

An Introduction to the Z Shell

version 5.9.999.3

**Paul Falstad
Bas de Bakker**

August 02, 2026

Contents

An Introduction to the Z Shell	1
Introduction	1
Filename Generation	1
Startup Files	5
Shell Functions	5
Directories	8
Directory Stacks	10
Command/Process Substitution	12
Redirection	15
Aliasing	16
History	19
Command Line Editing	21
Completion	24
Extended completion	29
Bindings	31
Parameter Substitution	32
Shell Parameters	36
Prompting	39
Login/logout watching	40
Options	42
Closing Comments	45
List of Examples	45
Index	51

An Introduction to the Z Shell

Paul Falstad
pf@software.com
Bas de Bakker
bas@phys.uva.nl

Introduction

zsh is a shell designed for interactive use, although it is also a powerful scripting language. Many of the useful features of bash, ksh, and tcsh were incorporated into zsh; many original features were added. This document details some of the unique features of zsh. It assumes basic knowledge of the standard UNIX shells; the intent is to show a reader already familiar with one of the other major shells what makes zsh more useful or more powerful. This document is not at all comprehensive; read the manual entry for a description of the shell that is complete and concise, although somewhat overwhelming and devoid of examples.

The text will frequently mention options that you can set to change the behaviour of zsh. You can set these options with the command

```
% setopt optionname
```

Set option with setopt

and unset them again with

```
% unsetopt optionname
```

Unset option with unsetopt

Case is ignored in option names, as are embedded underscores.

Filename Generation

Otherwise known as *globbing*, filename generation is quite extensive in zsh. Of course, it has all the basics:

```
% ls
Makefile  file.pro  foo.o     main.o    q.c       run234    stuff
bar.o     foo       link      morestuff run123     run240    sub
file.h    foo.c     main.h    pipe      run2       run303
% ls *.c
foo.c q.c
% ls *.[co]
bar.o  foo.c  foo.o  main.o  q.c
% ls foo.?
foo.c  foo.o
% ls *.[^c]
bar.o  file.h  foo.o  main.h  main.o
% ls *.[^oh]
foo.c  q.c
```

Basic globs with ls

Also, if the EXTENDED_GLOB option is set, some new features are activated. For example, the ^ character negates the pattern following it:

```
% setopt extendedglob
% ls -d ^*.c
Makefile  file.pro  link      morestuff  run2      run303
bar.o     foo      main.h    pipe      run234    stuff
```

```

file.h      foo.o      main.o      run123      run240      sub
% ls -d ^*.*
Makefile    link        pipe        run2        run240      stuff
foo         morestuff  run123      run234      run303      sub
% ls -d ^Makefile
bar.o       foo         link        morestuff   run123      run240      sub
file.h      foo.c       main.h      pipe        run2        run303
file.pro    foo.o       main.o      q.c         run234      stuff
% ls -d *.^c
.rhosts     bar.o       file.h      file.pro    foo.o       main.h      main.o

```

EXTENDED_GLOB negation

An expression of the form <x-y> matches a range of integers:

```

% ls run<200-300>
run234 run240
% ls run<300-400>
run303
% ls run<-200>
run123 run2
% ls run<300->
run303
% ls run<->
run123 run2 run234 run240 run303

```

Integer range globs

The NUMERIC_GLOB_SORT option will sort files with numbers according to the number. This will not work with ls as it resorts its arguments:

```

% setopt numericglobsort
% echo run<->
run2 run123 run234 run240 run303

```

NUMERIC_GLOB_SORT with echo

Grouping is possible:

```

% ls (foo|bar).*
bar.o foo.c foo.o
% ls *(c|o|pro)
bar.o file.pro foo.c foo.o main.o q.c

```

Grouped alternation globs

Also, the string **/ forces a recursive search of subdirectories:

```

% ls -R
Makefile file.pro foo.o main.o q.c run234 stuff
bar.o foo link morestuff run123 run240 sub
file.h foo.c main.h pipe run2 run303

morestuff:

stuff:
file xxx yyy

stuff/xxx:
foobar

```

```
stuff/yyy:
frobar
% ls **/*bar
stuff/xxx/foobar  stuff/yyy/frobar
% ls **/f*
file.h            foo            foo.o            stuff/xxx/foobar
file.pro          foo.c            stuff/file        stuff/yyy/frobar
% ls *bar*
bar.o
% ls **/*bar*
bar.o            stuff/xxx/foobar  stuff/yyy/frobar
% ls stuff/**/*bar*
stuff/xxx/foobar  stuff/yyy/frobar
```

Recursive directory globs

It is possible to exclude certain files from the patterns using the `~` character. A pattern of the form `*.c~bar.c` lists all files matching `*.c`, except for the file `bar.c`.

```
% ls *.c
foo.c    foob.c    bar.c
% ls *.c~bar.c
foo.c    foob.c
% ls *.c~f*
bar.c
```

Glob exclusion with tilde

One can add a number of *qualifiers* to the end of any of these patterns, to restrict matches to certain file types. A qualified pattern is of the form

```
pattern(...)
```

Qualifier pattern form

with single-character qualifiers inside the parentheses.

```
% alias l='ls -dF'
% l *
Makefile    foo*        main.h      q.c         run240
bar.o       foo.c       main.o      run123      run303
file.h      foo.o       morestuff/  run2        stuff/
file.pro    link@      pipe       run234      sub
% l *(/)
morestuff/  stuff/
% l *(@)
link@
% l *(*)
foo*        link@      morestuff/  stuff/
% l *(x)
foo*        link@      morestuff/  stuff/
% l *(X)
foo*        link@      morestuff/  stuff/
% l *(R)
bar.o       foo*       link@       morestuff/  run123      run240
file.h      foo.c      main.h      pipe        run2        run303
file.pro    foo.o      main.o      q.c         run234      stuff/
```

File type qualifiers

Note that `*(x)` and `*(*)` both match executables. `*(X)` matches files executable by others, as opposed to `*(x)`, which matches files executable by the owner. `*(R)` and `*(r)` match readable files; `*(W)` and `*(w)`, which checks for writable files. `*(W)` is especially important, since it checks for world-writable files:

```
% l *(w)
bar.o      foo*      link@      morestuff/  run123      run240
file.h     foo.c      main.h     pipe        run2        run303
file.pro   foo.o      main.o     q.c         run234      stuff/
% l *(W)
link@      run240
% l -l link run240
lrwxrwxrwx 1 pfalstad      10 May 23 18:12 link -> /usr/bin/
-rw-rw-rw- 1 pfalstad      0 May 23 18:12 run240
```

World-writable qualifiers

If you want to have all the files of a certain type as well as all symbolic links pointing to files of that type, prefix the qualifier with a `-`:

```
% l *(-/)
link@      morestuff/  stuff/
```

Dash prefix follows symlinks

You can filter out the symbolic links with the `^` character:

```
% l *(W^@)
run240
% l *(x)
foo*      link@      morestuff/  stuff/
% l *(x^@/)
foo*
```

Filter out symlinks

To find all plain files, you can use `..`:

```
% l *(.)
Makefile  file.h    foo*      foo.o      main.o     run123     run234     run303
bar.o     file.pro  foo.c      main.h     q.c        run2       run240     sub
% l *(^.)
link@      morestuff/  pipe      stuff/
% l s*(.)
stuff/     sub
% l *(p)
pipe
% l -l *(p)
prw-r--r-- 1 pfalstad      0 May 23 18:12 pipe
```

Plain file and pipe quals

`*(U)` matches all files owned by you. To search for all files not owned by you, use `*(^U)`:

```
% l -l *(^U)
-rw----- 1 subbarao      29 May 23 18:13 sub
```

Ownership qualifiers

This searches for setuid files:

```
% l -l *(s)
-rwsr-xr-x 1 pfalstad      16 May 23 18:12 foo*
```

Startup Files

Setuid qualifier

This checks for a certain user's files:

```
% 1 -1 *(u[subbarao])  
-rw----- 1 subbarao          29 May 23 18:13 sub
```

User qualifier match

Startup Files

There are five startup files that zsh will read commands from:

```
$ZDOTDIR/.zshenv  
$ZDOTDIR/.zprofile  
$ZDOTDIR/.zshrc  
$ZDOTDIR/.zlogin  
$ZDOTDIR/.zlogout
```

Five startup files

If ZDOTDIR is not set, then the value of HOME is used; this is the usual case.

.zshenv is sourced on all invocations of the shell, unless the -f option is set. It should contain commands to set the command search path, plus other important environment variables. .zshenv should not contain commands that produce output or assume the shell is attached to a tty.

.zshrc is sourced in interactive shells. It should contain commands to set up aliases, functions, options, key bindings, etc.

.zlogin is sourced in login shells. It should contain commands that should be executed only in login shells. .zlogout is sourced when login shells exit. .zprofile is similar to .zlogin, except that it is sourced before .zshrc. .zprofile is meant as an alternative to .zlogin for ksh fans; the two are not intended to be used together, although this could certainly be done if desired. .zlogin is not the place for alias definitions, options, environment variable settings, etc.; as a general rule, it should not change the shell environment at all. Rather, it should be used to set the terminal type and run a series of external commands (fortune, msgs, etc).

Shell Functions

zsh also allows you to create your own commands by defining shell functions. For example:

```
% yp () {  
>     ypmatch $1 passwd.byname  
> }  
% yp pfalstad  
pfalstad*:3564:35:Paul John Falstad:/u/pfalstad:/usr/princeton/bin/zsh
```

Define a simple function

This function looks up a user in the NIS password map. The \$1 expands to the first argument to yp. The function could have been equivalently defined in one of the following ways:

```
% function yp {  
>     ypmatch $1 passwd.byname  
> }  
% function yp () {  
>     ypmatch $1 passwd.byname  
> }  
% function yp () ypmatch $1 passwd.byname
```

function keyword forms

Note that aliases are expanded when the function definition is parsed, not when the function is executed. For example:

```
% alias ypmatch=echo
% yp pfalstad
pfalstad:~:3564:35:Paul John Falstad:/u/pfalstad:/usr/princeton/bin/zsh
```

Alias after function parse

Since the alias was defined after the function was parsed, it has no effect on the function's execution. However, if we define the function again with the alias in place:

```
% function yp () { ypmatch $1 passwd.byname }
% yp pfalstad
pfalstad passwd.byname
```

Redefine with alias active

it is parsed with the new alias definition in place. Therefore, in general you must define aliases before functions.

We can make the function take multiple arguments:

```
% unalias ypmatch
% yp () {
>     for i
>     do ypmatch $i passwd.byname
>     done
> }
% yp pfalstad subbarao sukthnkr
pfalstad:~:3564:35:Paul John Falstad:/u/pfalstad:/usr/princeton/bin/zsh
subbarao:~:3338:35:Kartik Subbarao:/u/subbarao:/usr/princeton/bin/zsh
sukthnkr:~:1267:35:Rahul Sukthankar:/u/sukthnkr:/usr/princeton/bin/tcsh
```

Loop over multiple args

The `for i` loops through each of the function's arguments, setting `i` equal to each of them in turn. We can also make the function do something sensible if no arguments are given:

```
% yp () {
>     if (( $# == 0 ))
>     then echo usage: yp name ...; fi
>     for i; do ypmatch $i passwd.byname; done
> }
% yp
usage: yp name ...
% yp pfalstad sukthnkr
pfalstad:~:3564:35:Paul John Falstad:/u/pfalstad:/usr/princeton/bin/zsh
sukthnkr:~:1267:35:Rahul Sukthankar:/u/sukthnkr:/usr/princeton/bin/tcsh
```

Check `$#` for usage

`$#` is the number of arguments supplied to the function. If it is equal to zero, we print a usage message; otherwise, we loop through the arguments, and `ypmatch` all of them.

Here's a function that selects a random line from a file:

```
% randline () {
>     integer z=$(wc -l <$1)
>     sed -n $[RANDOM % z + 1]p $1
> }
% randline /etc/motd
PHOENIX WILL BE DOWN briefly Friday morning, 5/24/91 from 8 AM to
```

```
% randline /etc/motd
SunOS Release 4.1.1 (PHOENIX) #19: Tue May 14 19:03:15 EDT 1991
% randline /etc/motd
| Please use the "msgs" command to read announcements. Refer to the |
% echo $z

%
```

Random line with RANDOM

randline has a local variable, z, that holds the number of lines in the file. `$(RANDOM % z + 1)` expands to a random number between 1 and z. An expression of the form `$(...)` expands to the value of the arithmetic expression within the brackets, and the RANDOM variable returns a random number each time it is referenced. % is the modulus operator, as in C. Therefore, `sed -n $(RANDOM%z+1)p` picks a random line from its input, from 1 to z.

