
```

Continue

```
for (var i = 0; i < 10; i++) {
    if (i == 5) { continue; }       // skips the rest of
    document.write(i + ", ");       // skips 5
}
```