Function definitions can be viewed with the functions builtin:

```
% functions randline
randline () {
    integer z=$(wc -l <$1)
    sed -n $(RANDOM % z + 1)p $1
}
% functions
yp () {
    if let $# == 0
    then
        echo usage: yp name ...
    fi
    for i
    do
        ypmatch $i passwd.byname
    done
}
randline () {
    integer z=$(wc -l <$1)
    sed -n $(RANDOM % z + 1)p $1
}
}
```

View with functions builtin

Here's another one:

```
% cx () { chmod +x $* }
% ls -l foo bar
-rw-r--r-- 1 pfalstad      29 May 24 04:38 bar
-rw-r--r-- 1 pfalstad      29 May 24 04:38 foo
% cx foo bar
% ls -l foo bar
-rwxr-xr-x 1 pfalstad      29 May 24 04:38 bar
-rwxr-xr-x 1 pfalstad      29 May 24 04:38 foo
```

chmod +x as function

Note that this could also have been implemented as an alias:

```
% chmod 644 foo bar
% alias cx='chmod +x'
% cx foo bar
% ls -l foo bar
-rwxr-xr-x  1 pfalstad      29 May 24 04:38 bar
-rwxr-xr-x  1 pfalstad      29 May 24 04:38 foo
```

Same trick as an alias

Instead of defining a lot of functions in your `.zshrc`, all of which you may not use, it is often better to use the `autoload` builtin. The idea is, you create a directory where function definitions are stored, declare the names in your `.zshrc`, and tell the shell where to look for them. Whenever you reference a function, the shell will automatically load it into memory.

```
% mkdir /tmp/funs
% cat >/tmp/funs/yp
ypmatch $1 passwd.byname
^D
% cat >/tmp/funs/cx
chmod +x $*
^D
% FPATH=/tmp/funs
% autoload cx yp
% functions cx yp
undefined cx ()
undefined yp ()
% chmod 755 /tmp/funs/{cx,yp}
% yp egsirer
egsirer:~:3214:35:Emin Gun Sirer:/u/egsirer:/bin/sh
% functions yp
yp () {
    ypmatch $1 passwd.byname
}
```

autoload with FPATH

This idea has other benefits. By adding a `#!` header to the files, you can make them double as shell scripts. (Although it is faster to use them as functions, since a separate process is not created.)

```
% ed /tmp/funs/yp
25
i
#! /usr/local/bin/zsh
.
w
42
q
% </tmp/funs/yp
#! /usr/local/bin/zsh
ypmatch $1 passwd.byname
% /tmp/funs/yp sukthnkr
sukthnkr:~:1267:35:Rahul Sukthankar:/u/sukthnkr:/usr/princeton/bin/tcsh
```

Function as shell script

Now other people, who may not use `zsh`, or who don't want to copy all of your `.zshrc`, may use these functions as shell scripts.

Directories

One nice feature of `zsh` is the way it prints directories. For example, if we set the prompt like this:

```
phoenix% PROMPT='%~> '
~> cd src
~/src>
```

Tilde in the prompt

the shell will print the current directory in the prompt, using the ~ character. However, zsh is smarter than most other shells in this respect:

```
~/src> cd ~subbarao
~subbarao> cd ~maruchck
~maruchck> cd lib
~maruchck/lib> cd fun
~maruchck/lib/fun> foo=/usr/princeton/common/src
~maruchck/lib/fun> cd ~foo
~foo> cd ..
/usr/princeton/common> cd src
~foo> cd news/nntp
~foo/news/nntp> cd inews
~foo/news/nntp/inews>
```

Named directory shortcuts

Note that zsh prints *other* users' directories in the form ~user. Also note that you can set a parameter and use it as a directory name; zsh will act as if foo is a user with the login directory /usr/princeton/common/src. This is convenient, especially if you're sick of seeing prompts like this:

```
phoenix:/usr/princeton/common/src/X.V11R4/contrib/clients/xv/docs>
```

Long path in prompt

If you get stuck in this position, you can give the current directory a short name, like this:

```
/usr/princeton/common/src/news/nntp/inews> inews=$PWD
/usr/princeton/common/src/news/nntp/inews> echo ~inews
/usr/princeton/common/src/news/nntp/inews
~inews>
```

Shorten path via parameter

When you reference a directory in the form ~inews, the shell assumes that you want the directory displayed in this form; thus simply typing echo ~inews or cd ~inews causes the prompt to be shortened. You can define a shell function for this purpose:

```
~inews> namedir () { $1=$PWD ; : ~$1 }
~inews> cd /usr/princeton/bin
/usr/princeton/bin> namedir pbin
~pbin> cd /var/spool/mail
/var/spool/mail> namedir spool
~spool> cd .msgs
~spool/.msgs>
```

namedir helper function

You may want to add this one-line function to your .zshrc. zsh can also put the current directory in your title bar, if you are using a windowing system. One way to do this is with the chpwd function, which is automatically executed by the shell whenever you change directory. If you are using xterm, this will work:

```
chpwd () { print -Pn '^[]2;%~^G' }
```

chpwd for xterm title

Directory Stacks

The `-P` option tells `print` to treat its arguments like a prompt string; otherwise the `%~` would not be expanded. The `-n` option suppresses the terminating newline, as with `echo`.

If you are using an IRIS `wsh`, do this:

```
chpwd () { print -Pn '\2201.y%~\234' }
```

`chpwd` for IRIS `wsh`

The `print -D` command has other uses. For example, to print the current directory to standard output in short form, you can do this:

```
% print -D $PWD
~subbarao/src
```

`print -D` shortens `PWD`

and to print each component of the path in short form:

```
% print -D $path
/bin /usr/bin ~locbin ~locbin/X11 ~/bin
```

`print -D` shortens `path`

Directory Stacks

If you use `csch`, you may know about directory stacks. The `pushd` command puts the current directory on the stack, and changes to a new directory; the `popd` command pops a directory off the stack and changes to it.

```
phoenix% cd
phoenix% PROMPT='Z %~> '
Z ~> pushd /tmp
/tmp ~
Z /tmp> pushd /usr/etc
/usr/etc /tmp ~
Z /usr/etc> pushd /usr/bin
/usr/bin /usr/etc /tmp ~
Z /usr/bin> popd
/usr/etc /tmp ~
Z /usr/etc> popd
/tmp ~
Z /tmp> pushd /etc
/etc /tmp ~
Z /etc> popd
/tmp ~
```

`pushd` and the directory stack

`zsh`'s directory stack commands work similarly. One difference is the way `pushd` is handled if no arguments are given. As in `csch`, this exchanges the top two elements of the directory stack:

```
Z /tmp> dirs
/tmp ~
Z /tmp> pushd
~ /tmp
```

`pushd` swaps top two

unless the stack only has one entry:

```
Z ~> popd
/tmp
```

```
Z /tmp> dirs
/tmp
Z /tmp> pushd
~ /tmp
Z ~>
```

pushd with one entry

or unless the PUSHD_TO_HOME option is set:

```
Z ~> setopt pushdtohome
Z ~> pushd
~ ~ /tmp
```

PUSHD_TO_HOME option

As an alternative to using directory stacks in this manner, we can get something like a *directory history* by setting a few more options and parameters:

```
~> DIRSTACKSIZE=8
~> setopt autopushd pushdminus pushdsilent pushdtohome
~> alias dh='dirs -v'
~> cd /tmp
/tmp> cd /usr
/usr> cd bin
/usr/bin> cd ../pub
/usr/pub> dh
0      /usr/pub
1      /usr/bin
2      /usr
3      /tmp
4      ~
/usr/pub> cd -3
/tmp> dh
0      /tmp
1      /usr/pub
2      /usr/bin
3      /usr
4      ~
/tmp> ls ~2/df
/usr/bin/df
/tmp> cd -4
~>
```

Directory history with AUTOPUSHD

Note that ~2 expanded to the second directory in the history list, and that cd -3 recalled the third directory in the list.

You may be wondering what all those options do. AUTO_PUSHD made cd act like pushd. (alias cd=pushd is not sufficient, for various reasons.) PUSHD_MINUS swapped the meaning of cd +1 and cd -1; we want them to mean the opposite of what they mean in csh, because it makes more sense in this scheme, and it's easier to type:

```
~> dh
0      ~
1      /tmp
2      /usr/pub
3      /usr/bin
4      /usr
~> unsetopt pushdminus
~> cd +1
/tmp> dh
```

```
0      /tmp
1      ~
2      /usr/pub
3      /usr/bin
4      /usr
/tmp> cd +2
/usr/pub>
```

PUSHD_MINUS with cd +n

PUSHD_SILENT keeps the shell from printing the directory stack each time we do a cd, and PUSHD_TO_HOME we mentioned earlier:

```
/usr/pub> unsetopt pushdsilent
/usr/pub> cd /etc
/etc /usr/pub /tmp ~ /usr/bin /usr
/etc> cd
~ /etc /usr/pub /tmp ~ /usr/bin /usr
~> unsetopt pushdtohome
~> cd
/etc ~ /usr/pub /tmp ~ /usr/bin /usr
/etc>
```

PUSHD_SILENT hides stack

DIRSTACKSIZE keeps the directory stack from getting too large, much like HISTSIZE:

```
/etc> setopt pushdsilent
/etc> cd /
/> cd /
/> cd /
/> cd /
/> cd /
/> cd /
/> cd /
/> cd /
/> cd /
/> dh
0      /
1      /
2      /
3      /
4      /
5      /
6      /
7      /
```

DIRSTACKSIZE limits stack

Command/Process Substitution

Command substitution in zsh can take two forms. In the traditional form, a command enclosed in backquotes (`...`) is replaced on the command line with its output. This is the form used by the older shells. Newer shells (like zsh) also provide another form, \$(...). This form is much easier to nest.

```
% ls -l `echo /vmunix`
-rwxr-xr-x  1 root      1209702 May 14 19:04 /vmunix
% ls -l $(echo /vmunix)
-rwxr-xr-x  1 root      1209702 May 14 19:04 /vmunix
% who | grep mad
subbarao ttyt7    May 23 15:02    (mad55sx15.Prince)
```

```
pfalstad ttyu1    May 23 16:25    (mad55sx14.Prince)
subbarao ttyu6    May 23 15:04    (mad55sx15.Prince)
pfalstad ttyv3    May 23 16:25    (mad55sx14.Prince)
% who | grep mad | awk '{print $2}'
ttyt7
ttyu1
ttyu6
ttyv3
% cd /dev; ls -l $(who |
> grep $(echo mad) |
> awk '{ print $2 }')
crwx-w----  1 subbarao  20,  71 May 23 18:35 ttyt7
crw--w----  1 pfalstad  20,  81 May 23 18:42 ttyu1
crwx-w----  1 subbarao  20,  86 May 23 18:38 ttyu6
crw--w----  1 pfalstad  20,  99 May 23 18:41 ttyv3
```

Backtick and \$(...) forms

Many common uses of command substitution, however, are superseded by other mechanisms of zsh:

```
% ls -l `tty`
crw-rw-rw-  1 root      20,  28 May 23 18:35 /dev/ttyqc
% ls -l $TTY
crw-rw-rw-  1 root      20,  28 May 23 18:35 /dev/ttyqc
% ls -l `which rn`
-rwxr-xr-x  1 root      172032 Mar  6 18:40 /usr/princeton/bin/rn
% ls -l =rn
-rwxr-xr-x  1 root      172032 Mar  6 18:40 /usr/princeton/bin/rn
```

TTY and =cmd shortcuts

A command name with a = prepended is replaced with its full pathname. This can be very convenient. If it's not convenient for you, you can turn it off:

```
% ls
=foo    =bar
% ls =foo =bar
zsh: foo not found
% setopt noequals
% ls =foo =bar
=foo    =bar
```

Setting NO_EQUALS

Another nice feature is process substitution:

```
% who | fgrep -f =(print -l root lemke shgchan subbarao)
root      console May 19 10:41
lemke     ttyq0    May 22 10:05    (narnia:0.0)
lemke     ttyr7    May 22 10:05    (narnia:0.0)
lemke     ttyrd    May 22 10:05    (narnia:0.0)
shgchan   ttys1    May 23 16:52    (gaudi.Princeton.)
subbarao  ttyt7    May 23 15:02    (mad55sx15.Prince)
subbarao  ttyu6    May 23 15:04    (mad55sx15.Prince)
shgchan   ttyvb    May 23 16:51    (gaudi.Princeton.)
```

Process sub with =(...)

A command of the form =(...) is replaced with the name of a *file* containing its output. (A command substitution, on the other hand, is replaced with the output itself.) `print -l` is like `echo`, excepts that it prints its arguments one per line, the way `fgrep` expects them:

Command/Process Substitution

```
% print -l foo bar
foo
bar
```

print -l one per line

We could also have written:

```
% who | fgrep -f =(echo 'root
> lemke
> shgchan
> subbarao')
```

=(...) with multiline echo

Using process substitution, you can edit the output of a command:

```
% ed =(who | fgrep -f ~/.friends)
355
g/lemke/d
w /tmp/filbar
226
q
% cat /tmp/filbar
root      console May 19 10:41
shgchan   ttys1    May 23 16:52   (gaudi.Princeton.)
subbarao  ttyt7    May 23 15:02   (mad55sx15.Prince)
subbarao  ttyu6    May 23 15:04   (mad55sx15.Prince)
shgchan   ttyvb    May 23 16:51   (gaudi.Princeton.)
```

Edit process sub output

or easily read archived mail:

```
% mail -f =(zcat ~/mail/oldzshmail.Z)
"/tmp/zsha06024": 84 messages, 0 new, 43 unread
> 1  U  TO: pfalstad, zsh (10)
    2  U  nytim!tim@uunet.uu.net, Re: Zsh on Sparc1 /SunOS 4.0.3
    3  U  JAM%TPN@utrcgw.utc.com, zsh fix (15)
    4  U  djm@eng.umd.edu, way to find out if running zsh? (25)
    5  U  djm@eng.umd.edu, Re: way to find out if running zsh? (17)
    6  r  djm@eng.umd.edu, Meta . (18)
    7  U  jack@cs.glasgow.ac.uk, Re: problem building zsh (147)
    8  U  nytim!tim@uunet.uu.net, Re: Zsh on Sparc1 /SunOS 4.0.3
    9  U  urisa!jmd, Another fix... (61)
   10  U  pplacewa@bbn.com, Re: v18i084: Zsh 2.00 - A small complaint (36)
   11  U  lubkin@cs.rochester.edu, POSIX job control (34)
   12  U  yale!bronson!tan@uunet.UU.NET
   13  U  brett@rpi.edu, zsh (36)
   14  S  subbarao, zsh sucks!!!! (286)
   15  U  snibru!d241s008!d241s013!ala@relay.EU.net, zsh (165)
   16  U  nytim!tim@uunet.UU.NET, Re: Zsh on Sparc1 /SunOS 4.0.3
   17  U  subbarao, zsh is a junk shell (43)
   18  U  amaranth@vela.acs.oakland.edu, zsh (33)
43u/84 1: x
% ls -l /tmp/zsha06024
/tmp/zsha06024 not found
```

Read compressed mail folder

Note that the shell creates a temporary file, and deletes it when the command is finished.

```
% diff =(ls) =(ls -F)
3c3
< fortune
---
> fortune*
10c10
< strfile
---
> strfile*
```

diff two process outputs

If you read zsh's man page, you may notice that `<(…)` is another form of process substitution which is similar to `=(…)`. There is an important difference between the two. In the `<(…)` case, the shell creates a named pipe (FIFO) instead of a file. This is better, since it does not fill up the file system; but it does not work in all cases. In fact, if we had replaced `=(…)` with `<(…)` in the examples above, all of them would have stopped working except for `fgrep -f <(…)`. You can not edit a pipe, or open it as a mail folder; `fgrep`, however, has no problem with reading a list of words from a pipe. You may wonder why `diff <(foo) bar` doesn't work, since `foo | diff - bar` works; this is because `diff` creates a temporary file if it notices that one of its arguments is `-`, and then copies its standard input to the temporary file.

`>(…)` is just like `<(…)` except that the command between the parentheses will get its input from the named pipe.

```
% dvips -o >(lpr) zsh.dvi
```

Output process sub `>(…)`

Redirection

Apart from all the regular redirections like the Bourne shell has, zsh can do more. You can send the output of a command to more than one file, by specifying more redirections like

```
% echo Hello World >file1 >file2
```

Redirect to multiple files

and the text will end up in both files. Similarly, you can send the output to a file and into a pipe:

```
% make > make.log | grep Error
```

Redirect and pipe together

The same goes for input. You can make the input of a command come from more than one file.

```
% sort <file1 <file2 <file3
```

Input from multiple files

The command will first get the contents of `file1` as its standard input, then those of `file2` and finally the contents of `file3`. This, too, works with pipes.

```
% cut -d: -f1 /etc/passwd | sort <newnames
```

Pipe then more input

The sort will get as its standard input first the output of `cut` and then the contents of `newnames`.

Suppose you would like to watch the standard output of a command on your terminal, but want to pipe the standard error to another command. An easy way to do this in zsh is by redirecting the standard error using `2> >(…)`.

```
% find / -name games 2> >(grep -v 'Permission' > realerrors)
```

Stderr via process sub

The above redirection will actually be implemented with a regular pipe, not a temporary named pipe.

Aliasing

Often-used commands can be abbreviated with an alias:

```
% alias uc=uncompress
% ls
hanoi.Z
% uc hanoi
% ls
hanoi
```

Simple command alias

or commands with certain desired options:

```
% alias fm='finger -m'
% fm root
Login name: root                In real life: Operator
Directory: /                   Shell: /bin/csh
On since May 19 10:41:15 on console 3 days 5 hours Idle Time
No unread mail
No Plan.

% alias lock='lock -p -60000'
% lock
lock: /dev/ttyr4 on phoenix. timeout in 60000 minutes
time now is Fri May 24 04:23:18 EDT 1991
Key:

% alias l='ls -AF'
% l /
.bash_history          kadb*
.bashrc                lib@
.cshrc                 licensed/
.exrc                  lost+found/
.login                 macsyma
...
```

Alias with fixed options

Aliases can also be used to replace old commands:

```
% alias grep=egrep ps=sps make=gmake
% alias whoami='echo root'
% whoami
root
```

Replace commands with aliases

or to define new ones:

```
% cd /
% alias sz='ls -l | sort -n +3 | tail -10'
% sz
drwxr-sr-x  7 bin          3072 May 23 11:59 etc
drwxrwxrwx 26 root        5120 May 24 04:20 tmp
drwxr-xr-x  2 root        8192 Dec 26 19:34 lost+found
drwxr-sr-x  2 bin        14848 May 23 18:48 dev
-r--r--r--  1 root     140520 Dec 26 20:08 boot
```

```
-rwxr-xr-x  1 root      311172 Dec  26 20:08 kadb
-rwxr-xr-x  1 root      1209695 Apr  16 15:33 vmunix.old
-rwxr-xr-x  1 root      1209702 May  14 19:04 vmunix
-rwxr-xr-x  1 root      1209758 May  21 12:23 vmunix.new.kernelmap.old
-rwxr-xr-x  1 root      1711848 Dec  26 20:08 vmunix.org
% cd
% alias rable='ls -AFtrd *(R)' nrable='ls -AFtrd *(^R)'
% rable
README      func/      bin/      pub/      News/      src/
nicedcolors etc/      scr/      tmp/      iris/      zsh*
% nrable
Mailboxes/  mail/      notes
```

New commands via aliases

(The pattern `*(R)` matches all readable files in the current directory, and `*(^R)` matches all unreadable files.)

Most other shells have aliases of this kind (*command* aliases). However, `zsh` also has *global* aliases, which are substituted anywhere on a line. Global aliases can be used to abbreviate frequently-typed usernames, hostnames, etc.

```
% alias -g me=pfalstad gun=egsirer mjm=maruchck
% who | grep me
pfalstad tty0    May 24 03:39    (mickey.Princeton)
pfalstad tty5    May 24 03:42    (mickey.Princeton)
% fm gun
Login name: egsirer                In real life: Emin Gun Sirer
Directory: /u/egsirer              Shell: /bin/sh
Last login Thu May 23 19:05 on ttyq3 from bow.Princeton.ED
New mail received Fri May 24 02:30:28 1991;
  unread since Fri May 24 02:30:27 1991
% alias -g phx=phoenix.princeton.edu warc=wuarchive.wustl.edu
% ftp warc
Connected to wuarchive.wustl.edu.
```

Global alias with `-g`

Here are some more interesting uses.

```
% alias -g M='| more' GF='| fgrep -f ~/.friends'
% who M      # pipes the output of who through more
% who GF     # see if your friends are on
% w GF       # see what your friends are doing
```

Global aliases for pipes

Another example makes use of `zsh`'s process substitution. If you run `NIS`, and you miss being able to do this:

```
% grep pfalstad /etc/passwd
```

grep local passwd file

you can define an alias that will seem more natural than `ypmatch pfalstad passwd`:

```
% alias -g PASS='<(ypcat passwd)'
% grep pfalstad PASS
pfalstad:*:3564:35:Paul John Falstad:/u/pfalstad:/usr/princeton/bin/zsh
```

Global alias for `ypcat`

If you're really crazy, you can even call it `/etc/passwd`:

```
% alias -g /etc/passwd='<(ypcat passwd)'  
% grep pfalstad /etc/passwd  
pfalstad:*:3564:35:Paul John Falstad:/u/pfalstad:/usr/princeton/bin/zsh
```

Alias over /etc/passwd

The last example shows one of the perils of global aliases; they have a lot of potential to cause confusion. For example, if you defined a global alias called `|` (which is possible), `zsh` would begin to act very strangely; every pipe symbol would be replaced with the text of your alias. To some extent, global aliases are like macros in C; discretion is advised in using them and in choosing names for them. Using names in all caps is not a bad idea, especially for aliases which introduce shell metasyntax (like `M` and `GF` above).

Note that `zsh` aliases are not like `csh` aliases. The syntax for defining them is different, and they do not have arguments. All your favorite `csh` aliases will probably not work under `zsh`. For example, if you try:

```
alias rm mv '!*' /tmp/wastebasket'
```

`csh` alias syntax fails

no aliases will be defined, but `zsh` will not report an error. In `csh`, this line defines an alias that makes `rm` safe—files that are `rm`'d will be moved to a temporary directory instead of instantly destroyed. In `zsh`'s syntax, however, this line asks the shell to print any existing alias definitions for `rm`, `mv`, or `!* /tmp/wastebasket`. Since there are none, most likely, the shell will not print anything, although `alias` will return a nonzero exit code. The proper syntax is this:

```
alias rm='mv !*' /tmp/wastebasket'
```

`csh` history in alias

However, this won't work either:

```
% rm foo.dvi  
zsh: no matches found: !*
```

History bang expands as glob

While this makes `rm` safe, it is certainly not what the user intended. In `zsh`, you must use a shell function for this:

```
% unalias rm  
% rm () { mv $* /tmp/wastebasket }  
% rm foo.dvi  
% ls /tmp/wastebasket  
foo.dvi
```

Safe `rm` as a function

While this is much cleaner and easier to read (I hope you will agree), it is not `csh`-compatible. Therefore, a script to convert `csh` aliases and variables has been provided. You should only need to use it once, to convert all your `csh` aliases and parameters to `zsh` format:

```
% csh  
csh> alias  
l      ls -AF  
more   less  
on     last -2 !:1 ; who | grep !:1  
csh> exit  
% c2z >neat_zsh_aliases  
% cat neat_zsh_aliases  
alias l='ls -AF'  
alias more='less'  
on () { last -2 $1 ; who | grep $1 }  
...
```

Convert aliases with `c2z`

The first two aliases were converted to regular zsh aliases, while the third, since it needed to handle arguments, was converted to a function. c2z can convert most aliases to zsh format without any problems. However, if you're using some really arcane csh tricks, or if you have an alias with a name like `do` (which is reserved in zsh), you may have to fix some of the aliases by hand.

The `c2z` script checks your csh setup, and produces a list of zsh commands which replicate your aliases and parameter settings as closely as possible. You could include its output in your startup file, `.zshrc`.

History

There are several ways to manipulate history in zsh. One way is to use csh-style `! history`:

```
% /usr/local/bin/! :0 !-2*:s/foo/bar/ >>!$
```

csh-style bang history

If you don't want to use this, you can turn it off by typing `setopt nobanghist`. If you are afraid of accidentally executing the wrong command you can set the `HIST_VERIFY` option. If this option is set, commands that result from history expansion will not be executed immediately, but will be put back into the editor buffer for further consideration.

If you're not familiar with `! history`, here follows some explanation. History substitutions always start with a `!`, commonly called "bang". After the `!` comes an (optional) designation of which "event" (command) to use, then a colon, and then a designation of what word of that command to use. For example, `!-n` refers to the command *n* commands ago.

```
% ls
foo bar
% cd foo
% !-2
ls
baz bam
```

Repeat with `!-n`

No word designator was used, which means that the whole command referred to was repeated. Note that the shell will echo the result of the history substitution. The word designator can, among other things, be a number indicating the argument to use, where `0` is the command.

```
% /usr/bin/ls foo
foo
% !:0 bar
/usr/bin/ls bar
bar
```

Word designator `:0`

In this example, no event designator was used, which tells zsh to use the previous command. A `$` specifies the last argument

```
% mkdir /usr/local/lib/emacs/site-lisp/calc
% cd !:$
cd /usr/local/lib/emacs/site-lisp/calc
```

Last argument with `!:$`

If you use more words of the same command, only the first `!` needs an event designator.

```
% make prig >> make.log
make: *** No rule to make target `prig'. Stop.
% cd src
```

History

```
% !-2:0 prog >> !:$
make prog >> make.log
```

Reuse words across events

This is different from csh, where a bang with no event designator always refers to the previous command. If you actually like this behaviour, set the CSH_JUNKIE_HISTORY option.

```
% setopt cshjunkiehistory
% !-2:0 prog2 >> !:$
make prog2 >> cshjunkiehistory
```

Setting CSH_JUNKIE_HISTORY

Another way to use history is to use the fc command. For example, if you type an erroneous command:

```
% for i in `cat /etc/clients`
do
  rpu $i
done
zsh: command not found: rpu
zsh: command not found: rpu
zsh: command not found: rpu
...
```

Erroneous command for fc

typing fc will execute an editor on this command, allowing you to fix it. (The default editor is vi, by the way, not ed).

```
% fc
49
/rpu/s//rup/p
rup $i
w
49
q
for i in `cat /etc/clients`
do
  rup $i
done
      beam    up  2 days, 10:17,    load average: 0.86, 0.80, 0.50
      bow     up  4 days,  8:41,    load average: 0.91, 0.80, 0.50
      burn    up           17:18,    load average: 0.91, 0.80, 0.50
      burst   up  9 days,  1:49,    load average: 0.95, 0.80, 0.50
      tan     up           11:14,    load average: 0.91, 0.80, 0.50
      bathe   up  3 days, 17:49,    load average: 1.84, 1.79, 1.50
      bird    up  1 day,  9:13,    load average: 1.95, 1.82, 1.51
      bonnet   up  2 days, 21:18,    load average: 0.93, 0.80, 0.50
...
```

Edit history with fc

A variant of the fc command is r, which redoes the last command, with optional changes:

```
% echo foo
foo
% r
echo foo
foo

% echo foo
```

```
foo
% r foo=bar
echo bar
bar
```

Redo with r command

Command Line Editing

zsh's command line editor, ZLE, is quite powerful. It is designed to emulate either emacs or vi; the default is emacs. To set the bindings for vi mode, type `bindkey -v`. If your `EDITOR` or `VISUAL` environment variable is vi, zsh will use vi emulation by default. You can then switch to emacs mode with `bindkey -e`.

In addition to basic editing, the shell allows you to recall previous lines in the history. In emacs mode, this is done with **Ctrl+P** (control-P) or (on many terminals) with the cursor-up key:

```
% ls ~
-          README      file          mail          pub          tmp
Mailboxes  bin             func          nicecolors    scr          zsh
News       etc             iris          notes          src
% echo foobar
foobar
% «Ctrl+P»
% echo foobar«Ctrl+P»
% ls ~_
```

History recall with Ctrl+P

Pressing **Ctrl+P** once brings up the previous line (`echo foobar`); pressing it again brings up the line before that (`ls ~`). The cursor is left at the end of the line, allowing you to edit the line if desired before executing it. In many cases, ZLE eliminates the need for the `fc` command, since it is powerful enough to handle even multiline commands:

```
% for i in a b c d e
> do
> echo $i
> done
a
b
c
d
e
% «Ctrl+P»
% for i in a b c d e
do
echo $i
done_
```

Recall multiline with Ctrl+P

Now you can just move up to the part you want to change...

```
% for i in _ b c d e
do
echo $i
done
```

Edit recalled multiline command

change it, and execute the new command.

```
% for i in f g h i j
do
echo $i
done
f
g
h
i
j
```

Execute edited multiline loop

Also, you can search the history for a certain command using **Esc+P**, this will look for the last command that started with the (part of the) word at the beginning of the current line. Hitting **Esc+P** another time gets you the command before that, etc.

```
% set «Esc+P»
% setopt autolist «Esc+P»
% setopt nocomple_
```

History search with Esc+P

Another way is to do an incremental search, emacs-style:

```
% «Ctrl+R»
% _
i-search:

% l_ /usr/bin
i-search: l

% date > foofile_c
i-search: le
```

Incremental history search

Suppose you have retrieved an old history event in one of these ways and would like to execute several consecutive old commands starting with this one. **Ctrl+O** will execute the current command and then put the next command from the history into the editor buffer. Typing **Ctrl+O** several times will therefore reexecute several consecutive commands from the history. Of course, you can edit some of those commands in between.

In addition to completion (see below), **Tab** performs expansion if possible.

```
% ls *.c«Tab»
% ls foofile.c fortune.c rnd.c strfile.c unstr.c_
```

Expand globs with Tab

For example, suppose you have a bunch of weird files in an important directory:

```
% ls
* * * ; & % $??foo dspfok foo.c
!"foo"! ` \ ` foo rrr
```

List awkward filenames

You want to remove them, but you don't want to damage foo.c. Here is one way to do this:

```
% rm *«Tab»
% rm \ \ \* \* \* \ \ \!\"foo\"! \; \ & \ % \ \$'
,,
'foo \ \ \ \ \` dspfok foo foo.c rrr_
```

Command Line Editing

Expand star with quoting

When you expand `*`, `zsh` inserts the names of all the files into the editing buffer, with proper shell quoting. Now, just move back and remove `foo.c` from the buffer:

```
% rm \ \ \*\ \*\ \*\ \ \ \!"foo\"!\ \;\ \&\ %\ \$'
,,
'foo \ \ \ \ \ ` dspfok foo _rr
```

Delete all but `foo.c`

and press return. Everything except `foo.c` will be deleted from the directory. If you do not want to actually expand the current word, but would like to see what the matches are, type **Ctrl+X**`g`.

```
% rm f*«Ctrl+X»g
foo      foo.c
% rm f*_
```

List matches with **Ctrl+X** `g`

Here's another trick; let's say you have typed this command in:

```
% gcc -o x.out foob.c -g -Wpointer-arith -Wtrigraphs_
```

Incomplete `gcc` command line

and you forget which library you want. You need to escape out for a minute and check by typing `ls /usr/lib`, or some other such command; but you don't want to retype the whole command again, and you can't press return now because the current command is incomplete. In `zsh`, you can put the line on the *buffer stack*, using **Esc+Q**, and type some other commands. The next time a prompt is printed, the `gcc` line will be popped off the stack and put in the editing buffer automatically; you can then enter the proper library name and press return (or, **Esc+Q** again and look for some other libraries whose names you forgot).

A similar situation: what if you forget the option to `gcc` that finds bugs using AI techniques? You could either use **Esc+Q** again, and type `man gcc`, or you could press **Esc+H**, which essentially does the same thing; it puts the current line on the buffer stack, and executes the command `run-help gcc`, where `run-help` is an alias for `man`.

Another interesting command is **Esc+A**. This executes the current line, but retains it in the buffer, so that it appears again when the next prompt is printed. Also, the cursor stays in the same place. This is useful for executing a series of similar commands:

```
% cc grok.c -g -lc -lg1 -lsun -lmalloc -Bstatic -o b.out
% cc fubar.c -g -lc -lg1 -lsun -lmalloc -Bstatic -o b.out
% cc fooble.c -g -lc -lg1 -lsun -lmalloc -Bstatic -o b.out
```

Repeat similar `cc` commands

The **Esc+'** command is useful for managing the shell's quoting conventions. Let's say you want to print this string:

```
don't do that; type 'rm -rf \*', with a \ before the *.
```

String needing shell quotes

All that is necessary is to type it into the editing buffer:

```
% don't do that; type 'rm -rf \*', with a \ before the *.
```

Type unquoted awkward string

press **Esc+''** (escape-quote):

```
% 'don\'\'t do that; type \'\'rm -rf \'\'\', with a \ before the *.'
```

After **Esc+''** quote-line

then move to the beginning and add the echo command.

```
% echo 'don't do that; type 'rm -rf \*', with a \ before the *.'
don't do that; type 'rm -rf \*', with a \ before the *.
```

Echo after quote-line

Let's say you want to create an alias to do this echo command. This can be done by recalling the line with **Ctrl+P** and pressing **Esc+'** again:

```
% 'echo \''don't do that; type 'rm -rf \*', with a \ before the *.'\'
\'don't do that; type 'rm -rf \*', with a \ before the *.'
```

Re-quote recalled echo

and then move to the beginning and add the command to create an alias.

```
% alias zoof='echo \''don't do that; type 'rm -rf \*', with a \ before the *.'\'
-rf \*', with a \ before the *.'\'
% zoof
don't do that; type 'rm -rf \*', with a \ before the *.
```

Quoted line as alias

If one of these fancy editor commands changes your command line in a way you did not intend, you can undo changes with **Ctrl+Z**, if you can get it out of your keyboard, or **Ctrl+X Ctrl+U**, otherwise.

Another use of the editor is to edit the value of variables. For example, an easy way to change your path is to use the vared command:

```
% vared PATH
> /u/pfalstad/scr:/u/pfalstad/bin/sun4:/u/maruchck/scr:/u/subbarao/bin:/u/maruc
hck/bin:/u/subbarao/scripts:/usr/princeton/bin:/usr/ucb:/usr/bin:/bin:/usr/host
s:/usr/princeton/bin/X11:./usr/lang:./usr/etc:./etc
```

Edit PATH with vared

You can now edit the path. When you press return, the contents of the edit buffer will be assigned to PATH.

Completion

Another great zsh feature is completion. If you hit **Tab**, zsh will complete all kinds of stuff. Like commands or filenames:

```
% comp«Tab»
% compress _

% ls nic«Tab»
% ls nicecolors _

% ls /usr/pr«Tab»
% ls /usr/princeton/_

% ls -l =com«Tab»
% ls -l =compress _
```

Complete commands and filenames

If the completion is ambiguous, the editor will beep. If you find this annoying, you can set the NO_LIST_BEEP option. Completion can even be done in the middle of words. To use this, you will have to set the COMPLETE_IN_WORD option:

```
% setopt completeinword
% ls /usr/p_on«Tab»
% ls /usr/prince_on/
% setopt alwaystoend
% ls /usr/p_on«Tab»
% ls /usr/princeton/_
```

Complete in middle of word

You can list possible completions by pressing **Ctrl+D**:

```
% ls /vmu«Tab» -beep-
% ls /vmunix_
% ls /vmunix«Ctrl+D»
vmunix                vmunix.old
vmunix.new.kernelmap.old  vmunix.org
```

List completions with Ctrl+D

Or, you could just set the AUTO_LIST option:

```
% setopt autolist
% ls /vmu«Tab» -beep-
vmunix                vmunix.old
vmunix.new.kernelmap.old  vmunix.org
% ls /vmunix_
```

AUTO_LIST completion listing

If you like to see the types of the files in these lists, like in `ls -F`, you can set the LIST_TYPES option. Together with AUTO_LIST you can use LIST_AMBIGUOUS. This will only list the possibilities if there is no unambiguous part to add:

```
% setopt listambiguous
% ls /vmu«Tab» -beep-
% ls /vmunix_«Tab» -beep-
vmunix                vmunix.old
vmunix.new.kernelmap.old  vmunix.org
```

LIST_AMBIGUOUS with Tab

If you don't want several of these listings to scroll the screen so much, the ALWAYS_LAST_PROMPT option is useful. If set, you can continue to edit the line you were editing, with the completion listing appearing beneath it.

Another interesting option is MENU_COMPLETE. This affects the way **Tab** works. Let's look at the /vmunix example again:

```
% setopt menucomplete
% ls /vmu«Tab»
% ls /vmunix«Tab»
% ls /vmunix.new.kernelmap.old«Tab»
% ls /vmunix.old_
```

Menu completion with Tab

Each time you press **Tab**, it displays the next possible completion. In this way, you can cycle through the possible completions until you find the one you want.

The AUTO_MENU option makes a nice compromise between this method of completion and the regular method. If you set this option, pressing **Tab** once completes the unambiguous part normally, pressing the **Tab** key repeatedly after an ambiguous completion will cycle through the possible completions.

Another option you could set is REC_EXACT, which causes exact matches to be accepted, even if there are other possible completions:

```
% setopt recexact
% ls /vmu«Tab» -beep-
vmunix                vmunix.old
vmunix.new.kernelmap.old  vmunix.org
% ls /vmunix_«Tab»
% ls /vmunix _
```

Accept exact matches

To facilitate the typing of pathnames, a slash will be added whenever a directory is completed. Some computers don't like the spurious slashes at the end of directory names. In that case, the `AUTO_REMOVE_SLASH` option comes to rescue. It will remove these slashes when you type a space or return after them.

The *ignore* variable lists suffixes of files to ignore during completion.

```
% ls foo«Tab» -beep-
foofile.c  foofile.o
% ignore=( .o \~ .bak .junk )
% ls foo«Tab»
% ls foofile.c _
```

Ignore suffixes with ignore

Since `foofile.o` has a suffix that is in the `ignore` list, it was not considered a possible completion of `foo`.

Username completion is also supported:

```
% ls ~pfal«Tab»
% ls ~pfalstad/_
```

Complete tilde usernames

and parameter name completion:

```
% echo $ORG«Tab»
% echo $ORGANIZATION _
% echo ${ORG«Tab»
% echo ${ORGANIZATION _
```

Parameter name completion

Note that in the last example a space is added after the completion as usual. But if you want to add a colon or closing brace, you probably don't want this extra space. Setting the `AUTO_PARAM_KEYS` option will automatically remove this space if you type a colon or closing brace after such a completion.

There is also option completion:

```
% setopt nocl«Tab»
% setopt noclobber _
```

Option name completion

and binding completion:

```
% bindkey '^X^X' pu«Tab»
% bindkey '^X^X' push-line _
```

Completing bindkey arguments

The `compctl` command is used to control completion of the arguments of specific commands. For example, to specify that certain commands take other commands as arguments, you use `compctl -c`:

Completion

```
% compctl -c man nohup
% man upt«Tab»
% man uptime _
```

compctl commands for man

To specify that a command should complete filenames, you should use `compctl -f`. This is the default. It can be combined with `-c`, as well.

```
% compctl -cf echo
% echo upt«Tab»
% echo uptime _

% echo fo«Tab»
% echo foo.c
```

compctl commands and files

Similarly, use `-o` to specify options, `-v` to specify variables, and `-b` to specify bindings.

```
% compctl -o setopt unsetopt
% compctl -v typeset vared unset export
% compctl -b bindkey
```

compctl options variables bindings

You can also use `-k` to specify a custom list of keywords to use in completion. After the `-k` comes either the name of an array or a literal array to take completions from.

```
% ftphosts=(ftp.uu.net wuarchive.wustl.edu)
% compctl -k ftphosts ftp
% ftp wu«Tab»
% ftp wuarchive.wustl.edu _

% compctl -k '(cpirazzi subbarao sukthnr)' mail finger
% finger cp«Tab»
% finger cpirazzi _
```

compctl custom keyword lists

To better specify the files to complete for a command, use the `-g` option which takes any glob pattern as an argument. Be sure to quote the glob patterns as otherwise they will be expanded when the `compctl` command is run.

```
% ls
letter.tex letter.dvi letter.aux letter.log letter.toc
% compctl -g '*.tex' latex
% compctl -g '*.dvi' xdvi dvips
% latex l«Tab»
% latex letter.tex _
% xdvi l«Tab»
% xdvi letter.dvi _
```

compctl globs for TeX tools

Glob patterns can include qualifiers within parentheses. To `rmdir` only directories and `cd` to directories and symbolic links pointing to them:

```
% compctl -g '*(-/)' cd
% compctl -g '*(/)' rmdir
```

compctl directory-only globs

Completion

RCS users like to run commands on files which are not in the current directory, but in the RCS subdirectory where they all get ,v suffixes. They might like to use

```
% compctl -g 'RCS/*(:t:s/\,v//)' co rlog rcs
% ls RCS
builtin.c,v lex.c,v      zle_main.c,v
% rlog bu«Tab»
% rlog builtin.c _
```

compctl RCS ,v files

The :t modifier keeps only the last part of the pathname and the :s/\,v// will replace any ,v by nothing.

The -s flag is similar to -g, but it uses all expansions, instead of just globbing, like brace expansion, parameter substitution and command substitution.

```
% compctl -s '$(setopt)' unsetopt
```

compctl -s substitutions

will only complete options which are actually set to be arguments to unsetopt.

Sometimes a command takes another command as its argument. You can tell zsh to complete commands as the first argument to such a command and then use the completion method of the second command. The -l flag with a null-string argument is used for this.

```
% compctl -l '' nohup exec
% nohup comp«Tab»
% nohup compress _
% nohup compress fil«Tab»
% nohup compress filename _
```

compctl follow command args

Sometimes you would like to run really complicated commands to find out what the possible completions are. To do this, you can specify a shell function to be called that will assign the possible completions to a variable called reply. Note that this variable must be an array. Here's another (much slower) way to get the completions for co and friends:

```
% function getrcs {
> reply=()
> for i in RCS/*
> do
>   reply=($reply[*] $(basename $i ,v))
> done
> }
% compctl -K getrcs co rlog rcs
```

compctl via reply function

Some command arguments use a prefix that is not a part of the things to complete. The kill builtin command takes a signal name after a -. To make such a prefix be ignored in the completion process, you can use the -P flag.

```
% compctl -P - -k signals kill
% kill -H«Tab»
% kill -HUP _
```

compctl kill signal prefix

TeX is usually run on files ending in .tex, but also sometimes on other files. It is somewhat annoying to specify that the arguments of TeX should end in .tex and then not be able to complete these other files. Therefore you can specify things like "Complete to files ending in .tex if available, otherwise complete to any filename.". This is done with xored completion:

```
% compctl -g '*.tex' + -f tex
```

Xor completion for tex

The + tells the editor to only take the next thing into account if the current one doesn't generate any matches. If you have not changed the default completion, the above example is in fact equivalent to

```
% compctl -g '*.tex' + tex
```

Xor with default completion

as a lone + at the end is equivalent to specifying the default completion after the +. This form of completion is also frequently used if you want to run some command only on a certain type of files, but not necessarily in the current directory. In this case you will want to complete both files of this type and directories. Depending on your preferences you can use either of

```
% compctl -g '*.ps' + -g '*(-/)' ghostview
% compctl -g '*.ps *(-/)' ghostview
```

Xor files and directories

where the first one will only complete directories (and symbolic links pointing to directories) if no postscript file matches the already typed part of the argument.

Extended completion

If you play with completion, you will soon notice that you would like to specify what to complete, depending on what flags you give to the command and where you are on the command line. For example, a command could take any filename argument after a -f flag, a username after a -u flag and an executable after a -x flag. This section will introduce you to the ways to specify these things. To many people it seems rather difficult at first, but taking the trouble to understand it can save you lots of typing in the end. Even I keep being surprised when zsh manages to complete a small or even empty prefix to the right file in a large directory.

To tell zsh about these kinds of completion, you use “extended completion” by specifying the -x flag to compctl. The -x flag takes a list of patterns/flags pairs. The patterns specify when to complete and the flags specify what. The flags are simply those mentioned above, like -f or -g `:emphasis:`glob` pattern`.

As an example, the `r[string1, string2]` pattern matches if the cursor is after something that starts with *string1* and before something that starts with *string2*. The *string2* is often something that you do not want to match anything at all.

```
% ls
foo1  bar1  foo.Z  bar.Z
% compctl -g '^*.Z' -x 'r[-d,---]' -g '*.Z' -- compress
% compress f<Tab>
% compress foo1 _
% compress -d f<Tab>
% compress -d foo.Z _
```

`r[]` pattern for compress -d

In the above example, if the cursor is after the -d the pattern will match and therefore zsh uses the -g `*.Z` flag that will only complete files ending in .Z. Otherwise, if no pattern matches, it will use the flags before the -x and in this case complete every file that does not end in .Z.

The `s[string]` pattern matches if the current word starts with *string*. The *string* itself is not considered to be part of the completion.

```
% compctl -x 's[-]' -k signals -- kill
% kill -H<Tab>
% kill -HUP _
```

`s[]` pattern for kill signals

Extended completion

The `tar` command takes a tar file as an argument after the `-f` option. The `c[offset,string]` pattern matches if the word in position *offset* relative to the current word is *string*. More in particular, if *offset* is `-1`, it matches if the previous word is *string*. This suggests

```
% compctl -f -x 'c[-1,-f]' -g '*.tar' -- tar
```

`c[]` pattern for `tar -f`

But this is not enough. The `-f` option could be the last of a longer string of options. `C[...]` is just like `c[...]`, except that it uses glob-like pattern matching for *string*. So

```
% compctl -f -x 'C[-1,-*f]' -g '*.tar' -- tar
```

`C[]` pattern for `tar -f` cluster

will complete tar files after any option string ending in an `f`. But we'd like even more. Old versions of `tar` used all options as the first argument, but without the minus sign. This might be inconsistent with option usage in all other commands, but it is still supported by newer versions of `tar`. So we would also like to complete tar files if the first argument ends in an `f` and we're right behind it.

We can *and* patterns by putting them next to each other with a space between them. We can *or* these sets by putting comma's between them. We will also need some new patterns. `:literal:p[num]` will match if the current argument (the one to be completed) is the *num*th argument. `w[index,pattern]` will match if the argument in place *index* matches the *pattern*. This gives us

```
% compctl -f -x 'C[-1,-*f] , w[1,*f] p[2]' -g '*.tar' -- tar
```

`tar -f` and old-style options

In words: If the previous argument is an option string that ends in an `f`, or the first argument ended in an `f` and it is now the second argument, then complete only filenames ending in `.tar`.

All the above examples used only one set of patterns with one completion flag. You can use several of these pattern/flag pairs separated by a `-`. The first matching pattern will be used. Suppose you have a version of `tar` that supports compressed files by using a `-Z` option. Leaving the old `tar` syntax aside for a moment, we would like to complete files ending in `.tar.Z` if a `-Z` option has been used and files ending in `.tar` otherwise, all this only after a `-f` flag. Again, the `-Z` can be alone or it can be part of a longer option string, perhaps the same as that of the `-f` flag. Here's how to do it; note the backslash and the secondary prompt which are not part of the `compctl` command.

```
% compctl -f -x 'C[-1,-*Z*f] , R[-*Z*,---] C[-1,-*f]' -g '*.tar.Z' - \
> 'C[-1,-*f]' -g '*.tar' -- tar
```

`tar` compressed vs plain files

The first pattern set tells us to match if either the previous argument was an option string including a `Z` and ending in an `f` or there was an option string with a `Z` somewhere and the previous word was any option string ending in an `f`. If this is the case, we need a compressed tar file. Only if this is not the case the second pattern set will be considered. By the way, `R[pattern1,pattern2]` is just like `r[...]` except that it uses pattern matching with shell metacharacters instead of just strings.

You will have noticed the `--` before the command name. This ends the list of pattern/flag pairs of `-x`. It is usually used just before the command name, but you can also use an extended completion as one part of a list of xored completions, in which case the `--` appears just before one of the `+` signs.

Note the difference between using extended completion as part of a list of xored completions as in

```
% ls
foo bar
% compctl -x 'r[-d,---]' -g '*.Z' -- + -g '^*.Z' compress
% compress -d f«Tab»
% compress -d foo _
```

Xor after extended completion

and specifying something before the `-x` as in

```
% compctl -g '^*.Z' -x 'r[-d,---]' -g '*.Z' -- compress
% compress -d f«Tab»
% compress -d f_
```

Flags before -x fallback

In the first case, the alternative glob pattern (^*.Z) will be used if the first part does not generate any possible completions, while in the second case the alternative glob pattern will only be used if the r[...] pattern doesn't match.

Bindings

Each of the editor commands we have seen was actually a function bound by default to a certain key. The real names of the commands are:

```
expand-or-complete    «Tab»
push-line             «Esc+Q»
run-help              «Esc+H»
accept-and-hold       «Esc+A»
quote-line            «Esc+'»
```

Default editor key bindings

These bindings are arbitrary; you could change them if you want. For example, to bind accept-line to **Ctrl+Z**:

```
% bindkey '^Z' accept-line
```

bindkey for Ctrl+Z

Another idea would be to bind the delete key to delete-char; this might be convenient if you use **Ctrl+H** for backspace.

```
% bindkey '^?' delete-char
```

bindkey delete-char

Or, you could bind **Ctrl+X Ctrl+H** to run-help:

```
% bindkey '^X^H' run-help
```

bindkey Ctrl+X H run-help

Other examples:

```
% bindkey '^X^Z' universal-argument
% bindkey ' ' magic-space
% bindkey -s '^T' 'uptime
> '
% bindkey '^Q' push-line-or-edit
```

Miscellaneous bindkey examples

universal-argument multiplies the next command by 4. Thus **Ctrl+X Ctrl+Z Ctrl+W** might delete the last four words on the line. If you bind space to magic-space, then csh-style history expansion is done on the line whenever you press the space bar.

Something that often happens is that I am typing a multiline command and discover an error in one of the previous lines. In this case, push-line-or-edit will put the entire multiline construct into the editor buffer. If there is only a single line, it is equivalent to push-line.

The -s flag to bindkey specifies that you are binding the key to a string, not a command. Thus bindkey -s '^T' 'uptime\n' lets you VMS lovers get the load average whenever you press **Ctrl+T**.

Parameter Substitution

If you have a NeXT keyboard, the one with the `|` and `\` keys very inconveniently placed, the following bindings may come in handy:

```
% bindkey -s '\e/' '\\'  
% bindkey -s '\e=' '|'
```

NeXT keyboard Alt bindings

Now you can type **Alt+`/`** to get a backslash, and **Alt+`=`** to get a vertical bar. This only works inside `zsh`, of course; `bindkey` has no effect on the key mappings inside `talk` or `mail`, etc.

Some people like to bind **Ctrl+S** and **Ctrl+Q** to editor commands. Just binding these has no effect, as the terminal will catch them and use them for flow control. You could unset them as stop and start characters, but most people like to use these for external commands. The solution is to set the `NO_FLOW_CONTROL` option. This will allow you to bind the start and stop characters to editor commands, while retaining their normal use for external commands.

Parameter Substitution

In `zsh`, parameters are set like this:

```
% foo=bar  
% echo $foo  
bar
```

Assigning a scalar

Spaces before or after the `=` are frowned upon:

```
% foo = bar  
zsh: command not found: foo
```

Spaces around equals fail

Also, `set` doesn't work for setting parameters:

```
% set foo=bar  
% set foo = bar  
% echo $foo  
  
%
```

`set` leaves parameter empty

Note that no error message was printed. This is because both of these commands were perfectly valid; the `set` builtin assigns its arguments to the *positional parameters* (`$1`, `$2`, etc.).

```
% set foo=bar  
% echo $1  
foo=bar  
% set foo = bar  
% echo $3 $2  
bar =
```

`set` fills positional params

If you're really intent on using the `cs`h syntax, define a function like this:

```
% set () {  
>   eval "$1$2$3"  
> }  
% set foo = bar  
% set fuu=brrr
```

Parameter Substitution

```
% echo $foo $fuu
bar brrr
```

csh-style set function

But then, of course you can't use the form of set with options, like set -F (which turns off filename generation). Also, the set command by itself won't list all the parameters like it should. To get around that you need a case statement:

```
% set () {
>   case $1 in
>     -*|+*|'') builtin set $* ;;
>     *) eval "$1$2$3" ;;
>   esac
> }
```

set function with options

For the most part, this should make csh users happy.

The following sh-style operators are supported in zsh:

```
% unset null
% echo ${foo-xxx}
bar
% echo ${null-xxx}
xxx
% unset null
% echo ${null=xxx}
xxx
% echo $null
xxx
% echo ${foo=xxx}
bar
% echo $foo
bar
% unset null
% echo ${null+set}

% echo ${foo+set}
set
```

Default and alternate values

Also, csh-style : modifiers may be appended to a parameter substitution.

```
% echo $PWD
/home/learning/pf/zsh/zsh2.00/src
% echo $PWD:h
/home/learning/pf/zsh/zsh2.00
% echo $PWD:h:h
/home/learning/pf/zsh
% echo $PWD:t
src
% name=foo.c
% echo $name
foo.c
% echo $name:r
foo
% echo $name:e
c
```

Parameter Substitution

csh colon modifiers

The equivalent constructs in ksh (which are also supported in zsh) are a bit more general and easier to remember. When the shell expands `${foo#pat}`, it checks to see if *pat* matches a substring at the beginning of the value of *foo*. If so, it removes that portion of *foo*, using the shortest possible match. With `${foo##pat}`, the longest possible match is removed. `${foo%pat}` and `${foo%%pat}` remove the match from the end. Here are the ksh equivalents of the `:` modifiers:

```
% echo ${PWD%/*}
/home/learning/pf/zsh/zsh2.00
% echo ${PWD%/*/*}
/home/learning/pf/zsh
% echo ${PWD##*/}
src
% echo ${name%.*}
foo
% echo ${name#*.*}
c
```

ksh pattern removal

zsh also has upper/lowercase modifiers:

```
% xx=Test
% echo $xx:u
TEST
% echo $xx:l
test
```

Upper and lowercase modifiers

and a substitution modifier:

```
% echo $name:s/foo/bar/
bar.c
% ls
foo.c    foo.h    foo.o    foo.pro
% for i in foo.*; mv $i ${i:s/foo/bar/}
% ls
bar.c    bar.h    bar.o    bar.pro
```

String substitution modifier

There is yet another syntax to modify substituted parameters. You can add certain modifiers in parentheses after the opening brace like:

```
${(modifiers)parameter}
```

Parenthesized modifiers syntax

For example, `o` sorts the words resulting from the expansion:

```
% echo ${path}
/usr/bin /usr/bin/X11 /etc
% echo ${(o)path}
/etc /usr/bin /usr/bin/X11
```

Sorting with `o` modifier

One possible source of confusion is the fact that in zsh, the result of parameter substitution is *not* split into words. Thus, this will not work:

Parameter Substitution

```
% srcs='glob.c exec.c init.c'
% ls $srcs
glob.c exec.c init.c not found
```

No default word splitting

This is considered a feature, not a bug. If splitting were done by default, as it is in most other shells, functions like this would not work properly:

```
$ ll () { ls -F $* }
$ ll 'fuu bar'
fuu not found
bar not found

% ll 'fuu bar'
fuu bar not found
```

Preserving quoted arguments

Of course, a hackish workaround is available in sh (and zsh):

```
% setopt shwordsplit
% ll () { ls -F "$@" }
% ll 'fuu bar'
fuu bar not found
```

SH_WORD_SPLIT with \$@

If you like the sh behaviour, zsh can accommodate you:

```
% ls ${=srcs}
exec.c glob.c init.c
% setopt shwordsplit
% ls $srcs
exec.c glob.c init.c
```

Forced word splitting

Another way to get the \$srcs trick to work is to use an array:

```
% unset srcs
% srcs=( glob.c exec.c init.c )
% ls $srcs
exec.c glob.c init.c
```

Array assignment for files

or an alias:

```
% alias -g SRCS='exec.c glob.c init.c'
% ls SRCS
exec.c glob.c init.c
```

Global alias for files

Another option that modifies parameter expansion is RC_EXPAND_PARAM:

```
% echo foo/$srcs
foo/glob.c exec.c init.c
% setopt rcexpandparam
% echo foo/$srcs
foo/glob.c foo/exec.c foo/init.c
```

```
% echo foo/${^srcs}
foo/glob.c foo/exec.c foo/init.c
% echo foo/${^srcs}
foo/glob.c foo/exec.c foo/init.c
```

RC_EXPAND_PARAM prefixes

Shell Parameters

The shell has many predefined parameters that may be accessed. Here are some examples:

```
% sleep 10 &
[1] 3820
% echo $!
3820
% set a b c
% echo $#
3
% echo $ARGC
3
% ( exit 20 ) ; echo $?
20
% false; echo $status
1
```

Job and status parameters

(\$? and \$status are equivalent.)

```
% echo $HOST $HOSTTYPE
dendrite sun4
% echo $UID $GID
701 60
% cd /tmp
% cd /home
% echo $PWD $OLDPWD
/home /tmp
% ls $OLDPWD/.getwd
/tmp/.getwd
```

Host UID and directories

~+ and ~- are short for \$PWD and \$OLDPWD, respectively.

```
% ls ~-/.getwd
/tmp/.getwd
% ls -d ~+/learning
/home/learning
% echo $RANDOM
4880
% echo $RANDOM
11785
% echo $RANDOM
2062
% echo $TTY
/dev/ttyp4
% echo $VERSION
zsh v2.00.03
% echo $USERNAME
pf
```

Shell Parameters

Tilde shortcuts and misc

The `cdpath` variable sets the search path for the `cd` command. If you do not specify `.` somewhere in the path, it is assumed to be the first component.

```
% cdpath=( /usr ~ ~/zsh )
% ls /usr
5bin      dict      lang      net      sccs      sys
5include  etc       lector    nserve   services  tmp
5lib      export   lib       oed      share     ucb
adm       games    local     old      skel      ucbinclude
bin       geac     lost+found openwin  spool     ucblib
boot     hosts    macsyms_417 pat      src       xpg2bin
demo     include  man       princeton stand     xpg2include
diag     kvm      mdec      pub      swap      xpg2lib
% cd spool
/usr/spool
% cd bin
/usr/bin
% cd func
~/func
% cd
% cd pub
% pwd
/u/pfalstad/pub
% ls -d /usr/pub
/usr/pub
```

`cdpath` directory search

`PATH` and `path` both set the search path for commands. These two variables are equivalent, except that one is a string and one is an array. If the user modifies `PATH`, the shell changes `path` as well, and vice versa.

```
% PATH=/bin:/usr/bin:/tmp:.
% echo $path
/bin /usr/bin /tmp .
% path=( /usr/bin . /usr/local/bin /usr/ucb )
% echo $PATH
/usr/bin:./usr/local/bin:/usr/ucb
```

`PATH` and `path` linked

The same is true of `CDPATH` and `cdpath`:

```
% echo $CDPATH
/usr:/u/pfalstad:/u/pfalstad/zsh
% CDPATH=/u/subbarao:/usr/src:/tmp
% echo $cdpath
/u/subbarao /usr/src /tmp
```

`CDPATH` and `cdpath` linked

In general, predefined parameters with names in all lowercase are arrays; assignments to them take the form:

```
name=( elem ... )
```

Array assignment form

Predefined parameters with names in all uppercase are strings. If there is both an array and a string version of the same parameter, the string version is a colon-separated list, like `PATH`. You can use

```
% typeset -T F00 foo
```

Shell Parameters

typeset tied parameters

to create more of these yourself.

```
% foo=( a b c )
% echo $F00
a:b:c
```

Tied array and string

HISTFILE is the name of the history file, where the history is saved when a shell exits.

```
% zsh
phoenix% HISTFILE=/tmp/history
phoenix% SAVEHIST=20
phoenix% echo foo
foo
phoenix% date
Fri May 24 05:39:35 EDT 1991
phoenix% uptime
 5:39am up 4 days, 20:02,  40 users,  load average: 2.30, 2.20, 2.00
phoenix% exit
% cat /tmp/history
HISTFILE=/tmp/history
SAVEHIST=20
echo foo
date
uptime
exit
% HISTSIZE=3
% history
 28  rm /tmp/history
 29  HISTSIZE=3
 30  history
```

History file and size

If you have several instances of zsh running at the same time, like when using the X window system, it might be preferable to append the history of each shell to a file when a shell exits instead of overwriting the old contents of the file. You can get this behaviour by setting the APPEND_HISTORY option.

In zsh, if you say

```
% >file
```

Bare redirection alone

the command cat is normally assumed:

```
% >file
foo!
^D
% cat file
foo!
```

NULLCMD cat by default

Thus, you can view a file simply by typing:

```
% <file
foo!
```

Bare input shows file

Prompting

However, this is not csh or sh compatible. To correct this, change the value of the parameter NULLCMD, which is cat by default.

```
% NULLCMD=:  
% >file  
% ls -l file  
-rw-r--r--  1 pfalstad      0 May 24 05:41 file
```

NULLCMD as no-op

If NULLCMD is unset, the shell reports an error if no command is specified (like csh).

```
% unset NULLCMD  
% >file  
zsh: redirection with no command
```

Unset NULLCMD errors

Actually, READNULLCMD is used whenever you have a null command reading input from a single file. Thus, you can set READNULLCMD to more or less rather than cat. Also, if you set NULLCMD to : for sh compatibility, you can still read files with < file if you leave READNULLCMD set to more.

Prompting

The default prompt for zsh is:

```
phoenix% echo $PROMPT  
%m%#
```

Default host prompt

The %m stands for the short form of the current hostname, and the %# stands for a % or a #, depending on whether the shell is running as root or not. zsh supports many other control sequences in the PROMPT variable.

```
% PROMPT='%/> '  
/u/pfalstad/etc/TeX/zsh>  
  
% PROMPT='%~> '  
~/etc/TeX/zsh>  
  
% PROMPT='%h %~> '  
6 ~/etc/TeX/zsh>
```

Path and history escapes

%h represents the number of current history event.

```
% PROMPT='%h %~ %M> '  
10 ~/etc/TeX/zsh apple-gunkies.gnu.ai.mit.edu>  
  
% PROMPT='%h %~ %m> '  
11 ~/etc/TeX/zsh apple-gunkies>  
  
% PROMPT='%h %t> '  
12 6:11am>  
  
% PROMPT='%n %w tty%l> '  
pfalstad Fri 24 tty0>
```

Host time user escapes

Login/logout watching

PROMPT2 is used in multiline commands, like for-loops. The %_ escape sequence was made especially for this prompt. It is replaced by the kind of command that is being entered.

```
% PROMPT2='%_> '
% for i in foo bar
for>

% echo 'hi
quote>
```

Multiline PROMPT2 status

Also available is the R PROMPT parameter. If this is set, the shell puts a prompt on the *right* side of the screen.

```
% R PROMPT='%t '
%
6:14am

% R PROMPT='%~ '
%
~/etc/TeX/zsh

% PROMPT='%l %T %m[%h] ' R PROMPT='%~ '
p0 6:15 phoenix[5] ~/etc/TeX/zsh
```

Right-side R PROMPT

These special escape sequences can also be used with the -P option to print:

```
% print -P %h tty%l
15 ttyp1
```

print with -P escapes

The POSTEDIT parameter is printed whenever the editor exits. This can be useful for termcap tricks. To highlight the prompt and command line while leaving command output unhighlighted, try this:

```
% POSTEDIT=`echotc se`
% PROMPT='%S%% '

```

POSTEDIT highlight trick

This trick is mostly superseded by the zle_highlight array parameter.

Login/logout watching

You can specify login or logout events to monitor by setting the watch variable. Normally, this is done by specifying a list of usernames.

```
% watch=( pfalstad subbarao sukthnkr egsirer )
```

Watch list of users

The log command reports all people logged in that you are watching for.

```
% log
pfalstad has logged on p0 from mickey.
pfalstad has logged on p5 from mickey.
% ...
subbarao has logged on p8 from phoenix.
% ...
subbarao has logged off p8 from phoenix.
% ...
```

```
sukthnkr has logged on p8 from dew.  
% ...  
sukthnkr has logged off p8 from dew.
```

log reports watched users

If you specify hostnames with an @ prepended, the shell will watch for all users logging in from the specified host.

```
% watch=( @mickey @phoenix )  
% log  
djthongs has logged on q2 from phoenix.  
pfalstad has logged on p0 from mickey.  
pfalstad has logged on p5 from mickey.
```

Watch by hostname

If you give a tty name with a % prepended, the shell will watch for all users logging in on that tty.

```
% watch=( %ttyp0 %console )  
% log  
root has logged on console from .  
pfalstad has logged on p0 from mickey.
```

Watch by tty

The format of the reports may also be changed.

```
% watch=( pfalstad gettes eps djthongs jcorr bdavis )  
% log  
jcorr has logged on tf from 128.112.176.3:0.  
jcorr has logged on r0 from 128.112.176.3:0.  
gettes has logged on p4 from yo:0.0.  
djthongs has logged on pe from grumpy:0.0.  
djthongs has logged on q2 from phoenix.  
bdavis has logged on qd from BRUNO.  
eps has logged on p3 from csx30:0.0.  
pfalstad has logged on p0 from mickey.  
pfalstad has logged on p5 from mickey.  
% WATCHFMT='%n on tty%l from %M'  
% log  
jcorr on ttytf from 128.112.176.3:0.  
jcorr on ttyr0 from 128.112.176.3:0.  
gettes on ttyp4 from yo:0.0  
djthongs on ttype from grumpy:0.0  
djthongs on ttyq2 from phoenix.Princeto  
bdavis on ttyqd from BRUNO.ppp1.gov  
eps on ttyp3 from csx30:0.0  
pfalstad on ttyp0 from mickey.Princeton  
pfalstad on ttyp5 from mickey.Princeton  
% WATCHFMT='%n fm %m'  
% log  
jcorr fm 128.112.176.3:0  
jcorr fm 128.112.176.3:0  
gettes fm yo:0.0  
djthongs fm grumpy:0.0  
djthongs fm phoenix  
bdavis fm BRUNO  
eps fm csx30:0.0  
pfalstad fm mickey  
pfalstad fm mickey  
% WATCHFMT='%n %a at %t %w.'
```

```
% log
jcorr logged on at 3:15pm Mon 20.
jcorr logged on at 3:16pm Wed 22.
gettes logged on at 6:54pm Wed 22.
djthongs logged on at 7:19am Thu 23.
djthongs logged on at 7:20am Thu 23.
bdavis logged on at 12:40pm Thu 23.
eps logged on at 4:19pm Thu 23.
pfalstad logged on at 3:39am Fri 24.
pfalstad logged on at 3:42am Fri 24.
```

Custom WATCHFMT formats

If you have a `.friends` file in your home directory, a convenient way to make `zsh` watch for all your friends is to do this:

```
% watch=( $(< ~/.friends) )
% echo $watch
subbarao maruchck root sukthnkr ...
```

Watch from friends file

If `watch` is set to `all`, then all users logging in or out will be reported.

Options

Some options have already been mentioned; here are a few more:

Using the `AUTO_CD` option, you can simply type the name of a directory, and it will become the current directory.

```
% cd /
% setopt autocd
% bin
% pwd
/bin
% ../etc
% pwd
/etc
```

`AUTO_CD` without `cd`

With `CDABLE_VARS`, if the argument to `cd` is the name of a parameter whose value is a valid directory, it will become the current directory.

```
% setopt cdablevars
% foo=/tmp
% cd foo
/tmp
```

`CDABLE_VARS` via parameter

`CORRECT` turns on spelling correction for commands, and the `CORRECT_ALL` option turns on spelling correction for all arguments.

```
% setopt correct
% sl
zsh: correct `sl' to `ls' [nyae]? y
% setopt correctall
% ls x.v11r4
zsh: correct `x.v11r4' to `X.V11R4' [nyae]? n
/usr/princeton/src/x.v11r4 not found
```

```
% ls /etc/paswd
zsh: correct to `/etc/paswd' to `/etc/passwd' [nyae]? y
/etc/passwd
```

CORRECT and CORRECT_ALL

If you press y when the shell asks you if you want to correct a word, it will be corrected. If you press n, it will be left alone. Pressing a aborts the command, and pressing e brings the line up for editing again, in case you agree the word is spelled wrong but you don't like the correction.

Normally, a quoted expression may contain a newline:

```
% echo '
> foo
> '

foo

%
```

Newlines in quoted strings

With CSH_JUNKIE_QUOTES set, this is invalid, as it is in csh.

```
% setopt cshjunkiequotes
% ls 'foo
zsh: unmatched '
```

CSH_JUNKIE_QUOTES rejects

GLOB_DOTS lets files beginning with a . be matched without explicitly specifying the dot. This can also be specified for a particular pattern by appending (D) to it.

```
% ls -d *x*
Mailboxes
% ls -d *x*(D)
.exrc      .pnewsexpert  .xserverrc
.mushexpert .xinitrc      Mailboxes
% setopt globdots
% ls -d *x*
.exrc      .pnewsexpert  .xserverrc
.mushexpert .xinitrc      Mailboxes
```

GLOB_DOTS matches dots

HIST_IGNORE_DUPS prevents the current line from being saved in the history if it is the same as the previous one; HIST_IGNORE_SPACE prevents the current line from being saved if it begins with a space.

```
% PROMPT='%h> '
39> setopt histignoredups
40> echo foo
foo
41> echo foo
foo
41> echo foo
foo
41> echo bar
bar
42> setopt histignorespace
43> echo foo
foo
43> echo fubar
```

Options

```
fubar
43> echo fubar
fubar
```

HIST_IGNORE_DUPS and SPACE

IGNORE_BRACES turns off csh-style brace expansion.

```
% echo x{y{z,a},{b,c}d}e
xyze xyae xbde xcde
% setopt ignorebraces
% echo x{y{z,a},{b,c}d}e
x{y{z,a},{b,c}d}e
```

IGNORE_BRACES disables braces

IGNORE_EOF forces the user to type exit or logout, instead of just pressing **Ctrl+D**.

```
% setopt ignoreeof
% ^D
zsh: use 'exit' to exit.
```

IGNORE_EOF requires exit

INTERACTIVE_COMMENTS turns on interactive comments; comments begin with a #.

```
% setopt interactivecomments
% date # this is a comment
Fri May 24 06:54:14 EDT 1991
```

INTERACTIVE_COMMENTS with #

NO_BEEP makes sure the shell never beeps.

NO_CLOBBER prevents you from accidentally overwriting an existing file.

```
% setopt noclobber
% cat /dev/null > ~/.zshrc
zsh: file exists: /u/pfalstad/.zshrc
```

NO_CLOBBER blocks overwrite

If you really do want to clobber a file, you can use the >! operator. To make things easier in this case, the > is stored in the history list as a >!:

```
% cat /dev/null >! ~/.zshrc
% cat /etc/motd > ~/.zshrc
zsh: file exists: /u/pfalstad/.zshrc
% !!
cat /etc/motd >! ~/.zshrc
% ...
```

Force overwrite with >!

RC_QUOTES lets you use a more elegant method for including single quotes in a singly quoted string:

```
% echo '"don'\''t do that.'"
"don't do that."
% echo '"don''t do that.'"
"dont do that."
% setopt rcquotes
% echo '"don''t do that.'"
"don't do that."
```

RC_QUOTES doubled quotes

Finally, SUN_KEYBOARD_HACK wins the award for the strangest option. If a line ends with ```, and there are an odd number of them on the line, the shell will ignore the trailing ```. This is provided for keyboards whose RETURN key is too small, and too close to the ``` key.

```
% setopt sunkeyboardhack
% date`
Fri May 24 06:55:38 EDT 1991
```

SUN_KEYBOARD_HACK trailing backtick

Closing Comments

I (Bas de Bakker) would be happy to receive mail if anyone has any tricks or ideas to add to this document, or if there are some points that could be made clearer or covered more thoroughly. Please notify me of any errors in this document.

List of Examples

- [Example 1.1: Set option with setopt](#)
- [Example 1.2: Unset option with unsetopt](#)
- [Example 2.1: Basic globs with ls](#)
- [Example 2.2: EXTENDED_GLOB negation](#)
- [Example 2.3: Integer range globs](#)
- [Example 2.4: NUMERIC_GLOB_SORT with echo](#)
- [Example 2.5: Grouped alternation globs](#)
- [Example 2.6: Recursive directory globs](#)
- [Example 2.7: Glob exclusion with tilde](#)
- [Example 2.8: Qualifier pattern form](#)
- [Example 2.9: File type qualifiers](#)
- [Example 2.10: World-writable qualifiers](#)
- [Example 2.11: Dash prefix follows symlinks](#)
- [Example 2.12: Filter out symlinks](#)
- [Example 2.13: Plain file and pipe quals](#)
- [Example 2.14: Ownership qualifiers](#)
- [Example 2.15: Setuid qualifier](#)
- [Example 2.16: User qualifier match](#)
- [Example 3.1: Five startup files](#)
- [Example 4.1: Define a simple function](#)
- [Example 4.2: function keyword forms](#)
- [Example 4.3: Alias after function parse](#)
- [Example 4.4: Redefine with alias active](#)
- [Example 4.5: Loop over multiple args](#)
- [Example 4.6: Check \\$# for usage](#)
- [Example 4.7: Random line with RANDOM](#)
- [Example 4.8: View with functions builtin](#)

- [Example 4.9: chmod +x as function](#)
- [Example 4.10: Same trick as an alias](#)
- [Example 4.11: autoload with FPATH](#)
- [Example 4.12: Function as shell script](#)
- [Example 5.1: Tilde in the prompt](#)
- [Example 5.2: Named directory shortcuts](#)
- [Example 5.3: Long path in prompt](#)
- [Example 5.4: Shorten path via parameter](#)
- [Example 5.5: namedir helper function](#)
- [Example 5.6: chpwd for xterm title](#)
- [Example 5.7: chpwd for IRIS wsh](#)
- [Example 5.8: print -D shortens PWD](#)
- [Example 5.9: print -D shortens path](#)
- [Example 6.1: pushd and the directory stack](#)
- [Example 6.2: pushd swaps top two](#)
- [Example 6.3: pushd with one entry](#)
- [Example 6.4: PUSHD_TO_HOME option](#)
- [Example 6.5: Directory history with AUTOPUSHD](#)
- [Example 6.6: PUSHD_MINUS with cd +n](#)
- [Example 6.7: PUSHD_SILENT hides stack](#)
- [Example 6.8: DIRSTACKSIZE limits stack](#)
- [Example 7.1: Backtick and \\$\(...\) forms](#)
- [Example 7.2: TTY and =cmd shortcuts](#)
- [Example 7.3: Setting NO_EQUALS](#)
- [Example 7.4: Process sub with =\(...\)](#)
- [Example 7.5: print -l one per line](#)
- [Example 7.6: =\(...\) with multiline echo](#)
- [Example 7.7: Edit process sub output](#)
- [Example 7.8: Read compressed mail folder](#)
- [Example 7.9: diff two process outputs](#)
- [Example 7.10: Output process sub >\(...\)](#)
- [Example 8.1: Redirect to multiple files](#)
- [Example 8.2: Redirect and pipe together](#)
- [Example 8.3: Input from multiple files](#)
- [Example 8.4: Pipe then more input](#)
- [Example 8.5: Stderr via process sub](#)
- [Example 9.1: Simple command alias](#)
- [Example 9.2: Alias with fixed options](#)
- [Example 9.3: Replace commands with aliases](#)
- [Example 9.4: New commands via aliases](#)
- [Example 9.5: Global alias with -g](#)

- [Example 9.6: Global aliases for pipes](#)
- [Example 9.7: grep local passwd file](#)
- [Example 9.8: Global alias for ypcat](#)
- [Example 9.9: Alias over /etc/passwd](#)
- [Example 9.10: csh alias syntax fails](#)
- [Example 9.11: csh history in alias](#)
- [Example 9.12: History bang expands as glob](#)
- [Example 9.13: Safe rm as a function](#)
- [Example 9.14: Convert aliases with c2z](#)
- [Example 10.1: csh-style bang history](#)
- [Example 10.2: Repeat with !-n](#)
- [Example 10.3: Word designator :0](#)
- [Example 10.4: Last argument with !:\\$](#)
- [Example 10.5: Reuse words across events](#)
- [Example 10.6: Setting CSH_JUNKIE_HISTORY](#)
- [Example 10.7: Erroneous command for fc](#)
- [Example 10.8: Edit history with fc](#)
- [Example 10.9: Redo with r command](#)
- [Example 11.1: History recall with Ctrl+P](#)
- [Example 11.2: Recall multiline with Ctrl+P](#)
- [Example 11.3: Edit recalled multiline command](#)
- [Example 11.4: Execute edited multiline loop](#)
- [Example 11.5: History search with Esc+P](#)
- [Example 11.6: Incremental history search](#)
- [Example 11.7: Expand globs with Tab](#)
- [Example 11.8: List awkward filenames](#)
- [Example 11.9: Expand star with quoting](#)
- [Example 11.10: Delete all but foo.c](#)
- [Example 11.11: List matches with Ctrl+X g](#)
- [Example 11.12: Incomplete gcc command line](#)
- [Example 11.13: Repeat similar cc commands](#)
- [Example 11.14: String needing shell quotes](#)
- [Example 11.15: Type unquoted awkward string](#)
- [Example 11.16: After Esc+' quote-line](#)
- [Example 11.17: Echo after quote-line](#)
- [Example 11.18: Re-quote recalled echo](#)
- [Example 11.19: Quoted line as alias](#)
- [Example 11.20: Edit PATH with vared](#)
- [Example 12.1: Complete commands and filenames](#)
- [Example 12.2: Complete in middle of word](#)
- [Example 12.3: List completions with Ctrl+D](#)

- [Example 12.4: AUTO_LIST completion listing](#)
- [Example 12.5: LIST_AMBIGUOUS with Tab](#)
- [Example 12.6: Menu completion with Tab](#)
- [Example 12.7: Accept exact matches](#)
- [Example 12.8: Ignore suffixes with ignore](#)
- [Example 12.9: Complete tilde usernames](#)
- [Example 12.10: Parameter name completion](#)
- [Example 12.11: Option name completion](#)
- [Example 12.12: Completing bindkey arguments](#)
- [Example 12.13: compctl commands for man](#)
- [Example 12.14: compctl commands and files](#)
- [Example 12.15: compctl options variables bindings](#)
- [Example 12.16: compctl custom keyword lists](#)
- [Example 12.17: compctl globs for TeX tools](#)
- [Example 12.18: compctl directory-only globs](#)
- [Example 12.19: compctl RCS ,v files](#)
- [Example 12.20: compctl -s substitutions](#)
- [Example 12.21: compctl follow command args](#)
- [Example 12.22: compctl via reply function](#)
- [Example 12.23: compctl kill signal prefix](#)
- [Example 12.24: Xor completion for tex](#)
- [Example 12.25: Xor with default completion](#)
- [Example 12.26: Xor files and directories](#)
- [Example 13.1: r\[\] pattern for compress -d](#)
- [Example 13.2: s\[\] pattern for kill signals](#)
- [Example 13.3: c\[\] pattern for tar -f](#)
- [Example 13.4: C\[\] pattern for tar -f cluster](#)
- [Example 13.5: tar -f and old-style options](#)
- [Example 13.6: tar compressed vs plain files](#)
- [Example 13.7: Xor after extended completion](#)
- [Example 13.8: Flags before -x fallback](#)
- [Example 14.1: Default editor key bindings](#)
- [Example 14.2: bindkey for Ctrl+Z](#)
- [Example 14.3: bindkey delete-char](#)
- [Example 14.4: bindkey Ctrl+X H run-help](#)
- [Example 14.5: Miscellaneous bindkey examples](#)
- [Example 14.6: NeXT keyboard Alt bindings](#)
- [Example 15.1: Assigning a scalar](#)
- [Example 15.2: Spaces around equals fail](#)
- [Example 15.3: set leaves parameter empty](#)
- [Example 15.4: set fills positional params](#)

- [Example 15.5: csh-style set function](#)
- [Example 15.6: set function with options](#)
- [Example 15.7: Default and alternate values](#)
- [Example 15.8: csh colon modifiers](#)
- [Example 15.9: ksh pattern removal](#)
- [Example 15.10: Upper and lowercase modifiers](#)
- [Example 15.11: String substitution modifier](#)
- [Example 15.12: Parenthesized modifiers syntax](#)
- [Example 15.13: Sorting with o modifier](#)
- [Example 15.14: No default word splitting](#)
- [Example 15.15: Preserving quoted arguments](#)
- [Example 15.16: SH_WORD_SPLIT with \\$@](#)
- [Example 15.17: Forced word splitting](#)
- [Example 15.18: Array assignment for files](#)
- [Example 15.19: Global alias for files](#)
- [Example 15.20: RC_EXPAND_PARAM prefixes](#)
- [Example 16.1: Job and status parameters](#)
- [Example 16.2: Host UID and directories](#)
- [Example 16.3: Tilde shortcuts and misc](#)
- [Example 16.4: cdpath directory search](#)
- [Example 16.5: PATH and path linked](#)
- [Example 16.6: CDPATH and cdpath linked](#)
- [Example 16.7: Array assignment form](#)
- [Example 16.8: typeset tied parameters](#)
- [Example 16.9: Tied array and string](#)
- [Example 16.10: History file and size](#)
- [Example 16.11: Bare redirection alone](#)
- [Example 16.12: NULLCMD cat by default](#)
- [Example 16.13: Bare input shows file](#)
- [Example 16.14: NULLCMD as no-op](#)
- [Example 16.15: Unset NULLCMD errors](#)
- [Example 17.1: Default host prompt](#)
- [Example 17.2: Path and history escapes](#)
- [Example 17.3: Host time user escapes](#)
- [Example 17.4: Multiline PROMPT2 status](#)
- [Example 17.5: Right-side R PROMPT](#)
- [Example 17.6: print with -P escapes](#)
- [Example 17.7: POSTEDIT highlight trick](#)
- [Example 18.1: Watch list of users](#)
- [Example 18.2: log reports watched users](#)
- [Example 18.3: Watch by hostname](#)

- [Example 18.4: Watch by tty](#)
- [Example 18.5: Custom WATCHFMT formats](#)
- [Example 18.6: Watch from friends file](#)
- [Example 19.1: AUTO_CD without cd](#)
- [Example 19.2: CDABLE_VARS via parameter](#)
- [Example 19.3: CORRECT and CORRECT_ALL](#)
- [Example 19.4: Newlines in quoted strings](#)
- [Example 19.5: CSH_JUNKIE_QUOTES rejects](#)
- [Example 19.6: GLOB_DOTS matches dots](#)
- [Example 19.7: HIST_IGNORE_DUPS and SPACE](#)
- [Example 19.8: IGNORE_BRACES disables braces](#)
- [Example 19.9: IGNORE_EOF requires exit](#)
- [Example 19.10: INTERACTIVE_COMMENTS with #](#)
- [Example 19.11: NO_CLOBBER blocks overwrite](#)
- [Example 19.12: Force overwrite with >!](#)
- [Example 19.13: RC_QUOTES doubled quotes](#)
- [Example 19.14: SUN_KEYBOARD_HACK trailing backtick](#)

Index

A

ALWAYS_LAST_PROMPT
APPEND_HISTORY
AUTO_CD
AUTO_LIST
AUTO_MENU
AUTO_PARAM_KEYS
AUTO_PUSHD
AUTO_REMOVE_SLASH

C

CDABLE_VARS
COMPLETE_IN_WORD
CORRECT
CORRECT_ALL
CSH_JUNKIE_HISTORY
CSH_JUNKIE_QUOTES

E

EXTENDED_GLOB

G

GLOB_DOTS

H

HIST_IGNORE_DUPS
HIST_IGNORE_SPACE
HIST_VERIFY

I

IGNORE_BRACES
IGNORE_EOF
INTERACTIVE_COMMENTS

L

LIST_AMBIGUOUS
LIST_TYPES

M

MENU_COMPLETE

N

NO_BEEP
NO_CLOBBER
NO_FLOW_CONTROL
NO_LIST_BEEP
NUMERIC_GLOB_SORT

P

PUSHD_MINUS
PUSHD_SILENT
PUSHD_TO_HOME

R

RC_EXPAND_PARAM
RC_QUOTES
REC_EXACT

S

SUN_KEYBOARD_HACK